Technote 1190

Power Manager 2.0

CONTENTS

Participating in Power Management

Changes to Sleep and Wake

Device Power Handlers

New Routines in Power Manager 2.0

New Sleep Types

Sleep Messages

More New Messages

Wake on Network Activity

Adding Power Sources

Obtaining Microprocessor Temperature

Power Manager Apple Events

Server Mode

Further References

The Power Manager 2.0 is an update to Mac OS that facilitates the implementation of a more aggressive power management policy, and supports new capabilities in the latest Macintosh hardware and the NewWorld ROM architecture.

Power Manager 2.0 is available on all iMacs, Blue & White G3 and Power Mac G4 desktops, the "bronze keyboard" PowerBook G3 Series, the iBook, and future portables and desktops.

While internal changes have been made to the Mac OS to support Power Manager 2.0, there are a few new API routines that are available to developers. Thus, this document describes the features new or updated in Power Manager 2.0. For complete details on using Power Manager services, please see the chapter "Power Manager" in *Inside Macintosh: Devices* .

This Note is not only directed at application developers who previously had concerns about running on portables, but indeed to all Mac OS application developers. Device Driver writers on the other hand, should review *Updating Drivers for PM 2.0* for detailed information on how to support removal of power to the PCI slots.

Participating in Power Management

The new Power Manager is less and less a PowerBook-specific manager. Indeed, the new Power Manager performs the same power savings activities on desktop machines (where supported) as it does on portables, and will continue to do so in future products. It is important to use power efficiently in all computer systems, even those that never leave the den or office.

With the introduction of the new nanokernel that first became available in Mac OS 8.6 a task is allowed to block when it is not busy or does not require CPU time. As a result, the processor (or in some cases multiple processors), can be put into a low-power state until needed. This provides improved power savings, which translates to a noticeable improvement in battery life.

In order to take advantage of the new power-saving capability, applications and other software can help keep the system idle by simply following some basic guidelines.

WaitNextEvent

Do not use a zero sleep value in your calls to `WaitNextEvent`. Doing so causes your application to monopolize the CPU and prevents the system from becoming idle.

VBLs and Time Tasks

Reduce your use of VBLs and Time Manager tasks, especially if they trigger with high frequency. Again, the more tasks that are used and the higher the frequency, the more the CPU is burdened with executing possibly needless instructions.

Polling/Spin Looping

Do not sit in a loop and spin, waiting for an event. Do not use a spin loop to wait for `ioResult` to change in Device Manager calls. Instead, use the new, idle-friendly `PBWaitIOComplete` routine in the Device Manager.

```
OSStatus PBWaitIOComplete(IOParamPtr ioPB, Duration timeout)
```

`PBWaitIOComplete` will keep the system idle until either an interrupt occurs (one which possibly affects your wait on `ioResult`), or the specified timeout value has been reached. If the timeout has been reached, this routine will return `kMPTimeoutErr`.

Back to top

Changes to Sleep and Wake

To meet more aggressive power management requirements, the sleep/wake process has been enhanced to permit new hardware capabilities, including removing power from the PCI slots during sleep, if possible, and entering a new state called "deep sleep." However, the overall process is the same and the use of the sleep queue remains unchanged, other than additional messages that entries may or may not need to handle.

DriverServices has been updated to provide services, which, if used by PCI (and other) device drivers, allow devices to control their power during the sleep/wake process at a later stage than the traditional sleep queue. While using these new services is similar to using the classic sleep queue, the point at which device power handlers will be called is much later in the process, thereby allowing those devices to provide their services until the last possible moment before power is removed.

Note:
For complete details on using the traditional sleep queue, see the chapter "Power Manager" in *Inside Macintosh: Devices* .

Important:
Device Driver writers should review the separate document, *Updating Drivers for PM 2.0* , for detailed information on how to support removal of power to the PCI slots.

Back to top

Device Power Handlers

A device power handler is a routine called by the Power Manager that services power management requests on behalf of devices attached to the computer.

