This chapter describes the Power Manager, the part of the Macintosh Operating System that controls power to the internal hardware devices of battery-powered Macintosh computers (such as the Macintosh Portable, the Macintosh PowerBook computers, and the Macintosh Duo computers)

The Power Manager automatically shuts off power to internal devices to conserve power whenever the computer has not been used for a predetermined amount of time. In addition, the Power Manager allows your application or other software to

- install a procedure that is executed when power to internal devices is about to be shut off or when power has just been restored
- set a timer to wake up the computer at some time in the future
- set or disable the wakeup timer and read its current setting
- enable, disable, or delay the CPU idle feature
- read the current CPU clock speed
- control power to the internal modem and serial ports
- read the status of the internal modem
- read the state of the battery charge and the status of the battery charger

Most applications do not need to know whether they are executing on a battery-powered Macintosh computer because the transition between power states is largely invisible. As a result, most applications do not need to use Power Manager routines. You need the information in this chapter only if you are writing a program—such as a device driver—that must control power to some subsystem of a battery-powered Macintosh computer or that might be affected by the idle or sleep state. See "About the Power Manager," beginning on page 6-4, for a complete description of these power conservation states.

The Power Manager is available only in system software version 6.0.4 and later versions. You should use the `Gestalt` function to determine whether the Power Manager is available before calling it. See "Determining Whether the Power Manager Is Present," on page 6-14, for more information.

To use this chapter, you might need to be familiar with techniques for accessing information in your application's A5 world. The chapter "Introduction to Memory Management" in *Inside Macintosh: Memory* describes the A5 world and the routines you can use to manipulate the A5 register. This chapter provides complete code samples that illustrate how to access your application's A5 world in a sleep procedure. If you wish to display a dialog box from a sleep procedure, you also need to know about the Dialog Manager. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

This chapter begins with a preliminary description of the power conservation states controlled by the Power Manager and of the relationship between the power management hardware and software in portable Macintosh computers. It then discusses the power conservation states and the sleep queue in greater detail. The section "Using the Power Manager," beginning on page 6-13, describes how to use Power Manager routines to control the power conservation states and how to write and install sleep procedures.

The reference section is divided into three sections. The first section describes the data structures used by Power Manager routines. The second section, "Power Manager Routines," beginning on page 6-28, describes low-level Power Manager routines that you can use to control a variety of Power Manager functions. The third section, "Power Manager Dispatch Routines," beginning on page 6-40, describes high-level Power Manager routines that isolate you from the need to read or write directly to the Power Manager's private data structures and to parameter RAM. The Power Manager dispatch routines provide access to most of the Power Manager's internal parameters. Where a Power Manager dispatch routine duplicates the function of another Power Manager routine, the dispatch routine provides the preferred interface.

Whereas the Pascal programming language interface is used to describe the Power Manager routines in "Power Manager Routines," the C language interface is used for the newer routines described in "Power Manager Dispatch Routines." The section "Summary of the Power Manager," beginning on page 6-67, includes both Pascal and C interfaces for both sets of routines.

About the Power Manager

Battery-operated Macintosh computers (also known as ***portable Macintosh computers***) draw power from a built-in battery that can be charged from a voltage converter plugged into an electric socket. In order to prolong the battery charge and thereby increase the amount of time the computer can be operated from the battery, portable Macintosh computers contain software and hardware components that can put the computer into various power conservation states, known as the *power-saver, idle,* and *sleep states*.

The software that controls power to the internal devices of portable Macintosh computers is the ***Power Manager***. The Power Manager provides a software interface to the available power controlling hardware. On the Macintosh Portable computer, the power-management hardware is the 50753 microprocessor (known as the Power Manager integrated circuit or ***Power Manager IC***). On other portable Macintosh computers, other hardware may be used.

The Power Manager also provides some services unique to portable Macintosh computers—such as reading the current clock speed—that are not directly related to power control. The power management circuits and the microcode in the on-chip ROM of the Power Manager IC are described in the *Guide to the Macintosh Family Hardware,* second edition. The Power Manager provides routines that your program can use to enable and disable the idle state, to control power to some of the subsystems of the computer, and to ensure that your program is not adversely affected when the Power Manager puts the computer into the sleep state.

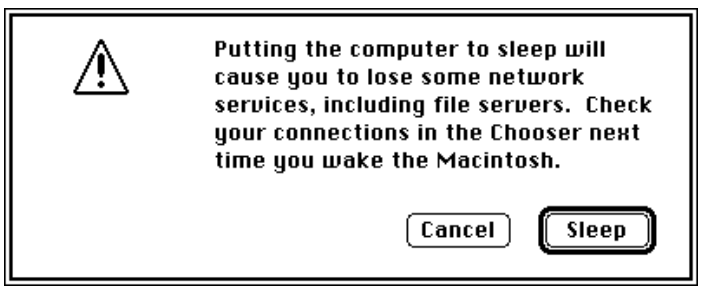
The ***power-saver state*** is a low power-consumption state of several portable Macintosh computers in which the processor slows from its normal clock speed to some slower clock speed. On the PowerBook 170 computer, for example, the CPU clock speed can be reduced from 25 MHz to 16 MHz in order to conserve power.

In the ***idle state,*** the Power Manager slows the computer even further, from its current clock speed to a 1 MHz clock speed. The Power Manager puts a portable Macintosh computer in the idle state when the system has been inactive for 15 seconds. When the computer has been inactive for an additional period of time (the user can set the length of this period), the Power Manager and the various device drivers shut off power or remove clocks from the computer's various subsystems, including the CPU, RAM, ROM, and I/O ports. This condition is known as the ***sleep state.***

No data is lost from RAM when a portable Macintosh computer is in the sleep state. Most applications can be interrupted by the idle and sleep states without any adverse effects. When the user resumes use of the computer (by pressing a key, for example), most of the applications that were running before the computer entered the sleep state are still loaded in memory and resume running as if nothing had happened. If your application or device driver cannot tolerate the sleep state, however, you can add an entry to an operating-system queue called the ***sleep queue.*** The Power Manager calls every sleep queue routine before the computer goes into the sleep state.

The user can also use the Battery desk accessory or a Finder menu item to cause a portable Macintosh computer to go into the sleep state immediately. If the user chooses Sleep from the Battery desk accessory (or from the Special menu in the Finder), the Power Manager checks to see if any network communications will be interrupted by going into the sleep state. If network communications will be affected, a built-in sleep procedure displays a dialog box (shown in Figure 6-1) giving the user the option of canceling the Sleep command.

Figure 6-1 A network driver's sleep dialog box



Note
Some portable Macintosh computers (for example, the Macintosh Portable) do not have a power switch. On these computers, if the user chooses Shut Down from Special menu in the Finder, the Power Manager puts the computer into the sleep state regardless of whether any network communication routines are running at the time. ◆

The power management circuits in portable Macintosh computers include a battery-voltage monitor, a voltage regulator and battery-charging circuit, and (on certain portable computers) the Power Manager IC. The Power Manager IC controls the clocks and power lines to the various internal components and external ports of the computer.

The microcode in the Power Manager IC implements many of the computer's power management features, such as power and clock control and the wakeup timer. A user or an application can set the ***wakeup timer*** to return the computer from the sleep state to the operating state at a specific time.

Note
The wakeup timer is not available on all portable Macintosh computers. ◆

The Power Manager firmware in the ROM of the computer provides an interface that allows your application to control some of the functions of the power control hardware. The power management hardware charges the battery, provides the voltages needed by the system, and automatically shuts down all power and clocks to the system if the battery voltage falls below a certain threshold. The automatic shutdown function helps to prevent possible damage to the battery resulting from low voltage.

At any given time, a portable Macintosh computer is in one of five power-consumption states:

- normal state
- power-saver state
- idle state
- sleep state
- shutdown state

When the computer is in its normal state, the CPU is running at its full clock speed and no measures are being taken to conserve power. The computer behaves exactly like any Macintosh computer that is not operated from a battery. Similarly, the shutdown state on a portable Macintosh computer is exactly like the shutdown state on any nonportable Macintosh computer, except that there is a very small drain on the battery to maintain the settings of the computer's parameter RAM.

The following sections provide more information about the three power conservation states (power-saver, idle, and sleep) managed by the Power Manager.

IMPORTANT
The exact implementation details—and indeed the very existence of one or more of the three power conservation states—is subject to variation across the entire line of portable Macintosh computers. In general, your application or other software should not be affected by any such variations. ▲

The Power-Saver State

The power-saver state, available on some portable Macintosh computers, is a power conservation state in which the processor slows from its normal clock speed to some slower clock speed. On the PowerBook 180 computer, for example, the user can use the PowerBook control panel to reduce the CPU clock speed from 33 MHz to 16 MHz.

There is currently no way for your application to put a portable Macintosh computer into the power-saver state or to return it to the normal (full-speed) state. Moreover, the power-saver state is not available on all portable Macintosh computers. If the operation of your application or other software component depends on the CPU clock speed, you can use the Power Manager's `GetCPUSpeed` function to determine the current speed. In general, of course, it's best to design your application so that it is unaffected by any changes in the clock speed of the CPU.

The Idle State

When a portable Macintosh computer has been inactive for some amount of time, the Power Manager causes the CPU to insert wait states into each RAM or ROM access. On the Macintosh Portable, for example, after 15 seconds of inactivity the Power Manager inserts 64 wait states, effectively changing the clock speed from 16 MHz to 1 MHz. This condition is referred to as the *idle state* or the *rest state.*

Note
The inactivity timeout interval, clock speed, and hardware implementation of the idle state are subject to variation across the entire line of portable Macintosh computers. ◆

For the purposes of determining whether to enter the idle state, inactivity is defined as the absence of any of the following:

- any execution of the `PBRead` or `PBWrite` function by the File Manager or Device Manager
- a call to the Event Manager's `PostEvent` or `OSEventAvail` function
- any access of the Apple Sound Chip (ASC) or other sound-producing hardware
- completion of an Apple Desktop Bus (ADB) transaction
- a call to the QuickDraw `SetCursor` procedure that changes the cursor
- the cursor displayed as the watch cursor

The Power Manager enters the idle state in one of two ways, depending on whether the computer supports a mode of idling called ***power cycling.*** If power cycling is available (for example, in the PowerBook 140 and later models), the CPU is turned off after two seconds of inactivity. After a short interval (on the order of one-half to three-fourths of a second), power is restored to the CPU. The Operating System then checks to see whether any relevant activity has occurred. If it has, the power cycling is stopped and the computer returns to the normal operating state. If, however, no activity has occurred, power cycling resumes with a slightly longer interval (up to several seconds). The CPU remains off for the duration of the cycling or until an interrupt occurs.

If power cycling is not available, the Power Manager uses an alternate method of entering the idle state. The Power Manager maintains an ***activity timer*** that measures the amount of time that has elapsed since the last relevant system activity. The activity timer is originally set to 15 seconds. When the timer counts down to 0, the Power Manager puts the computer into the idle state. Whenever the Power Manager detects

one of the relevant forms of activity, it resets the activity timer to 15 seconds and, if the computer is in the idle state, returns the computer to the operating state.

Neither the user nor your application can change the activity timer to use a period other than 15 seconds. However, the user can disable the activity timer through the Portable or PowerBook control panel, and your application can reset, enable, and disable the activity timer by using the `IdleUpdate`, `EnableIdle`, and `DisableIdle` routines. Your application can also use the `GetCPUSpeed` function to determine whether the computer is currently in the idle state. See "Enabling or Disabling the Idle State," beginning on page 6-15, for a further discussion of these routines.

The Sleep State

The Operating System sends a sleep command to the power management hardware when the user requests it (through the Battery desk accessory or the Finder), when the battery voltage falls below a preset level, or when the system has remained inactive for an amount of time that the user sets through the Portable or PowerBook control panel.

The Operating System uses the power management hardware to shut down power to the CPU, the ROM, and some of the control logic. Sufficient power is maintained to the RAM so that no data is lost. Before the Operating System sends the sleep command to the power management hardware, it performs the following tasks:

- It pushes the contents of all of the CPU's internal registers onto the stack.
- It calls all sleep procedures listed in the sleep queue to inform them that the system is about to be put into the sleep state. These procedures include the device drivers for the serial ports and floppy disk drives. Each device driver must call the power management hardware to stop power or clocks to the peripheral device controlled by that driver. If the device contains any internal registers, the device driver must save their contents before turning off power to the device. The sleep queue is described in the following section, "The Sleep Queue."
- It pushes onto the stack the Reset vector, the contents of the versatile interface adapter (VIA) chip, and the contents of the Apple Sound Chip (ASC) control registers.
- It saves the stack pointer in memory.

While a portable Macintosh computer is in the sleep state, the clock to the power management hardware (for example, the Power Manager IC) is off so that the hardware does no processing. On each rising edge of the 60 Hz clock signal (from one of the computer's logic chips), a hardware circuit restores the clock signal to the power management hardware, which updates the time in the real-time clock and checks the status of the system to determine whether to return the computer to its operating state. The power management hardware checks for the existence of the following conditions:

- A key on the keyboard has been pressed.
- The wakeup timer is enabled and the time to which the wakeup timer is set equals the time in the real-time clock.
- An internal modem is installed, the user has activated the ring-detect feature, and the modem has detected a ring (that is, someone has called the modem).

Note that use of the mouse or trackball cannot be detected by the power management hardware.

If the power management hardware does not detect any of these conditions, it deactivates its own clock until the next rising edge of the 60 Hz clock signal. If the power management hardware does detect one of these conditions, it restores power to the CPU, ROM, and any other hardware that was running when the computer entered the sleep state. Then the Power Manager's wakeup procedure reverses the procedure that put the computer into the sleep state, including calling each routine listed in the sleep queue to allow it to restore power to any subsystems it controls.

The Sleep Queue

The Power Manager maintains an operating-system queue called the *sleep queue.* The sleep queue contains pointers to all of the routines—called ***sleep procedures*** —that the Power Manager must call before it puts the computer into the sleep state or returns it to the operating state. Each device driver, for example, can place in the sleep queue a pointer to a routine that controls power to the subsystem that the driver controls. When the Power Manager is ready to put the computer into the sleep state, it calls each of the sleep procedures listed in the sleep queue. Each procedure performs whatever tasks are necessary to prepare for the sleep state, including calling Power Manager routines, and then returns control to the Power Manager. Similarly, the Power Manager calls each sleep procedure when it is returning the computer to the operating state.

If you are writing a device driver or if you want your program to be informed before the computer enters the sleep state, you can place an entry for your sleep procedure in the sleep queue. If you do place an entry in the sleep queue, remember to remove it before your device driver or application terminates. You use the `SleepQInstall` and `SleepQRemove` procedures to install and remove sleep queue entries, as described in "Installing a Sleep Procedure," beginning on page 6-18.

Your sleep procedure can be called at any of four different times, namely

- when the Power Manager wants to know whether it may put the computer into the sleep state (a sleep request)
- when the Power Manager is about to put the computer into the sleep state (a sleep demand)
- when the Power Manager has just returned the computer to the normal operating state (a wakeup demand)
- when the Power Manager has decided not to put the computer into the sleep state but has already issued a sleep request (a sleep-request revocation)

Your sleep procedure will need to respond differently, depending on the reason it is being called. The following four sections describe these cases.

