This chapter describes how your application can use the Device Manager to transfer information into and out of a Macintosh computer. The ***Device Manager*** controls the exchange of information between applications and hardware devices.

This chapter provides a brief introduction to devices and device drivers (the programs that control devices) and then explains how you can use the Device Manager functions to

- open, close, and exchange information with device drivers
- write your own device driver that can communicate with the Device Manager
- provide a user interface for your device driver by making it a Chooser extension or desk accessory.

You should read the sections "About the Device Manager" and "Using the Device Manager" if your application needs to use the Device Manager to communicate with a device driver. Applications often communicate with the Device Manager indirectly, by calling functions of other managers (for example, the File Manager) that use the Device Manager. However, sometimes applications must call Device Manager functions directly.

The sections "Writing a Device Driver," "Writing a Chooser-Compatible Device Driver," and "Writing a Desk Accessory," provide information you'll need if you are writing your own device driver.

If you writing a device driver, you should understand how memory is organized and allocated in Macintosh computers. See *Inside Macintosh: Memory*, for this information. You should also be familiar with resources and how the system searches resource files. You can find this information in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*. If your device driver is to perform background tasks, you'll need to understand how processes are scheduled. *Inside Macintosh: Processes* covers these topics. If your driver will control a hardware device, you should read *Designing Cards and Drivers for the Macintosh Family*, third edition.

Introduction to Devices and Drivers

A ***device*** is a physical part of the Macintosh, or a piece of external equipment, that can exchange information with applications or with the Macintosh Operating System. Input devices transfer information into the Macintosh, while output devices receive information from the Macintosh. An I/O device can transfer information in either direction.

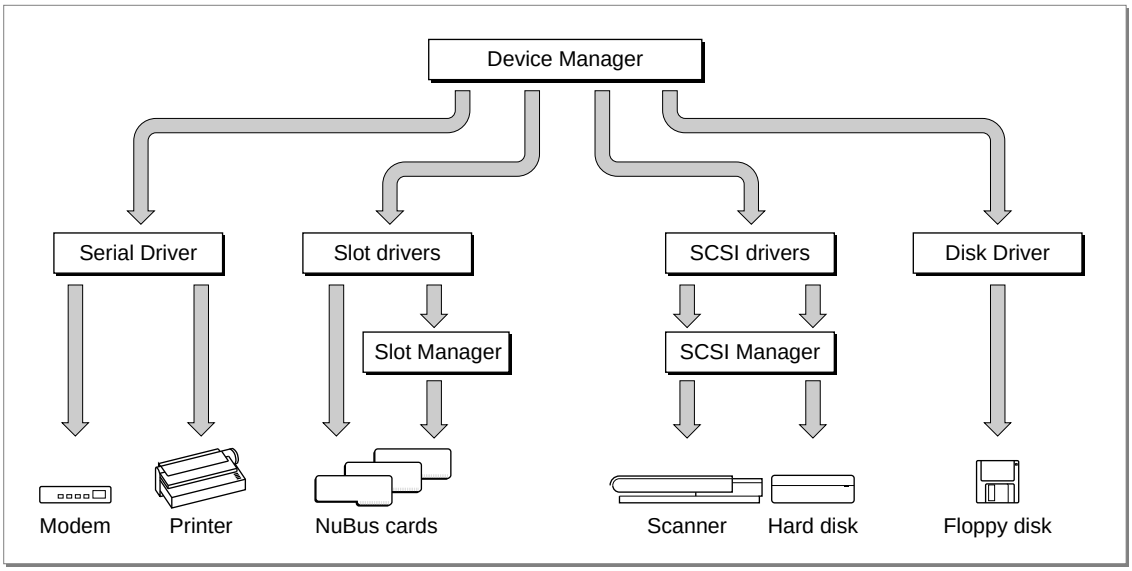
Devices transfer information in one of two ways. ***Character devices*** read or write a stream of characters, or bytes, one at a time. Character devices provide sequential access to data—they cannot skip over bytes in the data stream, and cannot go back to pick up bytes that have already passed. The keyboard and the serial ports are examples of character devices.

Block devices read and write blocks of bytes as a group. Disk drives, for example, can read and write blocks of 512 bytes or more. Block devices provide random access to data—they can read or write any block of data on demand.

Device Manager

Devices communicate with applications and with the Operating System through special programs called ***device drivers*** . A device driver typically controls a specific hardware device, such as a modem, hard disk, or printer. This type of device driver acts as a translator, converting software requests into hardware actions and hardware actions into software results. Figure 1-1 illustrates some of the hardware devices that communicate with the Macintosh through device drivers.

Figure 1-1 Devices and the Macintosh



Macintosh device drivers may be either synchronous or asynchronous. A ***synchronous device driver*** completes a requested transaction before returning control to the Device Manager. An ***asynchronous device driver*** can initiate a transaction and return control to the Device Manager before the transaction is complete. This type of device driver usually relies on interrupts from a hardware device to regain control of the processor and complete the transaction.

The Macintosh ROM and system software contain device drivers for controlling the standard devices included with every Macintosh computer, such as the mouse, serial ports, and floppy disk drive. Before deciding to write your own device driver, you should consider whether your device can be accessed using one of the standard device drivers. The section "Writing a Device Driver," beginning on page 1-24, discusses the reasons why you may want to use a standard device driver rather than writing your own.

Although device drivers are often used to control hardware, they are not restricted to this function. For example, Macintosh desk accessories and Chooser extensions are small programs that are written as device drivers, even though they may have nothing to do with controlling hardware. In general, a device driver is a program that conforms to a standard interface and provides access to a service through a standard set of routines.

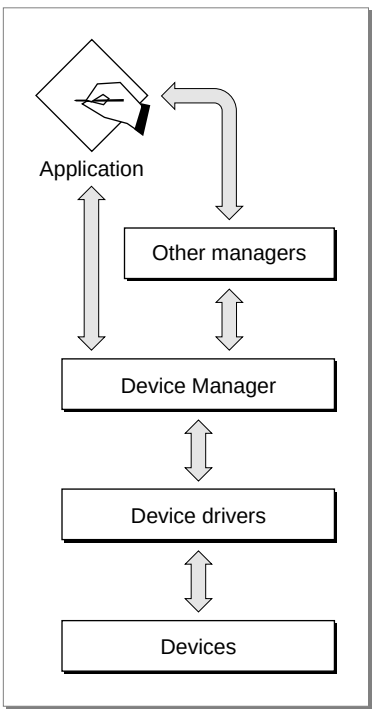
Your program can take advantage of this interface to perform tasks unrelated to actual physical devices.

About the Device Manager

The Device Manager provides a common programming interface for applications and other managers to use when communicating with device drivers. The Device Manager also includes support functions useful for writing your own device drivers.

Typically, your application won't communicate directly with device drivers; instead, it will call Device Manager functions or call the functions of another manager that calls the Device Manager. For example, your application can communicate with a disk driver by calling the Device Manager directly or by calling the File Manager, which calls the Device Manager. Figure 1-2 shows the relationship between applications, the Device Manager, other managers, device drivers, and devices.

Figure 1-2 Communication with devices



Before the Device Manager allows an application or another manager to communicate with a device driver, the driver must be open, which means the Device Manager has received a request to open the driver, has loaded the driver into memory, if necessary, and has successfully called the driver's open routine.

Your application opens a device driver using one of the Device Manager functions, `OpenDriver`, `OpenSlot`, or `PBOpen`. These functions return a ***driver reference number*** for the driver. You use the driver reference number to identify the driver in subsequent communication requests.

Your application communicates with a driver by calling Device Manager functions such as `FSRead` or `PBRead`, and supplying the driver reference number of the device. The Device Manager then invokes a corresponding routine in the device driver to perform the requested operation. The section "Driver Routines" on page 1-12 describes these routines and their relationship to the Device Manager functions.

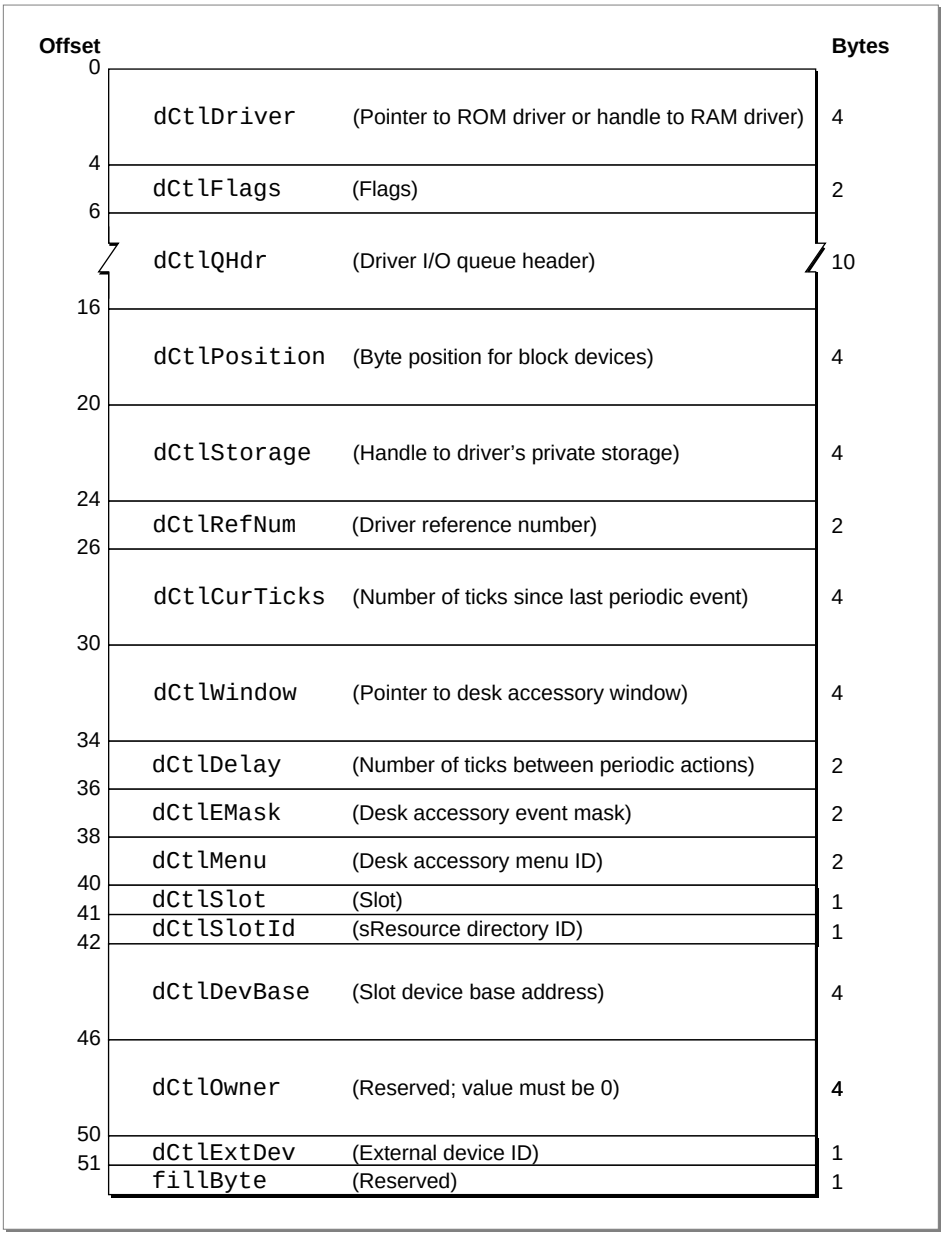
The Device Manager uses several data structures to locate, manage, and communicate with device drivers. These structures are described in the following sections.

The Device Control Entry

The Device Manager maintains a data structure called a ***device control entry*** (DCE) for each open driver. The device control entry is a relocatable block in the system heap that contains a handle or pointer to the device driver code, and additional information about the driver. Typically, the Device Manager maintains one device control entry for each open device driver, but it is possible for multiple entries to refer to the same driver.

Figure 1-3 shows the device control entry structure. See "Device Manager Reference," beginning on page 1-53, for descriptions of the fields within the device control entry structure.

Figure 1-3 The device control entry



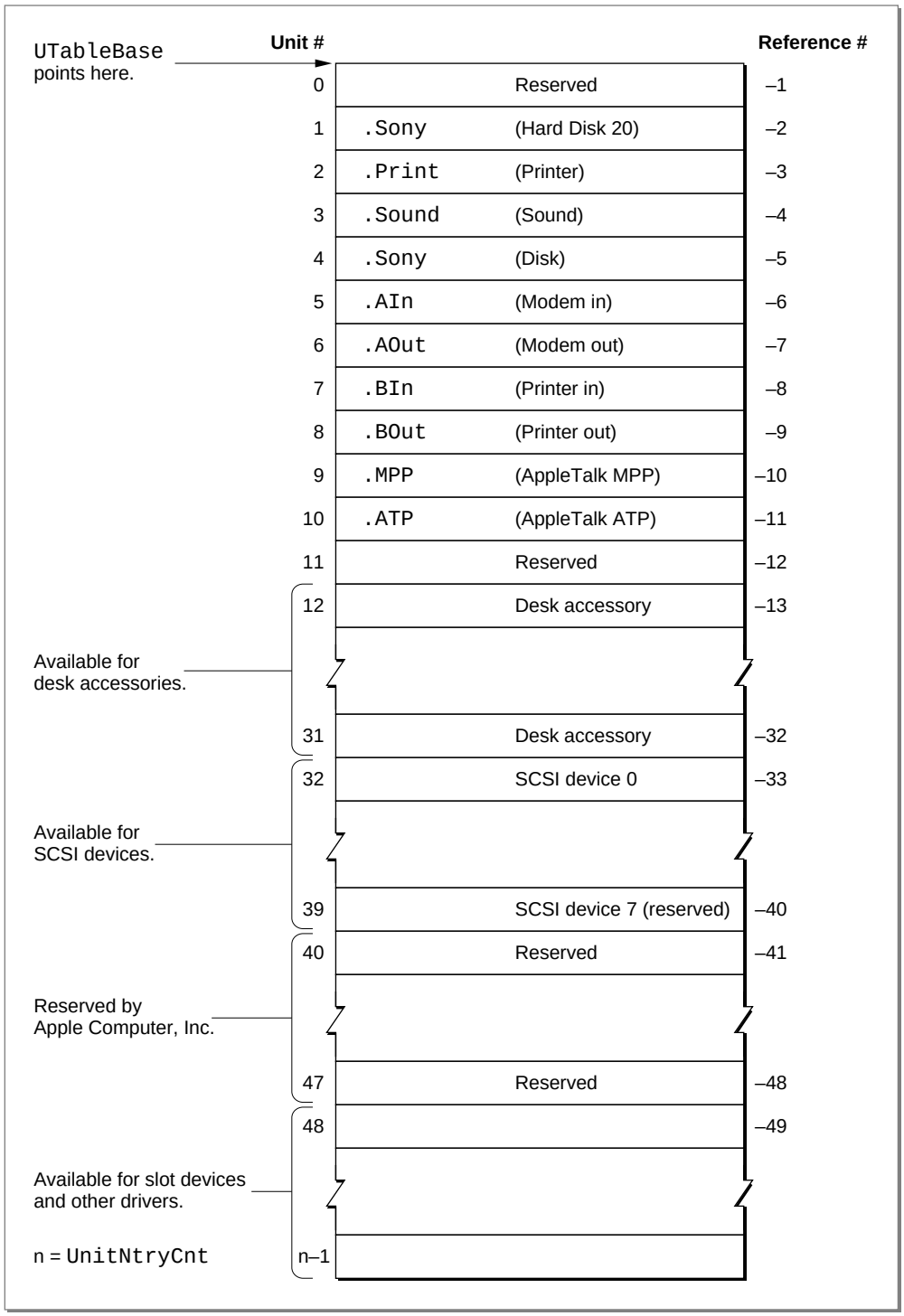
The Unit Table

The Device Manager uses a data structure called the ***unit table*** to organize and keep track of device control entries. The unit table is a nonrelocatable block in the system heap, containing an array of handles. Each handle points to the device control entry of an installed device driver. The location of a driver's device control entry handle in the unit table is called the driver's ***unit number*** . If the handle at a given unit number is `nil`, there is no device control entry installed in that position.

When you open a device driver, the Device Manager returns a driver reference number for the driver. The driver reference number is the one's complement (logical `NOT`) of the unit number.

The system global variable `UTableBase` points to the first entry of the unit table. The system global variable `UnitNtryCnt` contains the size of the unit table (that is, how many handles it can hold). Figure 1-4 shows the organization of the unit table, including the locations of some of the standard device drivers reserved by Apple Computer, Inc.

Figure 1-4 The unit table



The Driver I/O Queue

The Device Manager maintains an I/O queue for each open device driver. An I/O queue is a standard Macintosh Operating System queue of type `ioQType`, as described in the chapter "Queue Utilities" in *Inside Macintosh: Operating System Utilities*.

At the head of a device driver's I/O queue is the request currently being processed by the driver. The rest of the queue contains pending I/O requests—those the Device Manager has received but not yet sent to the device driver. This queue allows your application to request a data transfer with a busy device and accomplish other tasks while the device processes previous requests.

With respect to the I/O queue, the Device Manager allows you to make three types of requests: asynchronous, synchronous, and immediate.

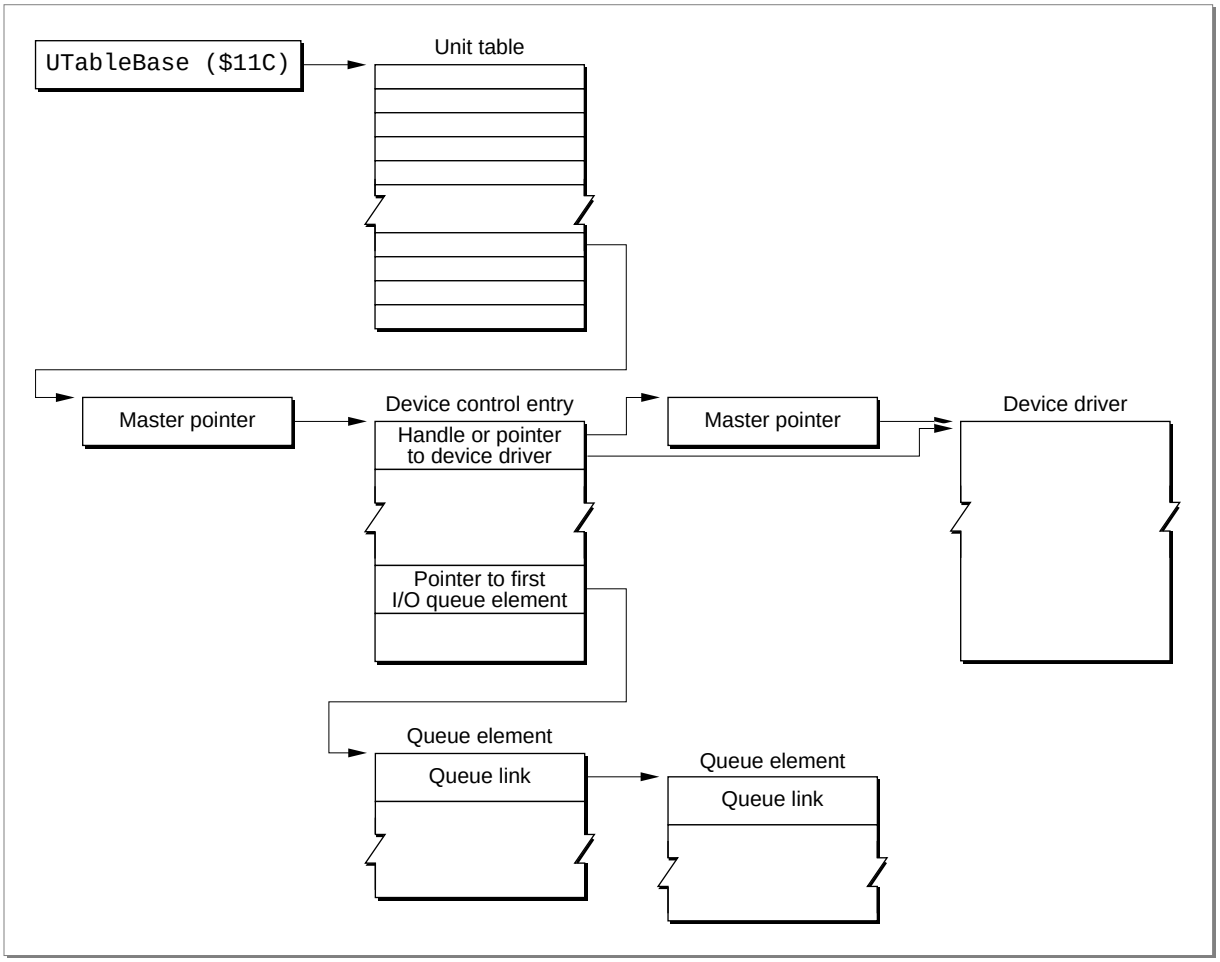
- ***Asynchronous requests.*** When you make an asynchronous request, the Device Manager places your request at the end of the driver I/O queue and returns control to your application—potentially before the request is processed. Your application is free to perform other tasks while the device driver processes the requests in its queue. The Device Manager provides mechanisms for your application to determine when the driver has processed the request.
- ***Synchronous requests.*** When you make a synchronous request, the Device Manager places your request at the end of the queue and waits until the device driver has handled every request in the queue, including the synchronous one, before returning control to your application. Notice there can never be more than one synchronous request in a driver I/O queue at any given time.
- ***Immediate requests.*** The Device Manager sends immediate requests directly to the device driver, bypassing the queue, and returns control to your application when the request is complete. Because the device driver might be in the middle of processing another request, you must make sure the driver is reentrant before making an immediate request. A ***reentrant driver*** is capable of handling multiple requests simultaneously. As some device drivers are not reentrant, you should always consult a driver's documentation to determine if it supports immediate requests.

IMPORTANT

The terms *synchronous* and *asynchronous* are used here to describe how the Device Manager queues your I/O requests. How a device driver processes these requests (synchronously or asynchronously) depends on the design of the driver. When you make a synchronous request to a device driver, the Device Manager waits for the driver to complete the request, regardless of whether the driver handles the request synchronously or asynchronously. ▲

Figure 1-5 shows the relationship of the unit table, device control entry, and I/O queue to a device driver.

Figure 1-5 Relationship of the Device Manager data structures



Driver Routines

Every device driver must provide a set of routines for handling requests from the Device Manager. When an application or another manager calls a Device Manager function, the Device Manager invokes one of the following routines in the designated device driver:

- The ***open routine*** allocates memory and initializes the device driver's data structures. It may also initialize a hardware device or perform any other tasks necessary to make the driver operational. All device drivers must implement an open routine.
- The ***close routine*** deactivates the device driver, releases any memory allocated by the driver, removes any patches installed by the driver, and performs any other tasks necessary to reverse the actions of the open routine. All drivers must implement a close routine.
- The ***control routine*** is usually used to send control information to the device driver. The function of this routine is driver-dependent. This routine is optional and need not be implemented.
- The ***status routine*** is usually used to return status information from the device driver. The function of this routine is driver-dependent. The status routine is optional and need not be implemented.
- The ***prime routine*** implements the input and output functions of the driver. This routine is optional. If the prime routine is implemented, it must support either read functions or write functions, or both.