The Power Manager defines a new low-level sleep queue called the **device sleep queue**. The device sleep queue is comprised of power handlers that are called with sleep-related power management requests very late in the sleep process and very early in the wake process. This queue is intended primarily for device drivers to set their power state in response to a sleep or wake event. The device sleep queue will receive the same messages sent to the traditional sleep queue, and both queues can receive some additional messages (please see "New Sleep Messages" below).

```
typedef pascal OSStatus (*PowerHandlerProcPtr)(UInt32  message,
                void * param,
                UInt32 refCon,
                RegEntryID * regEntryID);
```

This is the definition of a power handler. The parameter `message`is the current power management request. `param` is message-specific and is currently only used for `kGetDevicePowerLevel` and `kSetDevicePowerLevel` messages and the `kGetPowerInfo` and `kGetWakeOnNetInfo` messages. The `refCon` is provided for the power handler to use. Any value passed in this parameter during registration will be returned to the power handler each time it is called. The last parameter, `regEntryID`, is the device that the power handler controls and is the same as that presented to the Power Manager when the power handler was registered.

Important:
All power handlers should return the `kPowerMgtMessageNotHandled` if they do not handle a particular power management message.

Back to top

New Routines in Power Manager 2.0

AddDevicePowerHandler

You can use `AddDevicePowerHandler` to add a power handler to the device sleep queue. `regEntryID` is the address of your device's registry entry ID. Note that this is a required parameter and a valid `RegEntryID` must be provided. `handler` is the power handler you are registering for the given device. This is a pointer to PowerPC code and NOT a routine descriptor. Only PowerPC code can register a power handler, so a routine descriptor is not required. The `refCon` field is for your internal use and will be passed back to you on each power management request. `deviceType` is a string describing the type of device you have. Most power handlers wont need to use this parameter as the Power Manager will lookup your device type based on the `regEntryID` parameter.

```
    OSStatus AddDevicePowerHandler (RegEntryIDPtr regEntryID,
                    PowerHandlerProcPtr handler,
                    UInt32 refCon,
                    char * deviceType);
```

RemoveDevicePowerHandler

You can use `RemoveDevicePowerHandler` to uninstall your power handler. `regEntryID` is the registry entry ID for the device your power handler controls.

```
    OSStatus RemoveDevicePowerHandler (RegEntryIDPtr
 regEntryID);
```

GetDevicePowerLevel

You can use `GetDevicePowerLevel` to query a device's power handler for current power information.

```
    OSStatus GetDevicePowerLevel (RegEntryIDPtr regEntryID,
                    PowerLevel * devicePowerLevel);
```

SetDevicePowerLevel

You can use `SetDevicePowerLevel` to set a device's power level to the state you provide in the `devicePowerLevel` parameter. Please see "Updating Drivers for PM 2.0" for a description of power level definitions and how to use `GetDevicePowerLevel` and `SetDevicePowerLevel`.

```
     OSStatus SetDevicePowerLevel (RegEntryIDPtr regEntryID,
                    PowerLevel devicePowerLevel);
```

Back to top

New Sleep Types

With Power Manager 2.0 and the latest hardware from Apple, including the iBook and later machines, there are a few variations on sleep.

Simple Sleep

This form of sleep is what most have come to know as traditional PowerBook sleep. Most of the machine is powered off, but memory is placed into self-refresh mode so that the contents are not lost while the machine is asleep.

Safe Sleep

This form of sleep is similar to simple sleep, but a file is written to disk that represents the contents of memory at the point of sleeping so that -- should an unexpected loss of power occur -- the user's working context can be fully restored upon the next startup. Currently, this variation of sleep is only available if Virtual Memory is turned on in the Memory Control Panel. Safe sleep is enabled if the user checks the "Preserve memory contents on sleep" checkbox in the Advanced Settings panel of the Energy Saver Control Panel.

Deep Sleep

This form of sleep is one where the contents of memory are written to disk and the machine is fully powered off. When the user presses the power key the machine is booted to the point where it is determined if the preserved memory contents exist on disk and, if so, memory is reconstructed and the machine wakes up as though it had only just slept.