Sleep Requests

The Power Manager sends your sleep procedure a ***sleep request*** when it would like to put the computer into the sleep state. Your sleep procedure then has the option of denying the sleep request. If any procedure in the sleep queue denies the sleep request, the Power Manager sends a sleep-request revocation to each routine that it has already called with a sleep request, and the computer does not enter the sleep state. If, on the other hand, every sleep procedure in the sleep queue accepts the sleep request, then the Power Manager sends a sleep demand to each sleep procedure in the sleep queue. After every sleep procedure has processed the sleep demand, the Power Manager puts the computer into the sleep state.

Before sending a sleep request to any of the sleep procedures in the sleep queue, the Power Manager calls a built-in sleep procedure that checks the status of certain network services, as summarized in Table 6-1. Only if all of the network services permit sleep does the Power Manager continue by sending sleep requests to the routines in the sleep queue. The network services in Table 6-1 are described in *Inside Macintosh: Networking*.

The Power Manager issues a sleep request when a sleep timeout occurs (that is, when the period of inactivity set by the user in the Portable or PowerBook control panel has expired).

Table 6-1 Response of network services to sleep requests and sleep demands

Network service in use	Response to sleep request	Response to conditional sleep demand	Response to unconditional sleep demand
.MPP low-level protocol (DDP, NBP, RTMP, AEP)	Close driver if computer is on battery; else deny request	Close driver if user gives okay; else deny request	Close driver
.XPP extended protocol (ASP, AFP); no server volume mounted	Close driver if computer is on battery; else deny request	Close driver if user gives okay; else deny request	Close driver
.XPP; server volume mounted	Deny request	Close server sessions and close driver if user gives okay; else deny request	Close server sessions and close driver
An application is currently using AppleTalk	Deny request	Close server sessions and close driver if user gives okay; else deny request	Close server sessions and close driver

Sleep Demands

The Power Manager sends your sleep procedure a ***sleep demand*** when it is about to put the portable Macintosh computer into the sleep state. When a procedure in the sleep

queue receives a sleep demand, it must prepare for the sleep state as quickly as possible and return control to the Power Manager.

From the point of view of the Power Manager, there are two types of sleep demands—conditional and unconditional. The Power Manager might cancel a conditional sleep demand if certain network services are in use; an unconditional sleep demand cannot be canceled. When your sleep procedure receives a sleep demand, however, your procedure has no way to determine whether it originated as a conditional sleep demand or an unconditional sleep demand. Your device driver or application must prepare for the sleep state and return control promptly to the Power Manager when it receives a sleep demand.

The Power Manager processes a conditional sleep demand when the user chooses Sleep from the Battery desk accessory or from the Special menu in the Finder. When the Power Manager processes a conditional sleep demand, it first sends a sleep request to the network driver's sleep procedure (see Table 6-1). Whenever one of the network services is in use, the sleep procedure displays a dialog box requesting the user's permission to put the computer into the sleep state. The wording of the message in the dialog box depends on the nature of the network service in use. For example, if an .XPP driver protocol is in use, has opened a server, and has mounted a volume, then the message warns the user that the volume will be closed when the computer is put into the sleep state.

If the user denies permission to close the driver, the Power Manager does not send sleep demands to the routines in the sleep queue. If the user does give permission to close the driver, the Power Manager sends a sleep demand to the network driver's sleep procedure and then to every other sleep procedure in the sleep queue.

The Power Manager issues an unconditional sleep demand when the battery voltage falls below a preset level or when the user chooses Shut Down from the Special menu in the Finder. In this case, the Power Manager sends a sleep demand to the network driver's sleep procedure, which closes all network drivers. Then the Power Manager sends a sleep demand to every other sleep procedure in the sleep queue. As always for a sleep demand, each sleep procedure must prepare for the sleep state and return control to the Power Manager as quickly as possible. In this case, the Power Manager does not display any warnings or dialog boxes; neither the network services, the user, nor any application can deny the sleep demand.

Wakeup Demands

After restoring full power to the CPU, RAM, and ROM, the Power Manager's wakeup procedure calls each sleep procedure in the sleep queue with a wakeup demand. A ***wakeup demand*** informs your sleep procedure that it must reverse whatever steps it followed when it prepared for the sleep state. For example, a database application might reestablish communications with a remote database.

Sleep-Request Revocations

If any sleep procedure in the sleep queue denies a sleep request, the Power Manager sends a ***sleep-request revocation*** to every sleep procedure that it has already called with a sleep request. Your sleep procedure must reverse whatever steps it followed when it prepared to receive a sleep demand. A communications application that prevents users from opening new sessions while it is waiting to receive a sleep demand, for example, might once again allow users to open new sessions.

Power Manager Dispatch

Software that reads and writes directly to the Power Manager's private data structures and parameter RAM must be updated any time Apple Computer, Inc. makes a change to the internal operation of the Power Manager. The Power Manager for some versions of the Macintosh Operating System includes routines—referred to as the *Power Manager dispatch routines*—that eliminate the need for applications to deal directly with the Power Manager's data structures. These routines provide access to most of the Power Manager's internal parameters. The interface is extensible, and may grow over time to accommodate new kinds of functions.

You can use the routines described in "Power Manager Dispatch Routines," beginning on page 6-40, to isolate your application from future changes to the internal operation of the Power Manager software.

IMPORTANT

Apple Computer, Inc. reserves the right to change the internal operation of the Power Manager software. Applications should not depend on the Power Manager's internal data structures or parameter RAM. ▲

You should not depend on the Power Manager's internal data structures staying the same in future versions of system software. In particular, do not assume that

- timeout values such as the hard disk spindown time reside at the same locations in parameter RAM
- the power cycling process works the same way or uses the same parameters
- direct commands to the Power Manager microcontroller are supported in all models

Note

Whereas the Pascal programming language interface is used to describe the Power Manager routines in "Power Manager Routines," beginning on page 6-28, the C language interface is used for the newer routines described in "Power Manager Dispatch Routines," beginning on page 6-40. The section "Summary of the Power Manager," beginning on page 6-67, includes both Pascal and C interfaces for both sets of routines. ◆

Using the Power Manager

You can use the Power Manager to install a sleep procedure that is executed when power to internal devices is about to be shut off or after power has just been restored. Most applications or other software components that are sensitive to the power-consumption state of the computer can use sleep procedures to perform any necessary processing at those times. See "Writing a Sleep Procedure," beginning on page 6-20, and "Installing a Sleep Procedure," beginning on page 6-18, for complete details on how to write and install sleep procedures.

The Power Manager provides routines that you can use to monitor the state of the battery charge and the status of the battery charger. See "Monitoring the Battery and Battery Charger," beginning on page 6-26, for details. In all likelihood, only utility programs will need to use these routines.

If you are writing an application that is sensitive to the clock speed of the computer, you can use the Power Manager to disable the CPU idle state when necessary.

IMPORTANT
Do not disable the idle state except when executing a routine that must run at full speed. Disabling the idle state shortens the amount of time the user can operate the computer from a battery. ▲

If you want to ensure that a portable Macintosh computer is in the operating state at a particular time in the future, you can use the `SetWUTime` function to set the wakeup timer. You can use the wakeup timer in conjunction with the Time Manager, for example, when you want to use the computer to perform tasks that must be done at a specific time, like printing a large file in the middle of the night.

If you are writing a device driver for a portable Macintosh computer, you might need to use the Power Manager to control power to the subsystem that your driver controls. See "Switching Serial Power On and Off," on page 6-25, for a discussion of power control for the serial communications subsystem. For power control for other devices, consult Apple Developer Technical Support. The Power Manager cannot control power to external peripheral devices such as hard disks and CD-ROM drives because such devices have their own power supplies.

IMPORTANT
Because the Power Manager saves the contents of all of the CPU registers, including the stack pointer, before putting the computer into the sleep state, and because the contents of RAM are preserved while the computer is in the sleep state, most applications are not adversely affected by the sleep state. Because a portable Macintosh computer does not enter the idle state when almost any sort of activity is going on (or even when the watch cursor is being displayed), few programs are adversely affected by the idle state. Therefore, it is likely that your application will not have to make calls to the Power Manager. ▲

Determining Whether the Power Manager Is Present

You can use the `Gestalt` function with the `gestaltPowerMgrAttr` selector to determine whether the Power Manager is available on a particular computer and whether certain other devices in the computer can be put into the idle or sleep state. The `Gestalt` function returns in the `response` parameter a 32-bit value that may have some or all of the following bits set:

```
CONST
   gestaltPMgrExists         = 0;  {Power Manager is present}
   gestaltPMgrCPUIdle        = 1;  {CPU can idle}
   gestaltPMgrSCC            = 2;  {can stop SCC clock}
   gestaltPMgrSound          = 3;  {can shut off sound circuits}
   gestaltPMgrDispatchExists = 4;  {dispatch routines are present}
```

If the `gestaltPMgrExists` bit is set, the Power Manager is present. If the `gestaltPMgrCPUIdle` bit is set, the CPU is capable of going into a state of low power consumption. If the `gestaltPMgrSCC` bit is set, it is possible to stop the SCC clock, thus effectively turning off the serial ports. If the `gestaltPMgrSound` bit is set, it is possible to turn off power to the sound circuits. If the `gestaltPMgrDispatchExists` bit is set, the Power Manager dispatch routines are available; see the next section for more information.

Note
For complete details on using the `Gestalt` function, see the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*. ◆

Determining Whether the Power Manager Dispatch Routines are Present

You can use the `Gestalt` function with the `gestaltPowerMgrAttr` selector to determine whether the Power Manager dispatch routines are available on a particular computer. If the `gestaltPMgrDispatchExists` bit is set in the `response` parameter, the Power Manager dispatch routines are available.

Because more routines may be added in the future, the `PMSelectorCount` function (described on page 6-41) returns the number of dispatch routines that are implemented. The sample code in Listing 6-1 shows how you can use the `Gestalt` function to determine whether the Power Manager dispatch routines are present, and then use the `PMSelectorCount` function to find out which routines are supported. In this case, the sample code tests for the existence of the hard disk spindown routine (selector $07).

Listing 6-1 Determining which Power Manager dispatch routines exist

```
long pmgrAttributes;
Boolean routinesExist;

routinesExist = false;
if (! Gestalt(gestaltPowerMgrAttr, &pmgrAttributes))
if (pmgrAttributes & (1<<gestaltPMgrDispatchExists))
if (PMSelectorCount() >= 7) /* do the first 8 routines exist? */
   routinesExist = true;
```

▲ **WARNING**
If you call a routine that does not exist, the call to the public Power Manager trap (if the trap exists) will return an error code, which your program could misinterpret as data. ▲

Enabling or Disabling the Idle State

You can reset the activity timer to 15 seconds, disable or enable the idle state, and read the current CPU clock speed by using Power Manager routines.

IMPORTANT
Keep in mind that it is almost always better to design your code so that it is not affected by the idle state. If you do so, the computer can conserve power whenever possible. Note also that disabling the idle state does not disable the sleep state. To prevent your program from being adversely affected by the sleep state, you need to place a sleep procedure in the sleep queue, as described in "Installing a Sleep Procedure," beginning on page 6-18. ▲

To reset the activity timer to count down another 15 seconds before the Power Manager puts the computer into the idle state, use the `IdleUpdate` function. The `IdleUpdate` function takes no parameters and returns the value in the `Ticks` global variable at the time the function was called.

If you want to disable the idle state—that is, prevent the computer from entering the idle state—for more than 15 seconds, use the `DisableIdle` procedure. If your application cannot tolerate the idle state at all, you can call the `DisableIdle` procedure when your application starts up and then call the `EnableIdle` procedure when your application terminates.

The `EnableIdle` procedure cancels the last call to the `DisableIdle` procedure. Note that canceling the last call to the `DisableIdle` procedure is not always the same thing as enabling the idle state. For example, if the user has used the Portable control panel to disable the idle state, then a call to the `EnableIdle` procedure does not enable the idle state. Similarly, if your routine called the `DisableIdle` procedure more than once or if another routine has called the `DisableIdle` procedure, then a call to the `EnableIdle` procedure cancels only the last call to the `DisableIdle` procedure; it does not enable the idle state.

The Power Manager does not actually reenable the idle state until every call to the `DisableIdle` procedure has been matched by a call to the `EnableIdle` procedure, and then only if the user has not disabled the idle state through the Portable (or PowerBook) control panel. For this reason, you must be very careful to match each call to the `DisableIdle` procedure with a single call to the `EnableIdle` procedure. Be careful to avoid making extra calls to the `EnableIdle` procedure so that you do not inadvertently reenable the idle state while another application needs it to remain disabled.

Calls to the `EnableIdle` procedure are not cumulative; that is, after you make several calls to the `EnableIdle` procedure, a single call to the `DisableIdle` procedure still disables the idle state. Disabling the idle state always takes precedence over enabling the idle state. A call to the `DisableIdle` procedure disables the idle state no matter how many times the `EnableIdle` procedure has been called and whether or not the user has enabled the idle state through the Portable or PowerBook control panel.

The following examples should help to clarify the use of `EnableIdle` and `DisableIdle`:

- If an application calls the `EnableIdle` routine but the user disables or has disabled the idle state, the idle state is disabled.
- If an application calls the `DisableIdle` routine and the user enables or has enabled the idle state, the idle state is disabled.
- If an application calls the `DisableIdle` routine twice in a row and then calls the `EnableIdle` routine once, the idle state is disabled.
- If an application calls the `EnableIdle` routine twice in a row and then calls the `DisableIdle` routine once, the idle state is disabled.
- If the idle state is initially enabled and if an application calls the `DisableIdle` routine twice in a row and then calls the `EnableIdle` routine twice, the Power Manager first disables and then reenables the idle state.

To determine whether a portable Macintosh computer is currently in the idle state, read the current clock speed with the `GetCPUSpeed` function. If the value returned by the `GetCPUSpeed` function is 1, the computer is in the idle state.

Setting, Disabling, and Reading the Wakeup Timer

When a portable Macintosh computer is in the sleep state, the power management hardware updates the real-time clock and compares it to the wakeup timer once each second. When the real-time clock and the wakeup timer have the same setting, the power management circuits return the computer to the operating state. The Power Manager provides functions that you can use to set the wakeup timer, disable the wakeup timer, and read the wakeup timer's current setting.

IMPORTANT
In some portable Macintosh computers, the power management hardware does not receive this periodic "tickle." As a result, the wakeup timer cannot be used on those machines. To determine whether a particular portable Macintosh computer supports the use of the wakeup timer, call the `GetWUTime` function. An error is returned if the timer is not available. ▲

Use the `SetWUTime` function to set the wakeup timer. You pass one parameter to the `SetWUTime` function, an unsigned long word specifying the number of seconds since midnight, January 1, 1904. Setting the wakeup timer automatically enables it. Listing 6-2 illustrates how to call the `SetWUTime` function.

Listing 6-2 Setting the wakeup timer

```
FUNCTION WakeMeUp (when: LongInt): OSErr;
VAR
   myTime:  LongInt;
BEGIN
   GetDateTime(myTime);                    {get the current time}
   myTime := myTime + when;                {add desired delay}
   WakeMeUp := SetWUTime(LongInt(@myTime));
END;
```

The `when` parameter passed to the `WakeMeUp` function defined in Listing 6-2 specifies how long from the current time the wakeup timer should go off. The `WakeMeUp` function determines the current time by calling `GetDateTime` and then passes the appropriate value to `SetWUTime`. Note that the parameter passed to `SetWUTime` is the *address* of the desired wakeup time, not the wakeup time itself.