Each driver routine is responsible for handling specific types of Device Manager requests. Table 1-1 shows the Device Manager I/O functions and the driver routines responsible for handling them. The Device Manager I/O functions are described in "Using the Device Manager," beginning on page 1-14. The section "Writing a Device Driver," beginning on page 1-24, describes the driver routines.

Table 1-1 Device Manager I/O functions and responsible driver routines

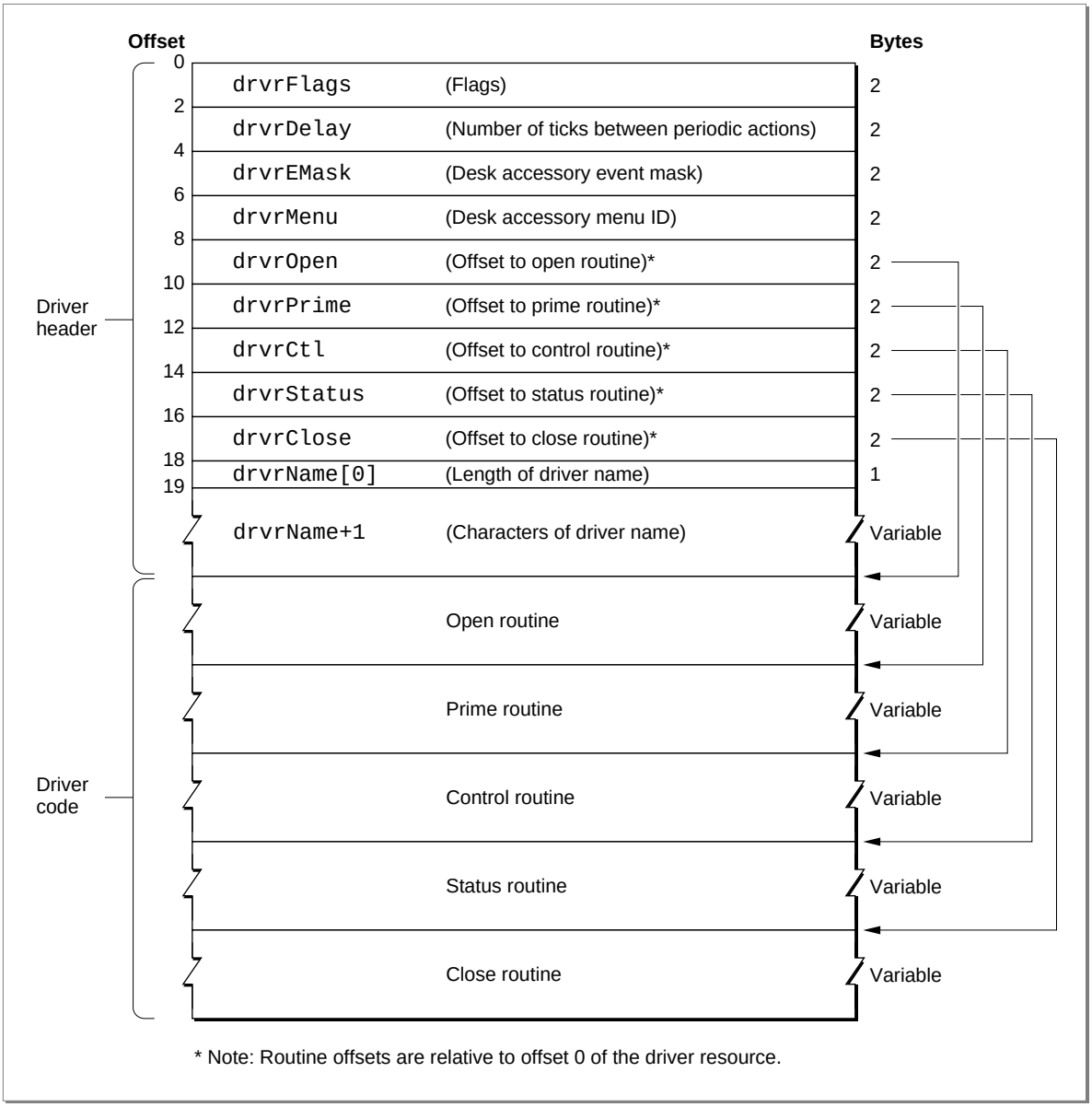
Device Manager function	Responsible driver routine
`OpenDriver, PBOpen, OpenSlot`	Open
`FSRead, PBRead`	Prime
`FSWrite, PBWrite`	Prime
`Control, PBControl`	Control
`Status, PBStatus`	Status
`KillIO, PBKillIO`	Control
`CloseDriver, PBClose`	Close

Driver Resources

Device drivers are usually stored in driver resources, which can be located in applications, system extension files, or the firmware of expansion cards. A driver

resource consists of a header followed by the driver code. The header contains information about the driver such as which driver routines are implemented and where the routines are located within the driver code. The Device Manager copies the relevant information from the header into the device control entry when you open the driver. Figure 1-6 shows the structure of a driver resource. The section "Creating a Driver Resource," beginning on page 1-24, describes driver resources in detail.

Figure 1-6 Structure of a driver resource



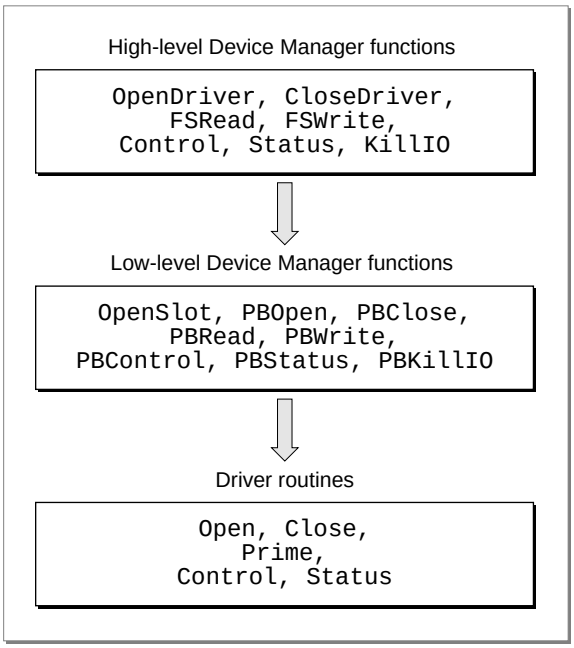
Using the Device Manager

Your application can use Device Manager functions to communicate with devices through their device drivers. This section describes the Device Manager functions that allow you to open, close, and control device drivers, exchange information with them, and monitor their status. The Device Manager also provides support functions useful for writing and installing device drivers. The section "Writing a Device Driver," beginning on page 1-24, describes these support functions.

The Device Manager includes high-level and low-level versions of most of its functions. The high-level versions are somewhat easier to use, but they allow less control of how the Device Manager processes the I/O request (for example, they are always handled synchronously) and they return less information to your application. Conversely, the low-level functions require some additional setup, but they allow you greater control and return more information.

The high-level Device Manager functions call the low-level functions, which in turn call the appropriate driver routine. For example, the Device Manager converts the high-level `FSRead` function to a low-level `PBRead` function before calling the driver's prime routine. Figure 1-7 depicts this hierarchy.

Figure 1-7 Hierarchy of Device Manager functions



The high-level functions differ in form, but the low-level functions all have the form:

```
pascal OSErr PBRoutineName (ParmBlkPtr paramBlock, Boolean async);
```

The `paramBlock` parameter is a pointer to a structure of type `ParamBlockRec`. You use the fields of this structure to pass more complete information to the driver than you can with high-level functions, and the driver uses the same structure to pass information back. The `ParamBlockRec` is defined in C as a union of six structures, but only the `IOParam` and `CntrlParam` types are used by the Device Manager. Figure 1-8 shows the fields of the `ParamBlockRec` structure used by the Device Manager. These fields are described in detail later in this section and in "Data Structures" on page 1-53.

The `async` parameter specifies whether the Device Manager should process the function asynchronously. For synchronous requests you set this parameter to `false`; the Device Manager adds the parameter block to the driver I/O queue and waits until the driver completes the request (which means it has completed all previously queued requests) before returning control to your application.

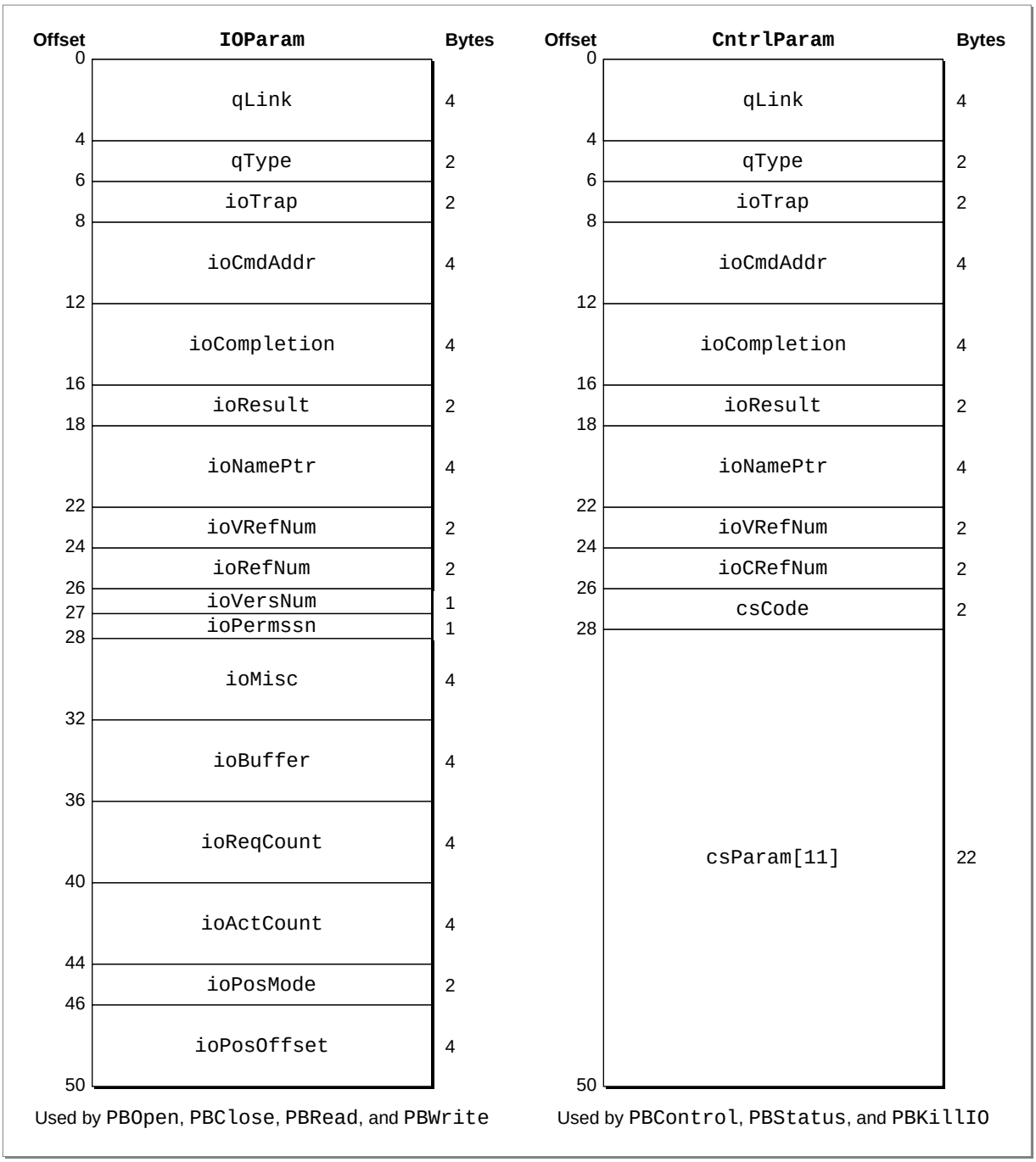
▲ **WARNING**
Never call any Device Manager function synchronously at interrupt time. A synchronous request at interrupt time may block other pending I/O requests. Because the device driver cannot begin processing the synchronous request until it completes the other requests in its queue, this situation can cause the Device Manager to loop indefinitely while it waits for the device driver to complete the synchronous request. ▲

If you set the `async` parameter to `true`, the Device Manager adds the parameter block to the driver I/O queue and returns control to your application immediately. In this case, a `noErr` result code signifies that the request was successfully queued, not that the request was successfully completed. The Device Manager sets the `ioResult` field of the parameter block to 1 when the request is queued, and stores the actual result code there when the driver indicates the request is complete.

When you make an asynchronous request you can also provide a pointer to a completion routine in the `ioCompletion` field of the parameter block. The Device Manager executes this routine when the driver completes the asynchronous request. Your completion routine could, for example, set a flag to signal your application that the I/O operation is complete. See "Handling Asynchronous I/O," beginning on page 1-37, for more information about completion routines and asynchronous operation.

Assembly-Language Note
You can call a Device Manager function immediately, bypassing the I/O queue, by setting bit 9 of the trap word. You can set or test this bit using the global constant `noQueueBit`. However, remember that the device driver might be processing another request, especially if you make an immediate request during interrupt time. The driver must be reentrant to handle this situation properly. You should always check a driver's documentation to make sure the driver is reentrant before making immediate requests. ◆

Figure 1-8 Device Manager parameter blocks



When you use a low-level Device Manager function, the Device Manager places the parameter block at the end of the driver I/O queue and then either waits for the driver to complete the request or returns control to your application, depending on the value of

the `async` parameter. For the high-level functions, the Device Manager creates a parameter block for you, filling the required fields with the values you supplied. The Device Manager then inserts the parameter block at the end of the I/O queue as a synchronous request. As previously-queued requests are processed, the parameter block moves forward in the I/O queue. When the parameter block is at the beginning of the queue, the Device Manager calls the appropriate driver routine and passes it a pointer to the parameter block and a pointer to the driver's device control entry.

For read and write requests, the Device Manager calls the driver's prime routine. This routine can execute synchronously, completing the requested read or write transaction before returning control to the Device Manager, or asynchronously, beginning the requested transaction but returning control to the Device Manager before completing it. For information about reading and writing data to devices, see "Communicating With Device Drivers" on page 1-20.

If you are writing a device driver and your driver's prime routine can execute asynchronously, your driver must use some mechanism to regain control of the processor to complete asynchronous requests. Your driver would typically use an interrupt handler for this purpose, and notify the Device Manager when the transaction is complete. See "Writing a Prime Routine" on page 1-33 and "Handling Asynchronous I/O" on page 1-37 for more information about writing asynchronous routines.

The Device Manager handles control and status requests in the same way as read and write requests, except that for control requests it calls the control routine and for status requests it calls the status routine. See "Controlling and Monitoring Device Drivers" on page 1-22 for information about making these requests. For information about providing status and control routines for your own driver, see "Writing Control and Status Routines" on page 1-34.

The Device Manager responds to `KillIO` requests by calling the device driver's control routine with a value of `killCode` for the `csCode` parameter. If the driver returns `noErr`, the Device Manager removes all parameter blocks from the queue, calling their completion routines with the result code `abortErr`. For more information about canceling I/O requests, see the description of the `KillIO` function on page 1-80. For information on how your driver can handle `KillIO` requests, see "Writing Control and Status Routines" on page 1-34.

In response to a close request, the Device Manager waits until the driver is inactive, then calls the driver's close routine. When the driver indicates it has processed the close request, the Device Manager unlocks the driver resource if the `dRAMBased` flag is set, and unlocks the device control entry if the `dNeedLock` flag is not set. The Device Manager does not release the driver resource or dispose of the device control entry unless you call the `DriverRemove` function. The next section describes how to open and close a device driver. See "Writing Open and Close Routines" on page 1-31 for information about how your driver should respond to open and close requests.

Opening and Closing Device Drivers

You must open a driver before your application can communicate with it. The Device Manager provides three functions for opening device drivers: `OpenDriver`, `OpenSlot`, and `PBOpen`. Each of these functions requires a driver name and returns a driver reference number.

A driver name consists of a period (.) followed by any sequence of 1 to 254 printing characters; for example, .ATP is the name of one of the high-level AppleTalk drivers. The initial period in a driver name allows the Device Manager and the File Manager, which share the `_Open` trap, to distinguish between driver names and filenames. Refer to a device driver's documentation to determine the driver name.

The `OpenDriver` function, which is the high-level function for opening a device driver, takes the driver name as its first parameter and returns the driver reference number in its second parameter. When an application or another manager calls the `OpenDriver` function, the Device Manager first searches the unit table to see if a driver with the specified name is already installed. If the name does not match any installed driver, the Device Manager searches the current Resource Manager search path for a driver resource with the specified name.

To open a device driver from a resource, the Device Manager

- creates a device control entry for the driver, filling in the DCE with values from the header of the driver resource
- installs a handle to the device control entry in the unit table at a location determined by the driver resource ID
- calls the driver's open routine

Listing 1-1 shows an application-defined function that uses the `OpenDriver` function to open a driver.

Listing 1-1 Opening a device driver

```
short    gDrvrRefNum; /* global variable for storing
                         my driver reference number */

OSErr MyOpenDriver(void)
{
   Handle   drvrHdl;
   short    drvrID;
   short    tempDrvrID;
   ResType  drvrType;
   Str255   drvrName;
   OSErr    myErr;

   tempDrvrID = MyFindSpaceInUnitTable(); /* see Listing 1-14 */
```

```
   if (tempDrvrID > 0)
   {
      drvrHdl = GetNamedResource((ResType)'DRVR', "\p.MYDRIVER");
      GetResInfo(drvrHdl, &drvrID, &drvrType, drvrName);
      SetResInfo(drvrHdl, tempDrvrID, drvrName);

      myErr = OpenDriver("\p.MYDRIVER", &gDrvrRefNum);

      if (myErr == noErr)
         DetachResource(drvrHdl);

      drvrHdl = GetNamedResource((ResType)'DRVR', drvrName);
      SetResInfo(drvrHdl, drvrID, drvrName);

      return(myErr);
   }
   else
      return(openErr); /* no space in the unit table */
}
```

The `OpenDriver` function uses the resource ID of the driver resource as the unit number for the device driver, which determines where the device control entry will be stored in the unit table. Because the `OpenDriver` function does not check to see if another device control entry is already located at that position in the unit table, the `MyOpenDriver` function begins by searching for an available space in the unit table. Listing 1-14 on page 1-39 shows the `MyFindSpaceInUnitTable` function.

If there is room in the unit table, the `MyOpenDriver` function calls `GetNamedResource` to load the resource into memory, then changes the ID of the driver resource in the resource map before calling the `OpenDriver` function.

After the driver is open, `MyOpenDriver` calls the `DetachResource` function to prevent the driver resource from being released. Finally, `MyOpenDriver` restores the original resource ID so that the driver's resource file remains unchanged.

You can use the `PBOpen` or `OpenSlot` functions instead of the `OpenDriver` function when you want more control over how the Device Manager opens the device driver. For example, you can set read and write permissions for the device with the `ioPermssn` field of the parameter block. Use the `OpenSlot` function to open drivers that serve slot devices, and the `PBOpen` function for all other drivers.

Because the Device Manager always opens device drivers synchronously, you must set the `async` parameter to `false` when using the `PBOpen` or `OpenSlot` functions. If a device driver is already open, the Device Manager simply returns the driver reference number.

The remaining Device Manager functions require your application to use the driver reference number, instead of the driver name, when referring to a device driver.

When you finish using a driver, you may want to close it. However, you do not normally close drivers that might be needed by the system or by other applications. Whether you should close a particular driver depends on the type of driver and how it is being used. Refer to the driver's documentation to determine if it should be closed. See the appropriate chapters in this book and other books in the *Inside Macintosh* series for information about standard Macintosh drivers.

If you do want to close a driver, you can use the high-level `CloseDriver` function or the low-level `PBClose` function. Listing 1-2 shows how to use the `PBClose` function to close the driver opened in Listing 1-1.

Listing 1-2 Closing a device driver

```
OSErr MyCloseDriver(short refNum)
{
   IOParam  paramBlock;

   paramBlock.ioRefNum = refNum;

   return(PBClose((ParmBlkPtr)&paramBlock, false));
}
```

The `MyCloseDriver` function specifies the driver to close by placing the driver reference number in the `ioRefNum` field of the parameter block and then calls the Device Manager `PBClose` function.

Communicating With Device Drivers

Once a device driver is open and you have its reference number, you can use Device Manager functions to exchange information with it. When you want to receive information from a device driver, you first allocate a data buffer to hold the information and then call the `FSRead` or `PBRead` function. To send information to a device driver, you first store the information in a data buffer and then call the `FSWrite` or `PBWrite` function. You must specify the number of bytes you want transferred when calling any of these functions.

The `PBRead` and `PBWrite` functions support asynchronous requests, and allow you to specify a completion routine. For block devices you specify the drive number, positioning mode, and positioning offset in the `ioVRefNum`, `ioPosMode`, and `ioPosOffset` fields of the parameter block. The Device Manager does not interpret these fields—they are used by the device driver to locate the desired data block.

The Macintosh Operating System defines three positioning modes for block devices:

- At the current position. Transfer begins at the current position on the medium—typically where the last transfer ended.

- Offset from the start. Transfer begins at the specified offset from the beginning of the medium.
- Offset from the mark. Transfer begins at the specified offset from the current position.

You specify the positioning mode by setting the `ioPosMode` field to one of the defined constants, `fsAtMark`, `fsFromStart`, or `fsFromMark`. Be sure you specify a mode that is compatible with the device.

On completion, the `PBRead` and `PBWrite` functions return in the `ioActCount` field of the parameter block the total number of bytes actually transferred. For block devices, these functions also return a new positioning offset in the `ioPosOffset` field.

Certain device drivers provide additional abilities with the read and write functions. For example, the Disk Driver allows you to use the `PBRead` function to verify that data written to a block device matches the data in memory. To do this, you add the read-verify constant `rdVerify` to the value in the `ioPosMode` field of the parameter block, as explained in the description of the `PBRead` function on page 1-70.

Listing 1-3 shows an example of how to read from a device driver.

