Symbian OS Internals/15. Power Management

- Symbian OS Internals Table of Contents

by Carlos Freitas

All power corrupts, but we need the electricity.
Anon

Mobile phones are battery-powered devices. In the majority of cases, the battery is the only available source of energy - the exception being the times when the mobile phone is being recharged. Even with today's most advanced developments, rechargeable batteries are still characterized by:

- The limited amount of power they can supply at any given time
- The limited period for which they can supply electrical energy before exhaustion - the depletion of the active materials inside a cell must be replenished by recharging
- The hysteresis of the depletion--recharge cycle, which shortens the life-span of a battery.

So it is fair to say that the supply of power on a mobile device is quite constrained.

The problem is compounded by the need to keep the size and weight of battery components as small as possible. At the same time, new features are being added that use more power, or users are operating phones in power-consuming states (such as gaming or audio/video playback) for longer periods of time.

Because of this, mobile phone hardware has gained new energy-saving features, which require software monitoring and control. It is the primary goal of the operating system's power management architecture to define and implement strategies to use energy efficiently, to extend the useful life-time of the batteries, to increase the period of time for which the device can be used between recharges and at the same time, to allow the use of services required by the user of the device at any given time and at an acceptable level.

From the point of view of the phone's hardware components, there are a number of factors that the power management policy and implementation need to address, such as:

- Each hardware component's requirement on power resources at a given time
- The component's state with respect to power consumption and availability
- The component's transition time to a *more available* state (for executing both externally initiated tasks and background tasks), including the restoration of the status prior to the transition to the *less available* state
- The component's current workload and how this maps to the range of possible states
- The component's response time to an input that requires processing (such as the input from an interface port) whilst in each state and its ability to keep up with a sudden inflow of data.

Power management also deals with the operational state of the mobile phone as perceived by its user. That perception is primarily based on the availability of the user interface. The device is seen to be operational when the user can interact with the UI. On the other hand, the device is perceived to be unavailable when the UI seems to be unavailable, for example when the display is off or is displaying a screen saving image. The user expects the transition from unavailable to available to be fast and invariable.

The user also has the perception of an *off* state from which it takes a considerably longer time to return back to an *on* state.

The user perception of the operational state of the device may differ from the actual state of the hardware or the interpretation the operating system has of that state. For example, the user may perceive the device to be *off* when the screen is off, but background tasks or data transactions may well be going on. In this case, the device may be able to readily return to a perceived operational state with no loss of data and state. Equally, the phone can present itself to its user as fully operational, because the UI is active and available, while in fact significant portions of the hardware may be powered off, or only a fraction of the total processing bandwidth may be available for utilization.

In summary, the power management implementation is responsible for, on behalf of the operating system:

- Controlling the state and the power requirements of the hardware
- Extending the useful life of the battery component and the period the device can be used in between recharges
- Managing the user's perception of the phone operational state.

From this, it is clear that power management must be implemented at all levels of the operating system. Let me give a couple of examples:

1. There may be a UI-specific policy that decides to switch the display backlight or the display itself off after a period of user inactivity
2. A client of the services provided by an input port may decide to allow the controlling device driver to move the input port hardware to a low power state after a period of inactivity (no transactions through that port).

Symbian OS favors a distributed approach to power management, with components at different layers of the OS responsible both for managing their requirements on system power, and the impact of their actions on the availability of the phone. They achieve this in co-operation with other interdependent components, which can be at any level of the OS.

This chapter is mostly concerned with the implementation of power management at the kernel level of Symbian OS, and its interface to user-side components, but I will refer to other parts of the framework whenever necessary.

Power states

The kernel-level implementation of power management is responsible for managing the power state of hardware components such as the CPU and peripherals. The factors that are used to identify the state the component is in include:

- Its ability to retain data while in that state

- Its requirement on the level of system power resources, such as voltage, clock, current, and so on
- The response time to internal or external events while in that state
- Its transition time to the next *more available* state
- Its internal processing load at the time of transition to that state, and what parts of the component are involved.

The kernel-side framework sees a hardware component's power state as one of:

1. Off - moving a hardware component to this state is a result of removing the power supply to the component. Data and state are lost, the component's power consumption becomes negligible and it has no requirement on any power resource. The component will have no ability to respond to external events, and will have the longest transition time to any other state
2. Standby - in this power state, the hardware component may not have the ability to preserve data and state, but the power framework software can restore them when it transitions the component to a *more available* state, so long as it saved the previous state's status elsewhere before the transition to standby. The requirements on power resources can be significantly lowered, for one of two reasons - either no status preservation is required, or the component has a fully static operation and no internal processing takes place. The component may preserve some ability to detect and service external events, but the response time is generally long and will impact the component's performance
3. Retention - in this state the requirements on power resources are reduced to only those necessary to preserve the component's data, internal state and the ability to detect external or generate internal events. The component is not performing any active tasks, it is not involved in any data transactions and no internal processing is taking place. The response times to external or internal events, as well as the transition time to a state where the events can be processed, may be long enough to impact the system's performance
4. Idle - this is usually a transitory state, or a mode. The hardware component has finished processing a task and no request for further service has been placed on it. No connected component is acting on any inputs, nor are new internal events being generated. The component has the capability to respond to new events and process them. Its state and data are preserved. Its power consumption may be reduced, but that does not lead to lowering the requirements it has on system power resources. Depending on a number of circumstances which I will explain later in this chapter, the power management framework may move the component to a lower power state or keep it in this state until a request for processing data or events is placed upon it (by the OS or by a connected component)
5. Active - a hardware component in this state is processing tasks, data or events generated internally and received from its inputs. Its requirement on power resources is as high as necessary to guarantee that this processing takes place at the level of availability required. It may have pending tasks or requests for processing.

Not all hardware components support all these power states, and some components are capable of intermediate states, usually different variations of the retention state.

Transitions between power states may be triggered by user actions, requests from the clients of the services provided by these components, the need to save power, changes to the state of power resources used by the component, and so on. These states apply to the CPU and peripherals independently, so it is possible that, at a given time, different hardware components of a phone will be in different power states.

For example, it is possible that the CPU might enter the idle state if it has no scheduled tasks and there are no anticipated events requiring its attention, while a peripheral has an outstanding request for servicing incoming data. The OS power management must be able to make a decision to either move the CPU to a lower power mode or to leave it in its present state. The decision depends on, for example, the response time to transition the CPU back to the active state and service any requests issued by the peripheral, and is also based on the permissible degradation of service provided by the peripheral for the request it is servicing.

Transitioning the CPU to and from some of these states has an impact on the rest of the system. For instance, transitioning the CPU to the off or standby state will result in open applications being terminated with loss of state and data (unless it is saved elsewhere). On returning from standby to the active state, the same applications must be restored to their previous context.

Given their wider impact, in Symbian OS we consider off, standby and active to be system-wide states, and the transitions between those states, system-wide transitions.

Peripheral transitions to low power states that are not the result of a system-wide transition may need to be agreed with their clients at other levels of the OS, given the possible degradation of the quality of service they provide to those clients.

Power framework

The kernel power framework is responsible for:

- Managing the transitions of processor core and peripherals between power states
- Making use of the energy-saving features of the hardware, detecting and responding to events which may trigger power state transitions
- Managing the hardware components' requirements on system power resources.

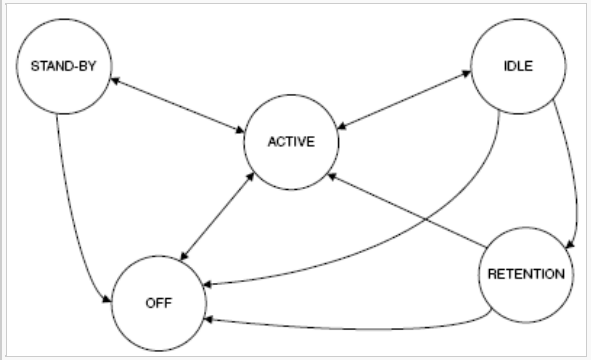

Figure 15.1 *Typical CPU state transition diagram*

The implementation of the framework straddles several software components, including the kernel, device and media drivers, extensions and peripheral controllers, the base port, the hardware adaptation layer (HAL).

The framework is made up of a built-in basic framework and mandatory and optional portions that must be implemented as part of the port of Symbian OS to a new hardware platform (Figure 15.1).

Basic framework

The basic power management framework is primarily concerned with system-wide power transitions - in other words, transitioning the processor core and peripherals as a group between the off, standby and active power states.

Power manager

The power manager is at the core of the framework and provides the API between the user and kernel levels of the framework. It co-ordinates the transition of the CPU, peripherals and hardware platform between the various system-wide power states, provides the interface to other parts of the kernel, and coordinates the interactions of the other components of the framework.

Power controller

The framework uses the variant-specific power controller to control the power behavior of CPU and other parts of the hardware platform. The power controller may also provide a way of controlling the power resources of the platform.

Power handlers

The framework uses power handlers to control the power behavior of peripherals. This means that power handlers are associated with peripheral drivers. The implementation of power handlers may be customized at the driver level and/or at the variant level.

Wakeup events

The basic power management framework provides support for wakeup events. These are hardware events specific to each low power state, which, if they occur during a system-wide power transition, may result in the transition being cancelled or even reverted. If they occur during standby, they may trigger a return to the active state. Wakeup events for the standby or off states usually relate to user interactions or timed sources. The framework provides support for tracking these events (at peripheral driver or platform-specific levels) and notifying the user-side software component responsible for initiating the system-wide power transition of their occurrence.

CPU idle

The basic framework provides support for transitioning the CPU in and out of idle mode. The transition to this mode is triggered when there are no threads ready to run, and the kernel schedules the null (also known as the idle) thread as a result. In specific circumstances, some CPUs can be moved to a more power-saving retention state, and that would be handled by the platform-specific implementation of the idle mode handling.

Power HAL

There is a power-related group of functions that can be executed kernel-side in response to a call to the HAL class's Get(...) or Set(...) APIs (or for certain functions, through calling UserHal or UserSvr class APIs). This group is identified by the enumerated value `EHalGroupPower`. The framework should also provide an object to handle this group of functions. This group of functions allow user-side components to gain access to the kernel power framework to obtain certain information or set the power behavior of selected hardware components.

Basic power model overview

The basic power framework only gives external visibility to the system-wide power states, which I will now enumerate:

```
enum TPowerState
    {
    EPwActive,
    EPwStandby,
    EPwOff,
    EPwLimit,
    };
```

The model relies on a user-side component to initiate the transitions to standby and off states. There should only be one such component in the system and it must have power management capabilities (for more on capabilities, see Chapter 8, *Platform Security*). This component is currently the shutdown server but that may change in the future to be the domain manager - see Chapter 16, *Boot Processes*, for more on this.

The kernel power framework was developed with the interface exported by the domain manager in mind (Figure 15.2), so I will assume throughout this chapter that the domain manager is the user-side component responsible for initiating system-wide transitions. In the majority of cases, the behavior of the kernel framework will be the same if the shutdown server is used instead - I will describe any exceptions where relevant.

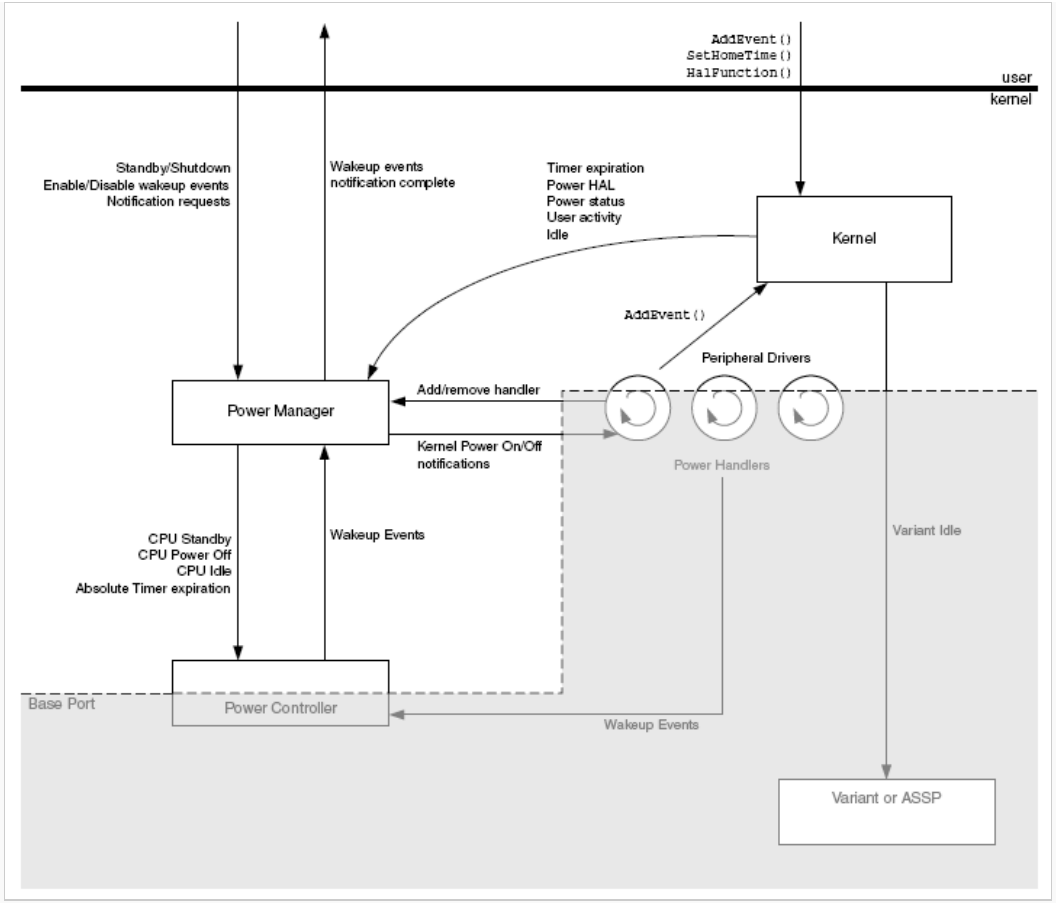

Figure 15.2 *Basic framework block diagram*

Transitions to standby or off are not instantaneous - from the moment the user requests the shutting down of the phone, until the framework is requested to perform the transition at kernel level there is typically a lengthy preparation phase in which UI state and application data are saved, and applications are shut down. We therefore want wakeup events that are detected kernel-side to be communicated to the initiator of the transition during the preparation phase. This component may on receiving a wakeup event, cancel or reverse the preparations for the system-wide transition - restoring the previous state of UI and applications.

Wakeup events are hardware-specific, and the kernel-level part of the framework maps a set of events to each target low power state. The shutdown server or domain manager must be able to set a target low power state for a system-wide transition and enable the wakeup events for that state. It must also be able to request notification of their occurrence, and in time, request the kernel framework to transition to that state.

When deciding to stop or reverse the preparations to a system-wide transition to a low power state, the initiator of the system-wide transition must be able to request the disabling of wakeup events for the previous target low power state, and set the target state to active. It must also be able to cancel the request for notification of wakeup events.

Once the kernel-side power framework has initiated a transition, the user-side initiator cannot stop that transition - although a wakeup event may still prevent it taking place.

The power manager manages the kernel-side transitions.

All of the user-side requests that I've mentioned are routed to the power manager. This receives a request to power off or go to standby state, and dispatches notifications to other components that manage the transition of CPU and peripherals to those states.

Peripheral drivers for peripherals that need to be powered down as a result of a system-wide transition to standby or off must own a power handler. When these peripheral drivers are started, they need to register with the power manager for notifications of system-wide power transitions - the power manager keeps a list of registered power handlers. When the peripheral driver object is destroyed, it should de-register its power handler with the power manager.

The power manager notifies every registered peripheral driver of an imminent power down through its power handler. Upon receiving these notifications, peripheral drivers should change the power state of the peripheral they control so as not to compromise the eventual system-wide power transition that is taking place.

As peripheral power down may take some time, each power handler owns a fast semaphore, which the power manager waits on, after requesting it to power down the peripheral. This semaphore is signaled upon completion of the peripheral power down.

After all peripherals have powered down, the power manager should request the CPU to power down. To do this, it calls down to the power controller.

If the target state of the system-wide power transition is off, instruction execution terminates soon after the call to the power controller is issued. If the target state is standby, the CPU is eventually brought back to the active state when a wakeup event occurs. Instruction execution is resumed inside the power controller call, and then control is returned to the power manager, which then powers up all peripheral drivers owning a registered power handler, and waits for them to power up, in a sequence that is the reverse of the power down that I explained previously.

Wakeup events may also occur during the user-side transition, and if they are enabled, should be propagated up to the component that initiated that transition.

Wakeup events are monitored at the variant-specific level, so every request to enable or disable them should be propagated down to the power controller.

Each system-wide low power state (standby and off) may have a different set of wakeup events. So, if the domain manager requests the enabling of wakeup events when the target state is already a low power state the power manager will disable the set corresponding to the previous low power state, before enabling the set corresponding to the new low power state. If the domain manager requests the disabling of wakeup events, the power manager assumes that it decided to stop or reverse the transition, so it is sets the target state to active.