To disable the wakeup timer, you can set the wakeup timer to any time earlier than the current setting of the real-time clock (that is, to some time in the past), or you can use the `DisableWUTime` function. To reenable the wakeup timer, you must use the `SetWUTime` function to set the timer to a new time in the future.

To get the current setting of the wakeup timer, use the `GetWUTime` function. This function returns two parameters: the time to which the wakeup timer is set (in seconds since midnight, January 1, 1904) and a flag indicating whether the wakeup timer is enabled.

If the computer is already in the operating state when the real-time clock reaches the setting in the wakeup timer, nothing happens.

Note
The power management circuits do not return the computer to the operating state while battery voltage is low, even if the wakeup timer and real-time clock settings coincide. ◆

Installing a Sleep Procedure

If you want your program to be notified before the Power Manager puts a portable Macintosh computer into the sleep state or returns it to the operating state, you can put an entry in the sleep queue. If you do place an entry in the sleep queue, remember to remove it before your device driver or application terminates.

The sleep queue is a standard operating-system queue, as described in *Inside Macintosh: Operating System Utilities*. The `SleepQRec` data type defines a ***sleep queue record*** as follows:

```
TYPE SleepQRec =                    {sleep queue record}
   RECORD
      sleepQLink:    SleepQRecPtr;  {next queue element}
      sleepQType:    Integer;       {queue type = 16}
      sleepQProc:    ProcPtr;       {pointer to sleep procedure}
      sleepQFlags:   Integer;       {reserved}
   END;
```

To add an entry to the sleep queue, fill in the `sleepQType` and `sleepQProc` fields of a sleep queue record. The `sleepQLink` and `sleepQFlags` fields are maintained privately by the Power Manager; your application should not modify these fields, except to initialize them before it calls the `SleepQInstall` procedure. `SleepQInstall` takes one parameter, a pointer to your sleep queue record. Listing 6-3 shows how to add an entry to the sleep queue.

Listing 6-3 Adding an entry to the sleep queue

```
VAR
   gSleepRec:     SleepQRec;        {a sleep queue record}

PROCEDURE MyInstallSleepProcedure;
BEGIN
   {Set up the record before installing it into the sleep queue.}
   WITH gSleepRec DO
   BEGIN
      sleepQLink := NIL;            {initialize reserved field}
      sleepQType := slpQType;       {set sleep queue type}
      sleepQProc := @MySleepProc;   {set address of sleep proc}
      sleepQFlags := 0;             {initialize reserved field}
   END;
   SleepQInstall(@gSleepRec);       {install the record}
END;
```

To remove your routine from the sleep queue, use the `SleepQRemove` procedure. This procedure also takes as its one parameter a pointer to your sleep queue record.

Using Application Global Variables in Sleep Procedures

When a sleep procedure installed by an application is called, the A5 world of that application might not be valid. That is to say, the A5 register might not point to the boundary between the application's global variables and its application parameters. When this happens, any attempt by the sleep procedure to read the application's global variables or to access any other information in the application's A5 world is likely to return erroneous information.

As a result, if you use an application to install a sleep procedure and your sleep procedure accesses any information in your application's A5 world, you'll need to make sure that, at the time you access that information, the A5 register points to your application's global variables. Your sleep procedure must also restore the A5 register to its previous value before exiting. This saving and restoring of the A5 register is necessary whenever your sleep procedure uses any information in your application's A5 world, such as your application global variables or any of your application's QuickDraw global variables.

Note
The techniques described in this section are relevant only to sleep procedures installed by applications. Sleep procedures installed from other kinds of code (for example, from system extensions) do not need to worry about saving and restoring the A5 register. ◆

It's easy enough to use the `SetA5` function to read the value of the A5 register when your sleep procedure begins executing and to restore the register immediately before your procedure exits. (See Listing 6-6 on page 6-21.) It's a bit harder to pass your application's A5 value to the sleep procedure. A standard way to do this in a high-level language like Pascal is to define a new data structure that contains both a sleep queue record and room for the A5 value. For example, you can define a structure of type `SleepInfoRec`, as follows:

```
TYPE SleepInfoRec =                 {sleep information record}
   RECORD
      mySleepQRec:   SleepQRec;     {a sleep queue record}
      mySlpRefCon:   LongInt;       {address of app's A5 world}
   END;
   SleepInfoRecPtr = ^SleepInfoRec;
```

Then, you simply need to call the `SetCurrentA5` function at a time that your application is the current application and pass the result of that function to your sleep procedure (via the `mySlpRefCon` field of the sleep information record). Listing 6-4 shows how to do this.

Listing 6-4 Installing a sleep procedure that uses application global variables

```
VAR
   gSleepInfoRec:    SleepInfoRec;  {a sleep information record}

PROCEDURE MyInstallSleepProc;
BEGIN
   {Set up the record before installing it into the sleep queue.}
   WITH gSleepInfoRec.mySleepQRec DO
   BEGIN
      sleepQLink := NIL;            {initialize reserved field}
      sleepQType := slpQType;       {set sleep queue type}
      sleepQProc := @MySleepProc;   {set address of sleep proc}
      sleepQFlags := 0;             {initialize reserved field}
   END;

   {Install app's A5 value into expanded sleep record.}
   gSleepInfoRec.mySlpRefCon := SetCurrentA5;

   SleepQInstall(@gSleepInfoRec));  {install the record}
END;
```

The Power Manager puts the address you pass to `SleepQInstall` into register A0 when your sleep procedure is called. Thus, the sleep procedure simply needs to retrieve the `SleepInfoRec` record and extract the appropriate value of the application's A5 world. See the next section, "Writing a Sleep Procedure," for a sample sleep procedure that does this.

Note
For more information about your application's A5 world and routines you can use to manipulate the A5 register, see the chapter "Introduction to Memory Management" in *Inside Macintosh: Memory.* ◆

Writing a Sleep Procedure

After you've added an entry to the sleep queue, the Power Manager calls your sleep procedure when the Power Manager issues a sleep request, a sleep demand, a wakeup demand, or a sleep-request revocation. Whenever the Power Manager calls your routine, the A0 register contains a pointer to your sleep queue record and the D0 register contains

a ***sleep procedure selector code*** indicating the reason your routine is being called. One of four selector codes will be in the D0 register:

```
CONST
   sleepRequest      = 1;     {sleep request}
   sleepDemand       = 2;     {sleep demand}
   sleepWakeUp       = 3;     {wakeup demand}
   sleepRevoke       = 4;     {sleep-request revocation}
```

When your routine receives a sleep request, it must either allow or deny the request and place its response in the D0 register. To allow the sleep request, clear the D0 register to 0 before returning control to the Power Manager. To deny the sleep request, return a nonzero value in the D0 register. (Note that you cannot deny a sleep demand.) Listing 6-5 defines two assembly-language glue routines that you can use to accept or deny the request from a high-level language.

Listing 6-5 Accepting and denying a sleep request

```
PROCEDURE MyAllowSleepRequest;
INLINE
   $7000;      {MOVEQ #0, D0}

PROCEDURE MyDenySleepRequest;
INLINE
   $7001;      {MOVEQ #1, D0}
```

If your routine or any other routine in the sleep queue denies the sleep request, the Power Manager sends a sleep-request revocation to each routine that it has already called with a sleep request. If none of the routines denies the sleep request, the Power Manager sends a sleep demand to each routine in the sleep queue. Because your routine will be called a second time in any case, it is not necessary to prepare for sleep in response to a sleep request; your routine need only allow or deny the sleep request by returning a result in the D0 register. Listing 6-6 shows a sample sleep procedure.

Listing 6-6 A sleep procedure

```
PROCEDURE MySleepProc;
VAR
   mySleepInfoPtr:   SleepInfoRecPtr;
   mySleepCommand:   LongInt;
   myOldA5:          LongInt;          {A5 upon entry to procedure}
   myCurA5:          LongInt;
```

```
BEGIN
   mySleepInfoPtr := MyGetSleepInfoPtr; {get the address of the sleep record}
   mySleepCommand := MyGetSleepCommand; {get the task we are to perform}

   {Set A5 register to app's A5 value, and save the original A5 value.}
   myOldA5 := SetA5(mySleepInfoPtr^.mySlpRefCon);

   CASE mySleepCommand OF              {do the right thing}
      sleepRequest:
         MySleepRequest;
      sleepDemand:
         MySleepDemand;
      sleepWakeUp:
         MyWakeupDemand;
      sleepRevoke:
         MySleepRevoke;
      OTHERWISE
         ;
   END; {CASE}

   myOldA5 := SetA5(myOldA5);          {restore original A5}
END;
```

The `MySleepProc` sleep procedure defined in Listing 6-6 retrieves the address of the sleep queue record contained in register A0 and the selector code contained in register D0. Then it calls the appropriate application-defined routine to handle the selector code. `MySleepProc` uses two assembly-language glue routines, defined in Listing 6-7, to get those values from the appropriate registers.

Listing 6-7 Retrieving the sleep queue record and the selector code

```
{Retrieve the address of our sleep info record from A0.}
FUNCTION MyGetSleepInfoPtr: SleepInfoRecPtr;
INLINE
   $2E88;      {MOVE.L A0, (A7)}

{Retrieve the command code for the sleep procedure from D0.}
FUNCTION MyGetSleepCommand: LongInt;
INLINE
   $2E80;      {MOVE.L D0, (A7)}
```

When your sleep procedure receives a sleep demand, it must prepare for the sleep state and return control to the Power Manager as quickly as possible. Because sleep demands are never sent by an interrupt handler, your sleep procedure can perform whatever tasks

are necessary to prepare for sleep, including making calls to the Memory Manager. You can, for example, display an alert box to inform the user of potential problems, or you can even display a dialog box that requires the user to specify the action to be performed. However, if several applications display alert or dialog boxes, the user might become confused or alarmed. More important, if the user is not present to answer the alert box or dialog box, control is never returned to the Power Manager and the computer does not go to sleep. Listing 6-8 defines a procedure that displays a dialog box whenever a sleep demand is received.

Listing 6-8 Displaying a dialog box in response to a sleep demand

```
PROCEDURE MySleepDemand;
VAR
   myItem:        Integer;       {item number for ModalDialog}
   myRect:        Rect;          {rectangle for NewDialog}
   myOrigPort:    GrafPtr;       {original graphics port}
BEGIN
   myItem := 0;
   gOrigTime := TickCount;       {initialize timer}

   IF gDialog = NIL THEN         {create a dialog window}
   BEGIN
      SetRect(myRect, 50, 50, 400, 150);
      gDialog := NewDialog(NIL, myRect, '', FALSE, dBoxProc,
                           WindowPtr(-1), FALSE, 0, gItemHandle);
   END;

   IF gDialog <> NIL THEN
   BEGIN
      GetPort(myOrigPort);       {remember current port}
      ShowWindow(gDialog);       {make dialog visible}
      SelectWindow(gDialog);
      SetPort(gDialog);
      REPEAT
         ModalDialog(@MyTimeOutFilter, myItem);
      UNTIL myItem = 1;

      HideWindow(gDialog);
      SetPort(myOrigPort);       {restore original port}
   END;
END;
```

To display a dialog box, you need to build the dialog box from within the sleep procedure itself to ensure that the newly created dialog box appears frontmost on the

screen. You can facilitate this process by passing a handle to the dialog item list to your sleep procedure. In Listing 6-8, the global variable `gItemHandle` is assumed to contain a handle to the dialog item list. You can execute the following line of code early in your application's execution to set `gItemHandle` to the correct value:

```
gItemHandle := Get1Resource('DITL', kAlertDITL);
```

▲ **WARNING**
If your sleep procedure displays an alert box or modal dialog box, the computer does not enter the sleep state until the user responds. If the computer remains in the operating state until the battery voltage drops below a preset value, the power management hardware automatically shuts off all power to the system, without preserving the state of open applications or data that has not been saved to disk. To prevent this from happening, you should automatically remove your dialog box after several minutes have elapsed. ▲

An easy way to implement this time-out feature is to pass the `ModalDialog` procedure the address of a modal dialog filter function that intercepts null events until the desired amount of time has elapsed. Listing 6-9 illustrates such a filter function.

Listing 6-9 A modal dialog filter function that times out

```
FUNCTION MyTimeOutFilter (myDialog: DialogPtr;
                          VAR myEvent: EventRecord;
                          VAR myItem: Integer): Boolean;
CONST
   kTimeOutMax = 18000;    {remove dialog box after 5 minutes}
BEGIN
   MyTimeOutFilter := FALSE;

   CASE myEvent.what OF
      nullEvent:
         BEGIN
            IF (TickCount - gOrigTime) >= kTimeOutMax THEN
            BEGIN
               myItem := 1;
               MyTimeOutFilter := TRUE;
            END;
         END;
      {handle other relevant events here}
      OTHERWISE
         ;
   END; {CASE}
END;
```

The global variable `gOrigTime` is initialized in the `MySleepDemand` procedure; the modal dialog filter function defined in Listing 6-9 simply waits until the appropriate number of ticks (sixtieths of a second) have elapsed before simulating a click on the OK button (assumed to be dialog item number 1).

When your routine receives a wakeup demand, it must prepare for the operating state and return control to the Power Manager as quickly as possible.

When your routine receives a sleep-request revocation, it must reverse any changes it made in response to the sleep request that preceded it and return control to the Power Manager.

Switching Serial Power On and Off

The serial I/O subsystem of a portable Macintosh computer includes the following components:

- the Serial Communications Controller (SCC) chip
- the serial driver chips
- the –5 volt supply
- the internal modem (if installed)

Because serial drivers always use these components in certain combinations, the Power Manager provides five serial power procedures that perform the following tasks:

- The `AOn` procedure switches on power to serial port A and switches on power to the internal modem if it is installed.
- The `AOnIgnoreModem` procedure switches on power to serial port A (the modem port) but does not switch on power to the internal modem.
- The `BOn` procedure switches on power to serial port B.
- The `AOff` procedure switches off power to serial port A and to the internal modem if it is in use.
- The `BOff` procedure switches off power to serial port B.

If no internal modem is installed, then calling any of the power-on routines switches on power to the SCC, the serial driver chips, and the –5 volt supply.

To switch power on for port B whether or not there is an internal modem installed, use the `BOn` procedure. This procedure switches on power to the SCC, the serial driver chips, and the –5 volt supply.

If the internal modem is installed, then you can use the `AOn` procedure to switch on the modem. In this case, this procedure switches on power to the SCC, the –5 volt supply, and the modem; the internal modem does not use the serial driver chips.

If the internal modem is installed but you do not want to use it (whether or not the user has used the Portable control panel to disconnect the modem), then use the `AOnIgnoreModem` procedure to switch on power to the SCC, the serial driver chips, and the –5 volt supply.

Note
You can use the Power Manager's `ModemStatus` function to determine whether an internal modem is turned on or off. For details, see the description of `ModemStatus` beginning on page 6-36. ◆

Monitoring the Battery and Battery Charger

You can use the Power Manager to monitor the status of the battery and battery charger. To do so, use the `BatteryStatus` function to determine the current voltage in the battery.

For most accurate results, you might want to average the voltage over some extended period of time (anywhere from 30 seconds to several minutes). The power load within a portable Macintosh computer varies dynamically, and the current draw of the various subsystems affects the voltage read at any one moment.

Power Manager Reference

This section describes the data structures and routines provided by the Power Manager. See "Using the Power Manager," beginning on page 6-13, for detailed instructions on using these routines.