Listing 1-3 Reading from a device driver

```
OSErr MyReadFromDriver(short refNum)
{
   IOParam  paramBlock;
   char     buffer[256];

   paramBlock.ioRefNum = refNum;
   paramBlock.ioReqCount = 256;
   paramBlock.ioBuffer = (Ptr)buffer;

   return(PBRead((ParmBlkPtr)&paramBlock, false));
}
```

The `MyReadFromDriver` function uses a parameter block to specify the device driver (by its driver reference number), the number of bytes to read, and a pointer to a buffer to receive the data. When `MyReadFromDriver` calls the `PBRead` function, the Device Manager appends the parameter block to the end of the driver I/O queue. Because the `async` parameter is set to `false`, the Device Manager does not return control to `MyReadFromDriver` until the driver has completed every request in its queue.

Listing 1-4 shows an example of how to write to a device driver.

Listing 1-4 Writing to a device driver

```
OSErr MyWriteToDriver(short refNum)
{
   IOParam  paramBlock;
   char*    buffer;

   buffer = "Data to Write";

   paramBlock.ioCompletion = nil;
   paramBlock.ioRefNum = refNum;
   paramBlock.ioBuffer = (Ptr)buffer;
   paramBlock.ioReqCount = strlen(buffer);

   return(PBWrite((ParmBlkPtr)&paramBlock, false));
}
```

The `MyWriteToDriver` function also uses a parameter block to transfer information to the driver. After filling in the necessary fields, `MyWriteToDriver` sends the parameter block to the `PBWrite` function. Because the `async` parameter is `false`, the Device Manager appends the parameter block to the end of the I/O queue and does not return control to the `MyWriteToDriver` function until the driver has completed the request.

Controlling and Monitoring Device Drivers

In addition to the read and write functions, the Device Manager provides functions that allow your application to control and monitor device drivers in other ways.

The `Control` and `PBControl` functions send commands to a driver. Because the types of commands to which drivers respond varies, you need to consult a driver's documentation to determine what commands it accepts. As an example, you can send a command to the Disk Driver requesting that it eject a disk.

The `Status` and `PBStatus` functions return status information from a driver. Again, the type of information drivers provide varies widely. The Serial Driver, for example, can return a breakdown of the types of errors that have occurred recently.

The control and status functions use the `CntrlParam` structure of the `ParamBlockRec` union. This structure is defined in "Device Manager Parameter Block," beginning on page 1-53.

Because of the diversity of device drivers, the control and status functions have two general-purpose parameters: `csCode` and `csParamPtr` (or `csParam` for the low-level `PBControl` and `PBStatus` functions). You indicate the type of control or status information you are requesting by placing a driver-specific code in the `csCode` parameter. You send or receive information using the `csParamPtr` parameter.

Listing 1-5 shows an example of how to send control and status requests to a device driver using the `PBControl` and `PBStatus` functions.

Listing 1-5 Controlling and monitoring a device driver

```
OSErr MyIssueDriverControl(short refNum)
{
   CntrlParam  paramBlock;

   paramBlock.ioCRefNum = refNum;
   paramBlock.csCode = kClearAll;   /* driver-specific control request */

   return(PBControl((ParmBlkPtr)paramBlock, false));
}

OSErr MyGetDriverStatus(short refNum)
{
   CntrlParam  paramBlock;
   OSErr       myErr;
   short       count;

   paramBlock.ioCRefNum = refNum;
   paramBlock.csCode = kByteCount; /* driver-specific status request */

   myErr = PBStatus((ParmBlkPtr)&paramBlock, false);

   count = paramBlock.csParam[0]; /* value returned in csParam array */
   if (myErr == noErr)
      return(count);
   else
      return(myErr);
}
```

The `MyIssueDriverControl` and `MyGetDriverStatus` functions call the example device driver control and status routines shown in Listing 1-12 on page 1-35 and Listing 1-13 on page 1-36.

The `MyIssueDriverControl` function begins by setting up the fields of a parameter block. The `ioCRefNum` field specifies the driver reference number, and the `csCode` field specifies the type of control information being sent. The `MyDriverControl` function shown in Listing 1-12 interprets the driver-specific value `kClearAll` as a request for the device driver to clear the information in its private storage.

The `MyGetDriverStatus` function also begins by setting up the fields of a parameter block. The `ioCRefNum` field specifies the device driver reference number, and the `csCode` field specifies the type of status information being requested. The `MyDriverStatus` function shown in Listing 1-13 interprets a value of `kByteCount` as a request to return the number of bytes transferred by the last I/O operation. This information is returned in the `csParam` field of the parameter block.

Writing a Device Driver

This section shows you how to write a basic device driver—one that can respond to Device Manager requests. Although you will need to write some assembly-language interface code, you can write your device driver routines in a high-level language.

Before you decide to write your own device driver, you should consider whether your task can be more easily accomplished using one of the standard Macintosh drivers described in this book or other *Inside Macintosh* volumes. In general, you should consider writing a device driver only if your hardware device or system service needs to be accessed at unpredictable times or by more than one application.

For example, if you develop a new output device that you want to make available to any application, you might need to write a custom driver. On the other hand, if your product is a specialized device that can only be used by your application, it may be easier to control the device using private code within your application.

This section describes how to

- create a driver resource
- write the code in your driver resource so that it responds appropriately to Device Manager requests
- handle the special requirements of asynchronous I/O
- install and initialize your driver

Creating a Driver Resource

You will probably want to store your device driver in a driver resource, although if you are writing a driver for a slot device, you might want to store your driver in an sResource data structure in the declaration ROM of the expansion card. See the chapter "Slot Manager" in this book for information about sResource data structures.

Storing your driver in a driver resource allows the Device Manager to load your driver code into memory and install a device control entry for your driver in the unit table. Like all resources, your driver resource has a resource type, a resource ID, a resource name, and resource attributes.

- The resource type must be `'DRVR'` if you plan to use the Device Manager to load your driver into memory. If you write your own routine to load the driver, you can choose a different resource type.
- The resource ID determines where in the unit table the Device Manager installs the driver's device control entry. Because you must choose the resource ID when creating your driver resource, you cannot know which unit numbers are available until you open your driver. Therefore, your driver-opening routine must find an empty location in the unit table and change the resource ID accordingly. "Installing a Device Driver" on page 1-38 discusses appropriate values for the resource ID.

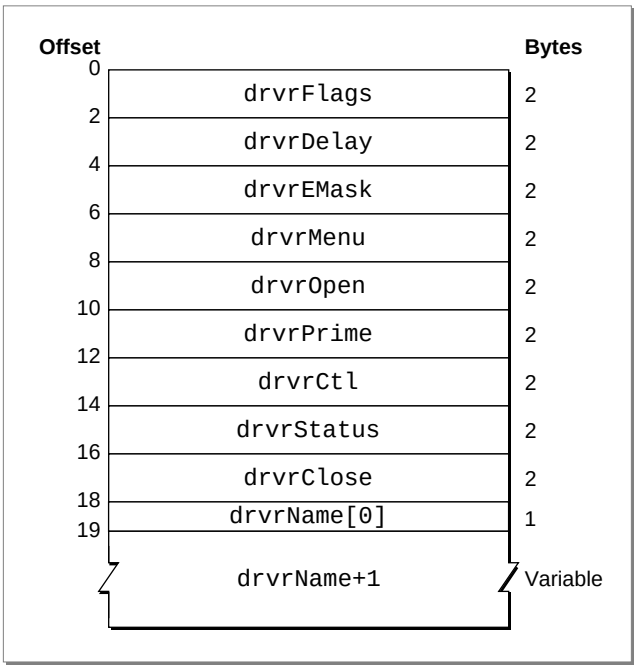
- The resource name should be the same as the driver name because the Device Manager calls `GetNamedResource` using this name if it can't find the driver in the unit table. A driver name consists of a period (.) followed by any sequence of 1 to 255 printing characters. The Device Manager ignores case (but not diacritical marks) when comparing names.
- The resource attributes of your driver resource depend on your driver. A typical driver might have these attributes: locked, since most drivers contain code that is called at interrupt time; in the system heap, so that the driver exists over launches of applications; and preloaded, which makes resource loading slightly more efficient.

A driver resource has two parts:

- a driver header that contains information about the driver
- the routines that do the work of the driver

The driver header contains a few words of flags and other data, offsets to the driver's routines, and an optional driver name. Figure 1-9 shows the format of a driver header.

Figure 1-9 The driver header



The elements of the driver header are:

Element	Description
`drvrFlags`	Flags in the high-order byte of this field specify certain characteristics of the driver. These flags are copied to the high-order byte of the `dCtlFlags` field of the device control entry when the

driver is opened. You can use the constants shown in Listing 1-6 to set or test the flags in this field.

Name	Bit	Meaning
`dReadEnable`	8	Set if the driver can respond to read requests.
`dWritEnable`	9	Set if the driver can respond to write requests.
`dCtlEnable`	10	Set if the driver can respond to control requests.
`dStatEnable`	11	Set if the driver can respond to status requests.
`dNeedGoodbye`	12	Set if the driver needs to be called before the application heap is reinitialized.
`dNeedTime`	13	Set if the driver needs time for performing periodic tasks.
`dNeedLock`	14	Set if the driver needs to be locked in memory as soon as it is opened.

`drvrDelay` If the `dNeedTime` flag is set, this field contains the requested number of ticks between periodic actions. This value is approximate and should not be used as a timing reference.

`drvrEMask` Used only by desk accessories, this field contains an event mask. See "Writing a Desk Accessory" on page 1-49 for information about this field.

`drvrMenu` Used only by desk accessories, this field contains a menu ID. See "Writing a Desk Accessory" on page 1-49 for more information.

`drvrOpen` The offset of the driver's open routine, relative to offset 0 of the driver header.

`drvrPrime` The offset of the driver's prime routine.

`drvrCtl` The offset of the driver's control routine.

`drvrStatus` The offset of the driver's status routine.

`drvrClose` The offset of the driver's close routine.

`drvrName` A Pascal string containing the driver's name, up to 255 characters.

See the section "Entering and Exiting From Driver Routines" on page 1-29 for more information about the routine offsets.

Note

Your driver routines, which follow the driver header, must be aligned on a word boundary. ◆

Listing 1-6 Driver flag constants

```
enum {
   /* flags used in the driver header and device control entry */
   dNeedLockMask    = 0x4000, /* set if driver must be locked in memory as
                                 soon as it's opened */
   dNeedTimeMask    = 0x2000, /* set if driver needs time for performing
                                 periodic tasks */
   dNeedGoodByeMask = 0x1000, /* set if driver needs to be called before the
                                 application heap is initialized */
   dStatEnableMask  = 0x0800, /* set if driver responds to status requests */
   dCtlEnableMask   = 0x0400, /* set if driver responds to control requests*/
   dWritEnableMask  = 0x0200, /* set if driver responds to write requests */
   dReadEnableMask  = 0x0100, /* set if driver responds to read requests */
};
```

The `dReadEnable`, `dWritEnable`, `dCtlEnable`, and `dStatEnable` flags indicate which Device Manager requests the device driver can respond to. The next section, "Responding to the Device Manager," describes these routines in detail.

Drivers in the application heap are lost when the heap is reinitialized. If you set the `dNeedGoodbye` flag, the Device Manager calls your driver before the heap is reinitialized so that you can perform any clean-up actions. See "Writing Control and Status Routines," beginning on page 1-34, for information about using this flag.

You set the `dNeedTime` flag if your device driver needs to perform some action periodically. For example, a network driver may want to poll its input buffer every 5 seconds to see if it has received any messages. The value of the `drvrDelay` field indicates how many ticks should pass between periodic actions. For example, a value of 0 in the `drvrDelay` field indicates that the action should happen as often as possible, a value of 1 means it should happen every sixtieth of a second, a value of 2 means at most every thirtieth of a second, and so on. Whether the action actually occurs this frequently depends on how often an application calls `WaitNextEvent` or `SystemTask`. See "Writing Control and Status Routines," beginning on page 1-34, for information about using this flag.

Note
If you do not want your driver to depend on applications to call `WaitNextEvent` or `SystemTask`, you can perform actions periodically by installing a VBL task, a Deferred Task Manager task, a Time Manager task, or a Notification Manager task. For more information, see *Inside Macintosh: Processes*. ◆

You need to set the `dNeedLock` flag if your device driver's code must be locked in memory. In particular, you need to set this flag in these two cases:

- If any part of your driver's code can be called at interrupt time. Because the Operating System may perform memory management at interrupt time, your driver must be locked to prevent it from being moved.

- If your driver provides the Operating System with a pointer to any part of its code. For example, if your driver uses the Device Manager to call another driver, you might provide the Device Manager with a pointer to a completion routine. If that completion routine is in your driver code, your driver code must be locked. Otherwise, that pointer might not be valid when the Device Manager calls the completion routine.

You can create your driver header in these ways:

- You can use a resource compiler. See "Resources" on page 1-89 for the Rez format of the driver resource.
- You can use the `DC` instruction, as shown in Listing 1-7, to position the header information directly in your assembly language code.

Listing 1-7 An assembly-language driver header

```
DHeader
DFlags   DC.W  0                 ;set by MyDriverOpen
DDelay   DC.W  0                 ;none
DEMask   DC.W  0                 ;DA event mask
DMenu    DC.W  0                 ;no menu
         DC.W  DOpen - DHeader   ;offset to Open
         DC.W  DPrime - DHeader  ;offset to Prime
         DC.W  DControl - DHeader ;offset to Control
         DC.W  DStatus - DHeader ;offset to Status
         DC.W  DClose - DHeader  ;offset to Close
Name     DC.B  '.MYDRIVER'       ;driver name
         ALIGN 2                 ;word alignment
```

In this example, the `drvrFlags` word is cleared to 0 because the flags are set by the `MyDriverOpen` function, shown in Listing 1-9 on page 1-32. This is an implementation decision—you can set the flags in the driver header or in your driver's open routine. The `drvrDelay` field is set to 0 because this driver does not perform any periodic actions using the `SystemTask` function. The `drvrEMask` and `drvrMenu` fields are set to 0, as this driver is not a desk accessory. The next five fields contain offsets to the driver routines, defined in the next section, "Responding to the Device Manager." The header ends with the driver name and the word alignment directive.

Responding to the Device Manager

The Device Manager calls a driver routine by setting up registers and jumping to the address indicated by the routine's offset in the driver header.

- Register A0 contains a pointer to the parameter block.
- Register A1 contains a pointer to the driver's device control entry.

This interface requires you to use some assembly language when writing a driver. However, you can write your driver routines in a high-level language if you provide an

assembly-language dispatching mechanism that acts as an interface between the Device Manager and your driver routines.

The next few sections discuss how you can provide a dispatching routine and how you can implement your driver routines in a high-level language.

Entering and Exiting From Driver Routines

Listing 1-8 shows an assembly-language dispatching routine that you can use as an interface between the Device Manager and your high-level language driver routines. This example properly handles synchronous, asynchronous, and immediate requests, as well as the special cases of open, close, and `KillIO`.

Listing 1-8 An assembly-language dispatching routine

```
DOpen
      MOVEM.L  A0-A1,-(SP)    ;save ParmBlkPtr, DCtlPtr across function call
      MOVEM.L  A0-A1,-(SP)    ;push ParmBlkPtr, DCtlPtr for C
      BSR      MyDriverOpen   ;call linked C function
      ADDQ     #8,SP          ;clean up the stack
      MOVEM.L  (SP)+,A0-A1    ;restore ParmBlkPtr, DCtlPtr
      RTS                     ;open is always immediate, must return via RTS

DPrime
      MOVEM.L  A0-A1,-(SP)    ;save ParmBlkPtr, DCtlPtr across function call
      MOVEM.L  A0-A1,-(SP)    ;push ParmBlkPtr, DCtlPtr for C
      BSR      MyDriverPrime  ;call linked C function
      ADDQ     #8,SP          ;clean up the stack
      MOVEM.L  (SP)+,A0-A1    ;restore ParmBlkPtr, DCtlPtr
      BRA.B    IOReturn

DControl
      MOVEM.L  A0-A1,-(SP)    ;save ParmBlkPtr, DCtlPtr across function call
      MOVEM.L  A0-A1,-(SP)    ;push ParmBlkPtr, DCtlPtr for C
      BSR      MyDriverControl;call linked C function
      ADDQ     #8,SP          ;clean up the stack
      MOVEM.L  (SP)+,A0-A1    ;restore ParmBlkPtr, DCtlPtr
      CMPI.W   #killCode,csCode(A0) ;test for KillIO call (special case)
      BNE.B    IOReturn
      RTS                     ;KillIO must always return via RTS

DStatus
      MOVEM.L  A0-A1,-(SP)    ;save ParmBlkPtr, DCtlPtr across function call
      MOVEM.L  A0-A1,-(SP)    ;push ParmBlkPtr, DCtlPtr for C
```

```
      BSR      MyDriverStatus ;call linked C function
      ADDQ     #8,SP          ;clean up the stack
      MOVEM.L  (SP)+,A0-A1    ;restore ParmBlkPtr, DCtlPtr

IOReturn
      MOVE.W   ioTrap(A0),D1
      BTST     #noQueueBit,D1 ;immediate calls are not queued, and must RTS
      BEQ.B    @Queued        ;branch if queued

@NotQueued
      TST.W    D0             ;test asynchronous return result
      BLE.B    @ImmedRTS      ;result must be ≤0
      CLR.W    D0             ;"in progress" result (> 0) not passed back

@ImmedRTS
      MOVE.W   D0,ioResult(A0)   ;for immediate calls you must explicitly
                                 ; place the result in the ioResult field
      RTS

@Queued
      TST.W    D0             ;test asynchronous return result
      BLE.B    @MyIODone      ;I/O is complete if result ≤ 0
      CLR.W    D0             ;"in progress" result (> 0) not passed back
      RTS

@MyIODone
      MOVE.L   JIODone,-(SP)  ;push IODone jump vector onto stack
      RTS

DClose
      MOVEM.L  A0-A1,-(SP)    ;save ParmBlkPtr, DCtlPtr across function call
      MOVEM.L  A0-A1,-(SP)    ;push ParmBlkPtr, DCtlPtr for C
      BSR      MyDriverClose  ;call linked C function
      ADDQ     #8,SP          ;clean up the stack
      MOVEM.L  (SP)+,A0-A1    ;restore ParmBlkPtr, DCtlPtr
      RTS                     ;close is always immediate, must return via RTS
```

In this example, `DOpen`, `DPrime`, `DControl`, `DStatus`, and `DClose` are the five entry points that the Device Manager locates using the offsets defined in the driver header. These in turn call the actual driver routines, which are written in C. The C functions return a result code if the I/O completed, or a positive value (usually 1) if the I/O is being handled asynchronously.

When the driver routine returns, the dispatching routine removes the parameters from the stack, restores the A0 and A1 registers, and then returns control to the Device Manager in one of two ways:

- Calling the `IODone` routine. This routine, described in detail on page 1-87, indicates to the Device Manager that the request is complete. The Device Manager removes the request from the I/O queue and calls the completion routine, if any. This is the normal method of returning from driver prime, control, and status routines.
- Returning with an `RTS` instruction. Use this method when you do not want the Device Manager to remove the request from the I/O queue. There are three cases where the `RTS` instruction should be used:
  - Returning from an asynchronous request that is not yet complete. After your device driver begins an asynchronous operation, it should return control to the Device Manager with an `RTS` instruction. The device driver can regain control of the processor using an interrupt handler, VBL task, or other method, and jump to `IODone` when the request is complete.
  - Returning from an immediate request. Because the Device Manager does not queue immediate requests, they should always return with an `RTS` instruction.
  - Returning from open, close, and `KillIO` requests. These requests are never queued and should always return with an `RTS` instruction.

To use this dispatching routine you would place it after the driver header in your assembly-language source file, and link it to your C-language driver routines. Listing 1-7 on page 1-28 shows the driver header. Sample driver routines are presented in the following sections.

Writing Open and Close Routines

You must provide both an open routine and a close routine for your device driver. The open routine should allocate any private storage your driver requires and place a handle to this storage in the `dCtlStorage` field of the device control entry. After allocating memory, the open routine should perform any other preparation required by your driver.

If your open routine installs an interrupt handler, you may want to store a pointer to the device control entry in private storage where it will be available for the interrupt handler. The section "Handling Asynchronous I/O" on page 1-37 discusses interrupt handling in more detail.

Listing 1-9 shows a sample open routine, `MyDriverOpen`. This function begins by checking whether the driver is already open (by examining the contents of the `dCtlStorage` field of the device control entry). If the driver is not already open, the `MyDriverOpen` function sets the appropriate flags in the device control entry and allocates memory in the system heap for private storage. The private storage of the driver in this example contains two fields, `byteCount` and `lastErr`, which store information about the last I/O function. The prime, control, and status routines described in the following sections use these fields.

If the `MyDriverOpen` function fails to allocate memory for private storage, it returns the `openErr` result code, which notifies the Device Manager that the driver did not open.

Listing 1-9 Example driver open routine

```
struct MyDriverGlobals {
   short    byteCount;
   short    lastErr;
};
typedef struct MyDriverGlobals MyDriverGlobals;
typedef struct MyDriverGlobals *MyDriverGlobalsPtr, **MyDriverGlobalsHdl;

OSErr MyDriverOpen(IOParamPtr pb, DCtlPtr dce)
{
   if (dce->dCtlStorage == nil)
   {
      /* set up flags in the device control entry */
      dce->dCtlFlags |= (dCtlEnableMask | dStatEnableMask | dWritEnableMask |
                         dReadEnableMask | dNeedLockMask | dRAMBasedMask );

      /* initialize dCtlStorage */
      dce->dCtlStorage = NewHandleSysClear(sizeof(MyDriverGlobals));
      if (dce->dCtlStorage == nil)
         return(openErr);
      else
         return(noErr);
   }
   else
   {
      /* the driver is already open */
      return(noErr);
   }
}
```

The close routine must reverse the effects of the open routine by releasing any memory allocated by the driver, removing interrupt handlers, removing any VBL or Time Manager tasks, and replacing changed interrupt vectors. If the close routine cannot complete the close request, it should return the `closErr` result code and the driver should continue to operate normally.

The Device Manager does not dispose of the device control entry when a driver is closed. If you want to save any information about the operational state of the driver until the next time the driver is opened, you can store a handle to the information in the `dCtlStorage` field of the device control entry.

Listing 1-10 shows a sample close routine, `MyDriverClose`. Because this device driver does not need to store any information until the next time it is opened, the `MyDriverClose` function disposes of the private storage allocated by `MyDriverOpen`.

Listing 1-10 Example driver close routine

```
OSErr MyDriverClose(IOParamPtr pb, DCtlPtr dce)
{
   if (dce->dCtlStorage != nil)
   {
      DisposeHandle(dce->dCtlStorage);
      dce->dCtlStorage = nil;
   }
   return(noErr);
}
```

Writing a Prime Routine

The prime routine implements I/O requests. You can write your prime routine to execute synchronously or asynchronously. While a synchronous prime routine completes an entire I/O request before returning to the Device Manager, an asynchronous prime routine can begin an I/O transaction but return to the Device Manager before the request is complete. In this case, the I/O request continues to be executed, typically when more data is available, by other routines such as interrupt handlers or completion routines. "Handling Asynchronous I/O" on page 1-37 discusses how to complete an asynchronous prime routine.