Deep sleep is currently only available if Virtual Memory is turned on in the Memory Control Panel. Currently, deep sleep is only entered when the machine goes into a Safe Sleep state and power is lost during that time (such as when battery power is fully drained while the machine is asleep).

To see if safe or deep sleep is a supported feature of a given machine, use the `PMFeatures` routine and check to see if the `hasDeepSleep` bit is set.

Back to top

Sleep Messages

Given the variations now possible for sleep, the messages sent to the sleep queue and the device sleep queue can also vary.

Messages for Simple Sleep

In this case, sleep queue entries and device sleep queue entries will receive the standard set of messages: `kSleepRequest` (previously `sleepRequest`) or `kSleepDemand` (previously `sleepDemand`) and `kSleepWakeup` on wake (previously `sleepWakeup`). These cases should be handled as before. Alternatively, these queues will receive the various doze messages on Blue & White G3s and iMacs: `kDozeRequest`, `kDozeDemand`, and `kDozeWakeup`.

Messages for Safe Sleep

In the safe sleep case, the standard set of messages described above is sent, but prior to a `kSleepRequest` or `kSleepDemand`, a sleep queue entry will receive a `kSuspendRequest` or `kSuspendDemand`, respectively. For the sleep queue, these latter new messages are sent not only in addition to the existing messages but also prior to them. The reason both messages are sent is that legacy software might not recognize a new message, and therefore might not properly prepare for the imminent sleep. The device sleep queue entries will receive the `kSuspendRequest` and/or `kSuspendDemand` message in lieu of the `kSleepRequest` or `kSleepDemand` messages.

On wakeup, if power was lost, then the sleep queue will receive a `kSuspendWakeup` message in addition and prior to a `kSleepWakeup` message. The device sleep queue entries will only receive a `kSuspendWakeup` if the machine is resuming from a powered-off state and is to restore the contents of memory that were saved to disk. If waking from normal sleep, both queues will receive a `kSleepWakeup` message.

Very Important!
Software registered in the traditional sleep queue must maintain its own state regarding these additional messages. For example, if software receives a `kSuspendDemand` message and proceeds to perform activities such that when a `kSleepDemand` message is received those activities do not need to be performed again, then it is up to that software to properly manage the sequence and not duplicate the activities. For the device sleep queue, since entries registered in it only receive one or the other message, this is not an issue.

Sleep Message Examples

In order to clarify the complexity of the new messages, the following examples are provided to show the order of events for each type of sleep. It is important to note that most software will not have to change how sleep is handled to run correctly with the new Power Manager, since most software typically should not care if safe sleep is being entered or not. In this case, follow the same guidelines as described in *Inside Macintosh: Devices* . Only device driver writers and those who must do something differently in the case of safe sleep need to understand when the new messages are sent and how to respond to them.

Note:
For older desktops (Blue & White G3s, iMacs) and machines that otherwise cannot enter the normal sleep state, replace "sleep" in the following examples with "doze" (but only in the simple sleep cases since safe sleep is not available to such machines). For example, `kSleepRequest` becomes `kDozeRequest`.

Simple Sleep Request (Idle Sleep)

This is the sequence of messages sent when the user-specified sleep timeout occurs:

1. Sleep queue sent `kSleepRequest`.
2. If a sleep queue element denies request, `kSleepRevoke` sent to sleep queue and sleep is aborted . Goto step 10.
3. Device sleep queue sent `kSleepRequest`.
4. If a device sleep queue element denies request, `kSleepRevoke` sent to device sleep queue and to sleep queue, and sleep is aborted. Goto step 10.
5. Sleep queue sent `kSleepDemand`.
6. Device sleep queue sent `kSleepDemand`.
7. Machine sleeps...then a wakeup event occurs.
8. Device sleep queue sent `kSleepWakeup`.
9. Sleep queue sent sleep `kSleepWakeup`.
10. Machine is awake.