Data Structures

This section describes the data structures used by the Power Manager. The sleep queue record is shown in Pascal. The other data structures, which are used by the functions described in "Power Manager Dispatch Routines," beginning on page 6-40, are shown in C.

Sleep Queue Record

The `SleepQInstall` and `SleepQRemove` procedures take as a parameter the address of a sleep queue record, which is defined by the `SleepQRec` data type.

```
TYPE SleepQRec =
   RECORD
      sleepQLink:    SleepQRecPtr;  {next queue element}
      sleepQType:    Integer;       {queue type = 16}
      sleepQProc:    ProcPtr;       {pointer to sleep procedure}
      sleepQFlags:   Integer;       {reserved}
   END;
   SleepQRecPtr = ^SleepQRec;
```

Field descriptions

`sleepQLink`	A pointer to the next element in the queue. This pointer is maintained internally by the Power Manager; your application should not modify this field.
`sleepQType`	The type of the queue, which must be the constant `slpQType` (16).
`sleepQProc`	A pointer to your sleep procedure. See "Sleep Procedures," on page 6-65, for details on this routine.
`sleepQFlags`	Reserved for use by Apple Computer, Inc.

Hard Disk Queue Structure

The `HardDiskQInstall` and `HardDiskQRemove` functions take as a parameter the address of a hard disk queue structure, which is defined by the `HDQueueElement` data type.

```
struct HDQueueElement {
   Ptr               hdQLink;       /* pointer to next queue element */
   short             hdQType;       /* queue type (must be HDPwrQType) */
   short             hdFlags;       /* reserved */
   HDSpindownProc    hdProc;        /* pointer to routine to call */
   long              hdUser;        /* user-defined private storage */
} HDQueueElement;
```

Wakeup Time Structure

The wakeup time structure used by the `GetWakeupTimer` and `SetWakeupTimer` functions is defined by the `WakeupTime` data type.

```
typedef struct WakeupTime {
 unsigned long wakeTime;   /* wakeup time as number of seconds since
                              midnight, January 1, 1904 */
 char wakeEnabled;         /* 1 = enable timer, 0=disable timer */
} WakeupTime;
```

Battery Information Structure

The `GetScaledBatteryInfo` function returns information about the battery in a data structure of type `BatteryInfo`.

```
typedef struct BatteryInfo {
   unsigned char     flags;           /* misc flags (see below) */
   unsigned char     warningLevel;    /* scaled warning level (0-255) */
```

```
   char              reserved;         /* reserved for internal use */
   unsigned char     batteryLevel;     /* scaled battery level (0-255) */
} BatteryInfo;
```

The values of the bits in the `flags` field are as follows:

Bit name	Bit number	Description
`batteryInstalled`	7	A battery is installed.
`batteryCharging`	6	The battery is charging.
`chargerConnected`	5	The charger is connected.

Battery Time Structure

The `GetBatteryTimes` function returns information about the time remaining on the computer's battery or batteries in a data structure of type `BatteryTimeRec`.

```
typedef struct BatteryTimeRec {
   unsigned long expectedBatteryTime;  /* estimated time remaining */
   unsigned long minimumBatteryTime;   /* minimum time remaining */
   unsigned long maximumBatteryTime;   /* maximum time remaining */
   unsigned long timeUntilCharged;     /* time until full charge */
} BatteryTimeRec;
```

Power Manager Routines

This section describes the routines provided by the Power Manager. You can use these routines to

- enable, disable, and read the idle state
- control and read the wakeup timer
- add and remove elements from the sleep queue
- control power to the serial ports
- read the status of the internal modem
- read the status of the battery and battery charger

Controlling the Idle State

The Power Manager provides routines that you can use to modify and control the idle state. See "The Idle State," on page 6-7, for a complete description of a computer's idle state and activity timer.

IdleUpdate

You can use the `IdleUpdate` function to reset the Power Manager's activity timer.

```
FUNCTION IdleUpdate: LongInt;
```

DESCRIPTION

The `IdleUpdate` function resets the activity timer. It takes no parameters and returns the value in the `Ticks` global variable at the time the function was called.

EnableIdle

You can use the `EnableIdle` procedure to enable the idle state.

```
PROCEDURE EnableIdle;
```

DESCRIPTION

The `EnableIdle` procedure cancels the effect of a call to the `DisableIdle` procedure. A call to the `EnableIdle` procedure enables the idle state only if the user has not used the Portable or PowerBook control panel to disable the idle state and if every call to the `DisableIdle` procedure has been balanced by a call to the `EnableIdle` procedure.

ASSEMBLY-LANGUAGE INFORMATION

The MPW development system provides an assembly-language macro to execute the `EnableIdle` routine. That macro calls the `_IdleState` trap. To call the `_IdleState` trap directly, you must first put a longword routine selector in the D0 register. For `EnableIdle`, the routine selector is 0.

SEE ALSO

See "Enabling or Disabling the Idle State," beginning on page 6-15, for more discussion of `EnableIdle`.

DisableIdle

You can use the `DisableIdle` procedure to disable the idle state.

```
PROCEDURE DisableIdle;
```

DESCRIPTION

The `DisableIdle` procedure disables the idle state, even if the user has used the Portable or PowerBook control panel to enable the idle state. Every call to the `DisableIdle` procedure must be balanced by a call to the `EnableIdle` procedure before the idle state is reenabled.

ASSEMBLY-LANGUAGE INFORMATION

The MPW development system provides an assembly-language macro to execute the `DisableIdle` routine. That macro calls the `_IdleState` trap. To call the `_IdleState` trap directly, you must first put a longword routine selector in the D0 register. For `DisableIdle`, the routine selector can be any value that is greater than 0.

SEE ALSO

See "Enabling or Disabling the Idle State," beginning on page 6-15, for more discussion of `DisableIdle`.

GetCPUSpeed

You can use the `GetCPUSpeed` function to read the current CPU clock speed.

```
FUNCTION GetCPUSpeed: LongInt;
```

DESCRIPTION

The `GetCPUSpeed` function returns the current effective clock speed (in megahertz) of the CPU.

ASSEMBLY-LANGUAGE INFORMATION

The MPW development system provides an assembly-language macro to execute the `GetCPUSpeed` routine. That macro calls the `_IdleState` trap. To call the `_IdleState` trap directly, you must first put a longword routine selector in the D0 register. For `GetCPUSpeed`, the routine selector can be any value that is less than 0. The CPU speed is returned as a single byte in register D0.

Controlling and Reading the Wakeup Timer

The Power Manager provides functions to set the wakeup timer, disable the wakeup timer, and read the current setting of the wakeup timer.

IMPORTANT

Some portable Macintosh computers do not support the wakeup timer. There is currently no direct way to determine whether a particular portable computer supports the wakeup timer. You can, however, inspect the result code from the `GetWUTime` function to see whether the call executed successfully. ▲

SetWUTime

You can use the `SetWUTime` function to set the wakeup timer.

```
FUNCTION SetWUTime (WUTime: LongInt): OSErr;
```

`WUTime` The time at which the wakeup timer is to wake up, specified as a number of seconds since midnight, January 1, 1904.

DESCRIPTION

The `SetWUTime` function sets and enables the wakeup timer. When a portable Macintosh computer is in the sleep state, the power management hardware updates the real-time clock and compares it to the wakeup timer once each second. When the real-time clock and the wakeup timer have the same setting, the power management hardware returns the computer to the operating state.

The `WUTime` parameter specifies the time at which the power management hardware will return the computer to the operating state. You specify the time as the number of seconds since midnight, January 1, 1904.

If the computer is not in the sleep state when the wakeup timer and the real-time clock settings coincide, nothing happens. If you set the wakeup timer to a time earlier than the current setting of the real-time clock, you effectively disable the wakeup timer.

RESULT CODES

`noErr`	0	No error

SEE ALSO

See "Setting, Disabling, and Reading the Wakeup Timer," beginning on page 6-16, for an example of calling `SetWUTime`.

You can use the `SetWakeupTimer` function (page 6-45) to explicitly enable and disable the wakeup timer.

DisableWUTime

You can use the `DisableWUTime` function to disable the wakeup timer.

```
FUNCTION DisableWUTime: OSErr;
```

DESCRIPTION

The `DisableWUTime` function disables the wakeup timer. You must set a new wakeup time to reenable the wakeup timer.

RESULT CODES

`noErr`	0	No error

GetWUTime

You can use the `GetWUTime` function to read the current setting of the wakeup timer.

```
FUNCTION GetWUTime (VAR WUTime: LongInt; VAR WUFlag: Byte): OSErr;
```

`WUTime`	On exit, the current setting of the wakeup timer, specified as the number of seconds since midnight, January 1, 1904.
`WUFlag`	On exit, a bit field encoding the state of the wakeup timer.

DESCRIPTION

The `GetWUTime` function returns the current setting of the wakeup timer and indicates whether the wakeup timer is enabled. The value returned in the `WUTime` parameter is the current setting of the wakeup timer, specified as the number of seconds since midnight, January 1, 1904. If the low-order bit (bit 0) of the `WUFlag` parameter is set to 1, the wakeup timer is enabled. The other bits in the `WUFlag` parameter are reserved.

SPECIAL CONSIDERATIONS

The `GetWUTime` function returns an error on machines that do not support the wakeup timer.

RESULT CODES

`noErr`	0	No error
`pmBusyErr`	–13001	Wakeup timer is not available on this machine

Controlling the Sleep Queue

The Power Manager allows you to install a sleep procedure that is executed whenever the machine is about to go into the sleep state or just after the machine returns from the sleep state.

SleepQInstall

You can use the `SleepQInstall` procedure to add an entry to the sleep queue.

```
PROCEDURE SleepQInstall (qRecPtr: SleepQRecPtr);
```

`qRecPtr` A pointer to a sleep queue record.

DESCRIPTION

The `SleepQInstall` procedure adds the specified sleep queue record to the sleep queue. The `qRecPtr` parameter is a pointer to a sleep queue record.

SPECIAL CONSIDERATIONS

You should make sure to remove any elements you installed in the sleep queue before your application or other software exits.

SEE ALSO

See "Sleep Queue Record," on page 6-26, for the structure of a sleep queue record. See "Sleep Procedures," beginning on page 6-65, for information about sleep procedures.

SleepQRemove

You can use the `SleepQRemove` procedure to remove an entry from the sleep queue.

```
PROCEDURE SleepQRemove (qRecPtr: SleepQRecPtr);
```

`qRecPtr` A pointer to a sleep queue record, which is described on page 6-26.

DESCRIPTION

The `SleepQRemove` procedure removes the specified sleep queue record from the sleep queue. The `qRecPtr` parameter is a pointer to the sleep queue record that you provided when you added your routine to the sleep queue.

Controlling Serial Power

The Power Manager provides five procedures that you can use to control power to the serial ports and internal modem.

Assembly-Language Note
Although MPW provides assembly-language macros to execute these routines, each of these macros calls the `_SerialPower` trap macro. To call the `_SerialPower` trap macro directly, you must first put a routine selector in the D0 register, setting the bits of the selector as follows:

Bit	Use
0	Set to 0 to use internal modem; set to 1 to ignore modem.
2	Set to 0 for port B; set to 1 for port A.
7	Set to 0 to switch on power; set to 1 to switch off power. ◆

AOn

You can use the `AOn` procedure to turn on the power to serial port A.

```
PROCEDURE AOn;
```

DESCRIPTION

The `AOn` procedure switches on power to the SCC and the –5 volt supply. If the internal modem is installed and is connected to port A, the `AOn` procedure also switches on power to the modem. If either of these conditions is not met, the `AOn` procedure switches on power to the serial driver chips.

ASSEMBLY-LANGUAGE INFORMATION

The MPW development system provides an assembly-language macro to execute the `AOn` routine. That macro calls the `_SerialPower` trap. To call the `_SerialPower` trap directly, you must first put a longword routine selector in the D0 register. For `AOn`, the routine selector is $4.

AOnIgnoreModem

You can use the `AOnIgnoreModem` procedure to turn on the power to serial port A but not to the internal modem.

```
PROCEDURE AOnIgnoreModem;
```

DESCRIPTION

The `AOnIgnoreModem` procedure switches on power to the SCC, the –5 volt supply, and the serial driver chips. This procedure does not switch on power to the internal modem, even if the user has used the Portable or PowerBook control panel to select the modem.

ASSEMBLY-LANGUAGE INFORMATION

The MPW development system provides an assembly-language macro to execute the `AOnIgnoreModem` routine. That macro calls the `_SerialPower` trap. To call the `_SerialPower` trap directly, you must first put a longword routine selector in the D0 register. For `AOnIgnoreModem`, the routine selector is $5.

BOn

You can use the `BOn` procedure to turn on the power to serial port B.

```
PROCEDURE BOn;
```

DESCRIPTION

The `BOn` procedure switches on power to the SCC, the –5 volt supply, and the serial driver chips.

ASSEMBLY-LANGUAGE INFORMATION

The MPW development system provides an assembly-language macro to execute the `BOn` routine. That macro calls the `_SerialPower` trap. To call the `_SerialPower` trap directly, you must first put a longword routine selector in the D0 register. For `BOn`, the routine selector is $0.

AOff

You can use the `AOff` procedure to turn off the power to serial port A and to the internal modem.

```
PROCEDURE AOff;
```

DESCRIPTION

The `AOff` procedure always switches off power to the SCC and the –5 volt supply if serial port B is not in use. If the internal modem is installed, connected to port A, and switched on, this procedure switches off power to the modem. If any of these conditions

are not met, it switches off power to the serial driver chips, unless they are being used by port B.

ASSEMBLY-LANGUAGE INFORMATION

The MPW development system provides an assembly-language macro to execute the `AOff` routine. That macro calls the `_SerialPower` trap. To call the `_SerialPower` trap directly, you must first put a longword routine selector in the D0 register. For `AOff`, the routine selector is $84.

BOff

You can use the `BOff` procedure to turn off the power to serial port B and to the internal modem.

```
PROCEDURE BOff;
```

DESCRIPTION

The `BOff` procedure switches off power to the SCC and the –5 volt supply if serial port A is not in use. If the internal modem is installed, connected to port B, and switched on, this procedure switches off power to the modem. Otherwise, the `BOff` procedure switches off power to the serial driver chips, unless they are being used by port A.

ASSEMBLY-LANGUAGE INFORMATION

The MPW development system provides an assembly-language macro to execute the `BOff` routine. That macro calls the `_SerialPower` trap. To call the `_SerialPower` trap directly, you must first put a longword routine selector in the D0 register. For `BOff`, the routine selector is $80.

Reading the Status of the Internal Modem

The Power Manager provides a function that allows you to determine the status of the internal modem.

ModemStatus

You can use the `ModemStatus` function to get information about the state of the internal modem.

```
FUNCTION ModemStatus (VAR Status: Byte): OSErr;
```

`Status` On exit, a byte value whose bits encode information about the current state of the internal modem. See the description below for the meaning of each bit.

DESCRIPTION

The `ModemStatus` function returns information about the internal modem in a portable Macintosh computer. Bits 0 and 2 through 5 of the `Status` parameter encode information about the state of the internal modem. (Currently, bits 6 and 7 are reserved; in addition, bit 1 is reserved and is always set.) The Power Manager recognizes the following constants for specifying bits in the `Status` parameter.

```
CONST
   modemOnBit           = 0;  {1 if modem is on}
   ringWakeUpBit        = 2;  {1 if ring wakeup is enabled}
   modemInstalledBit    = 3;  {1 if internal modem is installed}
   ringDetectBit        = 4;  {1 if incoming call is detected}
   modemOnHookBit       = 5;  {1 if modem is off hook}
```

Constant descriptions

`modemOnBit` The modem's power is on or off. If this bit is set, the modem is switched on. You can use the serial power control functions to control power to the modem. See "Switching Serial Power On and Off," beginning on page 6-25, for information about these functions.