The Device Manager indicates whether it is requesting a read or a write operation by placing one of the following constants in the low-order byte of the `ioTrap` field of the parameter block:

```
enum {
   aRdCmd   = 2,  /* read operation requested */
   aWrCmd   = 3   /* write operation requested */
};
```

The Device Manager includes two routines, `Fetch` and `Stash`, that provide low-level support for reading and writing characters to and from data buffers. Use of these routines is optional. "Writing and Installing Device Drivers," beginning on page 1-82, describes these functions.

The `Fetch` and `Stash` routines update the `ioActCount` field of the parameter block. If you do not use these routines, you are responsible for updating this field.

If your driver serves a block device, you should update the `dCtlPosition` field of the device control entry.

Listing 1-11 shows a sample prime routine. This routine determines whether a read or write operation is being requested, then calls the appropriate function. The reading and writing functions, which are not shown here, would transfer the data to or from the hardware device.

Listing 1-11 Example driver prime routine

```
OSErr MyDriverPrime(IOParamPtr pb, DCtlPtr dce)
{
   MyDriverGlobalsHdl   dStore;
   short                callType;
   long                 numBytes;
   short                myErr;

   dStore = (MyDriverGlobalsHdl)dce->dCtlStorage;
   numBytes = pb->ioReqCount;
   callType = 0x00ff & pb->ioTrap; /* get the low byte */
   switch (callType)
   {
      case aRdCmd:
         myErr = MyReadBytes(pb->ioBuffer, numBytes);
         break;
      case aWrCmd:
         myErr = MyWriteBytes(pb->ioBuffer, numBytes);
         break;
   }
   (*dStore)->byteCount = numBytes; /* save in private storage */
   (*dStore)->lastErr = myErr;
   pb->ioActCount = numBytes; /* update parameter block field */
   return(myErr);
}
```

After obtaining a handle to the device driver's private storage from the `dCtlStorage` field of the device control entry, the `MyDriverPrime` function examines the low-order byte of the `ioTrap` field of the parameter block to determine whether the Device Manager is requesting a read operation or a write operation. `MyDriverPrime` then calls either the `MyReadBytes` or `MyWriteBytes` function to move the requested number of bytes to or from the buffer designated by the parameter block.

The `MyDriverPrime` function stores the result code and byte count in its private storage. These values will be used by the example control and status routines described in the next section. Finally, `MyDriverPrime` updates the `ioActCount` field of the parameter block and returns the result code.

Writing Control and Status Routines

Control and status routines are usually used to send and receive driver-specific information. However, you can use these routines for any kind of data transfer as long as you implement the minimum functionality described in this section. Like the prime routine, the control and status routines that you write can execute synchronously or asynchronously.

The Device Manager passes information to the control routine in the `csCode` and `csParam` fields of the parameter block. The `csCode` field specifies the type of control request and the `csParam` field contains any additional information. The `csCode` values -32767 through 127 are reserved by Apple Computer, Inc. Within this range, the following constant values are defined for use by all device drivers:

Constant name	Value	Meaning
`killCode`	1	`KillIO` requested
`goodbye`	–1	Heap being reinitialized
`accRun`	65	Time for periodic action

When the Device Manager receives a `KillIO` request, it removes every parameter block from the driver I/O queue. If your driver responds to any requests asynchronously, the part of your driver that completes asynchronous requests (for example, an interrupt handler) might expect the parameter block for the pending request to be at the head of the queue. The Device Manager notifies your driver of `KillIO` requests so that it can take the appropriate actions to stop work on the pending request. Your driver must return control to the Device Manager by means of an `RTS` instruction and not by jumping to the `IODone` routine.

If you set the `dNeedGoodbye` flag in the `drvrFlags` field of the driver header (or the `dCtlFlags` field of the device control entry), the Device Manager will call your control routine with the value `goodbye` in the `csCode` parameter before the heap is reinitialized. You driver can respond by performing any clean-up actions necessary before heap reinitialization.

If you set the `dNeedTime` flag in the `drvrFlags` field of the driver header (or the `dCtlFlags` field of the device control entry), the Event Manager will periodically call your control routine with the value `accRun` in the `csCode` parameter. Because these calls are immediate, your driver must be reentrant to handle them properly. For more information about the `dNeedTime` flag and periodic actions, see the description of the driver header, beginning on page 1-25.

Your control routine must return the `controlErr` result code for any `csCode` values that are not supported. You can define driver-specific `csCode` values if necessary, as long as they are outside the range reserved by Apple Computer, Inc.

Listing 1-12 shows a sample control routine, `MyDriverControl`. This function interprets the driver-specific `csCode` value of `kClearAll` as a command to clear the information saved in the driver's private storage by the `MyDriverPrime` routine.

Listing 1-12 Example driver control routine

```
OSErr MyDriverControl(CntrlParamPtr pb, DCtlPtr dce)
{
   MyDriverGlobalsHdl   dStore;

   dStore = (MyDriverGlobalsHdl)dce->dCtlStorage;
```

```
   switch (pb->csCode)
   {
      case kClearAll:
         (*dStore)->byteCount = 0;
         (*dStore)->lastErr = 0;
         return(noErr);
      default: /* always return controlErr for unknown csCode */
         return(controlErr);
   }
}
```

Your status routine should work in a similar manner. The Device Manager uses the `csCode` field to specify the type of status information requested. The status routine should respond to whatever requests are appropriate for your driver and return the error code `statusErr` for any unsupported `csCode` value.

The Device Manager interprets a status request with a `csCode` value of 1 as a special case. When the Device Manager receives such a status request, it returns a handle to the driver's device control entry. Your driver's status routine never sees this request.

Listing 1-13 shows a sample status routine, `MyDriverStatus`, that implements two driver-specific status requests, `kByteCount` and `kLastErr`. When `MyDriverStatus` receives one of these requests, it returns the byte count or error code values saved in private storage by the `MyDriverPrime` routine. `MyDriverStatus` returns this information in the `csParam` field.

Listing 1-13 Example driver status routine

```
OSErr MyDriverStatus(CntrlParamPtr pb, DCtlPtr dce)
{
   MyDriverGlobalsHdl   dStore;

   dStore = (MyDriverGlobalsHdl)dce->dCtlStorage;
   switch (pb->csCode)
   {
      case kByteCount:
         pb->csParam[0] = (*dStore)->byteCount;
         return(noErr);
      case kLastErr:
         pb->csParam[0] = (*dStore)->lastErr;
         return(noErr);
      default: /* always return statusErr for unknown csCode */
         return(statusErr);
   }
}
```

Handling Asynchronous I/O

If you design any of your driver routines to execute asynchronously, you must provide a mechanism for your driver to complete the requests. Some examples of routines that you might use are:

- Completion routines. Your driver routine could call another driver to start the data transfer. In this case, you can provide that driver with a completion routine. When the other driver completes the request, the Device Manager executes the completion routine. In the completion routine, you could call the other driver again to execute the next part of the I/O operation. When the entire operation is complete, the completion routine should return by calling the `IODone` routine.
- Interrupt handlers. If your driver serves a hardware device that generates interrupts, you can create an interrupt handler that responds to these interrupts. Your interrupt handler must clear the source of the interrupt and return as quickly as possible, while preserving all registers other than D0 through D3 and A0 through A3. For more information about interrupts and how to install an interrupt handler, see *Inside Macintosh: Processes* and *Designing Cards and Drivers for the Macintosh Family*, third edition.
- VBL, Time Manager, and Deferred Task Manager tasks. Installing any of these tasks ensures that your driver receives system time at some point in the future. During this time, you can check to see if the I/O operation is ready to continue.

If your driver serves a device on a NuBus™ expansion card, you might want to use slot interrupts to signal your driver. When a NuBus card device signals a slot interrupt, the CPU can quickly detect which card requested the interrupt service, but not which device on the card. To determine which device caused the interrupt, the system uses a polling procedure. Your driver should provide a polling routine that checks if the device it serves caused the current interrupt, and if so, calls the proper driver routine to handle the interrupt. The Slot Manager maintains a queue of these polling routines for each slot. Your driver can install an element in this queue using the Slot Manager function `SIntInstall`. You can remove a queue element with the `SIntRemove` function. See the chapter “Slot Manager” in this book for information about these functions.

You should observe these guidelines when writing or using asynchronous routines:

- Once you pass a parameter block to an asynchronous routine it is out of your control. You should not examine or change the parameter block until your completion routine is called because you have no way of knowing the state of the parameter block.
- Do not dispose of or reuse a parameter block until the asynchronous request is completed. For example, if you declare the parameter block as a local variable, your function cannot return until the request is complete because local variables are allocated on the stack and released when a function returns.
- Use a completion routine to determine when an asynchronous routine has completed, rather than polling the `ioResult` field of the parameter block. Polling the `ioResult` field is not efficient and defeats the purpose of asynchronous operation.

Installing a Device Driver

There are a variety of ways to install a device driver, depending on where the driver code is stored and how much control you want over the installation process.

- You can store the device driver in a resource within an application and have the application install the driver.
- You can store the device driver, and the code to install it, in a system extension file. See the chapter "Start Manager" in *Inside Macintosh: Operating System Utilities* for information about creating system extensions.
- You can store the device driver in the declaration ROM of an expansion card. Slot device drivers can be designed to load automatically at startup, or you can use the Slot Manager `SGetDriver` function to load the driver into memory. Refer to *Designing Cards and Drivers for the Macintosh Family*, third edition, for information about writing and installing slot device drivers.

If you store your driver in a resource of type `'DRVR'` you can use the `OpenDriver` or `PBOpen` functions to install and open your driver. If you need more control over the installation process, you can use the `DriverInstall` function to create the device control entry and add it to the unit table, or you can create the device control entry yourself, install it in the unit table, and then use `OpenDriver` or `PBOpen` to open the driver. If the driver is already installed in the unit table, `OpenDriver` and `PBOpen` simply call the driver's open routine and return the driver reference number.

If you want to use the `OpenDriver` function to install your driver, you are responsible for examining the unit table and changing your driver resource ID so that the `OpenDriver` function installs your driver in an empty location in the unit table. If the handle at a given unit number is `nil`, there is no device control entry installed in that position. You can install your device control entry in any empty location in the unit table that is not listed as reserved by Apple Computer, Inc. Table 1-2 summarizes the unit numbers reserved for specific purposes.

Table 1-2 Reserved unit numbers

Unit number range	Reference number range	Purpose
0 through 11	–1 through –12	Reserved for serial, disk, AppleTalk, printer, and other drivers
12 through 31	–13 through –32	Available for desk accessories
32 through 38	–33 through –39	Available for SCSI devices
39 through 47	–40 through –48	Reserved
48 through 127	–49 through –128	Available for slot and other drivers

Listing 1-14 shows a method of searching the unit table for an appropriate location to install your driver. The `MyOpenDriver` function in Listing 1-1 on page 1-18 calls this function and then uses the `OpenDriver` function to install and open the device driver.

Listing 1-14 Finding space in the unit table

```
short MyFindSpaceInUnitTable(void);
{
   Ptr      curUTableBase, newUTableBase;
   short    curUTableEntries, newUTableEntries;
   short    refNum, unitNum;

   /* get current unit table values from low memory globals */
   curUTableEntries = *(short*)UnitNtryCnt;
   curUTableBase = *(Ptr*)UTableBase;

   /* search for empty space in the current unit table */
   for ( unitNum = curUTableEntries - 1;
         unitNum >= 48; /* lowest available unit number */
         unitNum-- )
   {
      refNum = ~(unitNum);
      if (GetDCtlEntry(refNum) == nil)
         return(unitNum); /* found a space */
   }

   /* no space in the current table, so make a new one */

   /* increase the size of the table by 16 (an arbitrary value) */
   newUTableEntries = curUTableEntries + 16;

   /* allocate space for the new table */
   newUTableBase =
      NewPtrSysClear((long)newUTableEntries * sizeof(Handle));
   if (newUTableBase == nil)
      return(memErr);

   /* copy the old table to the new table */
   BlockMove(curUTableBase, newUTableBase,
             (long)curUTableEntries * sizeof(Handle));

   /* set the new unit table values in low memory */
   *(Ptr*)UTableBase = newUTableBase;
   *(short*)UnitNtryCnt = newUTableEntries;

   unitNum = newUTableEntries - 1;
   return(unitNum);
}
```

Although rare, it is possible for the unit table to become completely full. If the `MyFindSpaceInUnitTable` function does not find an empty unit table entry, it creates a larger unit table and copies the contents of the old unit table into the new one. To avoid the need for every driver to create a larger table, this function increases the size of the table by 16 entries—a reasonable amount in most cases.

The `MyFindSpaceInUnitTable` function does not need to disable interrupts when changing the values of the `UTableBase` and `UnitNtryCnt` system global variables because both unit tables are valid and drivers are not opened or closed at interrupt time.

Note that this function does not check for empty locations in the space reserved for desk accessories or SCSI drivers. You may wish to modify the function if you are installing one of these.

Writing a Chooser-Compatible Device Driver

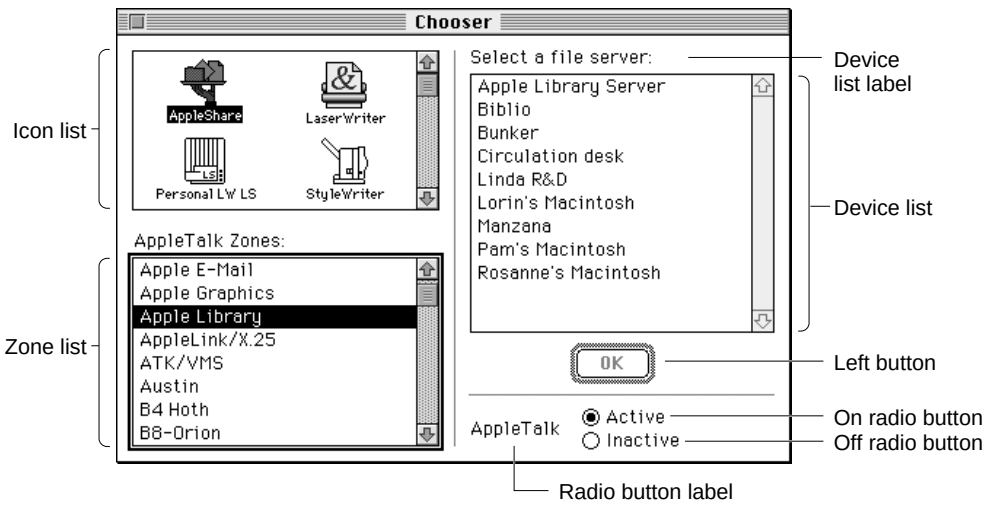
The Chooser is a desk accessory that helps provide a standard user interface for networking and printing device drivers. The Chooser allows the user to make choices such as which serial port to use, which AppleTalk zone to communicate with, and which LaserWriter to use.

This section describes how the Chooser works, how to create a Chooser extension, and how to respond to actions from the user. You should read the previous section, "Writing a Device Driver," before you read this section.

How the Chooser Works

The Chooser allows users to select which devices they want to use. When the user opens the Chooser, it displays a window containing lists and buttons for making device-related choices. Typically, users select a type of device from the icon list, then select the particular device they want to use from the device list. For AppleTalk devices, the user must also select an AppleTalk zone from the zone list. The Chooser window can also display buttons, such as an OK button; and radio buttons, such as the background printing On and Off buttons. Figure 1-10 shows an example of the Chooser window.

Figure 1-10 The Chooser window



The Chooser relies on the List Manager for creating, displaying, and manipulating possible user selections in this window. You may want to read the chapter "List Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information.

The Chooser does not communicate directly with device drivers; instead, it communicates with device packages. A ***device package*** is a resource similar to a driver resource, except a device package responds to Chooser messages instead of Device Manager requests. The device package is responsible for communicating the user's choices to the device driver.

Device packages are stored in Chooser extension files, which the Chooser looks for in the Extensions folder inside the System Folder of the startup disk. A Chooser extension file contains a number of resources in addition to the device package resource. These other resources contain information about the buttons, labels, and lists that the Chooser displays when the user selects the device icon from the icon list. You use these resources to define the following properties:

- The device list label. The Chooser displays this label over the device list.
- The buttons to use. The Chooser allows the device package to display up to four buttons, called the Left button, the Right button, the On radio button, and the Off radio button.
- The titles and positions of the buttons.
- The radio button label.
- The AppleTalk device type name. The Chooser searches the current AppleTalk zone for devices of this type.
- An AppleTalk Name-Binding Protocol (NBP) retry interval and a timeout count. The Chooser uses this information when searching for AppleTalk devices.

When a user selects the icon corresponding to a particular device package, the Chooser sends messages to that device package by calling the device package as if it were the following function:

```
pascal OSErr MyPackage (short message, short caller,
                        StringPtr objName, StringPtr zoneName,
                        long p1, long p2);
```

The Chooser passes the following parameters to the device package:

Parameter	Description

`message`	The operation to be performed; this parameter has one of the following values:
	`enum {`
	`  /* Chooser messages */`
	`  chooserInitMsg = 11,`
	`  newSelMsg = 12,`
	`  fillListMsg = 13,`
	`  getSelMsg = 14,`
	`  selectMsg = 15,`
	`  deselectMsg = 16,`
	`  terminateMsg = 17,`
	`  buttonMsg = 19`
	`};`
	Table 1-4 on page 1-47 explains the meaning of these messages.

`caller`	A number that identifies the application calling your device package. The value `chooserID` indicates the Chooser. Values in the range 0–127 are reserved; values outside this range may be used by applications.
`objName`	Additional information whose meaning depends on the value of the `message` parameter. See Table 1-4 on page 1-47 for more information.
`zoneName`	The name of the AppleTalk zone containing the devices in the device list. If the Chooser is being used with the local zone and bit 24 of the `flags` field of the device package header is not set, the string value is "*", otherwise, it is the actual zone name. See "Creating a Device Package" on page 1-45 for more information about the package header.
`p1`	A handle to the List Manager list that contains the device choices displayed in the device list box.
`p2`	Additional information whose meaning depends on the value of the `message` parameter. See Table 1-4 on page 1-47 for more details.

When the user opens the Chooser, the Chooser searches the Extensions folder for Chooser extension files. For each one it finds, it opens the file, fetches the device icon, reads the flags field of the device package header, and closes the file. The Chooser then displays each device icon, and dims the icons for AppleTalk devices if AppleTalk is not connected.

When the user selects a device icon that is not dimmed, the Chooser reopens the corresponding Chooser extension file and performs the following actions:

1. The Chooser labels the device list with the device list label.
2. The Chooser sends the `chooserInitMsg` message to the device package.
3. If the selected device package represents a serial printer, the Chooser places the two icons that represent the printer port and the modem port serial drivers into the device list box. When the user makes a selection, the Chooser records the user's choice in low memory and parameter RAM.
4. If the selected device icon represents an AppleTalk device and the corresponding device package does not accept `fillListMsg` messages, the Chooser initiates an asynchronous routine that interrogates the current AppleTalk zone for all devices whose type matches the AppleTalk device type name specified in the Chooser extension file. The asynchronous routine uses the retry interval and the timeout count. As responses arrive, the Chooser updates the device list.
5. If the device package does accept `fillListMsg` messages, the Chooser sends the `fillListMsg` message to the device package. The device package responds by filling the device list with the appropriate device choices.
6. To determine which devices in the device list should be selected, the Chooser calls the device package with the `getSelMsg` message. The device package responds by inspecting the list and setting the selected or unselected state of each entry. The Chooser may send the `getSelMsg` message frequently; for example, each time a new response to the AppleTalk zone interrogation arrives. The Chooser does not send the `getSelMsg` message for serial printers; it highlights the icon corresponding to the currently selected serial port, as recorded in low memory.
7. If the device package allows multiple devices to be active at once, the Chooser sets the appropriate List Manager bits. When the user selects or deselects a device, the Chooser calls the device package with the appropriate message. For packages that do not accept multiple active devices, the Chooser sends the `selectMsg` or `deselectMsg` message; otherwise, it sends the `newSelMsg` message. The device package mounts or unmounts the device, if appropriate, and records the user's choice.
8. When the user selects a different device icon or closes the Chooser, the Chooser calls the current device package with the `terminateMsg` message, if the package accepts this kind of message. At this time, the package can clean up, if necessary. The Chooser then calls the `UpdateResFile` function, closes the device resource file, and flushes the system startup volume.

Creating a Chooser Extension File

The Chooser uses three file types to identify different kinds of devices supported by Chooser extension files:

File type	Device type
`'PRES'`	Serial printer
`'PRER'`	Non-serial printer
`'RDEV'`	Other device

You can specify the creator of your Chooser extension file, which allows you to give your device its own icon.

You can include the following resources in your Chooser extension file:

Resource type	Resource ID	Description
'PACK'	–4096	Device package. This resource contains the device package header and code.
'STR '	–4096	Type name for AppleTalk devices. The Chooser searches the current AppleTalk zone for devices of this type.
'GNRL'	–4096	AppleTalk information. The first byte of this resource contains the Name-Binding Protocol (NBP) retry interval, the second byte contains the timeout count.
'STR '	–4091	List box label. The Chooser labels the device list with this string after the user has selected the device's icon.
'STR '	–4087	Radio button label.
'STR '	–4088	Off radio button title.
'STR '	–4089	On radio button title.
'STR '	–4092	Right button title.
'STR '	–4093	Left button title.
'ncrt'	–4096	Button positions.
'LDEF'	–4096	Alternate list definition function. You can supply this function to modify the device list—to include pictures or icons, for example.
'STR '	–4090	Reserved for use by the Chooser.

You should also include a 'BNDL' resource (and appropriate icon family resources) to give your device type a distinctive icon because this may be the only way that devices are identified in the Chooser window. The chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* describes the 'BNDL' resource.

The Chooser allows your device package to display two buttons, called the Left button and the Right button because of their default positions. The Left button has a double border and is highlighted (the title string is dark) when one or more devices are selected in the device list. When this button is highlighted, pressing the Return or Enter key, or double-clicking in the device list, is equivalent to clicking the button. The Right button has a single border and is always highlighted. The user can activate it only by clicking it.

The Chooser also allows you to display two radio buttons and a radio button label. These buttons are called the On radio button and the Off radio button because those are the titles the LaserWriter uses, but you can name them anything you want.

You can position these buttons by including a resource of type 'ncrt' with an ID of –4096. The first word in this type of resource specifies the number of rectangles, and the rest of the resource contains the rectangle definitions. The first rectangle positions the Left button, the second positions the Right button, the third positions the On radio

button, and the fourth positions the Off radio button. The fifth rectangle positions the radio button label.