Simple Sleep Demand (User-Demand Sleep)

This is the sequence of messages sent when the user (or a low-power condition) initiates sleep:

1. Sleep queue sent `kSleepDemand`.
2. Device sleep queue sent `kSleepRequest`.
3. If a device sleep queue element denies request, `kSleepRevoke` sent to device sleep queue and `kSleepWakeup` sent to sleep queue, and sleep is aborted. Goto step 8.
4. Device sleep queue sent `kSleepDemand`.
5. Machine sleeps...then a wakeup event occurs.

6. Device sleep queue sent `kSleepWakeup`.
7. Sleep queue sent sleep `kSleepWakeup`.
8. Machine is awake.

Safe Sleep Request

This is the sequence of messages sent when the user-specified sleep timeout occurs for safe sleep:

1. Sleep queue sent `kSuspendRequest`.
2. Sleep queue sent `kSleepRequest`.
3. If an sleep queue element denies request, `kSleepRevoke` sent to sleep queue and sleep is aborted. Goto step 14.
4. Device sleep queue sent `kSuspendRequest`.
5. If a device sleep queue element denies request, `kSuspendRevoke` sent to device sleep queue and `kSleepRevoke` sent to sleep queue and sleep is aborted then goto step 14.
6. Sleep queue sent `kSuspendDemand`.
7. Sleep queue sent `kSleepDemand`.
8. Device sleep queue sent `kSuspendDemand`.
9. Machine sleeps (and subsequently may or may not lose power)...then a wakeup/powerup event occurs.
10. If waking from normal sleep, device sleep queue sent `kSleepWakeup`. Goto step 13.
11. If waking from power off, device sleep queue sent `kSuspendWakeup`.
12. If waking from power off, sleep queue sent `kSuspendWakeup`.
13. Sleep queue sent `kSleepWakeup`.
14. Machine is awake.

Safe Sleep Demand

This is the sequence of messages sent when the user (or a low-power condition) initiates safe sleep:

1. Sleep queue sent `kSuspendDemand`.
2. Sleep queue sent `kSleepDemand`.
3. Device sleep queue sent `kSuspendRequest`.
4. If a device sleep queue element denies request, `kSleepRevoke` sent to device sleep queue and `kSleepWakeup` sent to sleep queue. Sleep is aborted, then goto step 11.
5. Device sleep queue sent `kSuspendDemand`.
6. Machine sleeps (and subsequently may or may not lose power)...then a wakeup/powerup event occurs.
7. If waking from normal sleep, device sleep queue sent `kSleepWakeup` then goto step 10.
8. If waking from power off, device sleep queue sent `kSuspendWakeup`.
9. If waking from power off, sleep queue sent `kSuspendWakeup`.
10. Sleep queue sent sleep `kSleepWakeup`.
11. Machine is awake.

Note:
A request is always sent to the device sleep queue regardless of whether the original sleep is a request (idle) or demand (user-initiated) sleep. This is because device sleep queue entries must always be given the opportunity to deny a sleep due to the fact that the device whose power they manage may not support low-power sleep (which can occur on Power Macintosh G4 desktops where power is removed from the PCI slots if all devices support it).

Back to top

More New Messages

WakeToDoze

The kWakeToDoze message is sent to both the sleep queue and the device sleep queue if the machine is to wake up to a certain point to service a network or other request that was the cause of waking the machine (rather than the user). Most sleep queue entries will not need to respond to this message separately and would return noErr. Most device sleep queue entries must treat a kWakeToDoze message in the same way as they treat a kSleepWakeup message. They should always handle and NOT ignore the message. Some device power handlers, video devices in particular (and possibly mass storage devices), should respond to the request differently and only perform the minimum wake tasks short of bringing the display back to life or spinning up the hard drive (unless necessary).