`ringWakeUpBit` The state of the ring-wakeup feature. If this bit is set, the ring-wakeup feature is enabled.

`modemInstalledBit` The modem is or is not installed. If this bit is set, an internal modem is installed.

`ringDetectBit` The ring-detect state. If this bit is set, the modem has detected an incoming call.

`modemOnHookBit` The modem is on or off hook. If this bit is set, the modem is off hook. The modem indicates that it is off hook whenever it is busy sending or receiving data or processing commands. The modem cannot receive an incoming call when it is off hook.

The Power Manager also defines these bit masks:

```
CONST
   modemOnMask          = $1;    {modem on}
   ringWakeUpMask       = $4;    {ring wakeup enabled}
   modemInstalledMask   = $8;    {internal modem installed}
   ringDetectMask       = $10;   {incoming call detected}
   modemOnHookMask      = $20;   {modem off hook}
```

The user can use the Portable or PowerBook control panel to enable or disable the ring-wakeup feature. When the ring-wakeup feature is enabled and the computer is in

the sleep state, the Power Manager returns the computer to the operating state when the modem receives an incoming call.

RESULT CODES

`noErr`	0	No error

Reading the Status of the Battery and the Battery Charger

The Power Manager monitors the voltage level of the internal battery and warns the user when the voltage drops below a threshold value stored in parameter RAM. If the voltage continues to drop and falls below another, lower value stored in parameter RAM, the Power Manager puts the computer into the sleep state. The Power Manager provides a function that allows you to read the state of charge of the battery and the status of the battery charger.

BatteryStatus

You can use the `BatteryStatus` function to get information about the state of the internal battery.

```
FUNCTION BatteryStatus (VAR Status: Byte; VAR Power: Byte): OSErr;
```

`Status`	On exit, a byte value whose bits encode information about the current state of the battery charger. See the description below for the meaning of each bit.
`Power`	On exit, a byte whose value indicates the current level of the battery voltage. See the description below for a method of calculating the voltage from this value.

DESCRIPTION

The `BatteryStatus` function returns the status of the battery charger (in the `Status` parameter) and the voltage level of the battery (in the `Power` parameter).

Bits 0 through 5 of the `Status` parameter encode information about the state of the battery charger. (Currently, bits 6 and 7 are reserved.) The Power Manager recognizes the following constants for specifying bits in the `Status` parameter.

```
CONST
   chargerConnBit          = 0;     {1 if charger is connected}
   hiChargeBit             = 1;     {1 if charging at hicharge rate}
   chargeOverFlowBit       = 2;     {1 if hicharge counter has overflowed}
```

```
batteryDeadBit          = 3;     {always 0}
batteryLowBit           = 4;     {1 if battery is low}
connChangedBit          = 5;     {1 if charger connection has changed}
```

Constant descriptions

`chargerConnBit` The charger is or is not connected. If this bit is set, the battery charger is connected to the computer.

`hiChargeBit` The charge rate. If this bit is set, the battery is charging at the hicharge rate.

`chargeOverFlowBit` The hicharge counter overflow. If this bit is set, the hicharge counter has overflowed. When the hicharge counter has overflowed, it indicates that the charging circuit is having trouble charging the battery.

`batteryDeadBit` The dead battery indicator. This bit is always 0, because the Power Manager automatically shuts the system down when the battery voltage drops below a preset level.

`batteryLowBit` The battery warning. If this bit is set, the battery voltage has dropped below the value set in parameter RAM. The power management hardware sends an interrupt to the CPU once every second when battery voltage is low.

`connChangedBit` The charger connection has or has not changed state. If this bit is set, the charger has been recently connected or disconnected.

The Power Manager also defines these bit masks:

```
CONST
   chargerConnMask          = $1;    {charger is connected}
   hiChargeMask             = $2;    {charging at hicharge rate}
   chargeOverFlowMask       = $4;    {hicharge counter has overflowed}
   batteryDeadMask          = $8;    {battery is dead}
   batteryLowMask           = $10;   {battery is low}
   connChangedMask          = $20;   {connection has changed}
```

Due to the nature of lead-acid batteries, the battery power remaining is difficult to measure accurately. Temperature, load, and other factors can alter the measured voltage by 30 percent or more. The Power Manager takes as many of these factors into account as possible, but the voltage measurement can still be in error by up to 10 percent. The measurement is most accurate when the computer has been in the sleep state for at least 30 minutes.

When the battery charger is connected to a portable Macintosh computer with a low battery, the battery is charged at the hicharge rate (1.5 amps) until battery voltage reaches its full charge (7.2 volts on most portable Macintosh computers). The Power Manager has a counter (the ***hicharge counter***) that measures the time required to raise the battery voltage to this level.

After the full charge level is reached, the power management circuits maintain the hicharge connection until the hicharge counter counts down to 0. This ensures that the battery is fully charged. At the end of that time, the power management circuits supply the battery with just enough current to replace the voltage lost through self-discharge.

RESULT CODES

`noErr`	0	No error

SEE ALSO

For more functions for determining the status of the battery and battery charger, see "Getting Information About the Internal Batteries," beginning on page 6-54.

Power Manager Dispatch Routines

This section describes the Power Manager dispatch routines. You can use these routines to

- determine what Power Manager features are available
- set and read the sleep and wakeup timers and disable or disable the sleep timer
- set, read, enable, and disable the timer that dims the screen
- control the hard disk
- get information about the battery
- get and set the state of the internal modem
- control the processing speed of the processor and processor cycling
- get and set the SCSI ID the computer uses in SCSI disk mode

Note
The functions in this section are described using the C language interface. The section "Summary of the Power Manager," beginning on page 6-67, includes both Pascal and C interfaces. ◆

Assembly-language note:
All the functions in this section share a single trap, `_PowerMgrDispatch` ($A09E). The trap is register based: parameters are passed in register D0 and sometimes also in A0. A routine selector value passed in the low word of register D0 determines which routine is executed. ◆

Determining the Power Manager Features Available

The functions in this section return the number of Power Manager dispatch functions available and return information about the Power Manager features available.

PMSelectorCount

You can use the `PMSelectorCount` function to determine which Power Manager dispatch functions are implemented.

```
short PMSelectorCount();
```

DESCRIPTION

The `PMSelectorCount` function returns the number of routine selectors present. Any function whose selector value is greater than the returned value is not implemented.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `PMSelectorCount` is 0 ($00) in the low word of register D0. The number of selectors is returned in the low word of register D0.

PMFeatures

You can use the `PMFeatures` function to find out which features of the Power Manager are implemented.

```
unsigned long PMFeatures();
```

DESCRIPTION

The `PMFeatures` function returns a 32-bit field describing hardware and software features associated with the Power Manager on a particular machine. If a bit value is 1, that feature is supported or available; if the bit value is 0, that feature is not available. Unused bits are reserved by Apple for future expansion.

Bit name	Bit number	Description
`hasWakeupTimer`	0	The wakeup timer is supported.
`hasSharedModemPort`	1	The hardware forces exclusive access to either SCC port A or the internal modem. (If this bit is not set, port A and the internal modem can be used simultaneously by means of the Communications Toolbox.)
`hasProcessorCycling`	2	Processor cycling is supported; that is, when the computer is idle, the processor power will be cycled to reduce power use.
`mustProcessorCycle`	3	The processor cycling feature must be left on (turn it off at your own risk).

Bit name	Bit number	Description
hasReducedSpeed	4	Processor can be started up at a reduced speed in order to extend battery life.
dynamicSpeedChange	5	Processor speed can be switched dynamically between its full and reduced speed at any time, rather than only at startup time.
hasSCSIDiskMode	6	The SCSI disk mode is supported.
canGetBatteryTime	7	The computer can provide an estimate of the battery time remaining.
canWakeupOnRing	8	The computer supports waking up from the sleep state when an internal modem is installed and the modem detects a ring.
hasDimmingSupport	9	The computer has dimming support built into the ROM.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `PMFeatures` is 1 ($01) in the low word of register D0. The 32-bit field of supported features is returned in register D0.

Controlling the Sleep and Wakeup Timers

The functions in this section read and set the sleep and wakeup timers and enable or disable the automatic sleep feature.

GetSleepTimeout

You can use the `GetSleepTimeout` function to find out how long the computer will wait before going to sleep.

```
unsigned char GetSleepTimeout();
```

DESCRIPTION

The `GetSleepTimeout` function returns the amount of time that the computer will wait after the last user activity before going to sleep. The value of `GetSleepTimeout` is expressed as the number of 15-second intervals that the computer will wait before going to sleep.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetSleepTimeout` is 2 ($02) in the low word of register D0. The sleep timeout value is returned in the low word of register D0.

SetSleepTimeout

You can use the `SetSleepTimeout` function to set how long the computer will wait before going to sleep.

```
void SetSleepTimeout(unsigned char timeout);
```

`timeout` The amount of time that the computer will wait after the last user activity before going to sleep expressed as a number of 15-second intervals.

DESCRIPTION

The `SetSleepTimeout` function sets the amount of time the computer will wait after the last user activity before going to sleep. The value of `SetSleepTimeout` is expressed as the number of 15-second intervals making up the desired time. If a value of 0 is passed in, the function sets the `timeout` value to the default value (currently equivalent to 8 minutes).

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SetSleepTimeout` is 3 ($03) in the low word of register D0. The sleep timeout value to set is passed in the high word of register D0.

AutoSleepControl

You can use the `AutoSleepControl` function to turn the automatic sleep feature on and off.

```
void AutoSleepControl(Boolean enableSleep);
```

`enableSleep`
A Boolean that specifies whether to enable the automatic sleep feature. Set this parameter to `true` to enable automatic sleep.

DESCRIPTION

The `AutoSleepControl` function enables or disables the automatic sleep feature that causes the computer to go into sleep mode after a preset period of time. When `enableSleep` is set to `true`, the automatic sleep feature is enabled (this is the normal state). When `enableSleep` is set to `false`, the computer will not go into the sleep mode unless it is forced to either by some user action—for example, by the user's selecting Sleep from the Special menu of the Finder—or in a low battery situation.

SPECIAL CONSIDERATIONS

Calling `AutoSleepControl` with `enableSleep` set to `false` multiple times increments the auto sleep disable level so that it requires the same number of calls to `AutoSleepControl` with `enableSleep` set to `true` to reenable the auto sleep feature. If more than one piece of software makes this call, auto sleep may not be reenabled when you think it should be.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `AutoSleepControl` is 13 ($0D) in the low word of register D0. The Boolean value is passed in the high word of register D0.

IsAutoSlpControlDisabled

You can use the `IsAutoSlpControlDisabled` function to find out whether automatic sleep control is enabled.

```
Boolean IsAutoSlpControlDisabled();
```

DESCRIPTION

The `IsAutoSlpControlDisabled` function returns a Boolean `true` if automatic sleep control is disabled, or `false` if automatic sleep control is enabled.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `IsAutoSlpControlDisabled` is 33 ($21) in the low word of register D0. The Boolean result is passed in the low byte of register D0.

GetWakeupTimer

You can use the `GetWakeupTimer` function to find out when the computer will wake up from sleep mode.

```
void GetWakeupTimer(WakeupTime *theTime);
```

`theTime` A pointer to a `WakeupTime` structure, which specifies whether the timer is enabled or disabled and the time at which the wakeup timer is set to wake the computer.

DESCRIPTION

The `GetWakeupTimer` function returns the time when the computer will wake up from sleep mode.

If the PowerBook model doesn't support the wakeup timer, `GetWakeupTimer` returns a value of 0.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetWakeupTimer` is 22 ($16) in the low word of register D0. The pointer to `WakeupTime` is passed in register A0.

SEE ALSO

The `WakeupTime` structure is described in "Wakeup Time Structure," on page 6-27.

SetWakeupTimer

You can use the `SetWakeupTimer` function to set the time when the computer will wake up from sleep mode.

```
void SetWakeupTimer(WakeupTime *theTime);
```

`theTime` A pointer to a WakeupTime structure, which specifies whether to enable or disable the timer and the time at which the wakeup timer is to wake the computer.

DESCRIPTION

The `SetWakeupTimer` function sets the time when the computer will wake up from sleep mode and enables or disables the timer. On a PowerBook model that doesn't support the wakeup timer, `SetWakeupTimer` does nothing.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SetWakeupTimer` is 23 ($17) in the low word of register D0. The pointer to `WakeupTime` is passed in register A0.

SEE ALSO

The `WakeupTime` structure is described in "Wakeup Time Structure," on page 6-27.

Controlling the Dimming Timer

The functions in this section read and set the dimming timer and enable or disable the automatic screen-dimming feature. The dimmer acts as a screen saver, dimming the screen after a preset time of user inactivity.

GetDimmingTimeout

You can use the `GetDimmingTimeout` function to find out how long the computer will wait before dimming the screen.

```
unsigned char GetDimmingTimeout();
```

DESCRIPTION

The `GetDimmingTimeout` function returns the amount of time that the computer will wait after the last user activity before dimming the screen. The value of `GetDimmingTimeout` is expressed as the number of 15-second intervals that the computer will wait before dimming the screen.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetDimmingTimeout` is 29 ($1D) in the low word of register D0. The dimming timeout value is returned in the low word of register D0.

SetDimmingTimeout

You can use the `SetDimmingTimeout` function to set how long the computer will wait before dimming the screen.

```
void SetDimmingTimeout(unsigned char timeout);
```

`timeout` The amount of time that the computer will wait after the last user activity before dimming the screen expressed as a number of 15-second intervals. Specify 0 to cause the screen to dim immediately.

DESCRIPTION

The `SetDimmingTimeout` function sets the amount of time the computer will wait after the last user activity before dimming the screen. The value of `SetDimmingTimeout` is expressed as the number of 15-second intervals making up the desired time.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SetDimmingTimeout` is 30 ($1E) in the low word of register D0. The dimming timeout value to set is passed in the high word of register D0.

SEE ALSO

You can use the `PMFeatures` function (page 6-41) to determine whether the computer supports automatic dimming.

DimmingControl

You can use the `DimmingControl` function to turn the automatic dimming feature on and off.

```
void DimmingControl(Boolean enableDimming);
```

`enableDimming`
A Boolean that specifies whether to enable the automatic dimming feature. Set this parameter to `true` to enable automatic dimming.

DESCRIPTION

The `DimmingControl` function enables or disables the automatic dimming feature that causes the computer to dim the screen after a preset period of time. When `enableDimming` is set to `true`, the automatic dimming feature is enabled (this is the normal state). When `enableDimming` is set to `false`, the computer will not dim the screen.

SPECIAL CONSIDERATIONS

Calling `DimmingControl` with `enableDimming` set to `false` multiple times increments the auto dimming disable level so that it requires the same number of calls to `DimmingControl` with `enableDimming` set to `true` to reenable the auto dimming

feature. If more than one piece of software makes this call, auto dimming may not be reenabled when you think it should be.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `DimmingControl` is 31 ($1F) in the low word of register D0. The Boolean value is passed in the high word of register D0.

SEE ALSO

You can use the `PMFeatures` function (page 6-41) to determine whether the computer supports automatic dimming.