Each rectangle definition is 8 bytes long and contains the rectangle coordinates in the order [*top*, *left*, *bottom*, *right*]. The default values are [112, 206, 132, 266] for the Left button and [112, 296, 132, 356] for the Right button. You could use the values [112, 251, 132, 331] to center a single button.

The Chooser uses the List Manager to produce and display the standard device list. You can supply a list definition function to modify this list. For example, you might want to include pictures or icons in your list. To do this, you must provide a resource of type `'LDEF'` with an ID of –4096. For complete information about list construction and data structures, see the chapter "List Manager" in *Inside Macintosh: More Macintosh Toolbox*.

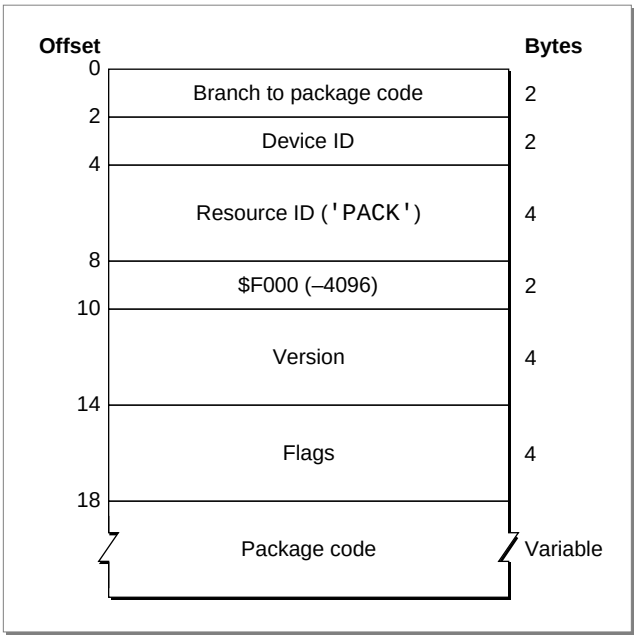
Creating a Device Package

Like a driver resource, a device package has two parts:

- a header that contains flags and other information about the driver
- the code that responds to Chooser messages

Figure 1-11 shows the structure of a device package.

Figure 1-11 Structure of a device package



Since the Chooser expects the package code to be at the beginning of the device package, the first field of the package header should be a `BRA.S` instruction to the package code.

The device ID is an integer that identifies the device. The version field differentiates versions of the driver code.

The flags field contains information about the device package and the device it serves. Table 1-3 lists the meaning of each bit of the flags field.

The package code should implement the `MyPackage` function described on page 1-42. The following section, "Responding to the Chooser," discusses how to implement this function.

Table 1-3 Device package flags

Bit	Meaning
31	Set if an AppleTalk device
30–29	Reserved (clear to 0)
28	Set if the device package can have multiple instances selected at once
27	Set if the device package uses the Left button
26	Set if the device package uses the Right button
25	Set if no zone name has been saved
24	Set if the device package uses actual zone names
23–21	Reserved (clear to 0)
20	Set if the device uses the On and Off radio buttons and radio button label
19–17	Reserved (clear to 0)
17	Set if the device package accepts the `chooserInitMsg` message
16	Set if the device package accepts the `newSelMsg` message
15	Set if the device package accepts the `fillListMsg` message
14	Set if the device package accepts the `getSelMsg` message
13	Set if the device package accepts the `selectMsg` message
12	Set if the device package accepts the `deselectMsg` message
11	Set if the device package accepts the `terminateMsg` message
10–0	Reserved (clear to 0)

Responding to the Chooser

This section gives more details about how your device package should respond when it receives a message from the Chooser.

When the Chooser sends your device package a message, the Chooser extension file is the current resource file and the Chooser window is the current graphics port. The

startup disk is the default volume and the System Folder of the startup disk is the default directory. Your device package must preserve all of these.

Table 1-4 lists the Chooser messages and how your device package should respond to them.

Table 1-4 Chooser messages and their meanings

Message	Meaning
`chooserInitMsg`	The Chooser sends this message to your device package when the user selects the icon representing your device in the icon list. The `objName` parameter contains a pointer to a data structure that contains a size word followed by four handles to structures of type `ControlRecord`. The size is at least 18 bytes (2 bytes for the size word and 4 bytes for each of the handles). The handles reference the Left and Right buttons and the On and Off radio buttons, in that order. Your device package can respond to this message by setting up the initial button configuration. To display any of the radio buttons, use the `ShowControl` function. To highlight them, use the `SetControlValue` function. The `p2` parameter is not used. For more information about controls, see the chapter "Control Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.
`newSelMsg`	If your device package allows multiple selections, the Chooser sends this message to your package when the user changes or adds a selection. The `objName` and `p2` parameters are not used.
`fillListMsg`	The Chooser sends this message when the user selects a device icon. The `p1` parameter contains a handle to a List Manager list. Your device package should use the List Manager to fill this list with choices for the particular type of device. The `objName` and `p2` parameters are not used.
`getSelMsg`	The Chooser sends this message to determine which devices in the device list should be selected. The `p1` parameter contains a handle to a List Manager list. Your device package should respond by inspecting the list and setting the selected or unselected state of each entry, using the `LSetSelect` function. You should alter only the entries that require updating. The Chooser does not send this message for serial printers.
`selectMsg`	If your device package does not allow multiple selections, the Chooser sends this message to your package when the user selects a device in the device list. You should record the user's selection, preferably in your Chooser extension file. Your device package may not call the List Manager in response to this message. If your device package accepts `fillListMsg` messages, the `objName` parameter is undefined and the `p2` parameter contains the row number of the selected device. If your device package does not accept `fillListMsg` messages, the `objName` parameter contains a pointer to a string containing the name of the device (up to 32 characters). If the device is an AppleTalk device, the `p2` parameter contains the `AddrBlock` value for the address of the selected AppleTalk device. For more information about AppleTalk devices, refer to *Inside Macintosh: Networking*.

continued

Table 1-4 Chooser messages and their meanings (continued)

Message	Meaning
`deselectMsg`	If your device package does not allow multiple selections, the Chooser sends this message to your package when the user deselects a device in the device list. Your device package may not call the List Manager in response to this message. If your device package accepts `fillListMsg` messages, the `objName` parameter is undefined and the `p2` parameter contains the row number of the device that was deselected. If your device package does not accept `fillListMsg` messages, the `objName` parameter contains a pointer to a string containing the name of the device (up to 32 characters). If the device is an AppleTalk device, the `p2` parameter contains the `AddrBlock` value for the address of the selected AppleTalk device. For more information about AppleTalk devices, refer to *Inside Macintosh: Networking*.
`terminateMsg`	The Chooser sends this message when the user selects a different device icon, closes the Chooser window, or changes zones. Your device package should perform any necessary cleanup tasks but should not dispose of the device list. The `objName` and `p2` parameters are not used.
`buttonMsg`	The Chooser sends this message when the user clicks one of the buttons in the Chooser window. The low-order byte of the `p2` parameter contains 1 if the user clicked the Left button, 2 if the user clicked the Right button, 3 if the user clicked the On radio button, and 4 if the user clicked the Off radio button. You must perform the appropriate highlighting for the radio buttons. The high-order word of the `p2` parameter contains the modifier bits from the mouse-up event. See the chapters "Control Manager" and "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for more information.

Allocating Private Storage

Device packages initially have no data space allocated. There are two ways your device package can acquire data space:

- Use the List Manager to allocate extra memory in the device list.
- Create a resource.

The Chooser uses column 0 of the device list structure to store the names displayed in the device list. For device packages that do not accept `fillListMsg` messages, the Chooser uses column 1 to store the 4-byte AppleTalk internet addresses of the devices in the list. Therefore, your device package can use column 1 and higher (if it accepts `fillListMsg` messages) or column 2 and higher to store private data. You can use standard List Manager functions to add these columns, store data in them, and retrieve the data stored there. Your device package can also use the `refCon` field of the device list for its own purposes.

Using the device list is limited by the fact that the Chooser disposes of the device list whenever the user changes device types or changes the current zone. However, the Chooser does call your device package with the `terminateMsg` message before it disposes of the list.

Also, if your device package does not accept `fillListMsg` messages, the Chooser disposes of the device list whenever a new response from the AppleTalk zone interrogation arrives. However, the Chooser does send the `getSelMsg` message immediately afterward.

The second way to obtain storage space is to create a resource in the device resource file. This file is always the current resource file when the Chooser sends a message to the package, so you can use the `GetResource` function to obtain a handle to the storage.

It is important for most device packages to record which devices the user has chosen. The recommended method for this is to create a resource in your driver resource file. This resource can be of any type; in fact, it's advantageous to provide your own resource type so that no other program will try to modify it. If you choose to use a standard resource type, you should use only resource IDs in the range –4080 through –4065.

Writing a Desk Accessory

Desk accessories are small applications designed like device drivers. Desk accessories typically provide a user interface with a window and a menu, perform some limited function, and are opened from the Apple menu. The Chooser is an example of a desk accessory.

Desk accessories were originally created for the Macintosh because they offered two distinct advantages over applications. They provided both a limited degree of multitasking and a primitive form of interapplication communication. However, modern Macintosh applications enjoy far more sophisticated versions of these capabilities. Users can even open applications from the Apple menu. For these reasons, you would be better served by writing a small application than by writing a desk accessory.

Control panels have largely replaced desk accessories as a user interface for device drivers. In addition to providing a more consistent and extensible interface, control panels can include an initialization (`'INIT'`) resource to load and execute your device driver at system startup. For more information about control panels, see the chapter "Control Panels" in *Inside Macintosh: More Macintosh Toolbox*.

If you're certain you need to write a desk accessory, you should read this section. You might also want to read the chapters "Event Manager," "Window Manager," "Dialog Manager," and "Menu Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

How Desk Accessories Work

When the user opens a desk accessory (or when an application calls the `OpenDeskAcc` function), the system performs a major context switch, loads the desk accessory into the system heap, and calls the desk accessory driver open routine. The desk accessory can respond by creating its window and menu.

When events occur, the Event Manager directs them to the desk accessory by calling its driver control routine. The Event Manager handles switching between applications and desk accessories in the system heap.

When the user closes the desk accessory (by closing its window or choosing Quit from its menu) or an application closes the desk accessory (by calling the `CloseDeskAcc` function), the desk accessory disposes of its window and any other data structures associated with it.

In a single-application environment in System 6, and in a multiple-application environment in which the desk accessory is launched in the application's partition (for example, a desk accessory opened by the user from the Apple menu while holding down the Option key), the Event Manager handles events for desk accessories in a slightly different manner, although it still translates them into control requests. For details, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Creating a Driver Resource for a Desk Accessory

You create a desk accessory by creating a driver resource and storing it in a resource file, as described in "Creating a Driver Resource," beginning on page 1-24. Typically, you store your desk accessory driver resource in a file of type `'dfil'`, which the user places in the Apple Menu Items folder.

Three fields of the driver resource header are of particular importance to desk accessories:

- The `drvrEMask` field. This field contains an event mask specifying which events your desk accessory can handle. If your desk accessory has a window, you should include keyboard, activate, update, and mouse-down events, but you should not include mouse-up events. When an event occurs, the Event Manager checks this field to determine whether the desk accessory can handle the type of event and, if so, calls the desk accessory driver control routine. See the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for more information about events and event masks.
- The `drvrMenu` field. This field contains the menu ID of your desk accessory's menu, if it has one, or any one of its menus, if it has more than one. Otherwise, it contains 0. A desk accessory menu ID must be negative and must be different from the menu ID for other desk accessories.
- The `drvrDelay` field and the `dNeedTime` flag of the `drvrFlags` field. Desk accessories sometimes need to perform certain actions periodically. For example, a clock desk accessory might change the time it displays every second. If your desk accessory needs to perform a periodic action, set the `dNeedTime` flag and use the `drvrDelay` field to indicate how often the action should occur. "Creating a Driver Resource," beginning on page 1-24, describes these fields in more detail.

All desk accessories must implement open, close, and control routines. Your desk accessory can implement a prime and status routine if needed.

Opening and Closing a Desk Accessory

When the user chooses an item from the Apple menu, the foreground application calls the `OpenDeskAcc` function, which determines whether the item is a desk accessory, application, or document, and schedules it for execution. Applications call the `CloseDeskAcc` function if the user chooses the Close menu item from the File menu when the foreground window does not belong to the application. These functions are described in "Device Manager Reference," beginning on page 1-53.

Opening a desk accessory is similar to launching an application. In your desk accessory driver open routine, you should do the following:

- Create the desk accessory's window. You can do this with the Dialog Manager function `GetNewDialog` or `NewDialog`. You should specify that the window be invisible because the `OpenDeskAcc` function will display it. You should set the `windowKind` field of the `windowRecord` structure to the desk accessory's driver reference number, which you can find in the device control entry. You should also store a copy of the window pointer in the `dCtlWindow` field of the device control entry.
- Allocate private storage as you would for any device driver.
- Create any menus needed by your desk accessory. You are responsible for adding your menus to the menu bar. See the chapter "Menu Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for more details.

If your driver open routine is unable to complete its tasks (because of insufficient memory, for example), you should modify the code so it doesn't respond to events, and display an alert indicating failure.

As for all drivers, your close routine should undo the actions taken by the open routine, dispose of the desk accessory's window and private storage, clear the window pointer in the device control entry, and remove any menus that were added to the menu bar.

Responding to Events

When the Event Manager determines an event has occurred that your desk accessory should handle, it checks the `drvrEMask` field of the driver header and, if that field indicates your desk accessory handles the event type, it passes the event to your desk accessory by calling your driver control routine.

The Event Manager passes one of nine values in the `csCode` field to indicate the action to take:

Constant name	Value	Meaning
`accEvent`	64	Handle a given event
`accRun`	65	Time for periodic action
`accCursor`	66	Change cursor shape if appropriate
`accMenu`	67	Handle a given menu item

Constant name	Value	Meaning
accUndo	68	Handle the Undo command
accCut	70	Handle the Cut command
accCopy	71	Handle the Copy command
accPaste	72	Handle the Paste command
accClear	73	Handle the Clear command

Along with the accEvent message, the Event Manager sends a pointer to an event record in the csParam field. Your desk accessory can respond to the event in whatever way is appropriate. For example, when your desk accessory becomes active, it might install its menu in the menu bar.

Note
If your desk accessory window is a modeless dialog box and you are calling the Dialog Manager function IsDialogEvent in response to the event, you should set the windowKind field of your window record to 2 before you call IsDialogEvent. Setting this field to 2 allows the Dialog Manager to recognize and handle the event properly. You should restore the original value of the windowKind field before returning from your control routine. ◆

The Event Manager periodically sends the accRun message if your desk accessory has requested time for background processing. To request this service, you set the dNeedTime flag in the drvrFlags field of your desk accessory driver header. See "Writing Control and Status Routines," beginning on page 1-34, for more information.

The accCursor message makes it possible to change the shape of the cursor when it is inside your desk accessory window and your desk accessory window is active. Your control routine should check whether the mouse location is in your window and, if so, should set the cursor appropriately by calling the QuickDraw function InitCursor.

If your desk accessory window is a dialog box, you should respond to the accCursor message by generating a null event (storing the event code for a null event in an event record) and passing it to the Dialog Manager function DialogSelect. This allows the Dialog Manager to blink the insertion point in editText items.

When the Event Manager sends an accMenu message, it provides the menu ID followed by the menu item number in the csParam field. You should take the appropriate action and then call the Menu Manager function HiliteMenu with a value of 0 for the menuID parameter to remove the highlighting from the menu bar.

You should respond to the last five messages, accUndo through accClear, by processing the corresponding editing command in the desk accessory window, if appropriate. The chapter "Scrap Manager" in *Inside Macintosh: More Macintosh Toolbox* contains information about cutting and pasting.

Your desk accessory routines should restore the current resource file and graphics port if it changes either one.

Device Manager Reference

This section describes the data structures, functions, and resources that are specific to the Device Manager.

The "Data Structures" section shows the C declarations for the data structures that are used by the Device Manager. The "Device Manager Functions" section describes the functions you use to communicate with device drivers and the functions that provide support for writing your own device drivers. The "Resources" section describes the driver resource.

Data Structures

This section describes the parameter block structure, the device control entry structure, and the enumerated types you use to define values within them.

Device Manager Parameter Block

The Device Manager provides both a high-level and a low-level interface for communicating with device drivers. You pass information to the low-level functions in a parameter block structure, defined by the `ParamBlockRec` union.

```
typedef union ParamBlockRec {
   IOParam        ioParam;
   FileParam      fileParam;
   VolumeParam    volumeParam;
   CntrlParam     cntrlParam;
   SlotDevParam   slotDevParam;
   MultiDevParam  multiDevParam;
} ParamBlockRec;
typedef ParamBlockRec *ParmBlkPtr;
```

The Device Manager uses two forms of the parameter block: one for the open, close, read, and write functions (the `IOParam` structure) and another for the control and status functions (the `CntrlParam` structure). Other managers use other structures of the `ParamBlockRec` union.

```
typedef struct IOParam {
 QElemPtr   qLink;         /* next queue entry */
 short      qType;         /* queue type */
 short      ioTrap;        /* routine trap */
 Ptr        ioCmdAddr;     /* routine address */
 ProcPtr    ioCompletion;  /* completion routine address */
 OSErr      ioResult;      /* result code */
 StringPtr  ioNamePtr;     /* pointer to driver name */
```

```
 short      ioVRefNum;     /* volume reference or drive number */
 short      ioRefNum;      /* driver reference number */
 char       ioVersNum;     /* not used by the Device Manager */
 char       ioPermssn;     /* read/write permission */
 Ptr        ioMisc;        /* not used by the Device Manager */
 Ptr        ioBuffer;      /* pointer to data buffer */
 long       ioReqCount;    /* requested number of bytes */
 long       ioActCount;    /* actual number of bytes completed */
 short      ioPosMode;     /* positioning mode */
 long       ioPosOffset;   /* positioning offset */
} IOParam;

typedef struct CntrlParam {
 QElemPtr   qLink;         /* next queue entry */
 short      qType;         /* queue type */
 short      ioTrap;        /* routine trap */
 Ptr        ioCmdAddr;     /* routine address */
 ProcPtr    ioCompletion;  /* completion routine address */
 OSErr      ioResult;      /* result code */
 StringPtr  ioNamePtr;     /* pointer to driver name */
 short      ioVRefNum;     /* volume reference or drive number */
 short      ioCRefNum;     /* driver reference number */
 short      csCode;        /* type of control or status request */
 short      csParam[11];   /* control or status information */
} CntrlParam;
```

The first eight fields are common to both structures. Each structure also includes its own unique fields.

Field descriptions for fields common to both structures

`qLink` A pointer to the next entry in the driver I/O queue. (This field is used internally by the Device Manager to keep track of asynchronous calls awaiting execution.)

`qType` The queue type. (This field is used internally by the Device Manager.)

`ioTrap` The trap number of the routine that was called. (This field is used internally by the Device Manager.)

`ioCmdAddr` The address of the routine that was called. (This field is used internally by the Device Manager.)

`ioCompletion` A pointer to a completion routine. When making asynchronous requests, you must set this field to `nil` if you are not specifying a completion routine. The Device Manager automatically sets this field to `nil` when you make a synchronous request.

`ioResult` A value indicating whether the routine completed successfully. The Device Manager sets this field to 1 when it queues an asynchronous request. When the driver completes the request, it places the actual

result code in this field. You can poll this field to detect when the driver has completed the request and to determine its result code. The Device Manager executes the completion routine after this field receives the result code.

`ioNamePtr` A pointer to the name of the driver. You use this field only when opening a driver.

`ioVRefNum` The drive number, if any. The meaning of this field depends on the device driver. The Disk Driver uses this field to identify disk devices.

Field descriptions for the `IOParam` structure

`ioRefNum` The driver reference number.

`ioVersNum` Not used.

`ioPermssn` The read/write permission of the driver. When you open a driver, you must supply one of the following values in this field:

```
enum {
   /* access permissions */
   fsCurPerm   = 0, /* retain current permission */
   fsRdPerm    = 1, /* allow reads only */
   fsWrPerm    = 2, /* allow writes only */
   fsRdWrPerm  = 3  /* allow reads and writes */
};
```

The Device Manager compares subsequent read and write requests with the read/write permission of the driver. If the request type is not permitted, the Device Manager returns a result code indicating the error.

`ioMisc` Not used.

`ioBuffer` A pointer to the data buffer for the driver to use for reads or writes.

`ioReqCount` The requested number of bytes for the driver to read or write.

`ioActCount` The actual number of bytes the driver reads or writes.

`ioPosMode` The positioning mode used by drivers of block devices. Bits 0 and 1 of this field indicate where an operation should begin relative to the physical beginning of the block-formatted medium. You can use the following constants to test or set the value of these bits:

```
enum {
   /* positioning modes */
   fsAtMark    = 0, /* at current position */
   fsFromStart = 1, /* offset from beginning */
   fsFromMark  = 3  /* offset from current
                       position  */
};
```

The Disk Driver allows you to add the following constant to this field to specify a read-verify operation:

```
enum {
   rdVerify = 64     /* read-verify mode */
};
```

See the description of the PBRead function on page 1-70.

`ioPosOffset` The byte offset, relative to the position specified by the positioning mode, where the driver should perform the operation. If you specify the `fsAtMark` positioning mode, the Device Manager ignores this field.

Field descriptions for the `CntrlParam` structure

`ioCRefNum` The driver reference number.

`csCode` A value identifying the type of control or status request. Each driver may interpret this number differently.

`csParam` The control or status information passed to or from the driver. This field is declared generically as an array of eleven integers. Each driver may interpret the contents of this field differently. Refer to the driver's documentation for specific information.

Device Control Entry

The device control entry structure, defined by the `AuxDCE` data type, stores information about each device driver in memory. The `AuxDCE` data type supersedes the original `DCtlEntry` data type, and provides additional fields for drivers that serve slot devices. See the chapter "Slot Manager" in this book for information about slot device drivers.

```
typedef struct AuxDCE {
   Ptr         dCtlDriver;    /* pointer or handle to driver */
   short       dCtlFlags;     /* flags */
   QHdr        dCtlQHdr;      /* I/O queue header */
   long        dCtlPosition;  /* current R/W byte position */
   Handle      dCtlStorage;   /* handle to private storage */
   short       dCtlRefNum;    /* driver reference number */
   long        dCtlCurTicks;  /* used internally */
   GrafPtr     dCtlWindow;    /* pointer to driver's window */
   short       dCtlDelay;     /* ticks between periodic actions */
   short       dCtlEMask;     /* desk accessory event mask */
   short       dCtlMenu;      /* desk accessory menu ID */
   char        dCtlSlot;      /* slot */
   char        dCtlSlotId;    /* sResource directory ID */
   long        dCtlDevBase;   /* slot device base address */
   Ptr         dCtlOwner;     /* reserved; must be 0 */
   char        dCtlExtDev;    /* external device ID */
```

```
   char        fillByte;      /* reserved */
} AuxDCE;
typedef AuxDCE *AuxDCEPtr, **AuxDCEHandle;
```

Field descriptions

`dCtlDriver` A pointer or handle to the driver, as determined by the `dRAMBased` flag (bit 6) of the `dCtlFlags` field.