Sample handler that doesn't do anything special for kWakeToDoze:

```
pascal OSStatus MyPowerHandler (UInt32 message,
                      void *param,
                      UInt32 refCon,
                      RegEntryID * regEntryID)
{
   switch (message)
        {
                case kWakeToDoze:
                case kSleepWakeup:
                       WakeMyDevice ();
                       break;

                case kDozeToFullWakeUp:
                  // We can ignore since we awoke fully on the kWakeToDoze msg
                       break;
                :
        }
}
```

Sample handler that DOES handle kWakeToDoze differently:

```
pascal OSStatus MyPowerHandler (UInt32 message,
                      void *param,
                      UInt32 refCon,
                      RegEntryID * regEntryID)
{
   switch (message)
        {
                case kWakeToDoze:
                       WakeMyDevicePartially();
                       break;

                case kSleepWakeup:
                       WakeMyDevice ();
                       break;

                case kDozeToFullWakeUp:
                       WakeMyDeviceFromPartialState ();
                       break;
                :
        }
}
```

DozeToFullWakeUp

The kDozeToFullWakeUp message follows the kWakeToDoze message if the user has elected to wake the machine manually while it is in the intermediate doze state. Again,most entries may choose to ignore this message, but those who responded to the kWakeToDoze message should perform whatever steps are required to bring their device to a full wake state.

If the machine is in the intermediate wake state (doze) and the normal sleep timeout occurs, the sleep queue entries will get the normal sleep or suspend request messages, and should handle them accordingly as the machine goes back to full sleep.

GetPowerLevel & GetPowerLevel

The kGetPowerLevel and kSetPowerLevel messages will only be sent to entries in the new device sleep queue. They are used to inform a power handler in the device sleep queue that a piece of software wishes to change the state of the devices it controls by using the new DriverServices GetDevicePowerLevel and SetDevicePowerLevel routines. See *Updating Drivers for PM 2.0* for a full description of the power level definitions.

DeviceInitiatedWake

The kDeviceInitiatedWake message is sent to all power handlers to query if they control a device that caused a wakeup to occur. Most power handlers will not respond to this request. Devices that can wake up the machine should respond to this message by returning an appropriate value that indicates if the device did initiate the wake up and if the wake up should be full using kDeviceRequestsFullWake or partial using kDeviceRequestsWakeToDoze. These values are to be returned via the param parameter of the power handler function.

Here is an example of a function that indicates to the Power Mgr that its device caused the wakeup:

```
pascal OSStatus MyPowerHandler (UInt32 message,
                void *param,
                UInt32 refCon,
                RegEntryID * regEntryID)
{
  OSStatus  status = kPowerMgtMessageNotHandled;
  switch (message)
  {
    case kDeviceInitiatedWake:
      *param = kDeviceDidNotWakeMachine;
      if (IWokeTheMachine())
        *param = kDeviceRequestsFullWake;
      status = noErr;
      break;
    // handle other cases
  }
  return err;
}
```

Back to top

Wake on Network Activity

New hardware (including iBook and Power Mac G4) has a new feature called "Wake on Network." This means that, based on user preference, certain types of network activity will cause a sleeping computer to awaken to respond to the network request. The user (via Energy Saver) or developer can specify two options insofar as what network activity can wake a sleeping machine (using the kWakeOnNetAdminAccessesBit and kWakeOnAllNetAccessesBit values described below). kConfigSupportsWakeOnNetBit is used only by power handlers for network interface cards in response to the kGetWakeOnNetInfo power management request where those cards support wake on network

activity.

```
    // Net Activity Wake Options
    enum
      {
      kConfigSupportsWakeOnNetBit   = 0, // current interface
supports wake on network
      kWakeOnNetAdminAccessesBit    = 1, // wake on network admin packet
      kWakeOnAllNetAccessesBit      = 2, // wake on any packet
      kUnmountServersBeforeSleepBit = 3, // disconnect from servers

      kConfigSupportsWakeOnNetMask  = (1<<kConfigSupportsWakeOnNetBit),
      kWakeOnNetAdminAccessesMask   = (1<<kWakeOnNetAdminAccessesBit),
      kWakeOnAllNetAccessesMask     = (1<<kWakeOnAllNetAccessesBit)
      kUnmountServersBeforeSleepMask=
(1<<kUnmountServersBeforeSleepBit);
      };
```

GetWakeOnNetworkOptions

`GetWakeOnNetworkOptions` can be used to retrieve what types of network accesses the system thinks are allowed to wake up the machine.

```
   OptionBits GetWakeOnNetworkOptions (void);
                  char * deviceType);
```