IsDimmingControlDisabled

You can use the `IsDimmingControlDisabled` function to find out whether automatic dimming is enabled.

```
Boolean IsDimmingControlDisabled();
```

DESCRIPTION

The `IsDimmingControlDisabled` function returns a Boolean `true` if automatic dimming is disabled, or `false` if dimming is enabled.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `IsDimmingControlDisabled` is 32 ($20) in the low word of register D0. The Boolean result is passed in the low byte of register D0.

SEE ALSO

You can use the `PMFeatures` function (page 6-41) to determine whether the computer supports automatic dimming.

Controlling the Hard Disk

The functions in this section return information about the hard disk timer and the state of the hard disk, and allow you to control the spin down of the hard disk. You can also use functions in this section to install and remove hard disk queue elements. The hard disk queue notifies your software when power to the internal hard disk is about to be turned off.

GetHardDiskTimeout

You can use the `GetHardDiskTimeout` function to find out how long the computer will wait before turning off power to the internal hard disk.

```
unsigned char GetHardDiskTimeout();
```

DESCRIPTION

The `GetHardDiskTimeout` function returns the amount of time the computer will wait after the last use of a SCSI device before turning off power to the internal hard disk. The value of `GetHardDiskTimeout` is expressed as the number of 15-second intervals the computer will wait before turning off power to the internal hard disk.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetHardDiskTimeout` is 4 ($04) in the low word of register D0. The hard disk timeout value is returned in the low word of register D0.

SetHardDiskTimeout

You can use the `SetHardDiskTimeout` function to set how long the computer will wait before turning off power to the internal hard disk.

```
void SetHardDiskTimeout(unsigned char timeout);
```

`timeout` The amount of time that the computer will wait after the last user activity before turning off the hard disk, expressed as a number of 15-second intervals.

DESCRIPTION

The `SetHardDiskTimeout` function sets how long the computer will wait after the last use of a SCSI device before turning off power to the internal hard disk. The value of `SetHardDiskTimeout` is expressed as the number of 15-second intervals the computer will wait before turning off power to the internal hard disk. If a value of 0 is passed in, the function sets the `timeout` value to the default value (currently equivalent to 4 minutes).

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SetHardDiskTimeout` is 5 ($05) in the low word of register D0. The hard disk timeout value to set is passed in the high word of register D0.

HardDiskPowered

You can use the `HardDiskPowered` function to find out whether the internal hard disk is on.

```
Boolean HardDiskPowered();
```

DESCRIPTION

The `HardDiskPowered` function returns a Boolean value indicating whether or not the internal hard disk is powered up. A value of `true` means that the hard disk is on, and a value of `false` means that the hard disk is off.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `HardDiskPowered` is 6 ($06) in the low word of register D0. The Boolean result is returned in the low word of register D0.

SpinDownHardDisk

You can use the `SpinDownHardDisk` function to force the hard disk to spin down.

```
void SpinDownHardDisk();
```

DESCRIPTION

The `SpinDownHardDisk` function immediately forces the hard disk to spin down and power off if it was previously spinning. Calling `SpinDownHardDisk` will not spin down the hard disk if spindown is disabled by calling the `SetSpindownDisable` function.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SpinDownHardDisk` is 7 ($07) in the low word of register D0.

IsSpindownDisabled

You can use the `IsSpindownDisabled` function to find out whether automatic hard disk spindown is enabled.

```
Boolean IsSpindownDisabled();
```

DESCRIPTION

The `IsSpindownDisabled` function returns a Boolean `true` if automatic hard disk spindown is disabled, or `false` if spindown is enabled.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `IsSpindownDisabled` is 8 ($08) in the low word of register D0. The Boolean result is passed in the low byte of register D0.

SetSpindownDisable

You can use the `SetSpindownDisable` function to disable hard disk spindown.

```
void SetSpindownDisable(Boolean setDisable);
```

`setDisable` A Boolean that specifies whether the spindown feature is enabled (`false`) or disabled (`true`).

DESCRIPTION

The `SetSpindownDisable` function enables or disables hard disk spindown, depending on the value of `setDisable`. If the value of `setDisable` is `true`, hard disk spindown is disabled; if the value is `false`, spindown is enabled.

Disabling hard disk spindown affects the `SpinDownHardDisk` function, as well as the normal spindown that occurs after a period of hard disk inactivity.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SetSpindownDisable` is 9 ($09) in the low word of register D0. The Boolean value to set is passed in the high word of register D0.

SEE ALSO

The `SpinDownHardDisk` function is described on page 6-50.

HardDiskQInstall

You can use the `HardDiskQInstall` function to notify your software when power to the internal hard disk is about to be turned off.

```
OSErr HardDiskQInstall(HDQueueElement *theElement);
```

`theElement` A pointer to an element for the hard disk power down queue.

DESCRIPTION

The `HardDiskQInstall` function installs an element into the hard disk power down queue to provide notification to your software when the internal hard disk is about to be powered off. For example, this feature might be used by the driver for an external battery-powered hard disk. When power to the internal hard disk is turned off, the external hard disk could be turned off as well.

When power to the internal hard disk is about to be turned off, the software calls the routine pointed to by the `hdProc` field so that it can do any special processing. The routine is passed a pointer to its queue element so that, for example, the routine can reference its variables.

Before calling `HardDiskQInstall`, the calling program must set the `hdQType` field to `HDPwrQType` or the queue element won't be added to the queue and `HardDiskQInstall` will return an error.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `HardDiskQInstall` is 10 ($0A) in the low word of register D0. The pointer to the `HDQueue` element is passed in register A0. The result code is returned in the low word of register D0.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `HDQueueElement` structure is defined in "Hard Disk Queue Structure," on page 6-27.

The application-defined hard disk spindown function is described in "Hard Disk Spindown Function," on page 6-66.

HardDiskQRemove

You can use the `HardDiskQRemove` function to discontinue notification of your software when power to the internal hard disk is about to be turned off.

```
OSErr HardDiskQRemove(HDQueueElement *theElement);
```

`theElement` A pointer to the element for the hard disk power down queue that you wish to remove.

DESCRIPTION

The `HardDiskQRemove` function removes a queue element installed by `HardDiskQInstall`. If the `hdQType` field of the queue element is not set to `HDPwrQType`, `HardDiskQRemove` simply returns an error.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `HardDiskQRemove` is 11 ($0B) in the low word of register D0. The pointer to the `HDQueue` element is passed in register A0. The result code is returned in the low word of register D0.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `HDQueueElement` structure is defined in "Hard Disk Queue Structure," on page 6-27.

The application-defined hard disk spindown function is described in "Hard Disk Spindown Function," on page 6-66.

Getting Information About the Internal Batteries

The functions in this section return information about the battery or batteries in the computer.

GetScaledBatteryInfo

You can use the `GetScaledBatteryInfo` function to find out the condition of the battery or batteries.

```
void GetScaledBatteryInfo(short whichBattery,
                          BatteryInfo *theInfo);
```

`whichBattery`
The battery for which you want information. Set this parameter to 0 to receive combined information about all the batteries in the computer.

`theInfo` A pointer to a `BatteryInfo` data structure, which returns information about the specified battery.

DESCRIPTION

The `GetScaledBatteryInfo` function provides a generic means of returning information about the battery or batteries in the system. Instead of returning a voltage value, the function returns the battery level as a fraction of the total possible voltage.

Note
Battery technologies such as nickel cadmium (NiCad) and nickel metal hydride (NiMH) have replaced sealed lead acid batteries in portable Macintosh computers. There is no single algorithm for determining the battery voltage that is correct for all portable Macintosh computers. ◆

The value of `whichBattery` determines whether `GetScaledBatteryInfo` returns information about a particular battery or about the total battery level. The value of `GetScaledBatteryInfo` should be in the range of 0 to `BatteryCount()`. If the value of `whichBattery` is 0, `GetScaledBatteryInfo` returns a summation of all the batteries, that is, the effective battery level of the whole system. If the value of `whichBattery` is out of range, or the selected battery is not installed, `GetScaledBatteryInfo` will return a result of 0 in all fields. Here is a summary of the effects of the `whichBattery` parameter:

Value of `whichBattery`	Information returned
0	Total battery level for all batteries
From 1 to `BatteryCount()`	Battery level for the selected battery
Less than 0 or greater than `BatteryCount()`	0 in all fields of `theInfo`

The `flags` character contains several bits that describe the battery and charger state. If a bit value is 1, that feature is available or is operating; if the bit value is 0, that feature is not operating. Unused bits are reserved by Apple for future expansion.

Bit name	Bit number	Description
`batteryInstalled`	7	A battery is installed.
`batteryCharging`	6	The battery is charging.
`chargerConnected`	5	The charger is connected.

The value of `warningLevel` is the battery level at which the first low battery warning message will appear. The function returns a value of 0 in some cases when it's not appropriate to return the warning level.

The value of `batteryLevel` is the current level of the battery. A value of 0 represents the voltage at which the Power Manager will force the computer into sleep mode; a value of 255 represents the highest possible voltage.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetScaledBatteryInfo` is 12 ($0C) in the low word of register D0. The `BatteryInfo` data are returned in the low word of register D0 as follows:

Bits	Contents
31–24	Flags
23–16	Warning level
15–8	Reserved
7–0	Battery level

SEE ALSO

The `BatteryInfo` data type is described in "Battery Information Structure," on page 6-27.

BatteryCount

You can use the `BatteryCount` function to find out how many batteries the computer supports.

```
short BatteryCount();
```

DESCRIPTION

The `BatteryCount` function returns the number of batteries that are supported internally by the computer. The value of `BatteryCount` returned may not be the same as the number of batteries currently installed.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `BatteryCount` is 26 ($1A) in the low word of register D0. The number of batteries supported is returned in the low word of register D0.

GetBatteryVoltage

You can use the `GetBatteryVoltage` function to find out the battery voltage.

```
Fixed GetBatteryVoltage(short whichBattery);
```

`whichBattery`
The battery for which you want a voltage reading.

DESCRIPTION

The `GetBatteryVoltage` function returns the battery voltage as a fixed-point number.

The value of `whichBattery` should be in the range 0 to `BatteryCount()`–1. If the value of `whichBattery` is out of range, or the selected battery is not installed, `GetBatteryVoltage` will return a result of 0.0 volts.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetBatteryVoltage` is 27 ($1B) in the low word of register D0. The battery number is passed in the high word of register D0. The 32-bit value of the battery voltage is returned in register D0.

GetBatteryTimes

You can use the `GetBatteryTimes` function to find out about how much battery time remains.

```
void GetBatteryTimes (short whichBattery,
                      BatteryTimeRec *theTimes);
```

`whichBattery`
The battery for which you want to know the time remaining. Specify 0 to get combined information about all the batteries.

theTimes A pointer to a battery time structure, which contains information about the time remaining for the batteries. The `BatteryTimeRec` data type is described on page 6-28.

DESCRIPTION

The `GetBatteryTimes` function returns information about the time remaining on the computer's battery or batteries. The time values are in seconds. The value of `theTimes.expectedBatteryTime` is the estimated time remaining based on current use patterns. The values of `theTimes.minimumBatteryTime` and `theTimes.maximumBatteryTime` are worst-case and best-case estimates, respectively. The value of `theTimes.timeUntilCharged` is the time that remains until the battery or batteries are fully charged.

The value of `whichBattery` determines whether `GetBatteryTimes` returns the time information about a particular battery or the total time for all batteries. The value of `GetScaledBatteryInfo` should be in the range of 0 to `BatteryCount()`. If the value of `whichBattery` is 0, `GetBatteryTimes` returns a total time for all the batteries, that is, the effective battery time for the whole system. If the value of `whichBattery` is out of range, or the selected battery is not installed, `GetBatteryTimes` will return a result of 0 in all fields. Here is a summary of the effects of the `whichBattery` parameter:

Value of `whichBattery`	Information returned
0	Total battery time for all batteries
From 1 to `BatteryCount()`	Battery time for the selected battery
Less than 0 or greater than `BatteryCount()`	0 in all fields of `theTimes`

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetBatteryTimes` is 28 ($1C) in the low word of register D0. The pointer to `BatteryTimeRec` is passed in register A0.

Controlling the Internal Modem

The functions in this section return information about the internal modem and configure the internal modem's state information.

GetIntModemInfo

You can use the `GetIntModemInfo` function to find out information about the internal modem.

```
unsigned long GetIntModemInfo();
```

DESCRIPTION

The `GetIntModemInfo` function returns a 32-bit field containing information that describes the features and state of the internal modem. It can be called whether or not a modem is installed and will return the correct information.

If a bit is set, that feature or state is supported or selected; if the bit is cleared, that feature is not supported or selected. Undefined bits are reserved by Apple for future expansion.

Bit name	Bit number	Description
`hasInternalModem`	0	An internal modem is installed.
`intModemRingDetect`	1	The modem has detected a ring on the telephone line.
`intModemOffHook`	2	The internal modem has taken the telephone line off hook (that is, you can hear the dial tone or modem carrier).
`intModemRingWakeEnb`	3	The computer will come out of sleep mode if the modem detects a ring on the telephone line and the computer supports this feature (see the `canWakeupOnRing` bit in `PMFeatures`).
`extModemSelected`	4	The external modem is selected (if this bit is set, then the modem port will be connected to port A of the SCC; if the modem port is not shared by the internal modem and the SCC, then this bit can be ignored).

Bits 15–31 contain the modem type, which can have one of the following values:

Value	Meaning
–1	Modem is installed but type not recognized.
0	No modem is installed.
1	Modem is a serial modem.
2	Modem is a PowerBook Duo–style Express Modem.
3	Modem is a PowerBook 160/180–style Express Modem.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetIntModemInfo` is 14 ($0E) in the low word of register D0. The bit field to set is passed in the high word of register D0.

SetIntModemState

You can use the `SetIntModemState` function to set some parts of the state of the internal modem.

```
void SetIntModemState(short theState);
```

`theState` A set of bits you can use to set the modem state. Set bit 15 of this parameter to 1 to set bits in the modem state. Clear bit 15 to 0 to clear bits in the modem state. The modem state bits are described in the preceding function description.

DESCRIPTION

The `SetIntModemState` function configures some of the internal modem's state information. Currently the only items that can be changed are the internal/external modem selection and the wakeup-on-ring feature.

To change an item of state information, the calling program sets the corresponding bit in the parameter `theState`. For example, to select the external modem, set bit 4 of `theState` to 1 and set bit 15 to 1. To select the internal modem, set bit 4 to 1 but set bit 15 to 0.

SPECIAL CONSIDERATIONS

In some PowerBook computers, there is a hardware switch to connect either port A of the SCC or the internal modem to the modem port. The two are physically separated, but software emulates the serial port interface for those applications that don't use the Communications Toolbox. You can check the `hasSharedModemPort` bit returned by `PMFeatures` to determine which way the computer is set up.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SetIntModemState` is 15 ($0F) in the low word of register D0. The bit field is returned in register D0.

Controlling the Processor

The functions in this section return information about the processor speed and processor cycling, set the processor speed, and enable or disable processor cycling.

MaximumProcessorSpeed

You can use the `MaximumProcessorSpeed` function to find out the maximum speed of the computer's microprocessor.

```
short MaximumProcessorSpeed();
```

DESCRIPTION

The `MaximumProcessorSpeed` function returns the maximum clock speed of the computer's microprocessor, in MHz.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `MaximumProcessorSpeed` is 16 ($10) in the low word of register D0. The processor speed value is returned in the low word of register D0.