The power controller may monitor wakeup events directly, or delegate this to a peripheral driver. In the latter case, the peripheral driver must notify the power controller of the occurrence of a wakeup event, and the power controller then propagates the notification to the power manager, which completes any pending user-side request for notification.

If the target low power state of a system-wide transition is standby, and a wakeup event happens after the kernel framework is requested to transition, but before the CPU is moved to that state, then the implementation should not complete the transition. If no event occurs, it will return when a detected wakeup event finally occurs.

Another important function of the kernel power framework is to detect the moment when the CPU idles. This can be used to move the CPU and platform to a power-saving state. Such decisions must be taken at variant-specific level, and therefore must involve the power controller.

The kernel notifies the power manager every time the null thread is scheduled to run. A power manager implementation calls down to the power controller's platform-specific implementation, which may decide to move the CPU to a low power retention state, possibly in cooperation with other components such as peripheral drivers.

There is an alternative mechanism to allow user-side components to communicate with the kernel-side framework - the HAL. This component provides APIs that any user-side component with power management capabilities may call to obtain information on power supplies and control the power behavior of certain peripherals (display, pointing devices, external case or flip, and so on).

Finally, the framework may include a battery-monitoring component, which is implemented at variant level. I will discuss this in greater detail later in the chapter.

Initialization

Early on during kernel boot (see Chapter 16, *Boot Processes*), when the microkernel is initialized, the global power manager object is created. As a power manager must own a pointer to a power controller, a dummy power controller is also created. The power manager will later replace this pointer with a pointer to a real power controller.

The power controller is typically implemented in a kernel extension; when this extension is started the power controller is created and registers with the power manager. Registering results in the power manager replacing the dummy pointer with a pointer to the real power controller.

It also sets up a global pointer in the kernel, K::PowerModel, to point to the power manager, in this way providing the kernel with the means to access the power framework.

The base porter may not want to have a power-controller-specific kernel extension, but instead have the Variant object, part of the variant DLL, create and own one.

The power controller extension entry point will usually create a platform-specific handling object for the `EHalGroupPower` group of functions and register it with the power manager.

Typically, it will also create a battery monitor component at this stage, if one is to be implemented.

Finally, as the kernel starts each peripheral driver, it will register its power handler with the power manager.

API description

Let's now look at each component and its public exported interface in detail.

The user-side interface The basic power management framework can be accessed from user-side via the Power class. This class provides static methods for enabling and disabling wakeup events, requesting or canceling notification of occurrence of wakeup events and moving the kernel-side components to one of the low power states, standby or off.

```
class Power
    {
public:
    IMPORT_C static TInt EnableWakeupEvents(TPowerState);
    IMPORT_C static void DisableWakeupEvents();
    IMPORT_C static void RequestWakeupEventNotification(TRequestStatus&);
    IMPORT_C static void CancelWakeupEventNotification();
    IMPORT_C static TInt PowerDown();
    };
```

All of these functions are exported by EUSER, and gain access to the kernel-side power framework in the usual way, via executive calls. Here is a description of the public API:

- `EnableWakeupEvents()`. This function is used to set the target low power state for a system-wide transition and to enable wakeup events for that state. If the target state is neither `EPwStandby` nor `EPwOff`, it returns `KErrArgument`
- `DisableWakeupEvents()`. This function is used to disable wakeup events for the current target low power state for the system-wide transition in progress. If the current target power state is neither `EPwstandby` nor `EPwoff`, the call returns immediately
- `RequestWakeupEventNotification()`. This is the only asynchronous function; it is used to request notification of any wakeup events that happen during the preparation to transition the system to a low power state, or after the system has entered standby. Only one pending request is allowed at a time - if another request is already pending, the function returns `KErrInUse`
- `CancelWakeupEventNotification()`. This call is used to cancel a pending wakeup event notification request. If, at the time this function is called, the notification request is still pending, then it returns `KErrCancel`
- `PowerDown()`. This function requests the kernel framework to move the CPU and peripherals to a low power state. If the target low power state is standby, this function returns when a wakeup event occurs. If the target low power state is off, this call never returns.

The power manager The power manager has no public exported APIs. The kernel-level power management framework offers an abstract class (DPowerModel) as a template for the implementation of a power manager:

```
class DPowerModel : public DBase
    {
public:
    virtual void AbsoluteTimerExpired() = 0;
    virtual void RegisterUserActivity(const TRawEvent& anEvent) = 0;
    virtual void CpuIdle() = 0;
    virtual void SystemTimeChanged(TInt anOldTime, TInt aNewTime) = 0;
    virtual TSupplyStatus MachinePowerStatus() = 0;
    virtual TInt PowerHalFunction(TInt aFunction, TAny* a1, TAny* a2) = 0;
    };
```

This class defines the interface between the power framework and the rest of the kernel. It mandates a number of functions that should be implemented by a power manager. The kernel uses the global pointer I mentioned earlier, K::PowerModel, to call these functions. Here is a description of the `DPowerModelAPI`:

- `CpuIdle()`. The kernel calls this function every time the null thread is scheduled to run
- `RegisterUserActivity()`. The kernel calls this function every time an event is added to the event queue. Peripheral drivers that monitor user interaction (such as pressing a key, tapping the touch screen, opening or closing the phone) may add events kernel-side. A user-side component may also add events using the userSvr API `AddEvent()`, which is exported from EUSER.DLL. The function takes a reference to the raw event as a parameter, so a power manager implementation may choose to respond differently to different events
- `PowerHalFunction()`. The kernel's HAL function that handles `EHalGroupPower` calls this function, passing an identifier to the function to be executed. The power manager implementation should call a platform-specific handling function
- `AbsoluteTimerExpired()`. The kernel calls this function every time an absolute timer completes. (An absolute timer is one that expires at a specific date and time.) A power manager implementation should call a power controller's platform-specific implementation, which may regard it as a wakeup event for an impending system-wide transition
- `SystemTimeChanged()`. The kernel calls this function every time the software RTC (and eventually the hardware RTC, if one exists) is updated in response to a call to `user::SetHomeTime()`
- `MachinePowerStatus()`. The kernel calls this function whenever the framework's exported API `Kern::MachinePowerStatus()` is called. `MachinePowerStatus()` should query the battery monitoring component if one is implemented kernel-side. The current implementation of the power manager in Symbian OS also offers:
- A kernel-side implementation of the corresponding user-side Power class APIs
- Management of and interface to power handlers
- Management of and interface to the power controller.

Here's the make up of the current Symbian OS power manager:

```
class DPowerManager : public DPowerModel
    {
public:
    void CpuIdle();
    void RegisterUserActivity(const TRawEvent& anEvent);
    TInt PowerHalFunction(TInt aFunction, TAny* a1, TAny* a2);
    void AbsoluteTimerExpired();
    void SystemTimeChanged(TInt anOldTime, TInt aNewTime);
    TSupplyStatus MachinePowerStatus();
public:
    static DPowerManager* New();
    TInt EnableWakeupEvents(TPowerState);
    void DisableWakeupEvents();
    void RequestWakeupEventNotification(TRequestStatus*);
    void CancelWakeupEventNotification();
    TInt PowerDown();
    void AppendHandler(DPowerHandler*);
    void RemoveHandler(DPowerHandler*);
    void WakeupEvent();
    ...
    };
```

Kernel-side implementation of user-side API The following methods are called in response to corresponding Power class calls:

- `DPowerManager::EnableWakeupEvents()` enables tracking of wakeup events for a valid target low power state (standby or off)
- `DPowerManager::DisableWakeupEvents()` disables tracking of wakeup events for the target low power state
- `DPowerManager::RequestWakeupEventNotification()` enables the delivery of wakeup event notifications to the client that requested it, whenever one occurs
- `DPowerManager::CancelWakeupEventNotification()` stops the power manager from delivering wakeup event notifications to the client that requested them
- `DPowerManager::PowerDown()` initiates the kernel-side transition of CPU and peripherals to the target low power state. If the target state is standby, when a wakeup event arrives, it delivers a notification to the client if a request is pending.

These functions need access to the platform-specific powercontroller, which is protected against concurrent access and re-entrance with a mutex. Therefore, the corresponding Power class functions execute inside a critical section to prevent the calling thread that holds the mutex from being suspended or killed.

Management and interface to power handlers

The `DPowerManager::AppendHandler` API adds the power handler to the list of controlled objects, and the `DPowerManager::RemoveHandlerAPI` removes it.

Management and interface to power controller

`DPowerManager::WakeupEvent()` checks if the power state is valid and completes any pending client's request for wakeup event notification.

The power handler

The `DPowerHandler` class is intended for derivation. The software component that owns the power handler must implement the pure virtual functions and may include other APIs (for example, to allow the handler to request power related resources):

```
class DPowerHandler : public DBase
    {
public:
    // to be implemented by kernel-side power framework
```

```
    IMPORT_C ~DPowerHandler();
    IMPORT_C DPowerHandler(const TDesC& aName);
    IMPORT_C void Add();
    IMPORT_C void Remove();
    IMPORT_C void PowerUpDone();
    IMPORT_C void PowerDownDone();
    IMPORT_C void SetCurrentConsumption(TInt aCurrent);
    IMPORT_C void DeltaCurrentConsumption(TInt aCurrent);

public: // to be implemented at component-specific level
    virtual void PowerDown(TPowerState) = 0;
    virtual void PowerUp() = 0;
    ...
    };
```

The APIs (exported from EKERN.EXE) with a default implementation are:

- A constructor to allow the creation of power handler objects owned by peripheral drivers. The constructor simply sets the name for this power handler from the argument passed in. (The name is only used for debug purposes.) Typically the peripheral driver-specific derived constructor will set up other relevant parameters
- `Add()`. Called by the component that owns the power handler to add it to the list of power handlers that receive notifications of power state changes. Calls `DPowerManager::AppendHandler()`
- `Remove()`. Called by the component that owns the power handler to remove it from the list of power handlers that receive notifications of power state changes. Calls `DPowerManager::RemoveHandler()`. Like the `Add()`, this function acquires a mutex that is also held by the implementation of `PowerUp()` and `PowerDown()`. Hence, the device driver writer must guarantee these calls are issued from inside a critical section to prevent the calling thread from being suspended or killed when owning a mutex
- `PowerDownDone()`. This is called by the component that owns the power handler, after it has performed all the required internal actions to guarantee that the system-wide power transition that is taking place can be accomplished
- `PowerUpDone()`. This is called by the component that owns the power handler, after it has performed all the required internal actions to guarantee that the system-wide transition that is taking place can be accomplished
- `SetCurrentConsumption()` and `DeltaCurrentConsumption()`. These APIs have been deprecated and should not be used
- A destructor to allow destruction from peripheral driver code.

Ownership of power handlers Next I will discuss the ownership of power handlers. The following kernel-side software components may own power handlers:

- Kernel extensions that control simple peripherals (for example display, digitizer and keyboard) and which can be accessed from user-side through the HAL or through unique kernel interfaces, and peripherals which provide services to other peripherals (for example, an internal inter-component bus) usually enforce a policy of having a single client at a time and therefore may own, or in the majority of cases derive from, `DPowerHandler`
- Device drivers may control more than one unit of the same type of peripheral and so they allow multiple simultaneous clients or channels (one per unit). In this case the channel usually owns the power handler. If you are implementing an LDD/PDD split, then the logical (or physical) channel object will create and own a pointer to the power handler
- Some device drivers may enforce a single channel policy. In this case, the logical device may own the power handler
- Peripheral bus controllers are used to extend access to the system bus (or any bus internal to the device) to external peripherals. They may be able to support multiple physical interfaces, in which case each interface implementation should own a power handler. Examples of these are the PCMCIA and MMC/SD/SDIO bus controllers
- In other instances, controllers only support one physical interface but multiple logical instances. In this case the controller itself will own the power handler - this is the case of the USB controller.

The power controller The power controller object must derive from the `DPowerController` class. This class provides APIs for initiating CPU-specific preparations for going to low power states, enabling/disabling tracking of wakeup events at platform-specific level and allowing peripheral drivers to notify the occurrence of wakeup events that they track:

```
class DPowerController : public DBase
    {
public: // Framework
    IMPORT_C DPowerController();
    IMPORT_C void Register();
    IMPORT_C void WakeupEvent();
    ...
public: // Platform-specific power component
    virtual void Cpuidle() = 0;
    virtual void EnableWakeupEvents() = 0;
    virtual void DisableWakeupEvents() = 0;
    virtual void AbsoluteTimerExpired() = 0;
    virtual void PowerDown(TTimeK aWakeupTime) = 0;
    ...
    };
```

The APIs (exported from EKERN.EXE) with default implementation are:

- A constructor to allow a platform-specific component (such as the variant DLL or the power controller kernel extension) to create a power controller. The default implementation sets the power controller power state to active. The platform-specific constructor will usually register the power controller with the power manager
- `Register()`. This API registers a power controller with the power manager. The implementation of `Register()` replaces the power controller object pointer with a pointer to this one
- `WakeupEvent()`. Calls the power manager to notify it of a wakeup event.

The power HAL handler This is the prototype of a platform-specific handling object for the power HAL group of functions:

```
class DPowerHal : public DBase
    {
public:
    IMPORT_C DPowerHal();
    IMPORT_C void Register();
public:
    virtual TInt PowerHalFunction(TInt aFunction, TAny* a1, TAny* a2) = 0;
    };
```

The APIs (exported from EKERN.EXE) with default implementation are:

- An exported constructor to allow a platform-specific component to create a handling object
- `Register()`. This registers the power handling function with the power manager.

Walkthrough of user-initiated shutdown

Now that I have explained the role of each component of the framework and their APIs, let's look at how they are used on a user-initiated transition to standby or off.

The shutting down of the phone is typically triggered by the user pressing a power button. In other cases, defined by a UI policy, it may be triggered by a user inactivity timer. These events are detected at kernel level and propagated to the user-side component that manages the system shutdown, currently the shutdown server.

The shutdown server starts the transition by notifying active applications that a transition is imminent, allowing them to save status and shut down.

It is only after all this is done that the shutdown server requests the kernel framework to transition the CPU, peripherals and the hardware platform to the target state.

The reverse applies to the transition from standby to active, with the CPU and peripherals transitioning first, and then a notification generated at the kernel framework level being propagated upwards to the shutdown server which is responsible for transitioning the rest of the system.

I will describe the processes of shutting down and restarting and the user-level components involved in more detail in the next chapter, *Boot Processes*.

Let us now look at the sequence of events in the kernel level framework during a transition to standby or off.

The way in which the user-side shutdown initiator hooks into the kernel framework varies, with the shutdown server calling `UserHal::SwitchOff()` (see Section 15.2.3.6) which then calls the Power class APIs, and the domain manager calling those APIs directly. In either case, the calls are always made in the sequence I will now describe.

The sequence starts with a call to `Power::EnableWakeupEvents()`, passing the target low power state as an argument (`EPwstandby` or `EPwoff`). This goes through an executive call to the kernel in a critical section, and ends up in the power manager. The power manager sets the target state, and then calls the derived `DPowerController` object's platform-specific implementation of `EnableWakeupEvents()`. As I have said before, this will either enable wakeup events directly in hardware, or call to relevant drivers.

The next function to be called in the sequence is `Power::RequestWakeupEventNotification()`. This goes through an executive call, inside a critical section, to the power manager. The power manager simply saves the pointers to the `TRequestStatus` object and the requester client thread.

If the power manager receives a notification that a wakeup event has occurred at any point during the transition to a low power state, or after the transition to standby, it uses the pointers to the request status object and the client thread to complete the request (with `KErrNone`).

The final function in the sequence is `Power::PowerDown()` which initiates the kernel-side shutdown. Again this goes through an executive call, inside a critical section, to the power manager. The power manager performs the following sequence:

1. Notifies every registered power handler of power down, by calling the driver-specific implementation of `DPowerHandler::PowerDown()` and passing the target power state. The driver-specific implementation may shut down the peripheral removing its power source (if the target state is off) or move it to a low power state, relinquishing its use on power resources, and possibly leaving some of its subsystems operational for detection of wakeup events (if the target state is standby)
2. Waits for all the power handlers to complete powering down. Completion is signaled by the peripheral driver calling `DPowerHandler::PowerDownDone()`
3. Acquires the tick queue mutex to stop tick timers being updated
4. Calls `DPowerController::PowerDown()`, passing the time for the next absolute timer expiration (in system ticks)
5. The platform-specific power controller function prepares the CPU for, and transitions it to, the low power state. If the target state is off, instruction execution terminates. If the target state is standby, execution is halted until a wakeup event occurs or an absolute timer expires, when execution resumes, the power controller restores the state of the CPU and core peripherals and control returns back to the power manager
6. Back in the power manager, it is safe to set the power state to active. The power manager wakes up the second queue (a different queue, used for second-based timers and driven by the tick queue), which will resynchronize the system time with the hardware RTC. If waking up on an absolute timer, this will queue a DFC to call back any timers which have expired and restart second queue. The power manager releases the timer mutex
7. At this point the power manager notifies all registered power handlers of the transition to the active state by calling the driver-specific implementation of `DPowerHandler::PowerUp()`. This may restore the peripheral state and power it up
8. As before, the power manager waits for all power handlers to finish powering up, which is signaled by the peripheral driver calling `DPowerHandler::PowerUpDone()`
9. Finally, the power manager simply completes the request for notification of wakeup events if one is pending.