`dCtlFlags` Flags describing the abilities and state of the driver. The high-order byte contains flags copied from the `drvrFlags` word of the driver resource. These flags are described in "Creating a Driver Resource," beginning on page 1-24.

The low-order byte of the `dCtlFlags` field contains the following run-time flags:

Name	Bit	Meaning
`dOpened`	5	Set by the Device Manager when the driver is opened, and cleared when it is closed.
`dRAMBased`	6	Set if the `dCtlDriver` field contains a handle.
`drvrActive`	7	Set by the Device Manager when the driver is executing a request, and cleared when the driver is inactive.

You can use the following constants to test or set the value of these flags:

```
enum {
   /* run-time flags in the device control entry */
   dOpenedMask    = 0x0020,
   dRAMBasedMask  = 0x0040,
   drvrActiveMask = 0x0080
};
```

`dCtlQHdr` A pointer to the header of the driver I/O queue, which is a standard Operating System queue. See the chapter "Queue Utilities" in *Inside Macintosh: Operating System Utilities* for more information about the `QHdr` data type.

`dCtlPosition` The current source or destination position for reading or writing. This field is used only by drivers of block devices. The value in this field is the number of bytes beyond the physical beginning of the medium used by the device, and must be a multiple of 512. For example, immediately after the Disk Driver reads the first block of data from a 3.5-inch disk, this field contains the value 512.

`dCtlStorage` A handle to a driver's private storage. A driver may allocate a relocatable block of memory and keep a handle to it in this field.

`dCtlRefNum` The driver reference number.

`dCtlCurTicks`	Used internally.
`dCtlWindow`	A pointer to the desk accessory window. See "Writing a Desk Accessory" on page 1-49 for more information.
`dCtlDelay`	The number of ticks to wait between periodic actions.
`dCtlEMask`	The desk accessory event mask. See "Writing a Desk Accessory" on page 1-49 for more information.
`dCtlMenu`	The menu ID of a desk accessory's menu, if any. See "Writing a Desk Accessory" on page 1-49 for more information.
`dCtlSlot`	The slot number of the slot device.
`dCtlSlotId`	The sResource directory ID of the slot device.
`dCtlDevBase`	The base address of the slot device. For a video card this field contains the address of the pixel map for the card's `GDevice` record.
`dCtlOwner`	Reserved. This field must be 0.
`dCtlExtDev`	The external device ID of the slot device.
`fillByte`	Reserved.

Device Manager Functions

This section describes the functions you use to

- open and close device drivers
- communicate with device drivers
- control and monitor device drivers
- write and install device drivers

The low-level Device Manager functions described in this section (those that use the parameter block structure to pass information) provide two advantages over the corresponding high-level functions:

- These functions can be executed asynchronously, returning control to your application before the operation is completed.
- In most cases, these functions provide more extensive information or perform advanced operations.

All of these functions exchange parameters with your application through a parameter block of type `ParamBlockRec`. When you call a low-level function, you pass the address of the parameter block to the function.

There are three versions of most low-level functions. The first takes two parameters: a pointer to the parameter block and a Boolean parameter that specifies whether the function is to execute asynchronously (`true`) or synchronously (`false`). For example, the first version of the low-level `PBRead` function has this declaration:

```
pascal OSErr PBRead(ParmBlkPtr paramBlock, Boolean async);
```

The second version does not take a second parameter; instead, it adds the suffix `Sync` to the name of the function.

```
pascal OSErr PBReadSync(ParmBlkPtr paramBlock);
```

Similarly, the third version of the function does not take a second parameter; instead, it adds the suffix `Async` to the name of the function.

```
pascal OSErr PBReadAsync(ParmBlkPtr paramBlock);
```

Only the first version of each function is documented in this section. Note, however, that the second and third versions of these functions do not use the glue code that the first version uses and are therefore more efficient. See "Summary of the Device Manager," beginning on page 1-91, for a listing of all three versions of these functions.

Assembly-Language Note

All Device Manager functions are synchronous by default. If you want a function to be executed asynchronously, set bit 10 of the trap word. To execute a function immediately, set bit 9 of the trap word. You can set these bits by appending the word `ASYNC` or `IMMED` as the second argument to the trap macro. For example:

```
_Read, ASYNC
_Control, IMMED
```

You can set or test bit 10 of a trap word using the global constant `asyncTrpBit`. You can set or test bit 9 of the trap word using the global constant `noQueueBit`. ◆

▲ **WARNING**

Never call any synchronous Device Manager function at interrupt time. This includes all of the high-level functions and the synchronous versions of the low-level functions.

A synchronous request at interrupt time may block other pending I/O requests. Because the device driver cannot begin processing the synchronous request until it completes the other requests in its queue, this situation can cause the Device Manager to loop indefinitely while it waits for the device driver to complete the synchronous request. ▲

Opening and Closing Device Drivers

A device driver must be open before your application can communicate with it. You can use the `OpenDriver` or `PBOpen` function to open closed drivers or to determine the driver reference number of a driver that is already open. You use the `OpenSlot` function to open drivers that serve slot devices. To open a desk accessory or other Apple menu item from within your application, use the `OpenDeskAcc` function.

When you finish communicating with a device driver, you can close it if you are sure no other application or part of the system needs to use it. You can use the `CloseDriver` or `PBClose` function to close a driver. You use the `CloseDeskAcc` function to close a desk accessory.

The `PBOpen` and `PBClose` functions use the `IOParam` union of the Device Manager parameter block. The `OpenSlot` function uses the `IOParam` union fields and some additional fields that apply only to slot devices.

IMPORTANT
Device drivers cannot be opened or closed asynchronously. The `PBOpen`, `PBClose`, and `OpenSlot` functions include an asynchronous option because they share code with the File Manager. The `async` parameter must be set to `false` when these functions are used to open or close a device driver. ▲

OpenDriver

You can use the `OpenDriver` function to open a closed device driver or to determine the driver reference number of an open device driver.

```
pascal OSErr OpenDriver(ConstStr255Param name, short *drvrRefNum);
```

`name` The name of the driver to open. A driver name consists of a period (.) followed by any sequence of 1 to 255 printing characters. The Device Manager ignores case (but not diacritical marks) when comparing names.

`drvrRefNum` The driver reference number of the opened driver.

DESCRIPTION

The `OpenDriver` function opens the device driver specified by the `name` parameter and returns its driver reference number in the `drvrRefNum` parameter. To avoid replacing an open driver, the Device Manager searches the drivers that are already installed in the unit table before searching driver resources. If the specified driver is already open, this function simply returns the driver reference number.

If the driver is not already open, the Device Manager calls the `GetNamedResource` function using the specified name and the resource type `'DRVR'`. If the resource is found, the resource ID defines the unit number of the driver, which determines the location in the unit table where the Device Manager stores the handle to the driver's device control entry (DCE).

After loading the driver resource into memory, the Device Manager creates a DCE for the driver, copies the flags from the driver header to the `dCtlFlags` field, and places the driver reference number in the `dCtlRefNum` field.

The `OpenDriver` function is a high-level version of the low-level `PBOpen` function. Use the `PBOpen` function when you need to specify read/write permission for the driver. The next section describes the `PBOpen` function.

SPECIAL CONSIDERATIONS

Because another driver might already be installed in the unit table at the location determined by the driver's resource ID, you should first search for an unused location in the unit table and renumber the driver resource accordingly before calling this function. See Listing 1-1 on page 1-18 for an example.

The `OpenDriver` function may move memory; you should not call it at interrupt time.

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Driver reference number does not match unit table
`unitEmptyErr`	–22	Driver reference number specifies a `nil` handle in unit table
`openErr`	–23	Requested read/write permission does not match driver's open permission
`dInstErr`	–26	Driver resource not found

SEE ALSO

For information about the low-level functions for opening devices, see the next section, which describes the `PBOpen` function, and the description of the `OpenSlot` function on page 1-63. For an example of how to open a device driver using the `OpenDriver` function, see Listing 1-1 on page 1-18.

PBOpen

You can use the `PBOpen` function to open a closed device driver or to determine the driver reference number of an open device driver.

```
pascal OSErr PBOpen(ParmBlkPtr paramBlock, Boolean async);
```

`paramBlock` A pointer to an `IOParam` structure of the Device Manager parameter block.

`async` A Boolean value that indicates whether the request is asynchronous. You must set this field to `false` because device drivers cannot be opened asynchronously.

Parameter block

←	`ioResult`	`OSErr`	The device driver's result code.
→	`ioNamePtr`	`StringPtr`	A pointer to the driver name.
←	`ioRefNum`	`short`	The driver reference number.
→	`ioPermssn`	`char`	Read/write permission.

DESCRIPTION

The `PBOpen` function opens the device driver specified by the `ioNamePtr` field and returns its driver reference number in the `ioRefNum` field. To avoid replacing an open driver, the Device Manager searches the drivers that are already installed in the unit table before searching driver resources. If the specified driver is already open, this function simply returns the driver reference number.

If the driver is not already open, the Device Manager calls the `GetNamedResource` function using the specified name and the resource type `'DRVR'`. If the resource is found, the resource ID defines the unit number of the driver, which determines the location in the unit table where the Device Manager stores the handle to the driver's device control entry (DCE).

After loading the driver resource into memory, the Device Manager creates a DCE for the driver, copies the flags from the driver header to the `dCtlFlags` field, and places the driver reference number in the `dCtlRefNum` field.

You specify the access permission for the device driver by placing one of the following constants in the `ioPermssn` field of the parameter block:

```
enum {
   /* access permissions */
   fsCurPerm         = 0,        /* retain current permission */
   fsRdPerm          = 1,        /* allow reads only */
   fsWrPerm          = 2,        /* allow writes only */
   fsRdWrPerm        = 3         /* allow reads and writes */
};
```

If the driver returns a negative result in register D0, the Device Manager returns the result code in the `ioResult` parameter and does not open the driver.

SPECIAL CONSIDERATIONS

Because another driver might already be installed in the unit table at the location determined by the driver's resource ID, you should first search for an unused location in the unit table and renumber the driver resource accordingly before calling this function. See Listing 1-1 on page 1-18 for an example.

The `PBOpen` function may move memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `PBOpen` function is `_Open` (0xA000). You must set up register A0 with the address of the parameter block. When `_Open` returns, register D0 contains the result code. Register D0 is the only register affected by this function.

Registers on entry

A0 Address of the parameter block

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Driver reference number does not match unit table
`unitEmptyErr`	–22	Driver reference number specifies a `nil` handle in unit table
`openErr`	–23	Requested read/write permission does not match driver's open permission
`dInstErr`	–26	Driver resource not found

SEE ALSO

For information about the high-level function for opening device drivers, see the description of the `OpenDriver` function on page 1-60. For information about the low-level function for opening device drivers that serve devices on expansion cards, see the next section, which describes the `OpenSlot` function. For an example of opening a device driver, see Listing 1-1 on page 1-18.

OpenSlot

You can use the `OpenSlot` function to open a device driver that serves a slot device.

```
pascal OSErr OpenSlot(ParmBlkPtr paramBlock, Boolean async);
```

`paramBlock` A pointer to a `SlotDevParam` or `MultiDevParam` structure of the `ParamBlockRec` union.

`async` A Boolean value that indicates whether the request is asynchronous. You must set this field to `false` because device drivers cannot be opened asynchronously.

Parameter block

←	`ioResult`	`OSErr`	The device driver's result code.
→	`ioNamePtr`	`StringPtr`	A pointer to the driver name.
←	`ioRefNum`	`short`	The driver reference number.
→	`ioPermssn`	`char`	Read/write permission.

Additional fields for a single device

→	`ioMix`	`Ptr`	Reserved for use by the driver open routine.
→	`ioFlags`	`short`	Determines the number of additional fields.
→	`ioSlot`	`char`	The slot number.
→	`ioId`	`char`	The slot resource ID.

Additional fields for multiple devices

→	`ioMMix`	`Ptr`	Reserved for use by the driver open routine.
→	`ioMFlags`	`short`	The number of additional fields.
→	`ioSEBlkPtr`	`Ptr`	A pointer to an external parameter block.

DESCRIPTION

The `OpenSlot` function is equivalent to the `PBOpen` function, except that it sets bit 9 of the trap word, which signals the `_Open` routine that the parameter block includes additional fields.

If the sResource serves a single device, you should clear all the bits of the `ioFlags` field and include the slot number and slot resource ID in the `ioSlot` and `ioID` fields.

If the sResource serves multiple devices, you should set the `fMulti` bit (bit 0) of the `ioFlags` field (clearing all other bits to 0), and specify, in the `ioSEBlkPtr` field, an external parameter block that is customized for the devices installed in the slot.

SPECIAL CONSIDERATIONS

The `OpenSlot` function may move memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `OpenSlot` function is `_Open` (0xA200). Bit 9 of the trap word is set to signal that the parameter block contains additional fields for slot devices.

You must set up register A0 with the address of the parameter block. When `_Open` returns, register D0 contains the result code. Register D0 is the only register affected by this function.

Registers on entry

A0 Address of the parameter block

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Driver reference number does not match unit table
`unitEmptyErr`	–22	Driver reference number specifies a `nil` handle in unit table
`openErr`	–23	Requested read/write permission does not match driver's open permission
`dInstErr`	–26	Driver resource not found

SEE ALSO

For information about the low-level function for opening other device drivers, see the description of the `PBOpen` function on page 1-61. For an example of opening a device

driver, see Listing 1-1 on page 1-18. Refer to the chapter "Slot Manager" in this book for more information about slot device drivers.

OpenDeskAcc

You can use the `OpenDeskAcc` function to open an item in the Apple menu.

```
pascal short OpenDeskAcc(ConstStr255Param deskAccName);
```

`deskAccName` A Pascal string containing the name of the Apple menu item.

DESCRIPTION

The `OpenDeskAcc` function opens the Apple menu item specified by the `deskAccName` parameter. If the item is already open, the `OpenDeskAcc` function schedules it for execution and returns to your application. Otherwise, it prepares to open the item. In either case, your application receives a suspend event and the selected item is brought to the foreground.

You should ignore the value returned by `OpenDeskAcc`. If the menu item is a desk accessory and is successfully opened, the function result is a driver reference number for the desk accessory driver. Otherwise the function result is undefined. The desk accessory is responsible for informing the user of any errors.

Because some older desk accessories may not reset the current graphics port before returning, you should bracket your call to `OpenDeskAcc` with calls to the QuickDraw procedures `GetPort` and `SetPort`, to save and restore the current port.

SPECIAL CONSIDERATIONS

The `OpenDeskAcc` function may move memory; you should not call it at interrupt time.

SEE ALSO

For information about closing a desk accessory, see the description of the `CloseDeskAcc` function beginning on page 1-68.

CloseDriver

You can use the `CloseDriver` function to close an open device driver.

```
pascal OSErr CloseDriver(short refNum);
```

`refNum` The driver reference number returned by the driver-opening function.

DESCRIPTION

The `CloseDriver` function closes the device driver indicated by the `refNum` parameter. The Device Manager waits until the driver is inactive before calling the driver's close routine. When the driver indicates it has processed the close request, the Device Manager unlocks the driver resource if the `dRAMBased` flag is set, and unlocks the device control entry if the `dNeedLock` flag is not set. The Device Manager does not dispose of the device control entry or remove it from the unit table.

This function is a high-level version of the low-level `PBClose` function. Use the `PBClose` function when you want to specify a completion routine.

▲ **WARNING**
You should not close drivers that other applications may be using, such as a disk driver, the AppleTalk drivers, and so on. ▲

SPECIAL CONSIDERATIONS

The Device Manager does not queue close requests.

▲ **WARNING**
Do not call the `CloseDriver` function at interrupt time because if the driver was processing a request when the interrupt occurred the Device Manager may loop indefinitely, waiting for the driver to complete the request. ▲

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Driver reference number does not match unit table
`unitEmptyErr`	–22	Driver reference number specifies a `nil` handle in unit table
`closErr`	–24	Driver unable to complete close request
`dRemovErr`	–25	Attempt to remove an open driver

SEE ALSO

For information about the low-level function for closing device drivers, see the next section, which describes the `PBClose` function.

PBClose

You can use the `PBClose` function to close an open device driver.

```
pascal OSErr PBClose(ParmBlkPtr paramBlock, Boolean async);
```

`paramBlock` A pointer to an `IOParam` structure of the Device Manager parameter block.

`async` A Boolean value that indicates whether the request is asynchronous. You must set this field to `false` because device drivers cannot be closed asynchronously.

Parameter block

←	`ioResult`	`OSErr`	The device driver's result code.
→	`ioRefNum`	`short`	The driver reference number.

DESCRIPTION

The `PBClose` function closes the device driver specified by the `ioRefNum` field. The Device Manager waits until the driver is inactive before calling the driver's close routine. When the driver indicates it has processed the close request, the Device Manager unlocks the driver resource if the `dRAMBased` flag is set, and unlocks the device control entry if the `dNeedLock` flag is not set. The Device Manager does not dispose of the device control entry or remove it from the unit table.

If the driver returns a negative result in register D0, the Device Manager returns this result code in the `ioResult` field of the parameter block and does not close the driver.

▲ **WARNING**
You should not close drivers that other applications may be using, such as a disk driver, the AppleTalk drivers, and so on. ▲

SPECIAL CONSIDERATIONS

The Device Manager does not queue close requests.

▲ **WARNING**
Do not call the `PBClose` function at interrupt time because if the driver was processing a request when the interrupt occurred the Device Manager may loop indefinitely, waiting for the driver to complete the request. ▲

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `PBClose` function is `_Close` (0xA001).

You must set up register A0 with the address of the parameter block. When `_Close` returns, register D0 contains the result code. Register D0 is the only register affected by this function.

Registers on entry

A0 Address of the parameter block

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Driver reference number does not match unit table
`unitEmptyErr`	–22	Driver reference number specifies a `nil` handle in unit table
`closErr`	–24	Driver unable to complete close request
`dRemovErr`	–25	Attempt to remove an open driver

SEE ALSO

For information about the high-level function for closing device drivers, see the description of the `CloseDriver` function on page 1-65. For an example of how to close a device driver using the `PBClose` function, see Listing 1-2 on page 1-20.

CloseDeskAcc

You can use the `CloseDeskAcc` function to close a desk accessory.

```
pascal void CloseDeskAcc(short refNum);
```

`refNum` The driver reference number contained in the desk accessory's `WindowRecord`.

DESCRIPTION

The `CloseDeskAcc` function closes the desk accessory specified by the `refNum` parameter. Your application should call `CloseDeskAcc` only when the user selects the Close or Quit item from your File menu and the active window does not belong to your application.

You obtain the `refNum` parameter from the `windowKind` field of the desk accessory's `WindowRecord`. Do not use the driver reference number returned by `OpenDeskAcc`.

SPECIAL CONSIDERATIONS

The `CloseDeskAcc` function may move memory; you should not call it at interrupt time.

SEE ALSO

For information about opening a desk accessory or other Apple menu item, see the description of the `OpenDeskAcc` function on page 1-65.

Communicating With Device Drivers

You can use either the `FSRead` or `PBRead` function to read information from a device driver, and you can use the `FSWrite` or `PBWrite` function to write information to a device driver.

FSRead

You can use the `FSRead` function to read data from an open driver into a data buffer.

```
pascal OSErr FSRead(short refNum, long *count, void *buffPtr);
```

`refNum`	The driver reference number.
`count`	The number of bytes to read.
`buffPtr`	A pointer to a buffer to hold the data.

DESCRIPTION

Before calling the `FSRead` function, your application should allocate a data buffer large enough to hold the data to be read. The `FSRead` function attempts to read the number of bytes indicated by the `count` parameter and transfer them to the data buffer pointed to by the `buffPtr` parameter. The `refNum` parameter identifies the device driver. After the transfer is complete, the `count` parameter indicates the number of bytes actually read.

▲ **WARNING**
Be sure your buffer is large enough to hold the number of bytes specified by the `count` parameter, or this function may corrupt memory. ▲

The `FSRead` function is a high-level synchronous version of the low-level `PBRead` function. Use the `PBRead` function when you want to request asynchronous reading or need to specify a drive number or a positioning mode and offset. See the next section, which describes the `PBRead` function.

SPECIAL CONSIDERATIONS

Do not call the `FSRead` function at interrupt time. Synchronous requests at interrupt time may block other pending I/O requests and cause the Device Manager to loop indefinitely while it waits for the device driver to complete the interrupted requests.

RESULT CODES

`noErr`	0	No error
`readErr`	–19	Driver does not respond to read requests
`badUnitErr`	–21	Driver reference number does not match unit table
`unitEmptyErr`	–22	Driver reference number specifies a `nil` handle in unit table
`abortErr`	–27	Request aborted by `KillIO`
`notOpenErr`	–28	Driver not open

SEE ALSO

For information about the low-level function for reading from device drivers, see the next section, which describes the `PBRead` function.

PBRead

You can use the `PBRead` function to read data from an open driver into a data buffer.

```
pascal OSErr PBRead(ParmBlkPtr paramBlock, Boolean async);
```

`paramBlock` A pointer to an `IOParam` structure of the Device Manager parameter block.

`async` A Boolean value that indicates whether the request is asynchronous.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The device driver's result code.
→	`ioVRefNum`	`short`	The drive number.
→	`ioRefNum`	`short`	The driver reference number.
→	`ioBuffer`	`Ptr`	A pointer to a data buffer.
→	`ioReqCount`	`long`	The requested number of bytes to read.
←	`ioActCount`	`long`	The actual number of bytes read.
→	`ioPosMode`	`short`	The positioning mode.
↔	`ioPosOffset`	`long`	The positioning offset.

DESCRIPTION

Before calling the `PBRead` function, your application should allocate a data buffer large enough to hold the data to be read. The `PBRead` function attempts to read the number of bytes indicated by the `ioReqCount` field and transfer them to the data buffer pointed to by the `ioBuffer` field. The `ioRefNum` field identifies the device driver. After the transfer is complete, the `ioActCount` field indicates the number of bytes actually read.

▲ **WARNING**
Be sure your buffer is large enough to hold the number of bytes specified by the `count` parameter, or this function may corrupt memory. ▲

For block devices such as disk drivers, the `PBRead` function allows you to specify a drive number in the `ioVRefNum` field and specify a positioning mode and offset in the `ioPosMode` and `ioPosOffset` fields. Bits 0 and 1 of the `ioPosMode` field indicate where an operation should begin relative to the physical beginning of the block-formatted medium. You can use the following constants to test or set the value of these bits:

```
enum {
   /* positioning modes */
   fsAtMark          = 0,     /* at current position */
   fsFromStart       = 1,     /* offset from beginning */
   fsFromMark        = 3      /* offset from current position */
};
```

The `ioPosOffset` field specifies the positive or negative byte offset where the data is to be read, relative to the positioning mode. The offset must be a multiple of 512. The `ioPosOffset` field is ignored when `ioPosMode` is set to `fsAtMark`.