CurrentProcessorSpeed

You can use the `CurrentProcessorSpeed` function to find out the current clock speed of the microprocessor.

```
short CurrentProcessorSpeed();
```

DESCRIPTION

The `CurrentProcessorSpeed` function returns the current clock speed of the computer's microprocessor, in MHz. The value returned will be different from the maximum processor speed if the computer has been configured to run with a reduced processor speed to conserve power.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `CurrentProcessorSpeed` is 17 ($11) in the low word of register D0. The processor speed value is returned in the low word of register D0.

FullProcessorSpeed

You can use the `FullProcessorSpeed` function to find out whether the computer will run at full speed the next time it restarts.

```
Boolean FullProcessorSpeed();
```

DESCRIPTION

The `FullProcessorSpeed` function returns a Boolean value of `true` if, on the next restart, the computer will start up at its maximum processor speed; it returns `false` if the computer will start up at its reduced processor speed.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `FullProcessorSpeed` is 18 ($12) in the low word of register D0. The Boolean result is returned in the low byte of register D0.

SetProcessorSpeed

You can use the `SetProcessorSpeed` function to set the clock speed the microprocessor will use the next time it is restarted.

```
Boolean SetProcessorSpeed(Boolean fullSpeed);
```

`fullSpeed` A Boolean that sets the processor speed to full speed (`true`) or reduced speed (`false`).

DESCRIPTION

The `SetProcessorSpeed` function sets the processor speed that the computer will use the next time it is restarted. If the value of `fullSpeed` is set to `true`, the processor will start up at its full speed (the speed returned by `MaximumProcessorSpeed`, described on page 6-60). If the value of `fullSpeed` is set to `false`, the processor will start up at its reduced speed.

SPECIAL CONSIDERATIONS

For PowerBook models that support changing the processor speed dynamically, the current processor speed is also changed. If the speed is actually changed, `SetProcessorSpeed` returns `true`; if the speed is not changed, it returns `false`.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SetProcessorSpeed` is 19 ($13) in the low word of register D0. The Boolean value to set is passed in the high word of register D0. The Boolean result is returned in register D0.

SEE ALSO

You can use the `PMFeatures` function (page 6-41) to determine whether the computer supports changing the processor speed dynamically.

IsProcessorCyclingEnabled

You can use the `IsProcessorCyclingEnabled` function to find out whether processor cycling is enabled.

```
Boolean IsProcessorCyclingEnabled();
```

DESCRIPTION

The `IsProcessorCyclingEnabled` function returns a Boolean value of `true` if processor cycling is currently enabled, or `false` if it is disabled.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `IsProcessorCyclingEnabled` is 24 ($18) in the low word of register D0. The Boolean result is returned in register D0.

EnableProcessorCycling

You can use the `EnableProcessorCycling` function to turn the processor cycling feature on and off.

```
void EnableProcessorCycling(Boolean enable);
```

`enable` A Boolean that specifies whether to enable processor cycling.

DESCRIPTION

The `EnableProcessorCycling` function enables processor cycling if a value of `true` is passed in, and disables it if `false` is passed.

▲ **WARNING**
You should follow the advice of the `mustProcessorCycle` bit in the feature flags when turning processor cycling off. Turning processor cycling off when it's not recommended can result in hardware failures due to overheating. ▲

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `EnableProcessorCycling` is 25 ($19) in the low word of register D0. The Boolean value to set is passed in the high word of register D0.

SEE ALSO

You can use the `PMFeatures` function (page 6-41) to determine whether the computer supports processor cycling.

Getting and Setting the SCSI ID

The functions in this section return and set the SCSI ID the computer uses in SCSI disk mode.

GetSCSIDiskModeAddress

You can use the `GetSCSIDiskModeAddress` function to find out the SCSI ID the computer uses in SCSI disk mode.

```
short GetSCSIDiskModeAddress();
```

DESCRIPTION

The `GetSCSIDiskModeAddress` function returns the SCSI ID that the computer uses when it is started up in SCSI disk mode. The returned value is in the range 1 to 6.

Note
When the computer is in SCSI disk mode, the computer appears as a hard disk to another computer. ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetSCSIDiskModeAddress` is 20 ($14) in the low word of register D0. The SCSI ID is returned in the low word of register D0.

SEE ALSO

You can use the `PMFeatures` function (page 6-41) to determine whether the computer supports SCSI disk mode.

SetSCSIDiskModeAddress

You can use the `SetSCSIDiskModeAddress` function to set the SCSI ID for the computer to use in SCSI disk mode.

```
void SetSCSIDiskModeAddress(short scsiAddress);
```

`scsiAddress`
The SCSI ID that the computer uses if it is started up in SCSI disk mode. You must specify a value in the range of 1 to 6.

DESCRIPTION

The `SetSCSIDiskModeAddress` function sets the SCSI ID that the computer will use if it is started up in SCSI disk mode.

The value of `scsiAddress` must be in the range of 1 to 6. If any other value is given, the software sets the SCSI ID for SCSI disk mode to 2.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SetSCSIDiskModeAddress` is 21 ($15) in the low word of register D0. The SCSI ID to set is passed in the high word of register D0.

SEE ALSO

You can use the `PMFeatures` function (page 6-41) to determine whether the computer supports SCSI disk mode.

Application-Defined Routines

The Power Manager allows you to define a sleep procedure that is called at various stages of the sleep and wakeup processes. You install a sleep procedure by calling the `SleepQInstall` procedure.

Sleep Procedures

You pass the address of a sleep procedure in the `sleepQProc` field of a sleep queue record.

MySleepProc

A sleep procedure can perform any operations required to prepare your application (or other software) for the sleep state. Your sleep procedure is also called when the computer reawakens.

DESCRIPTION

Your sleep procedure is called at various stages in the Power Manager's sleep and wakeup processes. It is called in response to a sleep request, a sleep demand, a wakeup demand, and a sleep-request revocation. You can determine which of these messages the Power Manager is sending by inspecting the sleep procedure selector code passed in register D0. This code is one of four values:

```
enum {
   /* sleep procedure selector codes */
   sleepRequest            = 1,     /* sleep request */
   sleepDemand             = 2,     /* sleep demand */
   sleepWakeUp             = 3,     /* wakeup demand */
   sleepRevoke             = 4      /* sleep-request revocation */
};
```

When called in response to a sleep request, your procedure must either accept or deny the request by either clearing register D0 or leaving it alone. When passed any other selector code, your sleep procedure should take any appropriate actions.

SPECIAL CONSIDERATIONS

A sleep procedure is never executed at interrupt time. As a result, you can, if necessary, call Memory Manager routines or other routines that allocate memory. You can also interact with the user by displaying dialog or alert boxes.

If your sleep procedure displays a dialog or alert box, you should make sure to remove the box after a reasonable amount of time. Failure to do so will prevent the computer from going to sleep and may permanently damage the screen.

ASSEMBLY-LANGUAGE INFORMATION

When your sleep procedure is called, register A0 contains the address of the sleep queue record associated with that procedure and the D0 register contains a sleep procedure selector code.

SEE ALSO

See "Writing a Sleep Procedure," beginning on page 6-20, for instructions on writing a sleep procedure, and see "Installing a Sleep Procedure," beginning on page 6-18, for instructions on installing a sleep procedure.

Hard Disk Spindown Function

You pass the address of a hard disk spindown function in the `hdProc` field of a hard disk queue structure.

MyHDSpindownProc

A hard disk spindown function can perform any operations you require to prepare for the hard disk to spin down.

```
pascal void MyHDSpindownProc(HDQueueElement *theElement);
```

`theElement` A pointer to the element in the hard disk power down queue that was used to install this function.

DESCRIPTION

The `HardDiskQInstall` function installs an element into the hard disk power down queue to provide notification to your software when the internal hard disk is about to be powered off. For example, this feature might be used by the driver for an external battery-powered hard disk. When power to the internal hard disk is turned off, the external hard disk could be turned off as well.

When power to the internal hard disk is about to be turned off, the software calls the routine pointed to by the `hdProc` field so that it can do any special processing. The routine will be passed a pointer to its queue element so that, for example, the routine can reference its variables.

SEE ALSO

The hard disk power down queue elements are defined in "Hard Disk Queue Structure," on page 6-27.

The `HardDiskQInstall` function is described on page 6-52. The `HardDiskQRemove` function is described on page 6-53.

Summary of the Power Manager

Pascal Summary

Constants

```
CONST
   {Power Manager Gestalt selector}
   gestaltPowerMgrAttr        = 'powr';   {Power Manager attributes selector}

   {bit values in Gestalt response parameter}
   gestaltPMgrExists          = 0;     {Power Manager is present}
   gestaltPMgrCPUIdle         = 1;     {CPU can idle}
   gestaltPMgrSCC             = 2;     {can stop SCC clock}
   gestaltPMgrSound           = 3;     {can shut off sound circuits}
   gestaltPMgrDispatchExists  = 4;     {Power Manager dispatch exists }

   slpQType                   = 16;    {sleep queue type}
   sleepQType                 = 16;    {sleep queue type}

   {bit positions for ModemStatus}
   modemOnBit                 = 0;     {1 if modem is on}
   ringWakeUpBit              = 2;     {1 if ring wakeup is enabled}
   modemInstalledBit          = 3;     {1 if internal modem is installed}
   ringDetectBit              = 4;     {1 if incoming call is detected}
   modemOnHookBit             = 5;     {1 if modem is off hook}

   {masks for ModemStatus}
   modemOnMask                = $1;    {modem on}
   ringWakeUpMask             = $4;    {ring wakeup enabled}
   modemInstalledMask         = $8;    {internal modem installed}
   ringDetectMask             = $10;   {incoming call detected}
   modemOnHookMask            = $20;   {modem off hook}

   {bit positions for BatteryStatus}
   chargerConnBit             = 0;     {1 if charger is connected}
   hiChargeBit                = 1;     {1 if charging at hicharge rate}
   chargeOverFlowBit          = 2;     {1 if hicharge counter has overflowed}
   batteryDeadBit             = 3;     {always 0}
   batteryLowBit              = 4;     {1 if battery is low}
   connChangedBit             = 5;     {1 if charger connection has changed}
```

```
{masks for BatteryStatus}
chargerConnMask      = $1;    {charger is connected}
hiChargeMask         = $2;    {charging at hicharge rate}
chargeOverFlowMask   = $4;    {hicharge counter has overflowed}
batteryDeadMask      = $8;    {battery is dead}
batteryLowMask       = $10;   {battery is low}
connChangedMask      = $20;   {connection has changed}

{sleep procedure selector codes}
sleepRequest         = 1;     {sleep request}
sleepDemand          = 2;     {sleep demand}
sleepWakeUp          = 3;     {wakeup demand}
sleepRevoke          = 4;     {sleep-request revocation}

{bits in bitfield returned by PMFeatures}
hasWakeupTimer       = 0;  {1 = wakeup timer is supported}
hasSharedModemPort   = 1;  {1 = modem port shared by SCC and internal modem}
hasProcessorCycling  = 2;  {1 = processor cycling is supported}
mustProcessorCycle   = 3;  {1 = processor cycling should not be turned off}
hasReducedSpeed      = 4;  {1 = processor can be started up at reduced speed}
dynamicSpeedChange   = 5;  {1 = processor speed can be switched dynamically}
hasSCSIDiskMode      = 6;  {1 = SCSI Disk Mode is supported}
canGetBatteryTime    = 7;  {1 = battery time can be calculated}
canWakeupOnRing      = 8;  {1 = can wakeup when the modem detects a ring}
hasDimmingSupport    = 9;  {1 = has dimming support built into the ROM}

{bits in BatteryInfo.flags}
batteryInstalled     = 7;  {1 = battery is currently connected}
batteryCharging      = 6;  {1 = battery is being charged}
chargerConnected     = 5;  {1 = charger is connected to the PowerBook }
                           { (this does not mean the charger is plugged in)}

{bits in bitfield returned by GetIntModemInfo}
hasInternalModem     = 0;  {1 = internal modem installed}
intModemRingDetect   = 1;  {1 = internal modem has detected a ring}
intModemOffHook      = 2;  {1 = internal modem is off hook}
intModemRingWakeEnb  = 3;  {1 = wakeup on ring is enabled}
extModemSelected     = 4;  {1 = external modem selected}

modemSetBit          = 15; {1 = set bit, 0=clear bit}

HDPwrQType           ='HD';{hard disk notification queue element type}
```

Data Types

```
TYPE SleepQRec =
   RECORD
      sleepQLink:    SleepQRecPtr;  {next queue element}
      sleepQType:    Integer;       {queue type = 16}
      sleepQProc:    ProcPtr;       {pointer to sleep procedure}
      sleepQFlags:   Integer;       {reserved}
   END;
   SleepQRecPtr = ^SleepQRec;

TYPE HDQueueElement =
   RECORD
      hdQLink:       Ptr;           {pointer to next queue element}
      hdQType:       Integer;       {queue element type (must be HDQType)}
      hdFlags:       Integer;       {miscellaneous flags}
      hdProc:        ProcPtr;       {pointer to routine to call}
      hdUser:        LongInt;       {user-defined (variable storage, etc.)}
   END;

TYPE WakeupTime =
   PACKED RECORD
      wakeTime:      LongInt;       {wakeup time (same format as time)}
      wakeEnabled:   Byte;          {1 = enable, 0=disable wakeup timer}
   END;

TYPE BatteryInfo =
   PACKED RECORD
      flags:         Byte;          {misc flags (see above)}
      warningLevel:  Byte;          {scaled warning level (0-255)}
      reserved:      Byte;          {reserved for internal use}
      batteryLevel:  Byte;          {scaled battery level (0-255)}
   END;

TYPE BatteryTimeRec =
   RECORD

      expectedBatteryTime: LongInt; {estimated battery time remaining}
      minimumBatteryTime:  LongInt; {minimum battery time remaining}
      maximumBatteryTime:  LongInt; {maximum battery time remaining}
      timeUntilCharged:    LongInt; {time until battery is fully charged}
   END;
```

Power Manager Routines

Controlling the Idle State

```
FUNCTION IdleUpdate         : LongInt;
PROCEDURE EnableIdle;
PROCEDURE DisableIdle;
FUNCTION GetCPUSpeed        : LongInt;
```

Controlling and Reading the Wakeup Timer

```
FUNCTION SetWUTime          (WUTime: LongInt): OSErr;
FUNCTION DisableWUTime      : OSErr;
FUNCTION GetWUTime          (VAR WUTime: LongInt; VAR WUFlag: Byte): OSErr;
```

Controlling the Sleep Queue

```
PROCEDURE SleepQInstall     (qRecPtr: SleepQRecPtr);
PROCEDURE SleepQRemove      (qRecPtr: SleepQRecPtr);
```

Controlling Serial Power

```
PROCEDURE AOn;
PROCEDURE AOnIgnoreModem;
PROCEDURE BOn;
PROCEDURE AOff;
PROCEDURE BOff;
```

Reading the Status of the Internal Modem

```
FUNCTION ModemStatus        (VAR Status: Byte): OSErr;
```

Reading the Status of the Battery and the Battery Charger

```
FUNCTION BatteryStatus      (VAR Status: Byte; VAR Power: Byte): OSErr;
```

Power Manager Dispatch Routines

Determining the Power Manager Features Available

```
FUNCTION PMSelectorCount    : Integer;
FUNCTION PMFeatures         : LongInt;
```