Remapping standby state to off state

As I mentioned before, the shutdown server is currently the user-side component responsible for initiating a shutdown. It does that by calling `UserHal::SwitchOff()` which requests the kernel framework to transition to standby. It is likely that the device creator will want the shutdown sequence to end in the power supply to the CPU and peripherals being removed to prolong the life of the device's battery - unless the device includes a backup battery which could be used to power the self-refreshing SDRAM in standby state. In this case, the base porter may remap the standby state to off.

Customizing the basic framework

The framework includes several abstract classes that are intended as prototypes for platform-specific (or driver-specific) components. Those porting Symbian OS to new hardware would implement these framework components as part of the base port, or in the case of power handlers, as peripheral drivers.

This customization of the framework implements mandatory functions that deal with:

1. Peripheral transitions when a system-wide transition occurs, to or from the standby state, or to the off state
2. CPU transitions when a system-wide transition occurs, to or from the standby state, or to the off state
3. CPU transitions to and from the idle mode
4. Tracking of standby wakeup events
5. Handling of power-related HAL functions.

Peripheral power down and power up

A DPowerHandler class derived object requires the following functions to be implemented:

- PowerDown(). This requests peripheral power down. The power manager calls this function during a transition to the standby or off state
- PowerUp(). This notifies the peripheral of a system power up. The power manager calls the power handler's PowerUp() when returning from standby back to the active state.

After receiving a request to power down, a peripheral driver should execute the necessary actions to power down the peripheral and ancillary hardware (unless it is required for detection of wakeup events and the target state is standby). This may include requesting the removal of the power supply, and also releasing the requirements on other power resources such as clocks and power supplies.

After this is done, the driver should signal to the power manager that the peripheral has powered down by calling the power handler's PowerDownDone() method.

After it receives notification of system power up, a peripheral driver may decide to power up the peripheral and ancillary hardware. The decision depends on the internal operational state of the peripheral driver before the transition to standby. The peripheral driver should also signal to the power manager that the call has completed by calling the power handler's PowerUpDone() method.

PowerDown() and PowerUp() are called in the context of the user-side component that requested the system-wide transition. Power- DownDone() and PowerUpDone() can be called from that same thread, or from the peripheral driver's thread (before or after the corresponding PowerDown() or PowerUp() functions return).

Note that PowerUp() and PowerDown() are only used on transitions to and from the standby or active states or transitions to off state. The peripheral hardware is typically powered up on opening a peripheral driver and down on closing it, and its power state changes when the driver uses or releases it - and all of this should be fully managed by the driver software.

CPU power down and power up

A DPowerController class derived object requires an implementation of PowerDown() which deals with the CPU transition between the standby, active and off states. The power manager calls the power controller's PowerDown() function to move the CPU to a low power state. PowerDown() runs in the context of the shutdown server (or domain manager). If one or more wakeup events occur during execution of the call, but before the power state is entered, the PowerDown() call should return immediately.

PowerDown() takes an argument (aWakeupTime), which is a system time value; if it is not null and the target state is standby, it specifies the time when the system should wakeup. This is the time when the next absolute timer will expire. Typically the implementation starts by checking that this time is in the future, and then programs the RTC (real time clock) module to generate an event at the specified time, which will cause a return to the active state. For this to happen, the call should enable RTC event detection during standby. The implementation of PowerDown() must make sure that setting the RTC to wakeup in the future will not cause it to wrap around, as the maintenance of the system time depends on the knowledge of when this happens. In this case, the RTC should wakeup the CPU just before it is about to wrap.

If aWakeupTime is null, the system will only wake up from standby when a wakeup event occurs. When this happens, the CPU wakes up and the PowerDown() function resumes and restores the status that was saved before entering standby. At that point, there is no need to call WakeupEvent() - upon returning from this function the power manager will notify any client which requested notification of wakeup events.

If the target state is off, then PowerDown() will never return. Usually the power controller turns off the CPU power supply.

Preparation to go to standby state In the standby state, the CPU's and core peripherals' clocks and even their power supplies, may be suppressed. This means that their internal state is not preserved. In this case, PowerDown() should save this internal state, so that it can be restored when the system wakes up. This is done as follows:

- CPU state. Saves all registers (on ARM - the current mode, banked registers for each mode, and stack pointer for both the current mode and user mode)
- MMU state. On ARM saves the control register, translation table base address, domain access control (if supported)
- Flushes the data cache and drains the write buffer
- Core peripherals. Saves the state of interrupt controller, I/O pin function controller, external (memory) bus state controller, clock controller, and so on.

When this data is saved to SDRAM, PowerDown() should place the device in self-refresh mode. If the SDRAM device allows partial bank refresh, and support has been implemented to query bank usage, PowerDown() can set the used banks to self-refresh, and power down unused banks of memory. Obviously this uses less power. Usually PowerDown() would leave peripheral subsystems that are involved in the detection of wakeup events powered and capable of detection. PowerDown() should also disable tick timer events and save the current count of this and any other system timers; it should enable any relevant wakeup events, and disable any others. On entering the standby state, instruction execution halts. PowerDown() can do this simply by stopping the CPU clock, if this has used a fully static architecture. A wakeup event will restart the CPU clock, and execution resumes. On returning from standby state, when PowerDown() resumes execution, it should restore the CPU and core peripherals' state that it saved prior to going to standby.

CPU idle

A DPowerController class derived object requires an implementation of CpuIdle(), which deals with CPU transition to idle state.

The idle state is a transitional state, often the gateway to a power-saving retention mode. In Section 15.2.2.1, I will look at how the CPU can be moved to these retention states.

Variant-specific idle As I mentioned previously, the scheduling of the null thread is what signals the CPU idle condition. The null thread is the first thread to start on a device at boot time, and it runs before the power manager has been registered with the kernel. Therefore, an alternative to the power manager's own CpuIdle() function must be provided, as a pure virtual method of the Asic class:

```
class Asic
    {
public:
    ...
    // power management
    virtual void Idle()=0;
    ...
    };
```

This function is typically a dummy implementation, provided by the Asic class derived Variant class. Once the power manager has been registered, the kernel will call its `CpuIdle()` function instead.

Enabling access to power controller from other kernel-side components

It is common that other kernel-side components such as the variant, or peripheral drivers, need access to the power controller. This has no built-in accessible interfaces, other than to the power manager. When porting Symbian OS, the base porter may therefore wish to implement a derived power controller exported method to return a pointer to itself and an interface class, in this way:

```
class TXXXPowerControllerInterface
    {
public:
    ...
    // to allow Drivers access to power controller
    IMPORT_C static PowerController* PowerController();
    inline static void RegisterPowerController(
    DXXXPowerController* aPowerController)
    {iPowerController=aPowerController;}
public:
    ...
    static DXXXPowerController* iPowerController;
    };

EXPORT_C DXXXPowerController* TXXXPowerControllerInterface::PowerController()
    {
    return &iPowerController;
    }
```

The power controller derived object's constructor should register the power controller with the interface, which is best done at construction time:

```
DXXXPowerController::DXXXPowerController()
    {
    Register(); //register power ctrllr with power manager

    // register power controller with interface
    TXXXPowerController::RegisterPowerController(this);
    }
```

Handling of wakeup events

When the CPU and peripherals move to the standby state, their responsiveness and availability are greatly reduced. This is accepted by the user who has chosen to switch the phone off and the framework uses that acceptance to save power.

However, at the OS level, we need to enable a minimum capability to respond to user interactions, so that the framework can transition the phone back to a more available state when the user switches the phone back on. Also, some internal events, such as expiry of absolute timers, must be able to bring the phone back to a more available state. A `DPowerController`-derived object should implement the following pure virtual functions to handle wakeup events:

- `EnableWakeupEvents()`
- `DisableWakeupEvents()`
- `AbsoluteTimerExpired()`.

EnableWakeupEvents() Typically, the domain manager (or shutdown server) will start a transition to standby by requesting the kernel power framework to start monitoring wakeup events and notify it of their occurrence. As a result, the power manager calls the power controller's `EnableWakeupEvents()` to enable detection at platform-specific level.

Monitoring wakeup events The power controller may monitor some wakeup events directly. If that is the case, the implementation of `EnableWakeupEvents()` programs the hardware components involved in their detection, and hooks a handling function to service the event. This is commonly achieved with the use of an interrupt - the ISR should schedule a DFC to notify the power manager of the event. More commonly, peripheral drivers monitor wakeup events. In this case, the implementation of `EnableWakeupEvents()` should store whether the event is enabled, like so:

```
class DXXXPowerController : public DPowerController
    {
public: // from DPowerController
    ...
    void EnableWakeupEvents();
    void AbsoluteTimerExpired();
    void DisableWakeupEvents();
    ...
public:
    DXXXPowerController();
    ...
private:
    TInt iWakeupEventMask;
    ...
    };

void DXXXPowerController::EnableWakeupEvents()
    {
    ...
    // Set iWakeUpMask to a bit mask with one bit set for
    // each relevant wakeup event for the standby state
    if(iTargetState==EPwstandby)
        iWakeupEventMask=myMask;
    }
```

There are two possible schemes:

1. Upon the occurrence of the event, the driver checks with the power controller to see if the event is enabled, and if it is, notifies the power manager by calling the power controller's `WakeupEvent()` method. (It checks by calling an API such as the next example IsWakeupEventEnable(...), and passing a bit mask containing the wakeup event that it is interested in.)

```
public:
    inline TBool IsWakeupEventEnabled(Tint aWakeupEvent)
        {
        (iWakeupEventMask & aWakeupEvent) ? return ETrue : return EFalse;
    }
```

2. The driver notifies the power controller whenever a wakeup event it monitors occurs, using an API such as the next example `NotifyWakeupEvent()` and passing a bit mask containing the wakeup event that it monitors; the API checks to see if the wakeup event is enabled, and if it is, notifies the power manager by calling the `WakeupEvent()` method.

```
public:
    inline void NotifyWakeupEvent (Tint aWakeupEvent)
        {
        if(iWakeupEventMask & aWakeupEvent) WakeupEvent();
        }
```

Obviously, for either of these schemes to work, the peripheral driver must have access to the power controller as I described previously.

DisableWakeupEvents() DisableWakeupEvents() either disables the detection of wakeup events directly in hardware, if the power controller monitors them, or it signals to the peripheral driver that monitors them to stop notifying the power controller of their occurrence.

AbsoluteTimerExpired() Absolute timer expiration is typically a monitored wakeup event; the servicing of `AbsoluteTimerExpired()` should simply notify the power manager of a wakeup event:

```
void DXXXPowerController::AbsoluteTimerExpired()
    {
    if (iTargetState == EPwstandby) WakeupEvent();
    }
```

Handling of power HAL group of functions

A `DPowerHal`-derived object requires an implementation of `PowerHalFunction()`, which provides the platform-specific handling of a group of HAL functions.

The HAL component provides user-side access to certain platform-specific functions. It uses the following public exported APIs:

```
class HAL : public HALData
    {
public:
    IMPORT_C static TInt Get(TAttribute aAttribute, TInt& aValue);
    IMPORT_C static TInt Set(TAttribute aAttribute, TInt aValue);
    ...
    IMPORT_C static TInt Get(TInt aDeviceNumber,
    TAttribute aAttribute, TInt& aValue);
    IMPORT_C static TInt Set(TInt aDeviceNumber,
    TAttribute aAttribute, TInt aValue);
    };
```

These can be called with an attribute specifying what function is to be executed at platform-specific level.

The set of HAL attributes that may need to be handled by the `PowerHalFunction()` function are:

- `EPowerBatteryStatus` - used with `HAL::Get(...)` only, see Section 15.3.1.3
- `EPowerGood` - used with `HAL::Get(...)` only, see Section 15.3.1.3
- `EPowerBackupStatus` - used with `HAL::Get(...)` only, see Section 15.3.1.3
- `EPowerExternal` - used with `HAL::Get(...)` only, see Section 15.3.1.3
- `EPowerBackup` - used with `HAL::Get(...)` only, see Section 15.3.1.3
- `EAccessoryPower` - used with `HAL::Get(...)` only, see Section 15.3.1.3
- `EPenDisplayOn` - used with `HAL::Set(...)` to enable switching the display on when tapping the touch panel, or `HAL::Get(...)` to query if tapping the touch panel will switch the display on
- `ECaseSwitchDisplayOn` - used with `HAL::Set(...)` to enable switching the display on when opening the phone lid, or `HAL::Get(...)` to query if opening the phone lid will switch the display on
- `ECaseSwitchDisplayOff` - used with `HAL::Set(...)` to enable switching the display off when closing the phone lid, or `HAL::Get(...)` to query if closing the phone lid will switch the display off.

The `DPowerHal` derived object's PowerHalFunction(...) will be called in response to HAL calls with any of the previous attributes and is passed one of the following parameters (in place of the aFunction argument) to indicate what function to perform at this level:

- `EPowerHalSupplyInfo` - called in response to `HAL::Get(...)` with `EPowerBatteryStatus`, `EPowerGood`, `EPowerBackupStatus` or `EPowerExternal`. Returns a device-specific information structure that is usually kept by the battery-monitoring component (if one exists at this level - see Section 15.3.1.3)
- `EPowerHalBackupPresent` - called in response to `HAL::Get(...)` with `EPowerBackup`, used to query for the presence of a backup battery (see Section 15.3.1.3)
- `EPowerHalAcessoryPowerPresent` - called in response to `HAL::Get(...)` with `EAccessoryPower`, used to query for the presence of accessory power (see Section 15.3.1.3)
- `EPowerHalSetPointerSwitchesOn` - called in response to `HAL::Set(...)` with `EPenDisplayOn`, may be used to enable switching the display back when tapping the touch sensitive panel. On periods of user inactivity, the window server may request the switching off of the display and backlight to conserve power. This is part of a system-wide power policy, which is not the object of this chapter
- `EPowerHalPointerSwitchesOn` - called in response to `HAL::Get(...)` with `EPenDisplayOn`, used to query if tapping the screen will switch the display back on
- `EPowerHalSetCaseOpenSwitchesOn` - called in response to `HAL::Set(...)` with `ECaseSwitchDisplayOn`, may be used to enable switching the display back on when opening any external encasement (lid on a clam shell device, the sliding panel, and so on)
- `EPowerHalCaseOpenSwitchesOn` - called in response to HAL:: Get(...) with `ECaseSwitchDisplayOn`, used to query if opening the case will switch the display back on
- `EPowerHalSetCaseCloseSwitchesOff` - called in response to `HAL::Set(...)` with `ECaseSwitchDisplayoff`,may be used to enable the switching off of the display when closing the case. Again, this may be part of a system-wide power policy taken care by a component not covered by this chapter
- `EPowerHalCaseCloseSwitchesOff` - called in response to `HAL::Get(...)` with `ECaseSwitchDisplayoff`, used to query if closing the case will switch the display off.

As I mentioned previously, `PowerHalFunction` may be invoked in response to a user-side component call to a `UserSvr` class exported API, `HalFunction(...)` - an export from EUSER.DLL:

```
class UserSvr
    {
```

```
public: ...
    IMPORT_C static TInt HalFunction(TInt aGroup,
    TInt aFunction, TAny* a1, TAny* a2);
    IMPORT_C static TInt HalFunction(TInt aGroup,
    TInt aFunction, TAny* a1,
    TAny* a2, TInt aDeviceNumber);
    ...
    };
```

There are a number of other argument values that `PowerHalFunction` function may be invoked with if using `UserSvr::HalFunction(...)`, such as:

- `EPowerHalOnoffInfo` - used to read a `TOnoffInfoV1` structure. This structure is used to record the display switch on/switch off behavior
- `EPowerHalSwitchoff` - this may be used to request a system-wide transition to standby and is provided for binary compatibility with previous versions of the OS. When this is serviced, an `ESwitchoff` TRawEvent will be added to the event queue, from where the window server will pick it up and call `UserHal::Switchoff()`, which will request a transition to standby (see Section 15.2.2). This behavior is customizable at the UI level, and the UI integrator may change it to merely switch UI peripherals such as the display, keypad and touch screen off, leaving the rest of the phone operational
- `EPowerHalTestBootSequence` - this may be used to indicate if the machine is being booted in device-specific test mode.

The following argument values have been deprecated and do not require handling:

- `EPowerHalSetAutoSwitchoffBehavior`
- `EPowerHalAutoSwitchoffBehavior`
- `EPowerHalSetAutoSwitchoffTime`
- `EPowerHalAutoSwitchoffTime`
- `EPowerHalResetAutoSwitchoffTimer`
- `EPowerHalSetBatteryType`
- `EPowerHalBatteryType`
- `EPowerHalSetBatteryCapacity`
- `EPowerHalBatteryCapacity`
- `EPowerHalAutoSwitchoffType`.

Typical power management

Extending the basic framework

We can identify a number of areas where extending the existing basic framework will result in power savings or increased ability to control and monitor power consumption. These extensions can mostly be done at the base port level. The extensions I will propose next utilize the existing framework functionality.

Resource management

Recent mobile phone designs define a number of power resources, such as clock frequencies, voltages and switchable power rails. Software can control these resources independently for each hardware component (CPU and peripherals).