SetWakeOnNetworkOptions

`SetWakeOnNetworkOptions` can be used to instruct the system on what network events should allow the machine to be awakened for servicing. Pass a value with the `kWakeOnNetAdminAccessesBit` and the `kWakeOnAllNetAccessesBit` set to zero in the `inOptions` parameter to disable wake on network.

```
   OptionBits SetWakeOnNetworkOptions (OptionBits
inOptions);
```

Networking Drivers and Wake On Network

Drivers that control a network interface card should be sure to register a power handler with the Power Manager (see Device Power Handlers) so that they may respond to queries about whether they caused the system to be awakened.

The `kGetWakeOnNetInfo` message will be sent to see if the network device that is currently selected in fact supports waking on network activity.

The `kDeviceInitiatedWake` message will be sent to the networking device's power handler upon wakeup. The power handler should respond with a result indicating if the device which the power handler represents is responsible for waking the system. The power handler should return `kDeviceRequestsWakeToFull` if the system should be fully awakened to service the request and `kDeviceRequestsWakeToDoze` if the system should only be partially awakened to service the request. A partially awakened system is one that is put into the doze state instead of the full running state.

Back to top

Adding Power Sources

Power Manager 2.0 has the provision to control and interact with a number of devices such as AC,

batteries, or UPS devices that supply power to the system. As such, the Power Manager needs to be informed of the existence and status of all attached power sources to reliably provide the user important information, particularly if the system is running on battery or backup power alone (e.g., UPS).

The Power manager is made aware of the existence of these devices through the power source data structure (referred to simple as a power source).

Power Source data structure

The following describes the structure of a power source:

```
typedef SInt16 PowerSourceID;

 struct PowerSourceParamBlock
 {
 PowerSourceID sourceID;             // unique ID assigned by Power Mgr
 UInt16     sourceCapacityUsage;     // how currentCapacity is used
 UInt32     sourceVersion;           // use kVersionOnePowerSource
 OptionBits   sourceAttr;            // attributes (see below)
 OptionBits   sourceFlags;           // flags (see below)
 UInt32     currentCapacity;         // current capacity, in
                                     //  milliwatts, or as percentage
 UInt32     maxCapacity;             // full capacity, in milliwatts
 UInt32     timeRemaining;           // time left to deplete,
                                     //  in milliwatt-hours
 UInt32     timeToFullCharge;        // time to charge,
                                     //  in milliwatt-hours
 UInt32     voltage;                 // voltage in millivolts
 SInt32     current;                 // current in milliamperes
                                     // (negative if consuming,
                                     //  positive if charging)
 };
```

The power source can specify how the Power Manager should interpret the `currentCapacity` field of the power source relative to the `maxCapacity` field. The following `sourceCapacityUsage` constants have been defined:

```
 enum
 {
 kCapacityIsActual    = 0,           // capacity expressed in actual units
 kCapacityIsPercentOfMax = 1         // capacity expressed as percentage
                                     // of maxCapacity field
 };
```

The following power source attribute flags can be used:

```
enum
 {
 bSourceIsBattery   = 0,     // power source is battery
 bSourceIsAC     = 1,        // power source is AC
 bSourceCanBeCharged = 2,    // power source can be charged
 bSourceIsUPS     = 3        // power source is a UPS
 };
```

The following power source state flags can be used:

```
enum
 {
 bSourceIsAvailable  = 0,   // power source is installed
 bSourceIsCharging  = 1,   // power source being charged
 bChargerIsAttached  = 2,   // a charger is connected
 };
```

AddPowerSource

You can add a power source to the list of sources the Power Manager monitors by calling `AddPowerSource`.

```
 pascal OSStatus AddPowerSource (PowerSourceParamBlock *
ioSource);
```

The client will receive a unique `PowerSourceID` which can be used to manage this power source. `ioSource` is a parameter block containing information describing the power source. The Power Manager retains a local copy of this information in its internal list of power sources.