Controlling the Sleep and Wakeup Timers

```
FUNCTION GetSleepTimeout : Byte;
PROCEDURE SetSleepTimeout(timeout : Byte);
PROCEDURE AutoSleepControl(enableSleep : Boolean);
FUNCTION IsAutoSlpControlDisabled() : Boolean;
PROCEDURE GetWakeupTimer(VAR theTime : WakeupTime);
PROCEDURE SetWakeupTimer(theTime : WakeupTime);
```

Controlling the Dimming Timer

```
FUNCTION GetDimmingTimeout() : Byte;
PROCEDURE SetDimmingTimeout(timeout : Byte);
PROCEDURE DimmingControl(enableDimming : Boolean);
FUNCTION IsDimmingControlDisabled() : Boolean;
```

Controlling the Hard Disk

```
FUNCTION GetHardDiskTimeout : Byte;
PROCEDURE SetHardDiskTimeout(timeout : Byte);
FUNCTION HardDiskPowered : Boolean;
PROCEDURE SpinDownHardDisk;
FUNCTION IsSpindownDisabled : Boolean;
PROCEDURE SetSpindownDisable(setDisable : BOOLEAN);
FUNCTION HardDiskQInstall(VAR theElement : HDQueueElement) : OSErr;
FUNCTION HardDiskQRemove(VAR theElement : HDQueueElement) : OSErr;
```

Getting Information About the Battery

```
PROCEDURE GetScaledBatteryInfo(whichBattery : Integer; VAR theInfo :
                            BatteryInfo);
FUNCTION BatteryCount : Integer;
FUNCTION GetBatteryVoltage(whichBattery : Integer) : Fixed;
PROCEDURE GetBatteryTimes(whichBattery : INTEGER; VAR theTimes :
                            BatteryTimeRec);
```

Controlling the Internal Modem

```
FUNCTION GetIntModemInfo : LongInt;
PROCEDURE SetIntModemState(theState : Integer);
```

Controlling the Processor

```
FUNCTION MaximumProcessorSpeed : Integer;
```

```
FUNCTION CurrentProcessorSpeed : Integer;
FUNCTION FullProcessorSpeed : Boolean;
FUNCTION SetProcessorSpeed(fullSpeed : Boolean) : Boolean;
FUNCTION IsProcessorCyclingEnabled : Boolean;
PROCEDURE EnableProcessorCycling(enable : Boolean);
```

Getting and Setting the SCSI ID

```
FUNCTION GetSCSIDiskModeAddress : Integer;
PROCEDURE SetSCSIDiskModeAddress(scsiAddress : Integer);
```

Application-Defined Routines

```
PROCEDURE MySleepProc;
PROCEDURE MyHDSpindownProc(theElement : HDQueueElement);
```

C Summary

Constants and Data Types

```
/* Power Manager Gestalt selector */
#define gestaltPowerMgrAttr 'powr'  /* Power Manager attributes selector */

/* bit values in Gestalt response parameter */
enum {
   gestaltPMgrExists          = 0,     /* Power Manager is present */
   gestaltPMgrCPUIdle         = 1,     /* CPU can idle */
   gestaltPMgrSCC             = 2,     /* can stop SCC clock */
   gestaltPMgrSound           = 3,     /* can shut off sound circuits */
   gestaltPMgrDispatchExists  = 4      /* Power Manager dispatch exists */
};

enum {
   slpQType                   = 16,    /* sleep queue type */
   sleepQType                 = 16     /* sleep queue type */
};

enum {
   /* bit positions for ModemStatus */
   modemOnBit                 = 0,     /* 1 if modem is on */
   ringWakeUpBit              = 2,     /* 1 if ring wakeup is enabled */
```

```
    modemInstalledBit       = 3,     /* 1 if internal modem is installed */
    ringDetectBit           = 4,     /* 1 if incoming call is detected */
    modemOnHookBit          = 5,     /* 1 if modem is off hook */

    /* masks for ModemStatus */
    modemOnMask             = 0x1,   /* modem on */
    ringWakeUpMask          = 0x4,   /* ring wakeup enabled */
    modemInstalledMask      = 0x8,   /* internal modem installed */
    ringDetectMask          = 0x10,  /* incoming call detected */
    modemOnHookMask         = 0x20,  /* modem off hook */

    /* bit positions for BatteryStatus */
    chargerConnBit          = 0,     /* 1 if charger is connected */
    hiChargeBit             = 1,     /* 1 if charging at hicharge rate */
    chargeOverFlowBit       = 2,     /* 1 if hicharge counter has overflowed */
    batteryDeadBit          = 3,     /* always 0 */
    batteryLowBit           = 4,     /* 1 if battery is low */
    connChangedBit          = 5,     /* 1 if charger connection has changed */

    /* masks for BatteryStatus */
    chargerConnMask         = 0x1,   /* charger is connected */
    hiChargeMask            = 0x2,   /* charging at hicharge rate */
    chargeOverFlowMask      = 0x4,   /* hicharge counter has overflowed */
    batteryDeadMask         = 0x8,   /* battery is dead */
    batteryLowMask          = 0x10,  /* battery is low */
    connChangedMask         = 0x20,  /* connection has changed */

    /* sleep procedure selector codes */
    sleepRequest            = 1,     /* sleep request */
    sleepDemand             = 2,     /* sleep demand */
    sleepWakeUp             = 3,     /* wakeup demand */
    sleepRevoke             = 4      /* sleep-request revocation */
};

/* bits in bitfield returned by PMFeatures */
#define hasWakeupTimer 0            /* 1 = wakeup timer is supported */
#define hasSharedModemPort 1        /* 1 = modem port shared by SCC and */
                                    /*  internal modem */
#define hasProcessorCycling 2       /* 1 = processor cycling is supported */
#define mustProcessorCycle 3        /* 1 = processor cycling should not be */
                                    /*  turned off */
#define hasReducedSpeed 4           /* 1 = processor can be started up at */
                                    /*  reduced speed */
#define dynamicSpeedChange 5        /* 1 = processor speed can be */
                                    /*  switched dynamically */
```

```
#define hasSCSIDiskMode 6     /* 1 = SCSI Disk Mode is supported */
#define canGetBatteryTime 7   /* 1 = battery time can be calculated */
#define canWakeupOnRing 8     /* 1 = can wakeup when the modem detects */
                              /*  a ring */
#define hasDimmingSupport 9   /* 1 = has dimming support built into the ROM */

/* bits in bitfield returned by GetIntModemInfo and set by SetIntModemState */
#define hasInternalModem 0    /* 1 = internal modem installed */
#define intModemRingDetect 1  /* 1 = internal modem has detected a ring */
#define intModemOffHook 2     /* 1 = internal modem is off hook */
#define intModemRingWakeEnb 3 /* 1 = wakeup on ring is enabled */
#define extModemSelected 4    /* 1 = external modem selected */

#define modemSetBit 15        /* 1 = set bit, 0=clear bit (SetIntModemState) */

/* bits in BatteryInfo.flags */
#define batteryInstalled 7    /* 1 = battery is currently connected */
#define batteryCharging 6     /* 1 = battery is being charged */
#define chargerConnected 5    /* 1 = charger is connected to the PowerBook */
                              /*  (this does not mean the charger is */
                              /*  plugged in) */

struct SleepQRec {
   struct SleepQRec  *sleepQLink;   /* next queue element */
   short             sleepQType;    /* queue type = 16 */
   ProcPtr           sleepQProc;    /* pointer to sleep procedure */
   short             sleepQFlags;   /* reserved */
};
typedef struct SleepQRec SleepQRec;
typedef SleepQRec *SleepQRecPtr;

/* hard disk spindown notification queue element */
typedef struct HDQueueElement HDQueueElement;

typedef pascal void (*HDSpindownProc)(HDQueueElement *theElement);

struct HDQueueElement {
   Ptr hdQLink;            /* pointer to next queue element */
   short hdQType;          /* queue element type (must be HDQType) */
   short hdFlags;          /* miscellaneous flags */
   HDSpindownProc hdProc;  /* pointer to routine to call */
   long hdUser;            /* user-defined private storage */
};

#define HDPwrQType 'HD'    /* queue element type */
```

```
/* wakeup time record */
typedef struct WakeupTime {
   unsigned long wakeTime;    /* wakeup time (same format as current time) */
   char wakeEnabled;          /* 1 = enable wakeup timer, 0=disable */
} WakeupTime;

/* battery time information (in seconds) */
typedef struct BatteryTimeRec {
   unsigned long expectedBatteryTime; /* estimated battery time remaining */
   unsigned long inimumBatteryTime;  /* minimum battery time remaining */
   unsigned long maximumBatteryTime; /* maximum battery time remaining */
   unsigned long timeUntilCharged;   /* time until battery is fully charged */
} BatteryTimeRec;
```

Power Manager Functions

Controlling the Idle State

```
pascal long IdleUpdate      (void);
pascal void EnableIdle      (void);
pascal void DisableIdle     (void);
pascal long GetCPUSpeed     (void);
```

Controlling and Reading the Wakeup Timer

```
pascal OSErr SetWUTime      (long WUTime);
pascal OSErr DisableWUTime  (void);
pascal OSErr GetWUTime      (long *WUTime, Byte *WUFlag);
```

Controlling the Sleep Queue

```
pascal void SleepQInstall   (SleepQRecPtr qRecPtr);
pascal void SleepQRemove    (SleepQRecPtr qRecPtr);
```

Controlling Serial Power

```
pascal void AOn             (void);
pascal void AOnIgnoreModem  (void);
pascal void BOn             (void);
pascal void AOff            (void);
pascal void BOff            (void);
```

Reading the Status of the Internal Modem

```
pascal OSErr ModemStatus    (Byte *Status);
```

Reading the Status of the Battery and the Battery Charger

```
pascal OSErr BatteryStatus  (Byte *Status, Byte *Power);
```

Power Manager Dispatch Functions

Determining the Power Manager Features Available

```
short PMSelectorCount       (void);
unsigned long PMFeatures    (void);
```

Controlling the Sleep and Wakeup Timers

```
unsigned char GetSleepTimeout(void);
void SetSleepTimeout        (unsigned char timeout);
void AutoSleepControl       (Boolean enableSleep);
Boolean IsAutoSlpControlDisabled(void);
void GetWakeupTimer         (WakeupTime *theTime);
void SetWakeupTimer         (WakeupTime *theTime);
```

Controlling the Dimming Timer

```
unsigned char GetDimmingTimeout(void);
void SetDimmingTimeout      (unsigned char timeout);
void DimmingControl         (Boolean enableDimming);
Boolean IsDimmingControlDisabled(void);
```

Controlling the Hard Disk

```
unsigned char GetHardDiskTimeout(void);
void SetHardDiskTimeout     (unsigned char timeout);
Boolean HardDiskPowered     (void);
void SpinDownHardDisk       (void);
Boolean IsSpindownDisabled  (void);
void SetSpindownDisable     (Boolean setDisable);
OSErr HardDiskQInstall      (HDQueueElement *theElement);
OSErr HardDiskQRemove       (HDQueueElement *theElement);
```

Getting Information About the Battery

```
void GetScaledBatteryInfo   (short whichBattery, BatteryInfo *theInfo);
short BatteryCount          (void);
Fixed GetBatteryVoltage     (short whichBattery);
void GetBatteryTimes        (short whichBattery, BatteryTimeRec *theTimes);
```

Controlling the Internal Modem

```
unsigned long GetIntModemInfo(void);
void SetIntModemState       (short theState);
```

Controlling the Processor

```
short MaximumProcessorSpeed (void);
short CurrentProcessorSpeed (void);
Boolean FullProcessorSpeed  (void);
Boolean SetProcessorSpeed   (Boolean fullSpeed);
Boolean IsProcessorCyclingEnabled(void);
void EnableProcessorCycling (Boolean enable);
```

Getting and Setting the SCSI ID

```
short GetSCSIDiskModeAddress(void);
void SetSCSIDiskModeAddress (short scsiAddress);
```

Application-Defined Functions

```
void MySleepProc            (void);
void (*HDSpindownProc)(HDQueueElement *theElement);
```

Assembly-Language Summary

Data Structures

Sleep Queue Data Structure

0	`sleepQLink`	long	pointer to next element in the queue
4	`sleepQType`	word	queue type (should be 16)
6	`sleepQProc`	long	pointer to a sleep procedure
10	`sleepQFlags`	word	reserved

Hard Disk Queue Structure

0	`hdQLink`	long	pointer to next element in the queue
4	`hdQType`	word	queue type (should be `HDPwrQType`)
6	`hdFlags`	word	reserved
8	`hdProc`	long	pointer to a hard disk power-down procedure
12	`hdUser`	long	user defined

Wakeup Time Structure

0	`wakeTime`	long	wakeup time in seconds since `00:00:00, 1/1/1904`
4	`wakeEnabled`	byte	1 = enable wakeup timer, 0 = disable timer

Battery Information Structure

0	`flags`	byte	flags
1	`warningLevel`	byte	scaled warning level (0—255)
2	`reserved`	byte	reserved
3	`batteryLevel`	byte	scaled battery level (0—255)

Battery Time Structure

0	`expectedBatteryTime`	long	estimated battery time remaining in seconds
4	`minimumBatteryTime`	long	minimum battery time remaining
8	`maximumBatteryTime`	long	maximum battery time remaining
12	`timeUntilCharged`	long	time remaining until battery is fully charged

Trap Macros

Trap Macros Requiring Routine Selectors

`_IdleState`

Selector	Routine
0	`EnableIdle`
Any positive number	`DisableIdle`
Any negative number	`GetCPUSpeed`

`_SerialPower`

Selector	Routine
$04	`AOn`
$05	`AOnIgnoreModem`
$00	`BOn`
$84	`AOff`
$80	`BOff`

_PowerMgrDispatch

Selector	Routine
$00	PMSelectorCount
$01	PMFeatures
$02	GetSleepTimeout
$03	SetSleepTimeout
$04	GetHardDiskTimeout
$05	SetHardDiskTimeout
$06	HardDiskPowered
$07	SpinDownHardDisk
$08	IsSpindownDisabled
$09	SetSpindownDisable
$0A	HardDiskQInstall
$0B	HardDiskQRemove
$0C	GetScaledBatteryInfo
$0D	AutoSleepControl
$0E	GetIntModemInfo
$0F	SetIntModemState
$10	MaximumProcessorSpeed
$11	CurrentProcessorSpeed
$12	FullProcessorSpeed
$13	SetProcessorSpeed
$14	GetSCSIDiskModeAddress
$15	SetSCSIDiskModeAddress
$16	GetWakeupTimer
$17	SetWakeupTimer
$18	IsProcessorCyclingEnabled
$19	EnableProcessorCycling
$1A	BatteryCount
$1B	GetBatteryVoltage
$1C	GetBatteryTimes
$1D	GetDimmingTimeout
$1E	SetDimmingTimeout
$1F	DimmingControl
$20	IsDimmingControlDisabled
$21	IsAutoSlpControlDisabled

Result Codes

`noErr`	0	No error
`pmBusyErr`	–13000	Power Manager IC stuck busy
`pmReplyTOErr`	–13001	Timed out waiting to begin reply handshake
`pmSendStartErr`	–13002	Power Manager IC did not start handshake
`pmSendEndErr`	–13003	During send, Power Manager did not finish handshake
`pmRecvStartErr`	–13004	During receive, Power Manager did not start handshake
`pmRecvEndErr`	–13005	During receive, Power Manager did not finish handshake