After the transfer is complete, the `ioPosOffset` field indicates the current position of the block device.

The Disk Driver allows you to use the `PBRead` function to verify that data written to a block device matches the data in memory. To do this, call `PBRead` immediately after writing the data, and add the read-verify constant `rdVerify` to the `ioPosMode` field of the parameter block. The result code `ioErr` is returned if the data does not match.

SPECIAL CONSIDERATIONS

Do not call the `PBRead` function synchronously at interrupt time. Synchronous requests at interrupt time may block other pending I/O requests and cause the Device Manager to loop indefinitely while it waits for the device driver to complete the interrupted requests.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `PBRead` function is `_Read` (0xA002). Set bit 10 of the trap word to execute this function asynchronously. Set bit 9 to execute it immediately.

You must set up register A0 with the address of the parameter block. When `_Read` returns, register D0 contains the result code. Register D0 is the only register affected by this function.

Registers on entry

A0 Address of the parameter block

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`readErr`	–19	Driver does not respond to read requests
`badUnitErr`	–21	Driver reference number does not match unit table
`unitEmptyErr`	–22	Driver reference number specifies a `nil` handle in unit table
`abortErr`	–27	Request aborted by `KillIO`
`notOpenErr`	–28	Driver not open
`ioErr`	–36	Data does not match in read-verify mode

SEE ALSO

For information about the high-level function for reading from device drivers, see the description of the `FSRead` function beginning on page 1-69. For an example of how to read from a device driver using the `PBRead` function, see Listing 1-3 on page 1-21.

FSWrite

You can use the `FSWrite` function to write data from a data buffer to an open driver.

```
pascal OSErr FSWrite(short refNum, long *count,
                     const void *buffPtr);
```

`refNum`	The driver reference number.
`count`	The number of bytes to write.
`buffPtr`	A pointer to the buffer that holds the data.

DESCRIPTION

The `FSWrite` function attempts to write the number of bytes indicated by the `count` parameter from the data buffer pointed to by the `buffPtr` parameter to the device driver specified by the `refNum` parameter. After the transfer is complete, the `count` parameter indicates the number of bytes actually written.

The `FSWrite` function is a high-level synchronous version of the low-level `PBWrite` function. Use the `PBWrite` function when you want to request asynchronous writing or need to specify a drive number or a positioning mode and offset. See the next section, which describes the `PBWrite` function.

SPECIAL CONSIDERATIONS

Do not call the `FSWrite` function at interrupt time. Synchronous requests at interrupt time may block other pending I/O requests and cause the Device Manager to loop indefinitely while it waits for the device driver to complete the interrupted requests.

RESULT CODES

noErr	0	No error
writErr	–20	Driver does not respond to write requests
badUnitErr	–21	Driver reference number does not match unit table
unitEmptyErr	–22	Driver reference number specifies a `nil` handle in unit table
abortErr	–27	Request aborted by `KillIO`
notOpenErr	–28	Driver not open

SEE ALSO

For information about the low-level function for writing to device drivers, see the next section, which describes the `PBWrite` function.

PBWrite

You can use the `PBWrite` function to write data from a data buffer to an open driver.

```
pascal OSErr PBWrite(ParmBlkPtr paramBlock, Boolean async);
```

`paramBlock` A pointer to an `IOParam` structure of the Device Manager parameter block.

`async` A Boolean value that indicates whether the request is asynchronous.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The device driver's result code.
→	ioVRefNum	short	The drive number.
→	ioRefNum	short	The driver reference number.
→	ioBuffer	Ptr	A pointer to a data buffer.
→	ioReqCount	long	The requested number of bytes to write.
←	ioActCount	long	The actual number of bytes written.
→	ioPosMode	short	The positioning mode.
↔	ioPosOffset	long	The positioning offset.

DESCRIPTION

The `PBWrite` function attempts to write the number of bytes indicated by the `ioReqCount` field from the data buffer pointed to by the `ioBuffer` field to the device driver specified by the `ioRefNum` field. After the transfer is complete, the `ioActCount` field indicates the number of bytes actually written.

For block devices such as disk drivers, the `PBWrite` function allows you to specify a drive number in the `ioVRefNum` field and specify a positioning mode and offset in the `ioPosMode` and `ioPosOffset` fields. Bits 0 and 1 of the `ioPosMode` field indicate where an operation should begin relative to the physical beginning of the block-formatted medium. You can use the following constants to test or set the value of these bits:

```
enum {
   /* positioning modes */
   fsAtMark          = 0,     /* at current position */
   fsFromStart       = 1,     /* offset from beginning */
   fsFromMark        = 3      /* offset from current position */
};
```

The `ioPosOffset` field specifies the positive or negative byte offset where the data is to be written, relative to the positioning mode. The offset must be a multiple of 512. The `ioPosOffset` field is ignored when `ioPosMode` is set to `fsAtMark`.

After the transfer is complete, the `ioPosOffset` field indicates the new current position of a block device.

SPECIAL CONSIDERATIONS

Do not call the `PBWrite` function synchronously at interrupt time. Synchronous requests at interrupt time may block other pending I/O requests and cause the Device Manager to loop indefinitely while it waits for the device driver to complete the interrupted requests.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `PBWrite` function is `_Write` (0xA003). Set bit 10 of the trap word to execute this function asynchronously. Set bit 9 to execute it immediately.

You must set up register A0 with the address of the parameter block. When `_Write` returns, register D0 contains the result code. Register D0 is the only register affected by this function.

Registers on entry

A0 Address of the parameter block

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`writErr`	–20	Driver does not respond to write requests
`badUnitErr`	–21	Driver reference number does not match unit table
`unitEmptyErr`	–22	Driver reference number specifies a `nil` handle in unit table
`abortErr`	–27	Request aborted by `KillIO`
`notOpenErr`	–28	Driver not open

SEE ALSO

For information about the high-level function for writing to device drivers, see the description of the `FSWrite` function on page 1-72. For an example of how to write to a device driver using the `PBWrite` function, see Listing 1-4 on page 1-22.

Controlling and Monitoring Device Drivers

You can use either the `Control` or `PBControl` function to send control information to a device driver, and you can use the `Status` or `PBStatus` function to obtain status information from a device driver. The Device Manager also provides the `KillIO` and `PBKillIO` functions for terminating all requests in a driver I/O queue.

The `PBControl`, `PBStatus`, and `PBKillIO` functions use the `CntrlParam` structure, described on page 1-53.

Control

You can use the `Control` function to send control information to a device driver.

```
pascal OSErr Control(short refNum, short csCode,
                     const void *csParamPtr);
```

`refNum` The driver reference number.

`csCode` A driver-dependent code specifying the type of information sent.

`csParamPtr` A pointer to the control information.

DESCRIPTION

The `Control` function sends information to the device driver specified by the `refNum` parameter. The value you pass in the `csCode` parameter and the type of information pointed to by the `csParamPtr` parameter are defined by the driver you are calling. For more information, see the appropriate chapters for the standard device drivers in this book and other books in the *Inside Macintosh* series.

The `Control` function is a high-level synchronous version of the low-level `PBControl` function. Use the `PBControl` function if you need to specify a drive number or if you want the control request to be executed asynchronously.

SPECIAL CONSIDERATIONS

Do not call the `Control` function at interrupt time. Synchronous requests at interrupt time may block other pending I/O requests and cause the Device Manager to loop indefinitely while it waits for the device driver to complete the interrupted requests.

RESULT CODES

`noErr`	0	No error
`controlErr`	–17	Driver does not respond to this control request
`badUnitErr`	–21	Driver reference number does not match unit table
`unitEmptyErr`	–22	Driver reference number specifies a `nil` handle in unit table
`abortErr`	–27	Request aborted by `KillIO`
`notOpenErr`	–28	Driver not open

SEE ALSO

For information about the low-level function for controlling device drivers, see the next section, which describes the `PBControl` function.

PBControl

You can use the `PBControl` function to send control information to a device driver.

```
pascal OSErr PBControl(ParmBlkPtr paramBlock, Boolean async);
```

`paramBlock` A pointer to a `CntrlParam` structure of the Device Manager parameter block.

`async` A Boolean value that indicates whether the request is asynchronous.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The device driver's result code.
→	`ioVRefNum`	`short`	The drive number.
→	`ioCRefNum`	`short`	The driver reference number.
→	`csCode`	`short`	The type of control call.
→	`csParam`	`short[11]`	The control information.

DESCRIPTION

The `PBControl` function sends information to the device driver specified by the `ioCRefNum` field. The value you pass in the `csCode` field and the type of information in the `csParam` field are defined by the driver you are calling. For more information, see the appropriate chapters for the standard device drivers in this book and other books in the *Inside Macintosh* series.

SPECIAL CONSIDERATIONS

Do not call the `PBControl` function synchronously at interrupt time. Synchronous requests at interrupt time may block other pending I/O requests and cause the Device Manager to loop indefinitely while it waits for the device driver to complete the interrupted requests.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `PBControl` function is `_Control` (0xA004). Set bit 10 of the trap word to execute this routine asynchronously. Set bit 9 to execute it immediately.

You must set up register A0 with the address of the parameter block. When `_Control` returns, register D0 contains the result code. Register D0 is the only register affected by this routine.

Registers on entry

A0 Address of the parameter block

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`controlErr`	–17	Driver does not respond to this control request
`badUnitErr`	–21	Driver reference number does not match unit table
`unitEmptyErr`	–22	Driver reference number specifies a `nil` handle in unit table
`abortErr`	–27	Request aborted by `KillIO`
`notOpenErr`	–28	Driver not open

SEE ALSO

For information about the high-level function for controlling device drivers, see the description of the `Control` function on page 1-75. For an example of how to send control information to a device driver using the `PBControl` function, see Listing 1-5 on page 1-23.

Status

You can use the `Status` function to obtain status information from a device driver.

```
pascal OSErr Status(short refNum, short csCode,
                    void *csParamPtr);
```

`refNum` The driver reference number.

`csCode` A driver-dependent code specifying the type of information requested.

`csParamPtr` A pointer to a `csParam` array where the status information will be returned.

DESCRIPTION

The `Status` function returns information about the device driver specified by the `refNum` parameter. The value you pass in the `csCode` parameter and the received

information pointed to by the `csParamPtr` parameter are defined by the driver you are calling. For more information, see the appropriate chapters for the standard device drivers in this book and other books in the *Inside Macintosh* series.

The `Status` function is a high-level synchronous version of the low-level `PBStatus` function. Use the `PBStatus` function if you need to specify a drive number or if you want the status request to be asynchronous.

Note
The Device Manager interprets a `csCode` value of 1 as a special case. When the Device Manager receives a status request with a `csCode` value of 1, it returns a handle to the driver's device control entry. This type of status request is not passed to the device driver. ◆

SPECIAL CONSIDERATIONS

Do not call the `Status` function at interrupt time. Synchronous requests at interrupt time may block other pending I/O requests and cause the Device Manager to loop indefinitely while it waits for the device driver to complete the interrupted requests.

RESULT CODES

`noErr`	0	No error
`statusErr`	–18	Driver does not respond to this status request
`badUnitErr`	–21	Driver reference number does not match unit table
`unitEmptyErr`	–22	Driver reference number specifies a `nil` handle in unit table
`abortErr`	–27	Request aborted by `KillIO`
`notOpenErr`	–28	Driver not open

SEE ALSO

For information about the low-level function for monitoring device drivers, see the next section, which describes the `PBStatus` function.

PBStatus

You can use the `PBStatus` function to obtain status information from a device driver.

```
pascal OSErr PBStatus(ParmBlkPtr paramBlock, Boolean async);
```

`paramBlock` A pointer to a `CntrlParam` structure of the Device Manager parameter block.

`async` A Boolean value that indicates whether the request is asynchronous.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The device driver's result code.
→	`ioVRefNum`	`short`	The drive number.
→	`ioCRefNum`	`short`	The driver reference number.
→	`csCode`	`short`	The type of status call.
←	`csParam`	`short[11]`	The status information.

DESCRIPTION

The `PBStatus` function returns information about the device driver specified by the `ioCRefNum` field. The value you pass in the `csCode` field and the type of information received in the `csParam` field are defined by the driver you are calling. For more information, see the appropriate chapters for the standard device drivers in this book and other books in the *Inside Macintosh* series.

Note
The Device Manager interprets a `csCode` value of 1 as a special case. When the Device Manager receives a status request with a `csCode` value of 1, it returns a handle to the driver's device control entry. This type of status request is not passed to the device driver. ◆

SPECIAL CONSIDERATIONS

Do not call the `PBStatus` function synchronously at interrupt time. Synchronous requests at interrupt time may block other pending I/O requests and cause the Device Manager to loop indefinitely while it waits for the device driver to complete the interrupted requests.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `PBStatus` function is `_Status` (0xA005). Set bit 10 of the trap word to execute this function asynchronously. Set bit 9 to execute it immediately.

You must set up register A0 with the address of the parameter block. When `_Status` returns, register D0 contains the result code. Register D0 is the only register affected by this function.

Registers on entry

A0 Address of the parameter block

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`statusErr`	–18	Driver does not respond to this status request
`badUnitErr`	–21	Driver reference number does not match unit table
`unitEmptyErr`	–22	Driver reference number specifies a `nil` handle in unit table
`abortErr`	–27	Request aborted by `KillIO`
`notOpenErr`	–28	Driver not open

SEE ALSO

For information about the high-level function for monitoring device drivers, see the description of the `Status` function on page 1-77. For an example of how to request status information from a device driver using the `PBStatus` function, see Listing 1-5 on page 1-23.

KillIO

You can use the `KillIO` function to terminate all current and pending I/O requests for a device driver.

```
pascal OSErr KillIO(short refNum);
```

`refNum` The driver reference number.

DESCRIPTION

The `KillIO` function stops any current I/O request being processed by the driver specified by the `RefNum` parameter, and removes all pending requests from the I/O queue for that driver. The Device Manager calls the completion routine, if any, for each pending request, and sets the `ioResult` field of each request equal to the result code `abortErr`.

The Device Manager passes `KillIO` requests to a driver only if the driver is open and enabled for control calls. If the driver returns an error, the I/O queue is left unchanged and no completion routines are called.

▲ **WARNING**
The `KillIO` function terminates all pending I/O requests for a driver, including requests initiated by other applications. ▲

SPECIAL CONSIDERATIONS

The Device Manager always executes the `KillIO` function immediately; that is, it never places a `KillIO` request in the I/O queue.

Although the Device Manager imposes no restrictions on calling `KillIO` at interrupt time, you should consult a device driver's documentation to determine if it supports this.

RESULT CODES

`noErr`	0	No error
`controlErr`	–17	Driver does not respond to this control request
`badUnitErr`	–21	Driver reference number does not match unit table
`unitEmptyErr`	–22	Driver reference number specifies a `nil` handle in unit table
`notOpenErr`	–28	Driver not open

SEE ALSO

For information about the low-level function for terminating current and pending I/O requests for a driver, see the next section, which describes the `PBKillIO` function.

PBKillIO

You can use the `PBKillIO` function to terminate all current and pending I/O requests for a device driver.

```
pascal OSErr PBKillIO(ParmBlkPtr paramBlock, Boolean async);
```

`paramBlock` A pointer to a `CntrlParam` structure of the Device Manager parameter block.

`async` A Boolean value that indicates whether the request is asynchronous. You must set this field to `false` because the `PBKillIO` function does not support asynchronous requests.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The device driver's result code.
→	`ioCRefNum`	`short`	The driver reference number.

DESCRIPTION

The `PBKillIO` function stops any current I/O request being processed by the driver specified by the `ioCRefNum` field, and removes all pending requests from the I/O queue for that driver. The Device Manager calls the completion routine, if any, for each pending request, and sets the `ioResult` field of each request equal to the result code `abortErr`.

The Device Manager passes `PBKillIO` requests to a device driver only if the driver is open and enabled for control calls. If the driver returns an error, the I/O queue is left unchanged and no completion routines are called.

▲ **WARNING**
The `PBKillIO` function terminates all pending I/O requests for a driver, including requests initiated by other applications. ▲

SPECIAL CONSIDERATIONS

The Device Manager always executes the `PBKillIO` function immediately; that is, it never places a `PBKillIO` request in the I/O queue. However, you should not call this function immediately—always call the `PBKillIO` function synchronously.

Although the Device Manager imposes no restrictions on calling `PBKillIO` at interrupt time, you should consult a device driver's documentation to determine if it supports this.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `PBKillIO` function is `_KillIO` (0xA006). You must set up register A0 with the address of the parameter block. When `_KillIO` returns, register D0 contains the result code. Register D0 is the only register affected by this function.

Registers on entry

A0 Address of the parameter block

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`controlErr`	–17	Driver does not respond to this control request
`badUnitErr`	–21	Driver reference number does not match unit table
`unitEmptyErr`	–22	Driver reference number specifies a `nil` handle in unit table
`notOpenErr`	–28	Driver not open

SEE ALSO

For information about the high-level function for terminating current and pending I/O requests for a driver, see the description of the `KillIO` function on page 1-80.

Writing and Installing Device Drivers

The Device Manager includes a number of functions that provide low-level support for device drivers.

The `DriverInstall` and `DriverInstallReserveMem` functions create a device control entry and install it in the unit table. The `DriverInstallReserveMem` function is preferred because it allocates the device control entry as low as possible in the system heap. The `DriverRemove` function removes an existing device control entry.

The `GetDCtlEntry` function returns a handle to a driver's device control entry.

The `IODone` routine notifies the Device Manager that an I/O operation is done. Driver routines call `IODone` when the current request is completed and ready to be removed from the I/O queue.

The `Fetch` and `Stash` routines can be used to move characters into and out of data buffers. You pass a pointer to the device control entry in the A1 register to each of these three routines. The Device Manager uses the device control entry to locate the active request. If no such request exists, these routines generate system error `dsIOCoreErr`.

In the interest of speed, you invoke the `Fetch`, `Stash`, and `IODone` routines with jump vectors, stored in the global variables `JFetch`, `JStash`, and `JIODone`, rather than macros. You can use a jump vector by moving its address onto the stack and executing an `RTS` instruction. An example is:

```
MOVE.L   JIODone,-(SP)
RTS
```

The `Fetch` and `Stash` routines do not return a result code; if an error occurs, the System Error Handler is invoked.

DriverInstall

You can use the `DriverInstall` function to create a device control entry and install it in the unit table.

```
pascal OSErr DriverInstall(Ptr drvrPtr, short refNum);
```

`drvrPtr` A pointer to the device driver.

`refNum` The driver reference number.

DESCRIPTION

The `DriverInstall` function allocates a device control entry (DCE) in the system heap and installs a handle to this DCE in the unit table location specified by the `refNum` parameter. You pass a pointer to the device driver in the `drvrPtr` parameter.

In addition, this function copies the `refNum` parameter to the `dCtlRefNum` field of the DCE, sets the `dRAMBased` flag in the `dCtlFlags` field, and clears all the other fields.

SPECIAL CONSIDERATIONS

The `DriverInstall` function does not load the driver resource into memory, copy the flags from the driver header to the `dCtlFlags` field, or open the driver. You can write code to perform these tasks, or use the `OpenDriver`, `OpenSlot`, or `PBOpen` functions instead.

The `DriverInstall` function allocates memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `DriverInstall` function is `_DrvrInstall` (0xA03D).

You place a pointer to the device driver in register A0, and the driver reference number in register D0. When `_DrvrInstall` returns, register D0 contains the result code.

Registers on entry

A0 A pointer to the device driver

D0 The driver reference number

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Driver reference number does not match unit table

SEE ALSO

For information about the `DriverInstallReserveMem` function, which installs a driver as low as possible in the system heap, see the next section.

DriverInstallReserveMem

You can use the `DriverInstallReserveMem` function to create a device control entry and install it in the unit table.

```
pascal OSErr DriverInstallReserveMem(Ptr drvrPtr, short refNum);
```

`drvrPtr` A pointer to the device driver.

`refNum` The driver reference number.

DESCRIPTION

The `DriverInstallReserveMem` function is equivalent to the `DriverInstall` function, except that it calls the Memory Manager `ReserveMem` function to compact the heap before allocating memory for the device control entry (DCE).

After calling the `ReserveMem` function, the `DriverInstallReserveMem` function allocates a DCE in the system heap and installs a handle to this DCE in the unit table location specified by the `refNum` parameter. You pass a pointer to the device driver in the `drvrPtr` parameter.

In addition, this function copies the `refNum` parameter to the `dCtlRefNum` field of the DCE, sets the `dRAMBased` flag in the `dCtlFlags` field, and clears all the other fields.

SPECIAL CONSIDERATIONS

The `DriverInstallReserveMem` function does not load the driver resource into memory, copy the flags from the driver header to the `dCtlFlags` field, or open the driver. You can write code to perform these tasks, or use the `OpenDriver`, `OpenSlot`, or `PBOpen` functions instead.

The `DriverInstallReserveMem` function allocates memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `DriverInstallReserveMem` function is `_DrvrInstall` (0xA03D). You must set bit 10 of the trap word to signal the Device Manager to call the `ReserveMem` function before allocating memory for the DCE.

You place a pointer to the device driver in register A0, and the driver reference number in register D0. When `_DrvrInstall` returns, register D0 contains the result code.

Registers on entry

A0 A pointer to the device driver

D0 The driver reference number

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Driver reference number does not match unit table

DriverRemove

You can use the `DriverRemove` function to remove a device driver's device control entry from the unit table and release the driver resource.

```
pascal OSErr DriverRemove(short refNum);
```

`refNum` The driver reference number.

DESCRIPTION

The `DriverRemove` function removes a device driver's device control entry from the unit table and releases the driver resource. You specify the device driver using the `refNum` parameter. You must close the device driver before calling `DriverRemove`.

If the driver is closed, `DriverRemove` calls the Memory Manager function `DisposeHandle` to release the device control entry, then sets the corresponding handle

in the unit table to `nil`. If the driver's `dRAMBased` flag is set, `DriverRemove` calls the Resource Manager function `ReleaseResource` to release the driver resource.

SPECIAL CONSIDERATIONS

The `DriverRemove` function may move memory; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `DriverRemove` function is `_DrvrRemove` (0xA03E).

You place the driver reference number in register D0. When `_DrvrRemove` returns, register D0 contains the result code.

Registers on entry

D0 The driver reference number

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`dRemovErr`	–25	Attempt to remove an open driver

GetDCtlEntry

You can use the `GetDCtlEntry` function to obtain a handle to the device control entry of a device driver.

```
pascal DCtlHandle GetDCtlEntry (short refNum);
```

`refNum` The reference number of the driver.