Power resources vary in complexity, from simple binary resources that can be switched on or off almost instantaneously to resources that can be set at different voltage levels or that take a while to change state. There are even resources that may only be changed in conjunction with other resources.

And, of course, some resources are shared between hardware components, and controlling them should be based on tracking their usage.

The base port controls power resources. The base porter needs to provide interfaces for the use of:

1. Peripheral drivers, to be able to change the resources used by the peripherals they control
2. The software component responsible for setting the operating point of the CPU when processing a task (see Section 15.5.1)
3. The derived `DPowerController` `CpuIdle()` function. This routine maps resource state to retention state, and may need to change the state of other resources to achieve the CPU retention state desired (see Section 15.3.1.2).

Controllable power resources may be spread across several functional areas of the ASIC and external components. However, in most cases, it is possible to concentrate the control of power resources on a single software component, the resource manager (Figure 15.3), which offers a conceptual representation and interfaces for all resources. The base porter may also decide to include resource management as part of the power controller kernel extension.

Let's now look at a suggested template for the resource manager. This will be based on an XXXResourceManager class:

```
class XXXResourceManager
    {
public:
    enum TResource // a list of controllable resources (e.g clocks, voltages, power lines)
        {
        SynchBinResourceUsedByZOnly,
        AsynchBinResourceUsedByZOnly,
        // ... other non-shared binary resources, synchronous or asynchronous
        BinResourceSharedByZAndY,
        // ... other shared binary resources, synchronous or asynchronous
        SynchMlResourceUsedByXOnly,
        AsynchMlResourceUsedByXOnly,
        // ... other non-shared multilevel resources, synchronous or asynchronous
        MlResourceSharedByXAndW,
        // ... other shared multilevel resources, synchronous or asynchronous
    };

    void InitResources(); // initialises power Resources not initialised by Bootstrap
    // interface for non-shared resources
    inline void Modify(TResource aResource, TBool aOnoff);
    // for non-shared binary resources
    inline void ModifyToLevel(TResource aResource, Tint aLevel); // for non-shared multilevel resources
    // the following functions may be used by Drivers/Extensions or the idle routine to determine what resources are On or off or th
    inline TBool GetResourceState(TResource aResource);
    // for non-shared binary resources
    inline TUint GetResourceLevel(TResource aResource);
    // for non-shared multilevel resources
public:
    // interface for shared resources
```

```
    SharedBinaryResourceX iSharedBResource;
    inline SharedBinaryResourceX* SharedBResourceX()
        {return & iSharedBResource;}
    // ... other shared Binary resources, synchronous or asynchronous
    SharedMultilevelResourceY iSharedMlResource;
    inline SharedMultilevelResourceY* SharedMlResourceY()
        {return & iSharedMlResource;}
    // ... other shared Multilevel resources
    };
```

If the resource manager needs to be available to the Variant component, or used early in the boot sequence, I recommend that the entry point of the kernel extension be written as follows:

```
static XXXResourceManager TheResourceManager;
static DXXXPowerHal* XXXPowerHal;

GLDEF_C TInt KernelModuleEntry(TInt aReason)
    {
    if(aReason==KModuleEntryReasonVariantInit0)
```

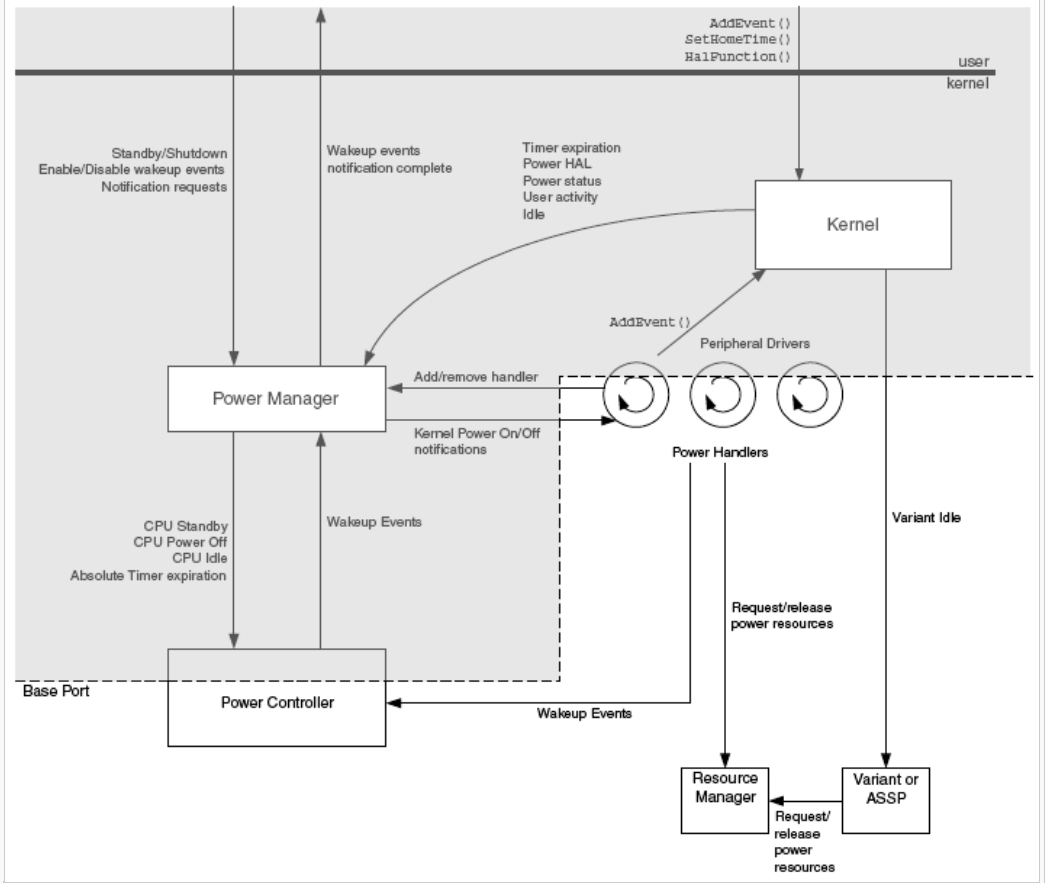

Figure 15.3 *Framework block diagram with resource manager*

```
        {
        // Start the Resource Manager earlier so that Variant and other extension could make use of Power Resources
        __KTRACE_OPT(KPOWER, Kern::Printf("Starting Resource controller"));
        new(&TheResourceManager) XXXResourceManager;
        TheResourceManager.InitResources();
        return KErrNone;
        }
    else if(aReason==KModuleEntryReasonExtensionInit0)
        {
        // Returning KErrNone here ensures we are called later with aReason==KModuleEntryReasonExtensionInit1.
        return KErrNone;
        }
    else if(aReason==KModuleEntryReasonExtensionInit1)
        {
        __KTRACE_OPT(KPOWER, Kern::Printf("Starting power controller"));
        XXXPowerHal = new DXXXPowerHal();
    if (!XXXPowerHal)
        return KErrNoMemory;
    DXXXPowerController* c = new DXXXPowerController();
    if(!c)
        return KErrNoMemory;
    return KErrNone;
    }
    return KErrArgument;
    }
```

This allows the kernel startup sequence to create the resource manager at Variant component creation time. The entry point is invoked again when other kernel extensions are initialized and creates the power controller.

Alternatively, the Variant component could create, and own the resource manager.

To give the variant and device drivers access to the resource manager, the power controller could export a method that returns a pointer to the resource manager object. This scheme is similar to the one used to give access to the power controller object as I explained in Section 15.2.3.5:

```
class TXXXPowerControllerInterface
    {
public:
    ...
    // to allow Variant/Drivers/other Extensions access to Resource Manager
    IMPORT_C static XXXResourceManager* ResourceManager();
    ...
    };

EXPORT_C XXXResourceManager* TXXXPowerControllerInterface::ResourceManager()
    {
    return &TheResourceManager;
    }
```

Resources may be shared by several hardware components; the existing framework already has a template to model the interface required by a binary-shared resource:

```
class MPowerInput
    {
public:
    virtual void Use() = 0;
    virtual void Release() = 0;
    };
```

A shared binary resource deriving from this class needs to implement the pure virtual functions:

- `Use()`. Signals that the power resource is in use. A driver calls this function when it needs the resource to be turned on. A typical implementation associates a counter, initially zero, with the object. `Use()` increments the counter and, if the counter's value changes from 0 to 1, turns the resource on
- `Release()`. Signals that the power resource is not in use. A driver calls this function when it no longer needs the source to be on. `Release()` would decrement the counter I mentioned previously. If the counter's value changes from 1 to 0, `Release()` turns the resource off.

The implementation may add other functions to get the current usage count or resource state. Usage count is especially important as some resources have a maximum acceptable load. When the cumulative load (usage count) on a resource equals its maximum, any attempt to increase its usage count should fail.

Multi-level resources may also be shared. The control model I mentioned previously is not appropriate for such resources - users will want to increase or decrease the level of the resource, rather than switch it on or off. The implementation needs to keep track of the current level of the resource and the requirement of each of the resource users. If a user asks to increase the level, then this is done (up to the maximum acceptable level). But if the user requests a lowering of the present level, then the level is reduced to the maximum requirement from all users. If the requestor does not have the highest level, then there will be no change.

The considerations made previously regarding the maximum cumulative load still apply; however in the case of multi-level resources, the maximum acceptable load may be different for different levels.

A generic shared multi-level API template could look like this:

```
class SharedMultilevelResource // Multilevel Shared Input
    {
public:
    virtual void IncreaseToLevel(TUint aLevel, TInt aRequester) = 0;
    virtual void ReduceToLevel(TUint aLevel, TInt aRequester) = 0;
    virtual TUint GetResourceLevel() = 0;
    };
```

The aRequester parameter on the APIs identifies the user that is requesting a level change.

Finally, there are power resources that cannot be instantaneously varied, requiring, for example, a stabilization period after being changed.

These need to be addressed differently. The software component that requested the resource change needs to wait for the resource to be stable before proceeding. Busy-waiting inside kernel-side components is strongly discouraged in EKA2, especially as the stabilization times may be long. A better alternative is to put the thread to sleep for a period of time, after which the thread can poll the resource again. The base porter can use `Kern::PollingWait()` for this purpose.

You should note that most device drivers use the same kernel thread and so when this thread sleeps, waiting for a resource to stabilize, other device drivers will be also be held up. If the resource stabilization time is long enough to impact the performance of other drivers on the same thread, the device driver which controls the resource may need to create its own kernel thread and change the resource from there. This thread can sleep without affecting the performance of other drivers, and then can call back to the main driver thread when the resource change finally takes place.

Given the multi-threaded nature of EKA2, we advise the base porter to write code that accesses resources with the kernel locked to guarantee their consistency. This is mandatory for shared resources, when accesses can be performed from different threads. If an interrupt service routine can read or change resources, interrupts should also be disabled around any access points.

Moving the CPU to retention from idle

Certain CPUs support a number of low power states distinguished by their ability to retain status, their different power requirements and their wakeup time.

Moving to one of these low power retention states is a non-system-wide power transition that can be wholly managed by the base port part of the kernel framework. In fact, transitions in and out of these low power retention states should be transparent to the rest of the system. If it is likely that a transition to a retention state may have an impact on other parts of the system at a given time then the base port code should not move the CPU to that state at that time, even if the opportunity presents itself.

Let's consider the actions needed to move the CPU to a low power retention state. I've said that the transition will happen in the power controller's platform-specific `CpuIdle()` function.

To guarantee the maximum uninterrupted idle time, some events need to be disabled during that period. The best example of such an event is the system tick - the periodic timed interrupt that is the basis for all timing in EKA2, and is provided by a hardware timer. This is commonly known as idle tick suppression.

The idle time can be predicted as the time until the next timer in the system is due to expire. The `CpuIdle()` implementation can examine the nanokernel timer queue (NTimerQ) which provides this information. The power framework already has an API to return the number of system ticks before the next `NTimer` expiration, the function `IdleTime()`, which is a member of the `NTimerQ` class. The `CpuIdle()` implementation can now suppress the system tick for that period, and program the hardware timer to skip the required number of ticks.

When the hardware timer finally wakes the CPU, `CpuIdle()` can simply adjust the system tick count and reset the hardware timer to produce the regular ticks again. To adjust the system tick count, `CpuIdle()` may use the function `Advance()`, which is a member of the `NTimerQ` class, passing it the number of suppressed ticks.

The CPU may wake up as a result of an event rather than the expiration of the hardware timer. In this case, the implementation of `CpuIdle()` needs to read the hardware timer, work out the number of integral system ticks suppressed and adjust the system tick count. It must also reprogram the hardware timer to generate the next (and subsequent) system ticks.

Sometimes waking up from a retention state can take longer than several system ticks. In that case, `CpuIdle()` should program the hardware timer to wake the CPU up at a time given by the next `NTimer` expiration minus the number of ticks it takes to wake up from that state.

This waking up from a retention state happens inside the null thread - this means that the post amble needed to restore the status should be kept as short as possible. Both preamble and post-amble routines should be executed with interrupts disabled, to stop them from being preempted.

It often happens that while the CPU is in the retention state, it is not able to perform the periodic refreshing that SDRAM needs. In this case, the SDRAM must be placed in self-refresh mode before going into the retention state, with the CPU reassuming control of refreshing it after waking up.

The choice of low power retention state is connected with the current status of the phone's power resources. The idle transition routine must have the ability to inspect the state of relevant resources, by interrogating the resource manager. This interface also allows the state of resources to be modified as needed.

The CPU is moved to a low power retention state by a wait-for-interrupt type instruction, which will suspend instruction execution until an enabled hardware event occurs.

Naturally, events other than the hardware timer interrupt may have the ability to wake the CPU up from the retention state; these are wakeup events for that state. Wakeup events for the retention state include not only hardware events that result from user interaction (screen tapping, key press, and so on) and timed alarms as for the standby state, but also the events that result from other peripherals' operation (such as receiving a unit of data from an input peripheral, a media change resulting from inserting or removing a removable media device) device timeouts, and more. These events should be left enabled or explicitly enabled prior to moving the CPU to retention state.

If a wakeup event other than the timer's expiration brings the CPU back from idle state, the `CpuIdle()` implementation must determine how many ticks were effectively skipped, and adjust the system tick count accordingly, before resetting the hardware timer to produce regular ticks. `CpuIdle()` can do this simply by examining the hardware timer's current count. However the adjustment needs to take in consideration that the effective elapsed time may not be an integral number of system ticks.

Finally, the base porter may decide, on longer periods of CPU idle, to transition the CPU to a state that is not capable of state retention, such as the standby state I described previously. To transition to this state, the `CpuIdle()` routine needs to save the status of the CPU and possibly that of some peripherals, as I described previously. Although this results in greater power savings, extreme care must be taken, as the transition into and out of such a state may severely impact the performance and real-time guarantees of the system.

Battery monitoring and management

The majority of Symbian OS mobile phones that were in the market as this book was written were based on a two-chip solution, with one processor dedicated to the telephony application and associated signaling stacks, and the other for Symbian OS. In this case, the telephony processor usually performs battery monitoring and management. Symbian OS gets the battery information through the communication channel between the two devices.

However, in the future we may see single-chip and even single-core solutions becoming more common. For single-core solutions, Symbian OS will provide battery monitoring and management. The base port will do the actual monitoring of battery levels. The framework must offer an interface to read the levels from the battery hardware-controlling component. It also needs to register and propagate any battery related events.

The management of the information provided by the battery monitoring involves notifying applications and other user-side components of level changes or critical conditions. For example, when the battery level drops below a certain level, the system-wide power policy might be that the window server must ask the screen driver to switch the display driver to a different mode, lowering the resolution and refresh rate to conserve power. The OS power policy must include provisions to keep the user of the phone informed of the battery level and warn him/her when the level drops below the safety threshold or when a charger has been connected.

The policy may even force a transition to a low power state, if the battery level drops below a critical threshold.

A user-side battery manager component should communicate with the battery monitoring part of the framework (a kernel-side component).

Certain device drivers may also have an interest in battery levels or notification of battery events.

The kernel framework has a template for a battery monitor as provided by `DBatteryMonitor` class:

```
class DBatteryMonitor
    {
public:
    IMPORT_C DBatteryMonitor();
    IMPORT_C void Register();
public:
    virtual TSupplyStatus MachinePowerStatus() = 0;
    virtual void systemTimeChanged(TInt anOldTime, TInt aNewTime) = 0;
};
```

This class includes an exported constructor to allow the platform-specific power kernel extension to create the monitor, and a `Register()` function, which the entry point of this extension should invoke after the monitor object is created, to register the battery monitor with the power manager. These two

public APIs are exported by EKERN.EXE.

The battery monitor object may derive from this class, and be owned by the power controller kernel extension.

In version 9.1 and below, Symbian OS allows the mapping of charge levels to four possible values: *zero*, *very low*, *low* and *good* as given in the `TSupplyStatus` enumeration:

```
enum TSupplyStatus
    {
    EZero,
    EVeryLow,
    ELow,
    EGood
    };
```

This is likely to change to a system that uses a percentage of charge level, as this would give finer graduations.