RemovePowerSource

If a device's power source is no longer available, you can tell the Power Manager to remove the source from its list using `RemovePowerSource` so that it is no longer a factor in the power summary calculation.

```
pascal OSStatus RemovePowerSource (PowerSourceID
inSourceID);
```

`inSourceID` is the ID of the power source to be removed. The routine will return `kNoSuchPowerSource` if the source with the specified `PowerSourceID` was not found in the Power Manager's list of power sources.

UpdatePowerSource

You can update a power source's vital statistics (power consumption rate, capacity remaining, etc.) by calling `UpdatePowerSource`. How the source information is obtained is internal to your software.

```
pascal OSStatus UpdatePowerSource (PowerSourceParamBlock *
ioSource);
```

`ioSource` contains the power source information, including the `PowerSourceID` assigned to your power source, which is to be updated. The routine will return `kNoSuchPowerSource` if the source with the specified `PowerSourceID` of this parameter block was not found in the Power Manager's list of power sources.

Example A simple UPS Power Source

Let's say the user attaches a UPS to the system that is capable of communicating its status to the Power Manager. The UPS driver software should register a power source with the Power Manager to indicate its existence:

```
static PowerSourceParamBlock mySource;

void RegisterUPSSource (void)
{
    OSStatus  status = noErr;

    mySource.sourceCapacityUsage    = kCapacityIsActual;
    mySource.sourceAttr             = (1<<bSourceIsUPS)
                    | (1<<bSourceIsAC)
                    | (1<<bSourceIsBattery);
               // note how we register that we're AC AND battery as
well as UPS

    mySource.sourceFlags            = (1<<bSourceIsAvailable);
    mySource.currentCapacity        = myCurrentCapacityInMilliWatts;
    mySource.maxCapacity            = myMaxCapacityInMilliWatts;
    mySource.timeRemaining          = secondsLeftToDischarge;
    mySource.timeToFullCharge       = secondsToFullChargeIfCharging;
    mySource.voltage                = myVoltageInMilliVolts;
    mySource.current                = -1234;
               // negative if consuming

    status = AddPowerSource (&mySource);
               // handle errors
}
```

When AC power is removed, the UPS can update the status by making the following change during its periodic update:

```
  void UpdateUPSSource (void)
{
    OSStatus  status = noErr;

    // Gather updated data
    if (NoACConnected())
      // note that we remove the AC attribute. This tells
      // the Power Manager to ignore internal AC and act like
      // were running off battery power
      mySource.sourceAttr &= ~(1<<bSourceIsAC);

      mySource.currentCapacity     = myCurrentCapacityInMilliWatts;
      mySource.timeRemaining       = secondsLeftToDischarge;
      mySource.timeToFullCharge    = secondsToFullChargeIfCharging;
      mySource.voltage             = myVoltageInMilliVolts;
      mySource.current             = -1234; // negative if consuming

      status = UpdatePowerSource (&mySource);
      // handle errors
}
```

When the UPS specifies that no AC is present, it is an indication to the Power Manager that any internal readings about the presence of AC are invalid and, hence, the Power Manager treats the system as though it were running off battery. As a result, under low-power conditions, the system will perform the same low-power actions that it does on PowerBooks with low battery power, that is, it will put the system into a low-power state. If the system supports deep sleep, it will try to put that system into deep sleep. If not, then the system is put into normal sleep unless it doesn't normally provide battery power (as is the case with desktops) in which case the system will be powered off. Note that Apple Events will be sent to active processes to indicate when a low-power condition exists and when low-power actions are imminent. Please see "Power Manager Apple Events" below for a description of these events.

Back to top

Obtaining Microprocessor Temperature

You can use the Power Manager to obtain the core temperature of the microprocessor or microprocessors on board.

```
SInt32 GetCoreProcessorTemperature (MPCpuID inCpuID);
```

The parameter, `inCpuID`, specifies from which processor the Power Mgr should get the core temperature. This ID can be obtained using the MP API (typically by calling `MPProcessors` to get the processor count and then using `MPGetNextCpuID` to iterate through the available processors). This method will work for both uniprocessor and multiprocessor systems.