DESCRIPTION

The `GetDCtlEntry` function returns a handle to the device control entry of the device driver indicated by the `refNum` parameter.

SEE ALSO

For a description of the device control entry structure see page 1-56.

IODone

You use the `IODone` routine to notify the Device Manager that an I/O request has completed.

DESCRIPTION

The `IODone` routine sets the `ioResult` field of the parameter block with the value returned by the driver in register D0. It then removes the current request from the driver I/O queue and marks the driver inactive. If there are no pending requests, and the `dNeedLock` bit of the `dCtlFlags` word is not set, `IODone` unlocks the driver and its device control entry. Finally, `IODone` executes the completion routine, if any.

The section "Entering and Exiting From Driver Routines," beginning on page 1-29, explains when to use this routine.

ASSEMBLY-LANGUAGE INFORMATION

Registers on entry

A1 Pointer to DCE

D0 Result code

Jump vector

`JIODone`

SEE ALSO

For an example of how to call the `IODone` routine from an assembly-language dispatching routine, see Listing 1-8 on page 1-29.

Fetch

You can use the `Fetch` routine to get the next character from the data buffer.

DESCRIPTION

The `Fetch` routine gets the next character from the data buffer pointed to by the `ioBuffer` field of the parameter block of the pending request. It increments the `ioActCount` field by 1. If the `ioActCount` field equals the `ioReqCount` field, this routine sets bit 15 of register D0. After receiving the last byte request, the driver should jump to the `IODone` routine.

Registers on entry

A1 Pointer to the device control entry

Registers on exit

D0 Character fetched; bit 15 = 1 if this is the last character in the buffer

Jump vector

`JFetch`

Stash

You can use the `Stash` routine to store the next character from the data buffer.

DESCRIPTION

The `Stash` routine places the character in register D0 into the data buffer pointed to by the `ioBuffer` field of the parameter block of the pending request and increments the `ioActCount` field by 1. If the `ioActCount` field equals the `ioReqCount` field, this routine sets bit 15 of register D0. After stashing the last byte requested, the driver should jump to the `IODone` routine.

ASSEMBLY-LANGUAGE INFORMATION

Registers on entry

A1 Pointer to DCE

D0 Character to stash

Registers on exit

D0 Bit 15 = 1 if this is the last character in the buffer

Jump vector

`JStash`

Resources

This section describes the driver resource, which you can use to store your device drivers and desk accessories. If your device driver requires a user interface, you can create a Chooser extension and store your driver in a device package resource. For more information, see "Creating a Device Package" on page 1-45.

The Driver Resource

Listing 1-15 shows the Rez format of the `'DRVR'` resource type.

Listing 1-15 `'DRVR'` resource format

```
type 'DRVR' {
   boolean = 0;
   boolean  dontNeedLock, needLock;       /* lock drvr in memory */
   boolean  dontNeedTime, needTime;       /* for periodic action */
   boolean  dontNeedGoodbye, needGoodbye; /* call before heap reinit */
   boolean  noStatusEnable, statusEnable; /* responds to Status */
   boolean  noCtlEnable, ctlEnable;       /* responds to Control */
   boolean  noWriteEnable, writeEnable;   /* responds to Write */
   boolean  noReadEnable, readEnable;     /* responds to Read */
   byte = 0;
   integer;                               /* driver delay */
   unsigned hex integer;                  /* DA event mask */
   integer;                               /* DA menu */
   unsigned hex integer;                  /* offset to Open */
   unsigned hex integer;                  /* offset to Prime */
   unsigned hex integer;                  /* offset to Control */
   unsigned hex integer;                  /* offset to Status */
   unsigned hex integer;                  /* offset to Close */
   pstring;                               /* driver name */
   hex string;                            /* driver code */
};
```

The driver resource begins with seven flags that specify certain characteristics of the driver.

You need to set the `dNeedLock` flag if your driver's code should be locked in memory.

You set the `dNeedTime` flag of the `drvrFlags` word if your device driver needs to perform some action periodically.

You need to set the `dNeedGoodbye` flag if you want your application to receive a `goodbye` control request before the heap is reinitialized.

The last four flags indicate which Device Manager requests the driver's routines can respond to.

The next element of the resource specifies the time between periodic tasks.

The next two elements provide an event mask and menu ID for desk accessories. The section "Writing a Desk Accessory" on page 1-49 describes these fields.

Offsets to the driver routines follow the desk accessory fields. See "Entering and Exiting From Driver Routines" on page 1-29 for more information about the routine offsets.

The next element of the driver resource is the driver name. You can use uppercase and lowercase letters when naming your driver, but the first character should be a period—`.MyDriver`, for example.

Your driver routines, which follow the driver name, must be aligned on a word boundary.

The section "Creating a Driver Resource" on page 1-24 discusses this structure in detail.

Summary of the Device Manager

C Summary

Constants

```
enum {
   /* request codes passed by the Device Manager to a driver's
      prime routine */
   aRdCmd            = 2,     /* read operation requested */
   aWrCmd            = 3      /* write operation requested */
};

enum {
   /* flags used in the driver header and device control entry */
   dNeedLockMask    = 0x4000, /* set if driver must be locked in memory as
                                 soon as it is opened */
   dNeedTimeMask    = 0x2000, /* set if driver needs time for performing
                                 periodic tasks */
   dNeedGoodByeMask = 0x1000, /* set if driver needs to be called before the
                                 application heap is initialized */
   dStatEnableMask  = 0x0800, /* set if driver responds to status requests */
   dCtlEnableMask   = 0x0400, /* set if driver responds to control requests */
   dWritEnableMask  = 0x0200, /* set if driver responds to write requests */
   dReadEnableMask  = 0x0100, /* set if driver responds to read requests */

   /* run-time flags used in the device control entry */
   drvrActiveMask   = 0x0080, /* driver is currently processing a request */
   dRAMBasedMask    = 0x0040, /* dCtlDriver is a handle (1) or pointer (0) */
   dOpenedMask      = 0x0020  /* driver is open */
};

enum {
   /* access permissions */
   fsCurPerm         = 0,     /* retain current permission */
   fsRdPerm          = 1,     /* allow reads only */
   fsWrPerm          = 2,     /* allow writes only */
   fsRdWrPerm        = 3,     /* allow reads and writes */
```

```
   /* positioning modes */
   fsAtMark          = 0,     /* at current position */
   fsFromStart       = 1,     /* offset from beginning */
   fsFromMark        = 3,     /* offset from current position */

   /* read modes */
   rdVerify          = 64     /* read-verify mode */
};

enum {
   /* control codes */
   goodbye           = -1,    /* heap being reinitialized */
   killCode          =  1,    /* KillIO requested */
   accEvent          = 64,    /* handle an event */
   accRun            = 65,    /* time for periodic action */
   accCursor         = 66,    /* change cursor shape */
   accMenu           = 67,    /* handle menu item */
   accUndo           = 68,    /* handle undo command */
   accCut            = 70,    /* handle cut command */
   accCopy           = 71,    /* handle copy command */
   accPaste          = 72,    /* handle paste command */
   accClear          = 73     /* handle clear command */
};

enum {
   /* Chooser messages */
   chooserInitMsg    = 11,    /* the user selected this device package */
   newSelMsg         = 12,    /* the user made new device selections */
   fillListMsg       = 13,    /* fill the device list with choices */
   getSelMsg         = 14,    /* mark one or more choices as selected */
   selectMsg         = 15,    /* the user made a selection */
   deselectMsg       = 16,    /* the user canceled a selection */
   terminateMsg      = 17,    /* allows device package to clean up */
   buttonMsg         = 19     /* the user selected a button */
};
```

Data Types

```
typedef union ParamBlockRec {
   IOParam        ioParam;
   FileParam      fileParam;
   VolumeParam    volumeParam;
   CntrlParam     cntrlParam;
```

```
   SlotDevParam   slotDevParam;
   MultiDevParam  multiDevParam;
} ParamBlockRec;
typedef ParamBlockRec *ParmBlkPtr;

typedef struct IOParam {
   QElemPtr    qLink;         /* next queue entry */
   short       qType;         /* queue type */
   short       ioTrap;        /* routine trap */
   Ptr         ioCmdAddr;     /* routine address */
   ProcPtr     ioCompletion;  /* completion routine address */
   OSErr       ioResult;      /* result code */
   StringPtr   ioNamePtr;     /* pointer to driver name */
   short       ioVRefNum;     /* volume reference or drive number */
   short       ioRefNum;      /* driver reference number */
   char        ioVersNum;     /* not used by the Device Manager */
   char        ioPermssn;     /* read/write permission */
   Ptr         ioMisc;        /* not used by the Device Manager */
   Ptr         ioBuffer;      /* pointer to data buffer */
   long        ioReqCount;    /* requested number of bytes */
   long        ioActCount;    /* actual number of bytes completed */
   short       ioPosMode;     /* positioning mode */
   long        ioPosOffset;   /* positioning offset */
} IOParam;

typedef struct CntrlParam {
   QElemPtr    qLink;         /* next queue entry */
   short       qType;         /* queue type */
   short       ioTrap;        /* routine trap */
   Ptr         ioCmdAddr;     /* routine address */
   ProcPtr     ioCompletion;  /* completion routine address */
   OSErr       ioResult;      /* result code */
   StringPtr   ioNamePtr;     /* pointer to driver name */
   short       ioVRefNum;     /* volume reference or drive number */
   short       ioCRefNum;     /* driver reference number */
   short       csCode;        /* type of control or status request */
   short       csParam[11];   /* control or status information */
} CntrlParam;

typedef struct AuxDCE {
   Ptr         dCtlDriver;    /* pointer or handle to driver */
   short       dCtlFlags;     /* flags */
   QHdr        dCtlQHdr;      /* I/O queue header */
   long        dCtlPosition;  /* current R/W byte position */
```

```
   Handle      dCtlStorage;   /* handle to private storage */
   short       dCtlRefNum;    /* driver reference number */
   long        dCtlCurTicks;  /* used internally */
   GrafPtr     dCtlWindow;    /* pointer to driver's window */
   short       dCtlDelay;     /* ticks between periodic actions */
   short       dCtlEMask;     /* desk accessory event mask */
   short       dCtlMenu;      /* desk accessory menu ID */
   char        dCtlSlot;      /* slot */
   char        dCtlSlotId;    /* sResource directory ID */
   long        dCtlDevBase;   /* slot device base address */
   Ptr         dCtlOwner;     /* reserved; must be 0 */
   char        dCtlExtDev;    /* external device ID */
   char        fillByte;      /* reserved */
} AuxDCE;
typedef AuxDCE *AuxDCEPtr, **AuxDCEHandle;
```

Functions

Opening and Closing Device Drivers

```
pascal OSErr OpenDriver     (ConstStr255Param name, short *drvrRefNum);
pascal OSErr PBOpen         (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBOpenSync     (ParmBlkPtr paramBlock);
pascal OSErr OpenSlot       (ParmBlkPtr paramBlock, Boolean async);
pascal short OpenDeskAcc    (ConstStr255Param deskAccName);
pascal OSErr CloseDriver    (short refNum);
pascal OSErr PBClose        (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBCloseSync    (ParmBlkPtr paramBlock);
pascal void  CloseDeskAcc   (short refNum);
```

Communicating With Device Drivers

```
pascal OSErr FSRead         (short refNum, long *count, void *buffPtr);
pascal OSErr PBRead         (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBReadSync     (ParmBlkPtr paramBlock);
pascal OSErr PBReadAsync    (ParmBlkPtr paramBlock);
pascal OSErr FSWrite        (short refNum, long *count, const void *buffPtr);
pascal OSErr PBWrite        (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBWriteSync    (ParmBlkPtr paramBlock);
pascal OSErr PBWriteAsync   (ParmBlkPtr paramBlock);
```

Controlling and Monitoring Device Drivers

```
pascal OSErr Control        (short refNum, short csCode, const void
                             *csParamPtr);
pascal OSErr PBControl      (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBControlSync  (ParmBlkPtr paramBlock);
pascal OSErr PBControlAsync (ParmBlkPtr paramBlock);
pascal OSErr Status         (short refNum, short csCode, void *csParamPtr);
pascal OSErr PBStatus       (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBStatusSync   (ParmBlkPtr paramBlock);
pascal OSErr PBStatusAsync  (ParmBlkPtr paramBlock);
pascal OSErr KillIO         (short refNum);
pascal OSErr PBKillIO       (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBKillIOSync   (ParmBlkPtr paramBlock);
pascal OSErr PBKillIOAsync  (ParmBlkPtr paramBlock);
```

Driver Support Functions

```
pascal OSErr DriverInstall  (Ptr drvrPtr, short refNum);
pascal OSErr DriverInstallReserveMem (Ptr drvrPtr, short refNum);
pascal OSErr DriverRemove   (short refNum);
pascal DCtlHandle GetDCtlEntry (short refNum);
```

Pascal Summary

Constants

```
CONST
   {request codes passed by the Device Manager to a driver's prime routine}
   aRdCmd            = 2;        {read operation requested}
   aWrCmd            = 3;        {write operation requested}

   {flags used in the driver header and device control entry}
   dNeedLockMask     = $4000;    {set if driver must be locked in memory as }
                                 { soon as it is opened}
   dNeedTimeMask     = $2000;    {set if driver needs time for performing }
                                 { periodic tasks}
   dNeedGoodByeMask  = $1000;    {set if driver needs to be called before }
                                 { the application heap is initialized}
   dStatEnableMask   = $0800;    {set if driver responds to status requests}
```

```
dCtlEnableMask    = $0400;    {set if driver responds to control requests}
dWritEnableMask   = $0200;    {set if driver responds to write requests}
dReadEnableMask   = $0100;    {set if driver responds to read requests}

{run-time flags used in the device control entry}
drvrActiveMask    = $0080;    {driver is currently processing a request}
dRAMBasedMask     = $0040;    {dCtlDriver is a handle (1) or pointer (0)}
dOpenedMask       = $0020;    {driver is open}

{access permissions}
fsCurPerm         = 0;        {retain current permission}
fsRdPerm          = 1;        {allow reads only}
fsWrPerm          = 2;        {allow writes only}
fsRdWrPerm        = 3;        {allow reads and writes}

{positioning modes}
fsAtMark          = 0;        {at current position}
fsFromStart       = 1;        {offset from beginning}
fsFromMark        = 3;        {offset from current position}

{read modes}
rdVerify          = 64;       {read-verify mode}

{control codes}
goodbye           = -1;       {heap being reinitialized}
killCode          =  1;       {KillIO requested}
accEvent          = 64;       {handle an event}
accRun            = 65;       {time for periodic action}
accCursor         = 66;       {change cursor shape}
accMenu           = 67;       {handle menu item}
accUndo           = 68;       {handle undo command}
accCut            = 70;       {handle cut command}
accCopy           = 71;       {handle copy command}
accPaste          = 72;       {handle paste command}
accClear          = 73;       {handle clear command}

{Chooser messages}
chooserInitMsg    = 11;       {the user selected this device package}
newSelMsg         = 12;       {the user made new device selections}
fillListMsg       = 13;       {fill the device list with choices}
getSelMsg         = 14;       {mark one or more choices as selected}
selectMsg         = 15;       {the user made a selection}
```

```
   deselectMsg       = 16;       {the user canceled a selection}
   terminateMsg      = 17;       {allows device package to clean up}
   buttonMsg         = 19;       {the user selected a button}
```

Data Types

```
TYPE  ParamBlkType = (IOParam, FileParam, VolumeParam, CntrlParam,
                      SlotDevParam, MultiDevParam);

      ParamBlockRec =
      RECORD
         qLink:         QElemPtr;   {next queue entry}
         qType:         Integer;    {queue type}
         ioTrap:        Integer;    {routine trap}
         ioCmdAddr:     Ptr;        {routine address}
         ioCompletion:  ProcPtr;    {completion routine address}
         ioResult:      OSErr;      {result code}
         ioNamePtr:     StringPtr;  {pointer to driver name}
         ioVRefNum:     Integer;    {volume reference or drive number}
      CASE ParamBlkType OF
      IOParam:
         (ioRefNum:     Integer;    {driver reference number}
         ioVersNum:     SignedByte; {not used}
         ioPermssn:     SignedByte; {read/write permission}
         ioMisc:        Ptr;        {not used}
         ioBuffer:      Ptr;        {pointer to data buffer}
         ioReqCount:    LongInt;    {requested number of bytes}
         ioActCount:    LongInt;    {actual number of bytes}
         ioPosMode:     Integer;    {positioning mode}
         ioPosOffset:   LongInt);   {positioning offset}
      CntrlParam:
         (ioCRefNum:    Integer;    {driver reference number}
         csCode:        Integer;    {type of control or status request}
         csParam:       ARRAY[0..10] OF Integer);  {control or status info}
      END;
      ParmBlkPtr = ^ParamBlockRec;

      AuxDCE =
      RECORD
         dCtlDriver:    Ptr;        {pointer or handle to driver}
         dCtlFlags:     Integer;    {flags}
         dCtlQHdr:      QHdr;       {driver I/O queue header}
         dCtlPosition:  LongInt;    {byte position}
```

```
    dCtlStorage:   Handle;     {handle to private storage}
    dCtlRefNum:    Integer;    {driver reference number}
    dCtlCurTicks:  LongInt;    {used internally}
    dCtlWindow:    GrafPtr;    {pointer to driver's window}
    dCtlDelay:     Integer;    {ticks between periodic actions}
    dCtlEMask:     Integer;    {event mask for desk accessories}
    dCtlMenu:      Integer;    {menu ID for desk accessories}
    dCtlSlot:      Byte;       {slot}
    dCtlSlotId:    Byte;       {sResource directory ID}
    dCtlDevBase:   LongInt;    {slot device base address}
    dCtlOwner:     Ptr;        {reserved; must be 0}
    dCtlExtDev:    Byte;       {external device ID}
    fillByte:      Byte;       {reserved}
  END;
  AuxDCEPtr        = ^AuxDCE;
  AuxDCEHandle     = ^AuxDCEPtr;
```

Routines

Opening and Closing Device Drivers

```
FUNCTION OpenDriver         (name: Str255; VAR refNum: Integer): OSErr;
FUNCTION PBOpen             (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBOpenSync         (paramBlock: ParmBlkPtr): OSErr;
FUNCTION OpenSlot           (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION OpenDeskAcc        (deskAccName: Str255): INTEGER;
FUNCTION CloseDriver        (refNum: Integer): OSErr;
FUNCTION PBClose            (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBCloseSync        (paramBlock: ParmBlkPtr): OSErr;
PROCEDURE CloseDeskAcc      (refNum: INTEGER);
```

Communicating With Device Drivers

```
FUNCTION FSRead             (refNum: Integer; VAR count: LongInt;
                             buffPtr: Ptr): OSErr;
FUNCTION PBRead             (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBReadSync         (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBReadAsync        (paramBlock: ParmBlkPtr): OSErr;
FUNCTION FSWrite            (refNum: Integer: VAR count: LongInt;
                             buffPtr: Ptr): OSErr;
FUNCTION PBWrite            (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBWriteSync        (paramBlock: ParmBlkPtr): OSErr;
```

```
FUNCTION PBWriteAsync       (paramBlock: ParmBlkPtr): OSErr;
```

Controlling and Monitoring Device Drivers

```
FUNCTION Control            (refNum: Integer; csCode: Integer;
                             csParamPtr: Ptr): OSErr;
FUNCTION PBControl          (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBControlSync      (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBControlAsync     (paramBlock: ParmBlkPtr): OSErr;
FUNCTION Status             (refNum: Integer; csCode: Integer;
                             csParamPtr: Ptr): OSErr;
FUNCTION PBStatus           (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBStatusSync       (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBStatusAsync      (paramBlock: ParmBlkPtr): OSErr;
FUNCTION KillIO             (refNum: Integer): OSErr;
FUNCTION PBKillIO           (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBKillIOSync       (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBKillIOAsync      (paramBlock: ParmBlkPtr): OSErr;
```

Driver Support Routines

```
FUNCTION DriverInstall      (drvrPtr: Ptr; refNum: Integer): OSErr;
FUNCTION DriverInstallReserveMem (drvrPtr: Ptr; refNum: Integer): OSErr;
FUNCTION DriverRemove       (refNum: Integer): OSErr;
FUNCTION GetDCtlEntry       (refNum: Integer): DCtlHandle;
```

Assembly-Language Summary

Data Structures

Device Manager Parameter Block Header

0	qLink	long	used internally by the Device Manager
4	qType	word	used internally by the Device Manager
6	ioTrap	word	used internally by the Device Manager
8	ioCmdAddr	long	used internally by the Device Manager
12	ioCompletion	long	completion routine
16	ioResult	word	result code
18	ioNamePtr	long	driver name
22	ioVRefNum	word	drive number

I/O Parameter Structure

24	`ioRefNum`	word	driver reference number
26	`ioVersNum`	byte	not used
27	`ioPermssn`	byte	read/write permission
28	`ioMisc`	long	not used
32	`ioBuffer`	long	pointer to data buffer
36	`ioReqCount`	long	requested number of bytes
40	`ioActCount`	long	actual number of bytes
44	`ioPosMode`	word	positioning mode
46	`ioPosOffset`	long	positioning offset

Control Parameter Structure

24	`ioCRefNum`	word	driver reference number
26	`csCode`	word	type of control or status request
28	`csParam`	22 bytes	control or status information

Trap Macros

Trap Macro Names

C and Pascal name	Trap macro name
`PBOpen`	`_Open`
`OpenSlot`	`_Open`
`PBClose`	`_Close`
`PBRead`	`_Read`
`PBWrite`	`_Write`
`PBControl`	`_Control`
`PBStatus`	`_Status`
`PBKillIO`	`_KillIO`
`DriverInstall`	`_DrvrInstall`
`DriverRemove`	`_DrvrRemove`

Routines Requiring Jump Vectors

Routine	Jump vector
`Fetch`	`JFetch`
`Stash`	`JStash`
`IODone`	`JIODone`

Result Codes

`noErr`	0	No error
`controlErr`	–17	Driver does not respond to this control request
`statusErr`	–18	Driver does not respond to this status request
`readErr`	–19	Driver does not respond to read requests
`writErr`	–20	Driver does not respond to write requests
`badUnitErr`	–21	Driver reference number does not match unit table
`unitEmptyErr`	–22	Driver reference number specifies a `nil` handle in unit table
`openErr`	–23	Requested read/write permission does not match driver's open permission
`closErr`	–24	Driver unable to complete close request
`dRemovErr`	–25	Attempt to remove an open driver
`dInstErr`	–26	Driver resource not found
`abortErr`	–27	Request aborted by `KillIO`
`notOpenErr`	–28	Driver not open
`ioErr`	–36	Data does not match in read-verify mode