There is one pure virtual function that must be implemented by the battery monitor, and that forms its mandatory interface to the kernel (the other function, `SystemTimeChanged()` has been deprecated):

- `MachinePowerStatus()`. This function should read and return the state of the battery with respect to charge level (as one of the `TSupplyStatus` enumerated values). If external power is connected, the function should return `EGood`. Device drivers call this function before starting operations whose successful conclusion depends on the battery charge level - for example, operations that lead to substantial increases in power consumption, or take a long time to complete. They access the function through another framework API, `Kern::MachinePowerStatus()`.

There is no built-in feature to notify device drivers of asynchronous battery events, such as a drop in charge beyond a critical level. The device creator could implement this at base port level: the battery monitor could provide an exported method to allow drivers to register an interest in being notified of battery events. The battery monitor would maintain a list of pointers to driver objects. Obviously, when a driver was closed, it should deregister with the battery monitor:

```
class DXXXBatteryMonitor : public DBatteryMonitor
    {
public:
    ...
    inline void RegisterForBatteryNotifications(DPowerHandler* aPowerHandler)
        {
        NKern::Lock();
        aPowerHandler->iNextBt=iHead;
        iHead=aPowerHandler;
        NKern::Unlock();
        }

    inline void DeRegisterForBatteryNotifications(DPowerHandler* aPowerHandler)
        {
        NKern::Lock();
        DPowerHandler** prev = &iHead;
        while (*prev != aPowerHandler)
            prev = &(*prev)->iNextBt;
        *prev = aPowerHandler->iNextBt;
        NKern::Unlock();
        }
public:
    ...
    DPowerHandler* iHead;
    };
```

Peripheral drivers could register with the battery monitor using a power controller exported API (implemented by the base port) which returns a pointer to the battery monitor:

```
class TXXXPowerControllerInterface
    {
public:
    ...
    // to allow Variant/Drivers/other Extensions access to battery monitor
    IMPORT_C static DXXXBatteryMonitor* BatteryMonitor();
    inline static void RegisterBatteryMonitor(DXXXBatteryMonitor* aBatteryMonitor)
        {iBatteryMonitor=aBatteryMonitor;}
public:
    ...
    static DXXXBatteryMonitor* iBatteryMonitor;
    };

EXPORT_C DXXXBatteryMonitor* TXXXPowerControllerInterface::BatteryMonitor()
    {
    return &iBatteryMonitor;
    }

// battery monitor constructor
DXXXBatteryMonitor::DXXXBatteryMonitor()
    {
    Register(); // register battery monitor with power manager
    TXXXPowerController::RegisterBatteryMonitor(this);

    // register battery monitor with the interface
    }
```

The driver's power handler-derived object could have a method that the battery monitor would call when an event occurs that the driver is interested in. This method could either execute the driver-specific handling of the event in the context of the battery monitor, or schedule a DFC to execute in the driver's thread. For example:

```
class DXXXPowerHandler : public DPowerHandler
    {
public:
    ...
    inline void NotifyBattEvent(TInt aEvent)
        {
        NKern::Lock();
        iBattEvent=aEvent;
        iBattEventDfc.Enque();
        NKern::Unlock();
        }
```

```
public:
    ...
    DPowerHandler* iNextBt;
    TInt NotificationMask;
    };
```

Here aEvent is a bit mask indicating what battery event has occurred. When an event occurs, the battery monitor could simply notify all drivers that are interested in that event by calling the previous API for their power handlers. This should be done from a thread context (for example a DFC):

```
// to be called after reading the event off the hardware battery component
DXXXBatteryMonitor::NotifyBattEvent(TInt aEvent)
    {
    DPowerHandler* ph = iHead;
    while (ph)
        {
        if(ph->NotificationMask&aEvent)
        ph->NotifyBattEvent(aEvent);
        ph = ph->iNextBt;
        }
    }
```

The scheme I have just described could be improved to have a priority associated with each driver, which will be reflected in the order the monitor notifies drivers.

The battery monitor should be responsible for maintaining a power supply information structure as summarized by the framework's existing `TSupplyInfoV1`:

```
class TSupplyInfoV1
    {
public:
    SInt64 iMainBatteryInsertionTime;
    TSupplyStatus iMainBatteryStatus;
    SInt64 iMainBatteryInUseMicroSeconds;
    TInt iCurrentConsumptionMilliAmps;
    TInt iMainBatteryConsumedMilliAmpSeconds;
    TInt iMainBatteryMilliVolts;
    TInt iMainBatteryMaxMilliVolts;
    TSupplyStatus iBackupBatteryStatus;
    TInt iBackupBatteryMilliVolts;
    TInt iBackupBatteryMaxMilliVolts;
    TBool iExternalPowerPresent;
    SInt64 iExternalPowerInUseMicroSeconds;
    TUint iFlags;
    };
```

This information is base-port specific and the monitoring component may decide to use these fields as it sees fit.

The power framework and the HAL provide the user-side software battery-management component with an embryonic interface to the battery monitor.

The following set of HAL attributes can be used:

- `EPowerBatteryStatus` - this is used to query the value of `iMainBatteryStatus` from the previous structure. This is the charge levelof the battery (normalized to one of the `TSupplyStatus` enumerated values)
- `EPowerGood` - this returns `ETrue` either if external power is connected or if the current battery charge level is above *low*
- `EPowerBackupStatus` - this is used to query the value of `iBackupBatteryStatus` which is the charge level of a backup battery, if present
- `EPowerExternal` - this is used to query the value of `iExternalPowerPresent` which is `ETrue` if external power, such as the charger, is connected
- `EPowerBackup` - this can be used to query for the presence of a backup battery
- `EAccessoryPower` - this can be used to query for presence of accessory power, such as for example, drawing power from a USB cable.

The implementation of `PowerHAL::PowerHalFunction(...)` should therefore call the battery monitor when it is passed one of the previous arguments.

It may be that this interface is not enough for the needs of a battery manager component. If that is the case, we suggest the use of a device driver for the purpose of communicating with the battery monitor. The battery monitor would then have a set of exported functions, which would be called by an LDD loaded by the battery manager, which offers a channel for interfacing to the battery manager (Figure 15.4).

Monitoring environmental inputs

Certain environmental factors such as temperature may have an impact on the power state of CPU and peripherals, and so need to be monitored. For example if the CPU temperature rises above a certain level, the power framework may need to reduce its clock speed to prevent damage. As a further example, certain mobile SDRAM devices have a temperature compensated self-refresh rate, for which software that monitors the case temperature needs to input the current temperature range.

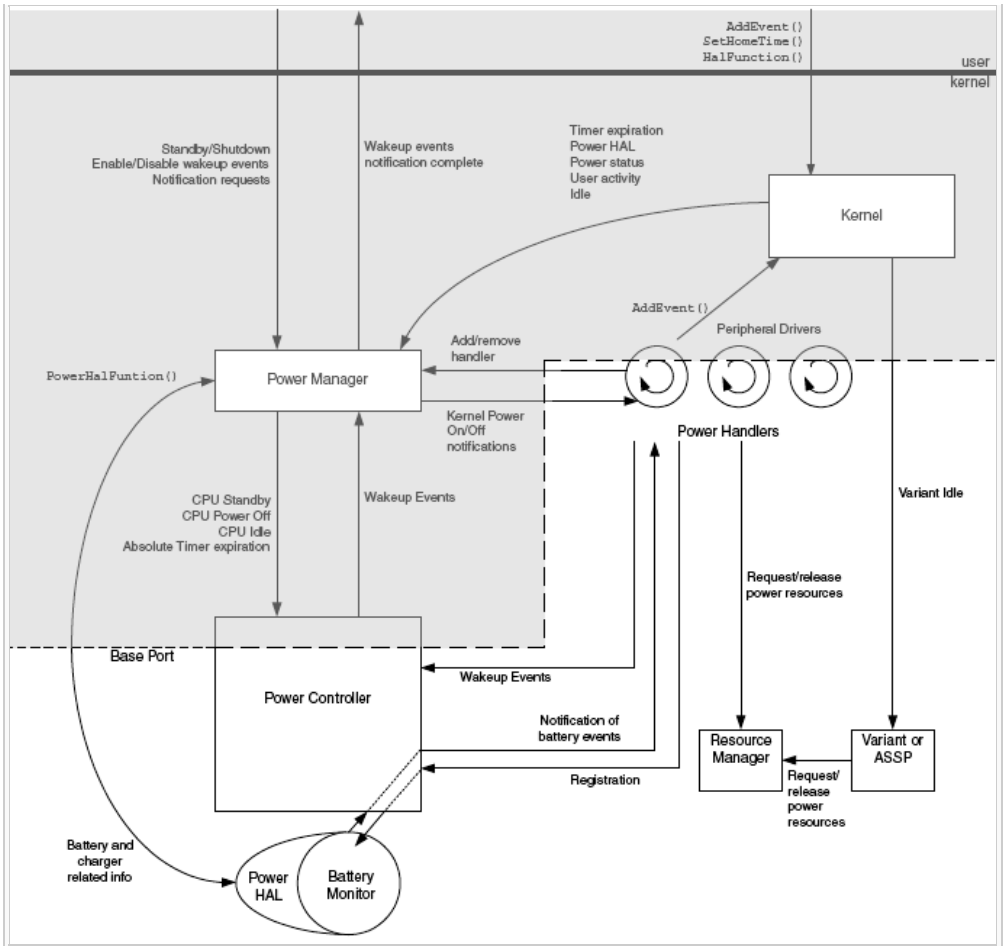

Figure 15.4 *Framework block diagram with battery monitor*

The base port may need to provide software routines to monitor the environmental inputs using hardware sensors and communicate the state of these to other parts of the kernel power framework.

Peripheral low power retention state support Peripheral devices, even those which are integrated as part of the main ASIC, may be capable of operation at low power, and may be transitioned to that mode of operation under software control. These low power states map to the retention state that I described in Section 15.1. Device driver software usually powers up the peripheral device it controls at channel creation time. If a peripheral is controlled by a kernel extension, it is usually powered up at kernel boot time. However, this does not mean that the peripheral device will be used immediately or that power resources used by that peripheral need to be turned on at the level corresponding to peripheral device activity. We recommend that if a peripheral device is idling, it should be moved to a low power state, if supported. The peripheral driver-specific part of the power framework should do this. The definition of peripheral idle may vary from peripheral to peripheral but may be generally defined as not servicing any requests from its clients and not performing any internal tasks not directly related to service of a client request. Any power-saving measures undertaken by the peripheral driver must be transparent to the users of the peripheral. If the time it takes a peripheral to return to a more available state and service a request has no impact on the performance of peripheral driver or their clients, then it is safe to move the peripheral to a low power state when it reaches an idle condition (Figure 15.5).

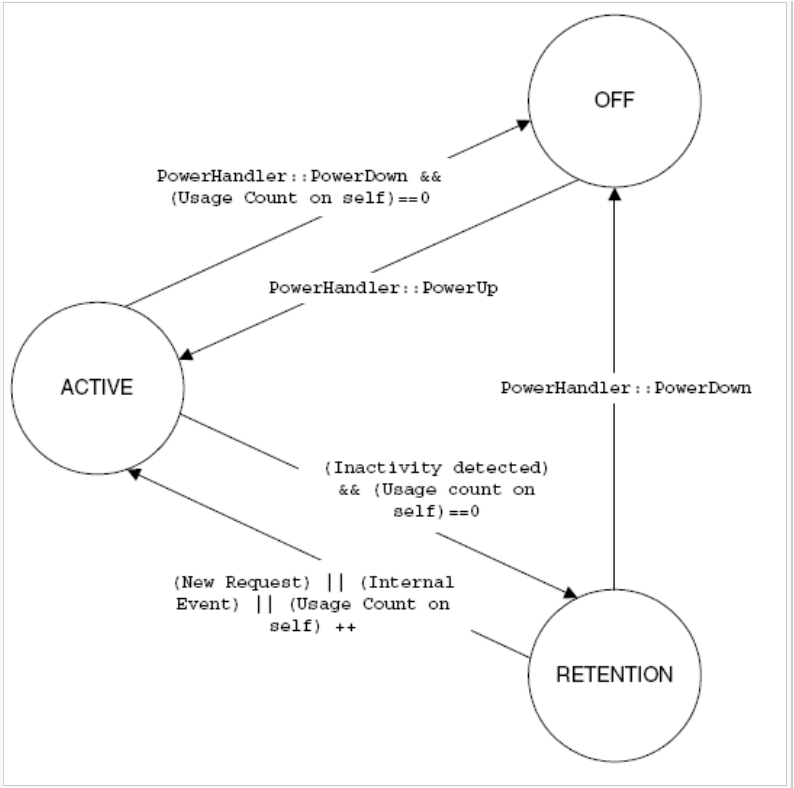

Figure 15.5 *Typical peripheral state transition diagram*

The SDIO bus controller implementation is a good example of peripheral inactivity monitoring:

1. When the bus power supply is turned on, a periodic inactivity timer with a period of one second is started
2. On timer expiration, the ensuing callback function checks if a device using the bus has locked the controller. If this is not the case, and the required number of seconds (iNotLockedTimeout) has expired since turning the power supply on, the bus is powered down. The inactivity timer is stopped
3. If the bus controller is locked (it is in use by a device on the bus) but a longer timeout period (given by `iInactivityTimeout`) expires, then the device that has locked the bus is notified every second from then on and may decide to deregister itself, which unlocks the controller, thus allowing it to power down on the next second tick, and move to a device-specific sleep mode.

Notifying peripheral drivers of imminent CPU transition to retention state

The base porter may want peripheral drivers to be notified that the CPU has entered the idle state. Depending on their current functional state, peripheral drivers may decide to either transition the peripheral to a retention state or stop the CPU transition to that state. In other cases, certain peripherals will have to be placed in a different mode of operation to track any events which will bring the CPU back from the retention state.

The kernel's null thread issues notifications that the CPU is idling. The base port should implement the notification mechanism. This mechanism should do nothing that results in scheduling another thread; it cannot block. At best it may initiate a power resource change, but may not wait for its completion.

Base porters could give their peripheral drivers a callback function, which would execute synchronously and would be called from the `CpuIdle()` routine. Next I will give an example of how this could be implemented. The platform-specific power controller object could have a method to allow driver-specific power handlers to register with the power controller for `CpuIdle()` callbacks. The power controller could then keep a list of pointers to registered drivers. When the power handler is destroyed, it should deregister with the power controller:

```
class DXXXPowerController : public DPowerController
    {
    ...
public:
    inline void RegisterWithCpuidle(DPowerHandler* aPowerHandler)
        {
        NKern::Lock();
        aPowerHandler->iNextCi=iHead;
        iHead=aPowerHandler;
        NKern::Unlock();
        }

    inline void DeRegisterWithCpuidle(DPowerHandler* aPowerHandler)
        {
        NKern::Lock();
        DPowerHandler** prev = &iHead;
        while (*prev != aPowerHandler) prev = &(*prev)->iNextCi;
        *prev = aPowerHandler->iNextCi;
    NKern::Unlock();
    }
public:
    ...
    DPowerHandler* iHead;
    };
```

The driver's power handler should keep pointers to the static synchronous, non-blocking, non-waiting callbacks that can be called from the power controller. There are two callbacks: one that is called when entering the CPU idle state, and the other that is called when leaving this routine, for example:

```
typedef void (*TCpuIdleInCallBack)(TAny* aPtr);
typedef void (*TCpuIdleOutCallBack)(TAny* aPtr);

inline static void EnterIdle(TAny* aPtr);
inline static void LeaveIdle(TAny* aPtr);

class DXXXPowerHandler : public DPowerHandler
    {
```

```
public:
    ...
public:
    ...
    TCpuIdleInCallBack iEnterIdleCallback;
    TCpuIdleOutCallBack iLeaveIdleCallback;
    };
```

At construction time, `iEnterIdleCallback` is set to point to `EnterIdle()` and `iLeaveIdleCallback` to `LeaveIdle()`. When entering the `CpuIdle()` function, the power controller calls the registered drivers, using the power handler callback pointer mentioned previously. The callback functions execute in the null thread context.

```
...
DPowerHandler* ph = iHandlers;
while (ph)
    {
    ph->iEnterIdle(ph);
    ph = ph->iNext;
    }
...
```

When the CPU wakes up, and just before leaving the `CpuIdle()` function, the power controller calls the registered drivers:

```
...
DPowerHandler* ph = iHandlers;
while (ph)
    {
    ph->iLeaveIdle(ph);
    ph = ph->iNext;
    }
...
```

Power management for peripherals that provide services to other peripherals

Some peripherals provide services that are used by other peripherals in the same system - these peripherals may require a separate driver to control them. Examples are intelligent internal buses such as I2C and SPI, DMA controllers, embedded PCI controllers and so on. The power state of these peripherals at any given time must be related to the power states of the peripherals they provide services to. It is important that their control model takes this into consideration.

If a peripheral provides services to another peripheral, it must not power down until the client peripheral has powered down - and of course it must power up before the dependent peripheral has any need for its services.

One way in which the base porter can guarantee this is to have the requests from its client drivers powering the slave peripheral up, and only powering down when the client driver powers down. If the slave peripheral driver's power handler's `PowerDown()` is called, it should wait until the all its client drivers power down before powering down the hardware it controls. Requests from the client peripheral's drivers will have to yield and wait for the slave peripheral to power up.