The temperature is expressed in degrees centigrade. The result will be a positive value if the temperature was correctly obtained. If the result is negative, then an error is being returned.

This routine will return `kCantReportProcessorTemperature` if the hardware in question does not support reporting the core processor temperature or if MP services are not available.

Back to top

Power Manager Apple Events

The Power Manager now provides limited support for Apple Events. Currently, only four events are broadcast to all active processes that indicate they are high-level event aware.

Power Management Event Class

All events are broadcast using an event class of `kAEMacPowerMgtEvt`:

```
enum
   {
   kAEMacPowerMgtEvt    = 'pmgt'
   }
```

Power Management Event IDs

The following event ids have been defined:

```
 enum
   {
   kAEMacToWake            = 'wake',
   kAEMacLowPowerSaveData  = 'pmsd',
   kAEMacEmergencySleep    = 'emsl',
   kAEMacEmergencyShutdown = 'emsd'
   };
```

The `kAEMacToWake` message is sent whenever the Macintosh is waking up from sleep.

The `kAEMacLowPowerSaveData` message is sent whenever the Macintosh is experiencing main power loss and the backup power is sufficiently low to warrant sleep or shutdown in the near future. Applications should do what is necessary to save the state of the services they provide (by saving unsaved documents to a temporary file for future restoration if necessary).

Important:
Software should avoid posting alerts or dialogs for user interaction when handling this Apple Event, or, if they must, they should be sure those dialogs have a brief time-out associated with them. This is because the machine may be unattended and to get the full benefit of power loss handling in the OS, the system must be able to be gracefully put to sleep or powered off.

One of two emergency low-power action warnings is broadcast when the machine is about to run out of backup power. Which message is sent depends on the machine and the features it supports.

The `kAEMacEmergencySleep` message is sent just before the computer is to go to sleep or deep sleep due to very little backup power remaining. No user interaction is allowed during this message. The low-power action will occur within several seconds of the broadcast of this event.

The `kAEMacEmergencyShutdown` message is sent just before the computer is to be shutdown due to very little backup power remaining. Some models (such as Blue & White G3s and most iMacs) will shutdown instead of going to sleep. No user interaction is allowed during this message. The low-power action will occur within several seconds of the broadcast of this event.

Important:
If a particular Macintosh model supports deep sleep, it will enter that state instead of sleep or shutdown, if possible. If a model indicates it supports deep sleep, but that option is unavailable at the time of the low-power action, then the machine will be put to sleep if it has battery support (such as iBook or PowerBooks) or powered off if not. Software should usually treat `kAEMacEmergencySleep` and `kAEMacEmergencyShutdown` the same because while one might be broadcast, the nature of the system at the time when the action is taken might cause the opposite to actually be implemented. That is, even though the system broadcasts a `kAEMacEmergencySleep` message, the given context might force a shutdown instead.

Back to top

Server Mode

The new Power Manager provides two new routines to allow server mode operation. This mode, most useful on servers (!), allows the machine to be started automatically in the event of unexpected loss of AC power and then subsequent restoration of AC power.

```
pascal void EnableServerMode (Boolean inEnable);
```

You can use `EnableServerMode` to enable or disable the server mode feature. When enabled, the machine will be automatically started when AC power is provided (only after an unexpected loss of AC power).

```
pascal Boolean IsServerModeEnabled (void);
```

You can use `IsServerModeEnabled` to determine if server mode is currently enabled. A return value of TRUE means that it is enabled, and FALSE means it is not enabled.

Back to top

Further References

- Inside Macintosh: Devices (Chapter 6 - Power Manager)
- TN1039: Disk Access & Power Manager.
- TN1046: PowerMgr Addenda.
- TN1079: Power Management & Servers.
- TN1086: Power Management & The Energy Saver API

Back to top

Downloadables

 Acrobat version of this Note (K).

 Power Manager 2.0 DDK (Coming soon).

Back to top

To contact us, please use the Contact Us page.
Updated: 22-November-1999