Peripherals that provide services to other peripherals may be capable of moving into a retention state. The principles of control discussed for general peripherals still apply: peripherals will be allowed to go to retention state if no request is being serviced or background task performed and if the latency of the retention state does not impact the performance of the client drivers.

Peripherals may provide services to more than one other peripheral, such as is usually the case with DMA controllers or inter-component buses (Figure 15.6). These peripherals can be seen as shared power resources, especially if they allow multiple simultaneous clients. They should implement a usage counting mechanism that will allow their drivers to know if the peripheral is in use, and decide when to power up or down, and if it is safe to go to retention state.

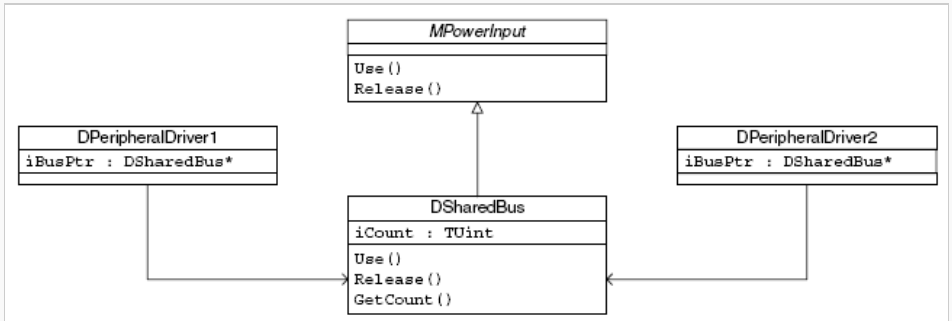

Figure 15.6 *Example shared peripheral*

In the previous example, the shared peripheral driver object derives from `MPowerInput` exposing a Use()/Release() interface to the client drivers.

If the shared peripheral's retention state latency does not have an impact on the performance of the client drivers, then the client drivers may call `Use()` whenever they issue a request for service to the shared driver, and `Release()` when the request is complete. If the impact of the latency cannot be dismissed, the client drivers will need to keep the shared peripheral in operation for longer periods, possibly for the entire duration of their own operational cycle.

Writing a power-aware device driver

Now let's look at how to implement power management for a *real life* device driver. I will use a simplified serial comms driver and will apply some of the concepts I have just described.

I make the following assumptions:

1. The peripheral hardware supports all five power states: off, standby, retention, idle and active. This is not a common situation: in most cases there is

no distinction between standby and off states and in some others, there is no support for retention state. I also assume that I can move the peripheral to a particular power state by setting requirements on certain power resources (clock, voltage and power supply), and by a hardware register programming sequence

2. The peripheral hardware uses a clock input that can only be on (when the peripheral is in active, idle or retention states) or off (when the peripheral is moved to standby or off states). This clock input is shared with another peripheral. The peripheral hardware operates at different voltages depending on the power state: 100% of maximum voltage for the active state, 50% of maximum voltage for the retention state, and 20% of maximum voltage for the standby state. And, finally, the power supply to the peripheral can be cut off or restored
3. The peripheral retention state is of negligible latency, that is, it can come back from retention to the active state quickly enough to service a request
4. In my example, the LDD software moves the peripheral to a different power state, for simplicity, while in a *real-life* device driver the LDD should call the PDD to perform the transition
5. The driver thread can change the power resources used by this peripheral instantaneously, which means that it can wait, with no impact on either its own performance, or that of its clients or the system.

The peripheral driver software routines implement a state machine:

1. When a channel is opened the peripheral is moved to the idle state
2. If the peripheral is in the idle state when a request is made, it moves to the active state
3. After the request is completed, the peripheral moves back to idle
4. When in the idle state, the peripheral waits for a period of time (the inactivity timeout) and if no request is made, it moves to the retention state
5. If the peripheral is in the retention state when a request is made, it moves to the active state
6. When the null (or idle) thread runs, it calls a driver function which checks if the peripheral is idling, in which case the driver callback initiates the peripheral's move to retention state and cancels the inactivity timer. It also delays any device timeouts until the CPU wakes up
7. If the peripheral is in one of the active, idle or retention states, the power manager may request a power down to either the standby or the off state
8. The peripheral can only leave standby if the power manager requests a power up, in which case the driver software moves it to idle and starts the inactivity timer
9. When the channel is closed, the driver software shuts down the peripheral (moving it to the off state).

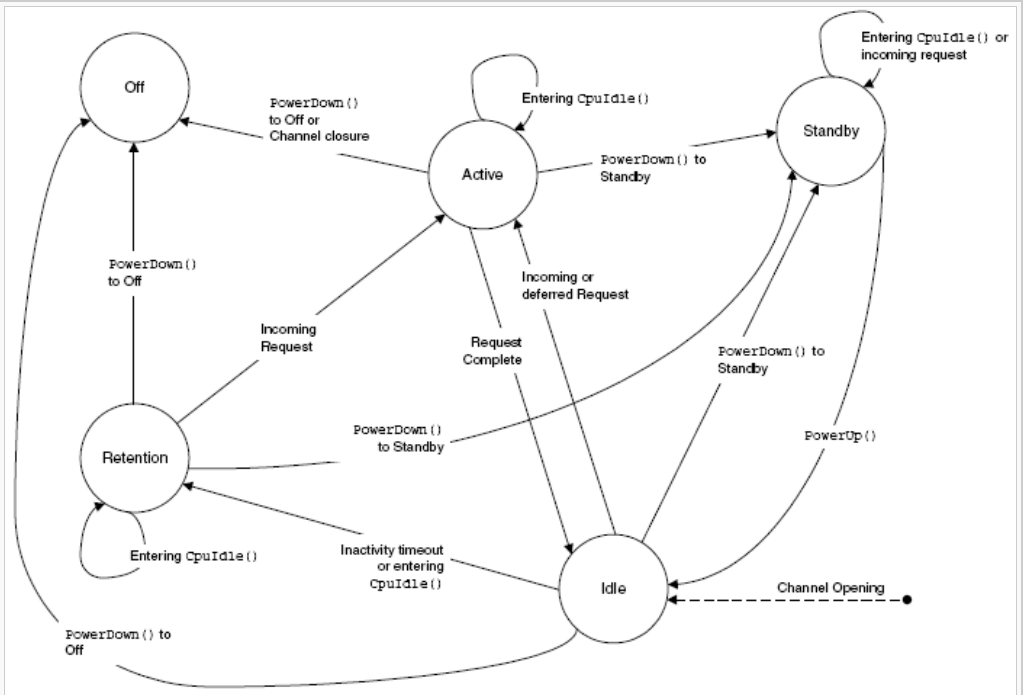

Figure 15.7 *Example serial comms driver state machine*

The state diagram shown in Figure 15.7 applies.

Thread and synchronization issues

The driver power management functions execute in different contexts:

1. The power manager's `PowerUp()` and `PowerDown()` are called from the thread of the user-side component responsible for the system transition
2. Requests from a client (including closing the channel) are issued from the client's thread but their servicing and their completion execute in the driver's thread. Channel opening executes in the client's context
3. The inactivity timer's expiration generates an interrupt
4. The callbacks that are called when entering or leaving `CpuIdle()` execute in the null thread.

Thus, we must take some care to guarantee that execution is performed in the right sequence:

- We must protect both the peripheral's and the power resource's state changes against the preemption of the executing thread
- The power manager's `PowerUp()` and `PowerDown()` must schedule DFCs to execute in the driver's thread
- The inactivity timer interrupt must schedule a DFC to execute in the driver's thread
- Transitions to the off state, or to and from the standby state, involve the calling of other power handlers and take some time: it might happen that a request comes in, or the null thread gets to run, after our driver's power handler moves to its low power state, and before the CPU reaches that state. This might also happen after the CPU wakes up but before our driver's power handler moves the peripheral back to the active state. Service requests and the `CpuIdle()` entry callback must check the current power state
- We must cancel the inactivity timer and the ensuing DFC on every state change, apart from when moving from the idle state to the retention state (since this is caused by its own expiration)
- The `CpuIdle()` entry and exit callbacks run with interrupts disabled (as I mentioned in Section 15.2.2.1). They cannot be preempted, and always run in sequence, even if the CPU never reaches the retention state
- The CPU idle (null) thread may run while a request is being serviced (for example, if the driver blocks waiting on a hardware event), or at any time during the power down or power up sequence. The driver's `CpuIdle()` entry callback needs to check if the peripheral state is idle
- When the client closes the channel to the driver, the kernel sends a request as a kernel-side message to the driver that needs to be completed before

the driver object is destructed. The completion of a kernel-side message may block, so the power-down or power-up DFCs, or the null thread, may run between the driver shutting down and the driver object (and the associated power handler) being destructed. We must check for this and skip any operations that result in attempting to operate on a peripheral that has already powered off.

Class definitions

The device driver class (DChannelSerialDriver) has a pointer to the power handler (DSerialDriverPowerHandler). It owns an `NTimer` that is used to track inactivity. It offers methods to power the peripheral hardware up and down, and move it to the retention and the active states.

The power handler has pointers to the power controller and resource manager. It has pointers to the two callbacks that will be called on entering and leaving the power controller's `CpuIdle()` function.

Driver object construction

```
DChannelSerialDriver::DChannelSerialDriver()
//
// Constructor
//
    :
    iPowerUpDfc(DChannelSerialDriver::PowerUpDfc,this,3),
    iPowerDownDfc(DChannelSerialDriver::PowerDownDfc,this,3),
    iTimerDfc(DChannelSerialDriver::TimerDfcFn,this,3),
    iTimer(DChannelSerialDriver::TimerCallBack,this)
    ...
    {
    ...
    iStatus=EOpen;
    }
```

When the driver DLL is loaded the kernel calls its entry point, which then creates the driver object.

The device driver object's constructor sets up the DFCs that will be issued when the power manager asks to power the peripheral up or down, and the DFC that is called when the inactivity timer expires. It also sets up the callback that the timer interrupt will call.

Channel opening

```
TInt DChannelSerialDriver::DoCreate(TInt aUnit, const TDesC8* /*anInfo*/, const TVersion &aVer)
//
// Create the channel from the passed info.
//
    {
    ...
    // set up the correct DFC queue
    SetDfcQ(((DComm*)iPdd)->DfcQ(aUnit)); // Sets the DFC queue (iDfcQ) to be used by this logical channel
    iPowerUpDfc.SetDfcQ(iDfcQ);
    iPowerDownDfc.SetDfcQ(iDfcQ);
    iTimerDfc.SetDfcQ(iDfcQ);
    ...
    iMsgQ.Receive();
    // create the power handler
    iPowerHandler=new DSerialDriverPowerHandler(this);
    if (!iPowerHandler)
        return KErrNoMemory;
    iPowerHandler->Add(); // add to power manager's list of power handlers
    iPowerHandler->RegisterCpuIdleCallback(ETrue);
    // register with CpuIdle
    DoPowerUp();
    return KErrNone;
    }
```

When the client creates a channel to access this driver the `DoCreate()` function above is called. This:

1. Sets the DFC queue to be used by the driver. The power up, power down and inactivity timer DFCs all execute in the context of that DFC queue, in this way avoiding any preemption problems
2. Activates the message delivery queue for this driver's requests

Creates the driver's power handler object and registers it with the power manager. Note that the device driver framework calls `DoCreate()` inside a critical section, making it possible to call the power handler `Add()` function from within it.

```
DSerialDriverPowerHandler::DSerialDriverPowerHandler(DChannelSerialDriver* aChannel)
    :
    DPowerHandler(KLddName),
    iChannel(aChannel),
    iPowerState(EIdle),
    iEnterIdleCallback(EnterIdle)
    iLeaveIdleCallback(LeaveIdle)
    {
    iResourceManager= TXXXPowerControllerInterface::ResourceManager();
    // get pointer to Resource Manager
    }
```

The power handler constructor sets up the pointer to the device driver object, and also sets up the pointers to the two callback functions. It obtains a pointer to the resource manager to allow it to access the power resources controlled by it

2. The `DoCreate()` function calls a method provided by the power handler to register with the power controller. This allows the calling of the callback from `CpuIdle()`

```
void DSerialDriverPowerHandler::RegisterCpuIdleCallback( TBool aRegister)
    {
    if(aRegister) // register
        {
        iPowerController= TXXXPowerControllerInterface::PowerController();
        iPowerController->RegisterWithCpuIdle(this);
        }
    else // deregister
        {
        iPowerController->DeRegisterWithCpuIdle(this);
        iPowerController=NULL;
        }
    }
```

3. Finally the `DoCreate()` function calls `DoPowerUp()` to power up the peripheral hardware, setting the driver's power state to idle. It starts the inactivity-monitoring timer:

```
void DChannelSerialDriver::DoPowerUp()
    {
    iTimer.Cancel();
    iTimerDfc.Cancel();
    NKern::Lock();
    iPowerHandler->iPowerState=DSerialDriverPowerHandler::EIdle;
    iResourceManager->ModifyToLevel(XXXResourceManager::VoltageSerial, 100);
    // request 100% voltage level
    iResourceManager->SharedClock()->Use(); // assert request on shared clock
    iResourceManager->Modify(XXXResourceManager::PowerSupplySerial, ETrue);
    // turn power supply on (if off)
    // ...write to peripheral registers to set peripheral in active state
    NKern::Unlock();
    Complete(EAll, KErrAbort);
    iTimer.OneShot(KTimeout, ETrue); // restart inactivity timeout
    }
```

Incoming requests

Client requests are delivered as kernel-side messages, which will be executed in the driver's thread context (iDfcQ). So, when the function `HandleMsg()` is executed, it marks the start of the execution of a client's request by this driver:

```
void DChannelSerialDriver::HandleMsg(TMessageBase* aMsg)
    {
    TInt state=iPowerHandler->iPowerState;
    if(state==(TInt)DSerialDriverPowerHandler::Eoff) return;
    if(state==(TInt)DSerialDriverPowerHandler::Estandby)
        {
        // postpone message handling to transition from standby
        iMsgHeld=ETrue;
        return;
        }
    TThreadMessage& m=*(TThreadMessage*)aMsg;
    TInt id=m.iValue;
    if (id==(TInt)ECloseMsg)
        {
        Shutdown(EFalse); // off
        iStatus = EClosed;
        m.Complete(KErrNone, EFalse);
        return;
        }
    else
        {
        if(iPowerHandler->iPowerState!= DSerialDriverPowerHandler::EActive)
        // if already active, skip
        MoveToActive(); // a request has been made, move to active
        if (id==KMaxTInt)
            {
            // DoCancel
            DoCancel(m.Int0());
            m.Complete(KErrNone,ETrue);
            return;
            }
        if (id<0)
            {
            // DoRequest
            TRequestStatus* pS=(TRequestStatus*)m.Ptr0();
            TInt r=DoRequest(~id,pS,m.Ptr1(),m.Ptr2());
            if (r!=KErrNone)
                Kern::RequestComplete(iClient,pS,r);
            m.Complete(KErrNone,ETrue);
            }
        else
            {
            // DoControl
            TInt r=DoControl(id,m.Ptr0(),m.Ptr1());
            m.Complete(r,ETrue);
            }
        }
    }
```

As I explained earlier, a request may arrive while the driver is being powered down or powered up. The function checks to see if the peripheral is powering down: if it is then the request will not be serviced. If the peripheral is being transitioned to standby, or just returning from it, the function defers the servicing of the request until the peripheral has powered on.

If the client is not requesting the closure of the driver, the peripheral hardware must be moved to the active state in anticipation of performing request-related actions. We do this by calling the `MoveToActive()` function:

```
void DChannelSerialDriver::MoveToActive()
    {
    iTimer.Cancel();
    iTimerDfc.Cancel();
    NKern::Lock();
    iPowerHandler->iPowerState=DSerialDriverPowerHandler::EActive;
    iResourceManager->ModifyToLevel(XXXResourceManager::VoltageSerial, 100);
    // request 100% voltage level
    iResourceManager->SharedClock()->Use();
    // assert request on shared clock
    // ...write to peripheral registers to set peripheral in active state
    NKern::Unlock();
    }
```

This function requests power resources that are compatible with the active state, sets the driver's power state to active and writes to the peripheral register to move it to active state.

Inactivity detection

When a request is completed, the driver calls the `Complete()` function. This function is also called when shutting down. It checks the power state, and if this is active, sets it to idle and restarts the inactivity-monitoring timer.

```
void DChannelSerialDriver::Complete(TInt aMask, TInt aReason)
    {
    if (aMask & ERx)
        Kern::RequestComplete(iClient, iRxStatus, aReason);
    if (aMask & ETx)
        Kern::RequestComplete(iClient, iTxStatus, aReason);
    if (aMask & ESigChg)
        Kern::RequestComplete(iClient, iSigNotifyStatus, aReason);
    TInt state=iPowerHandler->iPowerState;
    if(state==(TInt)DSerialDriverPowerHandler::EActive)
        {
        iPowerHandler->iPowerState=DSerialDriverPowerHandler::EIdle;
        iTimer.OneShot(KTimeout, ETrue);
        }
    }
```

The timer callback is called in the context of the system tick interrupt. The callback simply schedules a DFC to execute in the driver's thread context:

```
void DChannelSerialDriver::TimerCallBack(TAny* aPtr)
    {
    // called from ISR when timer completes
    DChannelSerialDriver *pC=(DChannelSerialDriver*)aPtr;
    pC->iTimerDfc.Add();
    }

void DChannelSerialDriver::TimerDfcFn(TAny* aPtr)
    {
    DChannelSerialDriver *pC=(DChannelSerialDriver*)aPtr;
    pC->InactivityDfc();
    }
```

A state change might occur between the timer being started and the DFC executing, so this function needs to check if the power state is still idle. If it is, then the peripheral hardware is moved to the retention state:

```
void DChannelSerialDriver::InactivityDfc(TAny* aPtr)
    {
    DChannelSerialDriver *pC=(DChannelSerialDriver*)aPtr;
    if(pc->iPowerHandler->iPowerState== DSerialDriverPowerHandler::Eidle)
        pC->MoveToRetention();
    }
```

We move to retention state like this:

```
void DChannelSerialDriver::MoveToRetention()
    {
    // may be called from Null thread: must not block or schedule another thread
    NKern::Lock();
    iPowerHandler->iPowerState= DSerialDriverPowerHandler::ERetention;
    iResourceManager->ModifyToLevel(XXXResourceManager::VoltageSerial, 50);
    // request 50% voltage level
    iResourceManager->SharedClock()->Use();
    // assert request on shared clock
    // ...write to peripheral registers to set peripheral in retention state
    NKern::Unlock();
    }
```

Entering and leaving CPU idle

Because we registered for `CpuIdle()` callbacks, when the power controller's `CpuIdle()` is entered, the power controller calls the driver using the `iEnterIdleCallback` pointer. I discussed this in Section 15.2.2.1.

```
inline static void EnterIdle(TAny* aPtr)
    {
    // called with interrupts disabled
    DSerialDriverPowerHandler* d = (DSerialDriverPowerHandler*)aPtr;
    if (d->iChannel->iStatus != EClosed) // not closing
        {
        if(d->iPowerState==DSerialDriverPowerHandler::EIdle)
            {
            d->iChannel->iTimer.Cancel();
            d->iChannel->iTimerDfc.Cancel();
            // ...Cancel device timeouts
            d->iChannel->iCancelled=ETrue;
            d->iChannel->MoveToRetention();
            // this must be synchronous, non-blocking, non waiting
            }
        }
    else
        // race condition: driver was already closed (ECloseMsg) but the PowerHandler has not deregistered yet
        {}
    }
```

This function checks to see if the peripheral is in the idle state, and moves it to the retention state. It stops the inactivity-monitoring timer.

The null thread may run between the request to close the channel being serviced and the driver object being destroyed - at that time the power handler has not yet deregistered itself with the power controller. We need to check for that condition.

Just before leaving the `CpuIdle()` function, the power controller calls the driver using the `iLeaveIdleCallback` pointer:

```
inline static void LeaveIdle(TAny* aPtr)
    {
    // called with interrupts disabled
    DSerialDriverPowerHandler* d = (DSerialDriverPowerHandler*)aPtr;
    if (d->iChannel->iStatus != EClosed) // not closing
        {
        if(d->iChannel->iCancelled)
            {
            // ...Restarts device timeouts
            }
        }
    else
```

```
        // race condition: driver was already closed (ECloseMsg) but the PowerHandler has not deregistered yet
        {}
    }
```

If device timeouts were cancelled, the driver restarts them.

Power manager initiated power down and power up

The power manager calls the power handler's `PowerDown()` and `PowerUp()` functions in the power manager's client context. Their implementation may require lengthy operations or may even block. So it is best if they both schedule DFCs to execute in the driver's context, in this way also guaranteeing that they will not preempt each other.

```
void DSerialDriverPowerHandler::PowerUp()
    {
    iChannel->iPowerUpDfc.Enque();
    }

void DSerialDriverPowerHandler::PowerDown(TPowerState aState)
    {
    (aState==EPwStandby)? iStandby=ETrue:iStandby=EFalse;
    iChannel->iPowerDownDfc.Enque();
    }
```

The power-down DFC moves the peripheral hardware to either the standby or the off state, depending on the target power state, and then acknowledges the transition:

```
void DChannelSerialDriver::PowerDownDfc(TAny* aPtr)
    {
    DChannelSerialDriver* d=(DChannelSerialDriver*)aPtr;
    if (d->iStatus != EClosed)
        d->Shutdown(iStandby);
    else
        // race condition: driver was already closed (ECloseMsg) but the PowerHandler has not deregistered yet
        {}
    d->iPowerHandler->PowerDownDone();
    }
```

When shutting down, we abort all pending requests, cancel timers and DFCs, and stop the peripheral hardware function. The requirements on power resources are reduced to a level compatible to the standby state. If going to the off state, we turn off the power supply.

```
TInt DChannelSerialDriver::Shutdown(TBool astandby)
    {
    ...
    Complete(EAll, KErrAbort); // complete any pending requests
    iTimer.Cancel();
    iTimerDfc.Cancel();
    iPowerUpDfc.Cancel();
    iPowerDownDfc.Cancel();
    NKern::Lock();
    if(astandby)
        iPowerHandler->iPowerState= DSerialDriverPowerHandler::EStandby;
    else
        iPowerHandler->iPowerState= DSerialDriverPowerHandler::EOff;
    iResourceManager->ModifyToLevel(XXXResourceManager::VoltageSerial, 20);
    // request 20% voltage level
    iResourceManager->SharedClock()->Release();
    // relinquish requirement on shared clock
    if(aStandby)
        {
        // ...write to peripheral registers to set peripheral in standby state
        NKern::Unlock();
        return;
        }
    iResourceManager->Modify( XXXResourceManager::PowerSupplySerial, EFalse);
    // turn power supply off
    NKern::Unlock();
    }
```

The power-up DFC moves the peripheral hardware to the idle state by calling the `DoPowerUp()` function that I described previously.

A power manager-initiated move to standby is not instantaneous; a request may arrive after the peripheral has moved to standby or before it has powered back up. The `HandleMsg()` function will defer the request to be serviced until the power handler is powered back up.

```
void DChannelSerialDriver::PowerUpDfc(TAny* aPtr)
    {
    DChannelSerialDriver* d=(DChannelSerialDriver*)aPtr;
    if (d->iStatus != EClosed) // if not closing by client's request
        d->DoPowerUp();
    else
        // race condition: driver was already closed (ECloseMsg) but the PowerHandler has not deregistered yet
        {}
    d->iPowerHandler->PowerUpDone();
    if (d->iMsgHeld)
        {
        __PM_ASSERT(d->iStatus != EClosed);
        d->iMsgHeld = EFalse;
        d->HandleMsg(d->iMsgQ.iMessage);
        }
    }
```

Again, we check the window of opportunity between closing the channel and deregistering the power handler for any power manager initiated transitions, and if any, skip those transitions.

Channel closure and destruction

The closing of the channel results in the sending of an `ECloseMsg` message to the driver. This is serviced by the `HandleMsg()` and results in the shutting down of the driver and the powering off of the peripheral.

We also delete the driver object:

```
DChannelSerialDriver::~DChannelSerialDriver()
    //
    // Destructor
    //
    {
    if (iPowerHandler)
        {
        iPowerHandler->RegisterCpuIdleCallback(EFalse);
        // deregister with CPU idle
        iPowerHandler->Remove(); // deregister with power manager
        delete iPowerHandler;
        }
    ...
    }
```

The destructor deregisters the power handler with both the power manager and the power controller and calls the power handler's destructor.

Emergency shutdown (power loss)

Emergency shutdown is a situation that results from sudden loss of power supply, such as when the mobile phone battery is removed.

There are two possible approaches for handling an emergency shutdown situation:

1. If there is a short-term alternative power source, such as that provided by a SuperCap (high capacitance capacitor), which is capable of supplying power for a few milliseconds, and there is a mechanism for notifying drivers and user-side software components, then the emergency situation can be handled before power failure
2. If there is no alternative power source, and hence no time left to handle the emergency situation before power failure, then the power failure event should be dealt with after the device is rebooted. We must provide a mechanism to mark a shutdown as an *orderly* or *emergency*.

If we have a short-term alternative power source, then the notification of an emergency shutdown should be distributed to a chosen subset of the peripheral drivers:

- Drivers for peripherals that draw a significant amount of power should shut down first and as speedily as possible after the notification. These might include the display and backlight, hard disk drives, and so on
- Drivers for peripherals that may be affected by a sudden loss of power, such as media drivers for external storage media that are susceptible of media corruption in the event of power loss, can then finish their current operation. For example, they can complete their current sector write, and shut down gracefully.

If there is no short-term alternative power source, peripherals will just power down on power loss without any finalization. Upon rebooting, the system checks each of the critical peripherals for possible corruption, and attempts to fix it. The file server scans every internal persistent media drives that are marked as *not finalized*, and fixes up any errors.

Removable media may also be corrupted by sudden power loss, so they will be scanned on notification of insertion.

In a system where there is a backup power source capable of guaranteeing the preservation of the contents of volatile system memory, it is possible to complete any aborted writes to persistent memory, as the data will still be in SDRAM when rebooting.

The kernel framework does not currently have a built-in mechanism for the distribution of emergency shutdown notifications. However, if battery monitoring is implemented at the framework level, you can implement such a mechanism, along the lines of the one I described in Section 15.3.1.3. The sudden power loss event should be serviced as speedily as possible, which is done best if it is capable of interrupting instruction execution - this means that the battery monitor component should hook a hardware interrupt (on ARM, an FIQ) to the event.

The driver code that services the notification should also handle it as speedily as possible - for example, it should complete the minimum of work to guarantee that the media will be restored when rebooting, and then power down. No time should be wasted completing requests or waiting for freed resources to reach their final state.

Managing idle time

CPU idle time

The typical utilization profile of a hardware component, CPU or peripheral, is usually characterized by brief periods of intense activity with high requirements on processing bandwidth, followed by longer periods of idleness (Figure 15.8). This matches the usage model of most mobile phones: the device is left constantly on, even when it is in someone's pocket or left downstairs for the night.

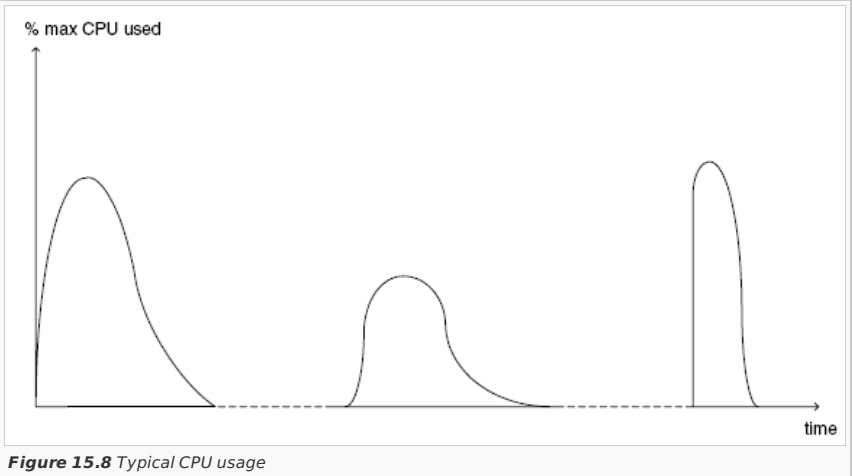

Figure 15.8 *Typical CPU usage*

Bursts of CPU activity may be triggered by user interaction, or interrupts from a peripheral or a timer working in the background. This activity is linked to

higher power consumption whilst the periods of idleness are linked with the entering of a power-saving mode, provided by most modern CPUs. These can be entered automatically, or under software control, to save power during periods of low activity.

I have previously provided a definition for these power-saving modes - or retention states - and I have examined the support that exists, or needs to be implemented in the framework, to move the CPU to them. I will now discuss the utilization of idle time.

Choosing a CPU retention state

As we have seen, the kernel notifies the power framework of when the CPU enters a period of inactivity by calling the power manager's `CpuIdle()` function. This function must decide whether to reduce the availability of the CPU by moving it to a low power retention state and reduce power consumption, or keep it in a more available state, with less or no power savings. There are also usually several gradations of retention that we can select.

The first factor in the determination of the retention state is the estimated uninterrupted length of idle time. We have already described how this idle time can be obtained from the kernel. Moving in (and out) of a retention state requires preparation, and usually, the more complex the preparations, the more the power savings that will result. A more power-efficient retention state usually requires a longer wakeup time too. Therefore, the longer the estimated idle time, the more power-efficient the selected retention state can be.

Another factor is the power resource utilization at the time when the decision is made; due to the interdependency between resources, it may only be possible to move the CPU to a low power retention state if certain resources are already off or being used only at a low level. The higher the power savings on a retention state, the lower should be the overall resource utilization profile. Also, on moving the CPU to retention state, more peripherals can be turned off or have their power requirement levels lowered, resulting in even greater power savings.

When choosing the CPU retention state, we need to take into consideration the state of certain peripherals - mainly the ones used for data input or the detection of unlatched external events.

Often the user will enable certain peripheral functions and leave them inactive but in an operational state for long periods of time: this often happens when infrared or Bluetooth are enabled, or when an I/O function card is connected to an externally accessible peripheral bus. If these peripherals are servicing a request when the null thread is entered, then they must be left operational, and this in itself could prevent the moving of the CPU to a more power-efficient retention state.

These peripherals have the ability to interrupt the CPU idle mode and request CPU processing time. Investigating the power resource state may give a view of what peripherals are in the active state and which have already powered down, thus helping us to choose the retention state.

Waking up from a retention state usually takes a fairly long time. We must choose a CPU retention state whose wakeup time will permit the correct operation of all peripherals that were left operational, and lead to the servicing of their requests for CPU attention on time - that is, with no data loss, correct and with the timely servicing of events.

The analysis of past CPU workload may be relevant for the choice of retention state, as it can give an indication of the future requirements. For example, if an episode requiring high CPU bandwidth is suddenly followed by a period of inactivity then it is probable that the task yielded, waiting for some hardware event to occur. Once the hardware event happens, the task can be expected to resume as soon as possible. In this case a retention state with a lower latency should be chosen.

The current battery charge level may also be important in the choice of a retention state. If the level is low, this does not necessarily lead to the choice of a less power hungry retention state. In fact, as the charge level approaches a critical threshold when applications and drivers should be notified, the decision to move the CPU to retention must be carefully weighed against the need to wake up on time to service the sending of the notifications.

SDRAM power management

When in a retention state, the CPU is usually unable to refresh SDRAM - and without cell refreshing, in which the cell charge level is periodically restored, the contents of memory will be lost. We can place SDRAM devices in a self-refresh mode, where their internal controller takes over the duty of refreshing the memory cell charge without the external CPU intervention. All we need is to maintain a power supply to the device and to supply a clock source to it - then we send a command to the SDRAM controller. Once this is done, the CPU can be moved to retention mode.

If we are using mobile SDRAM, we can reduce power consumption during the periods when the CPU is sleeping, by enabling self-refresh only for the memory banks that are actually in use.

To do this, we need to ensure that pages containing valid data are arranged to occupy as few memory banks as possible. This means that periodic re-organization of the memory pages, or defragmentation, needs to take place. Mobile SDRAM defragmentation typically happens during periods when the CPU has no other tasks to perform - that is, in the null thread. When the CPU is defragmenting memory, it cannot be moved to a retention state.

The power savings that result from a partial refresh of the memory device need to be carefully weighed against the increased overall power consumption stemming from the reduction of CPU idle time. You might achieve a balance by using only part of the idle time available for memory defragmentation, and using the rest as a power-saving retention state time.

Interaction between CPU retention and peripheral operation

If, when the CPU enters an idle period, the software investigates the state of peripherals and acts on them, it may further reduce power consumption:

- If the peripheral driver is not servicing any request, or if it is waiting on a signal from its client, it may be possible to move the peripheral to a retention mode. Its client will not request its services again until the CPU wakes up. Moving as many peripherals as possible to the retention state usually leads to freeing power resources
- Peripherals that are responsible for detecting wakeup events may power down to a high latency state and only leave the systems responsible for detecting those events powered up and operational. Given the wider time constraints associated with user input, the latency has no impact on the ability to service the input
- Even if the peripheral cannot be transitioned to a low power state, we may be able to take other actions that result in increasing the idle time, such as skipping periodic device timeouts or increasing their period
- We can turn off the LCD backlight, and lower the LCD refresh rate
- We can notify external devices on a peripheral bus such as MMC or USB, so that they may enter a low power mode.

Obviously, some of these actions need to be reversed when the CPU leaves the retention state.

Event reduction

Certain applications use periodic timers to poll the state of particular software resources. These timers will wake up the CPU from its retention mode, only for the application to realize that no change to the resource has been made. The effect of this is to shorten the period that the CPU can be in a retention state, making the choice of more complex and power-efficient states impossible. At the worst, if these timer ticks are too frequent, they may prevent

transitions to the power-saving retention states altogether.

Because of this, an effort has to be made to reduce the use of such timers and move to an event-driven architecture whenever possible.

Some peripheral drivers use interrupt driven I/O for data exchanges with the peripherals they control; this also has a negative impact on CPU idle time, as events are generated at a high rate, to signal transfers of small units of data. A better alternative is to use DMA, which enables transfers of larger amounts of data with a much lower signaling rate. This is especially relevant as some CPUs may go into a retention state while the DMA controllers are operational.

Display drivers for refreshed displays whose frame buffers are placed in system memory can be optimized for event reduction. During periods of CPU idle, no new display content is being generated, and no updates to the frame buffers occur. It is also unlikely that the user is interacting with the mobile phone. Therefore we recommend two different policies for LCD refresh rates: one that refreshes the LCD at the *normal* rate when the CPU is active, and another, for when the CPU is idling, that lowers the refresh rate and relies on the persistence of the display for longer periods of time in between refreshes. Obviously, lowering the LCD refresh rate increases the intervals between the CPU having to wake up and service DMA requests to refresh the display.

With the introduction of *Smart LCD* panels with their own controller and memory, the control model can be simplified; these displays can be placed in a mode in which they refresh from their internal frame buffer. This buffer keeps the last frame sent to the controller. The display can therefore be disconnected from the CPU bus during periods while this is in retention mode.

Peripheral idle time

Peripheral devices may spend considerable time idling. Even when a peripheral driver is controlled by a device driver that has an open channel, it might happen that no requests for service will be issued for considerable periods of time.

Earlier, I mentioned that if a peripheral is idling then it could be moved to a low power retention state. Peripheral drivers cannot estimate when their clients will issue requests for their services. Thus, the decision to move the peripheral to a retention state depends on that peripheral's ability to wakeup when a request is issued, and to service it on time without compromising the performance of the client.

Advanced power management

A number of improvements to the kernel power framework are being considered in line with the current developments.

CPU workload prediction and voltage and frequency scaling

Power consumption of an electronic component, such as a transistor or a gate, is directly proportional to the operating frequency and to the square of the operating voltage:

$P = K \times f \times V^2$

Hardware manufacturers have been taking advantage of this with improvements in the utilization of the physics of the silicon which allow electronic components to work at lower voltages and higher frequencies, without increasing the overall power consumption, as the previous formula clearly shows.

This static model has its limitations: as the transistor's operating threshold voltage is lowered, so the leakage current increases, resulting in the increase of static power and increased dissipation (which causes additional problems in removing the additional heat).

More recently, another approach based on dynamically varying the factors that contribute to power consumption has been favored:

- Hardware manufacturers design devices (CPU, peripherals) for which voltage and frequency can be dynamically adjusted without disruption of operation
- The operating system uses this feature to always require the lowest power consumption from the CPU without reducing the perceived performance of the system.

Another look at the physics of the silicon tells us that when reducing the supply voltage of a switching gate, the propagation delay across that gate increases. In other words, a reduction in operating voltage of a hardware component such as the CPU must be accompanied by a reduction of operating frequency.

The reverse of this principle may be used in favor of lowering the power consumption; if the frequency is reduced, the operating voltage can be reduced accordingly. Let us see how this could be beneficial.

Analyzing the operational cycle of the CPU reveals a *bursty* profile (Figure 15.9): tasks or episodes are executed at nominal clock frequency followed by *gaps* corresponding to periods of idle time.

If the clock frequency of the CPU was adjusted to allow each episode to complete before the next one, no degradation of system performance would occur (Figure 15.10).

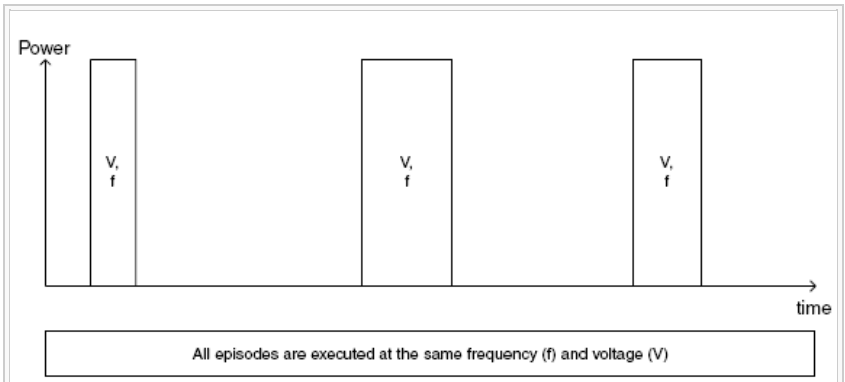

Figure 15.9 *Typical CPU episode profile (assuming low power idle time)*

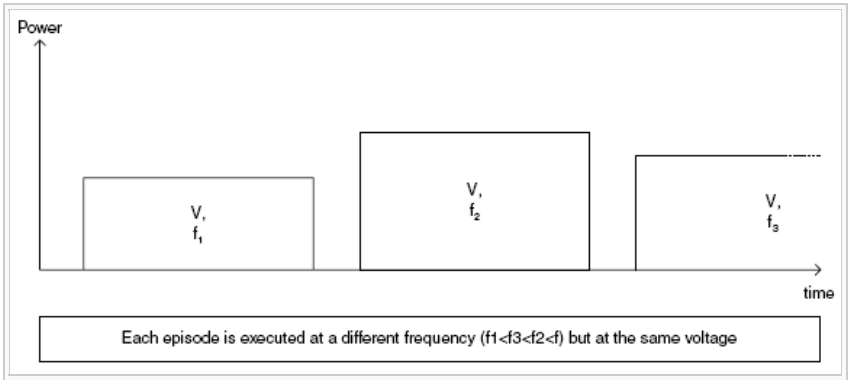

Figure 15.10 *Episode profile with frequency adjusted per episode*

It must be noted that if the total energy per task (the area inside each of the boxes) remained the same, no overall gain in power savings would occur. In fact the power performance would be poorer, as with the reduction of idle time no power savings could be made from moving the CPU to a retention state.

However, if we lower the clock frequency per task, we can lower the voltage supply to the CPU accordingly, resulting in a significant reduction in power consumption (Figure 15.11).

If, at the moment the CPU enters an idle period, it is possible to predict when the next episode is going to require CPU attention, then it is possible to continuously adjust the frequency (and voltage) and still allow each episode to complete before the next one is due to start.

Algorithms which perform an analysis of idle time and predict CPU workload in real time have recently been developed. These algorithms require the kernel, which is responsible for scheduling the tasks and tracking the idle condition, to be instrumented to collect the relevant information.

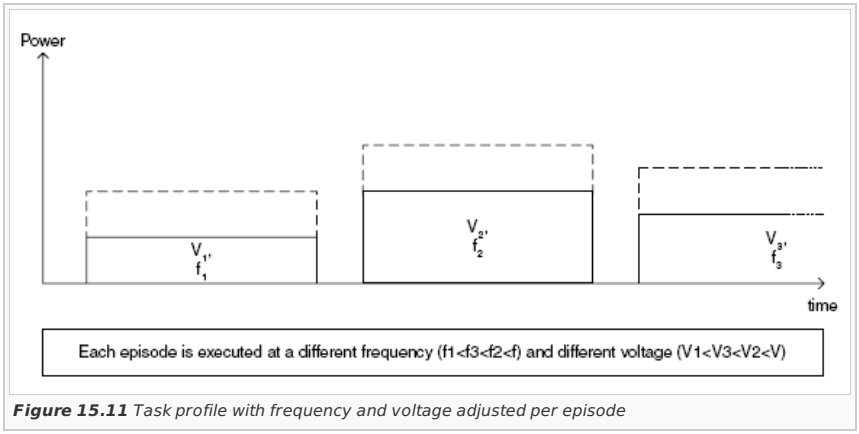

Figure 15.11 *Task profile with frequency and voltage adjusted per episode*

Usually the software component that contains the algorithms that perform the prediction and ultimately decide upon an operating point is a higher level component. It may be able to receive input from certain critical applications, which *hint* at a required performance level, and may support different switchable policies corresponding to different modes of device operation - such as *gaming*, *media playback*, *callonly mode* and so on.

The system has requirements on the kernel framework:

- APIs to allow collecting the workload information from the kernel
- An interface to the resource manager to allow the modification of the operating parameters (voltage, clock frequency)
- May have an interface to certain critical device drivers to allow those to request a performance level. Device drivers may have a knowledge of the probability of unpredictable future events that require CPU attention (data input, interrupts), and this information is not available to the workload prediction component.

A possible optimization to the *just-in-time* strategy I have just described might involve searching for periods in the CPU operational cycle when performance is independent of clock frequency. Examples of these include activities related to periods of intensive memory or I/O port access, when the CPU has to wait for these to return the data, or polling of an I/O port. These tend to be fairly common for wireless peripherals and disk I/O.

Identifying such periods will make it possible to lower the CPU clock frequency (and voltage) to match that of the peripheral it is accessing, resulting in energy saving without impacting the performance of the task.

Finally, it is possible that for certain CPU loads the strategy of lowering the clock frequency and operating voltage to a level that still allows deadlines to be met may result in less power savings than would be possible by running the episodes at a higher frequency and then saving power by transitioning the CPU to a low power mode when it is idling - even during the short periods in between tasks. It must be noted that neither transitioning the CPU to and from a low power mode nor changing the speed and operating voltage is cost free; energy is spent on both preparing the transition and in the transition itself and there are latencies associated with both. The software component responsible for setting the operational point for the CPU operation needs to be able to make the decision about what strategy results in greater economies of power but still meets the service requirements.

Peripheral low power states and quality of service

As I mentioned earlier in this chapter, some peripherals may be retention state capable. The retention states that they can be transitioned to are characterized not only by lower power consumption, but in some cases also by a higher latency, that is, a diminished ability to respond to external events within the time constraints required for the correct operation of the peripheral.

The peripheral driver should make the decision to move the peripheral to a low power state and choose a state based upon:

1. Which point of the operational cycle the driver is in: is the peripheral idling; are there any pending requests for service?

2. Is the peripheral able to detect incoming data or external events whilst in that state?
3. Is the response time of the peripheral to incoming data or events whilst in that state within the service constraints associated with a pending request for service?
4. For events that repeat, for example a stream of data, the transition time to the active state is important: even if the initial event is detected and transition to active started, will the peripheral be able to detect/service the next event?

It may be possible that a peripheral reaches an idle condition even when the peripheral driver has a pending request for services. It may be still possible to transition the peripheral to a low power state if that state's latency does not have an impact on the peripheral's ability to service the request. As an example of this, consider keyboard or touch screen drivers, where the associated peripheral still has the ability to detect a key press or touch sensitive panel tap and generate an interrupt, even when the peripheral is in a low power state. Given that the clients of these peripherals have very tolerant constraints for servicing those events and their repeat rate is usually of the order of tens of milliseconds (after debouncing filters have been applied), it is quite natural that these peripherals be moved to a low power state every time they finish processing an event (even though their operation implies they need to be ready for the next incoming event).

However it may also be that the client of the services provided by the peripheral can, in certain stages of its operational cycle, be more tolerant to the peripheral lowering its response time to input data or events, even though in other stages it has a much more stringent requirement on the peripheral. In other words, the quality of service required from the peripheral may not always be constant.

This situation is particularly common with peripherals used for tracking and servicing input data streams. As an example consider an IR peripheral: the constraints on response time during the discovery phase of the operation, when devices search for the presence of another device with which to initiate a transaction, are considerably lower than when the devices are already engaged in a transaction. The protocol even makes allowance for loss of data during that phase.

Another common situation relates to peripherals that even in their operational state may be able to work with different levels of power resources such as clocks or voltages on power lines. Their responsiveness to their client's requests varies according to their requirement on those power resources.

Therefore it is possible to envisage a system where clients of services provided by peripherals negotiate the quality of service provided by these peripherals with their drivers. To achieve this, special APIs need to be put in place. We will call them peripheral quality of service (QoS) APIs.

Peripheral QoS APIs allow peripheral drivers to know, at any time, the quality of service required by their clients. There are two ways in which peripheral quality of service specifications may be implemented:

1. The peripheral enters an idle period but has a pending request for service. The peripheral driver notifies its client. Then it's up to the client to allow or disallow the relaxing of the quality of service. When given permission to relax quality of service, the driver will adjust this according to its own needs
2. The client specifies the quality of service required for each request of service from the peripheral when placing the request. This may be expressed as, for example, a percentage of the maximum degradation allowed for the request, or as a range of discrete values, and it takes into consideration the requirements of the client, not the driver.

Peripheral QoS APIs allow the device driver to ensure the lowest requirement on platform power resources at all times and initiate the peripheral transition to a low power state whenever the quality of service required by its clients permits it. This can happen even with a pending request, not only when the peripheral is idling. This results in further lowering power consumption of the entire system.

Depending on which of the methodologies for setting the quality of service provided by peripherals is implemented, the strategy for transitioning peripherals to a low power state and the impact on system power and performance is different:

1. In the case where the peripheral driver notifies the client of an idle condition and receives permission to relax quality of service, it will transition the peripheral to a low power state, releasing the requirements on power resources (turn voltages or clocks off or lower their values). If the peripheral is in that state and the CPU enters its idle mode, the routine responsible for investigating the state of power resources sees the resources used by that peripheral as unused and assumes the CPU can safely be transitioned to a retention state without affecting the performance of the peripheral's driver or its client. Therefore the client of the services offered by the peripheral needs to make the decision to allow the relaxing of the quality of those services based not only on peripheral wakeup time but also upon CPU wakeup time, possibly from the retention state with the longer wakeup time
2. In the case where the client sets the quality of service required for each request, the peripheral driver will know if it can transition the peripheral to a low power state when the peripheral reaches an idle period whilst servicing the request. The choice of a peripheral low power state is determined by the requirement on service quality. This strategy allows a finer granularity of control, and supports multiple peripheral low power states. The drawback is that when the CPU reaches the idle mode the framework will need to investigate the quality of service required for the request the peripheral is servicing with each relevant peripheral driver before deciding on the CPU retention state to move to.

It is possible that for some peripherals one of these methods is preferable to the other; for others a combination of these two methods may be preferable, with the client setting the QoS for each request, allowing the driver to map to the lowest possible requirement on power resources, and then notify its client of periods of inactivity and receive from them confirmation that it is valid to transition the peripheral to a low power state.

Matching of energy sources and loads

The operational curve of a battery - relating the power it supplies to the current that is demanded of it - is only linear for a small region of that curve. Other factors such as temperature and age also affect its ability to release energy. Therefore the energy it releases for a given level of charge is not constant.

The duty cycle of a phone - the cycle of run/idle - creates a variable peak-to-average energy consumption which speeds up the discharge of the battery and reduces its useful life (Figure 15.12).

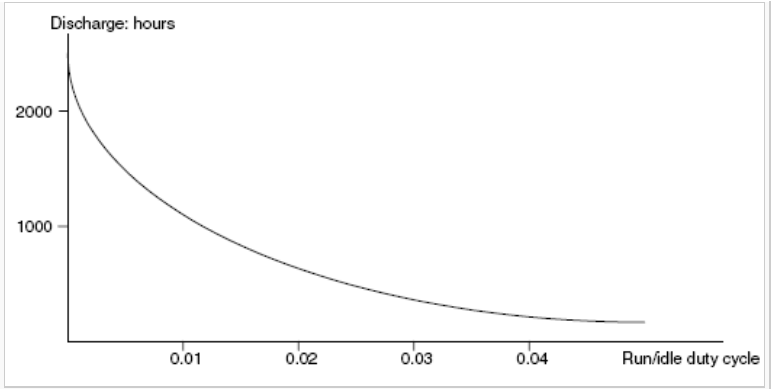

Figure 15.12 Battery *discharge versus duty cycle graph*

Mobile phone manufacturers may decide to incorporate multiple energy sources in their designs, to be used in conjunction with the main battery or as an alternative power source. Those energy sources may include SuperCaps, rechargeable buffer batteries, and so on.

These energy sources may be switched in and out of the energy supply to supplement or replace the main battery, matching corresponding load changes or providing backup for the main power supply. This should be done under software control.

Software will monitor the load and use a framework to switch the sources in or out.

SDRAM partial refresh

Mobile phone designs may include mobile SDRAM components capable of partial array self-refresh. This feature allows for the power consumed during self-refresh to be directly proportional to the amount of memory refreshed. The memory device is organized as a number of power banks and software can set the number of banks that can be self-refreshed, starting from one end (Figure 15.13).

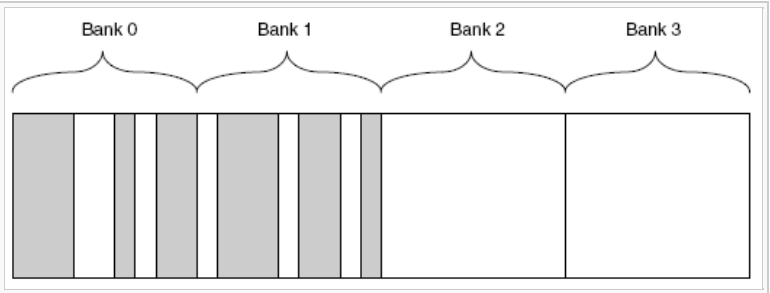

Figure 15.13 *Banks of SDRAM on a mobile phone*

SDRAM can be placed in self-refresh mode during the periods when the CPU is not accessing it, for example, CPU standby and retention states. To make use of the features provided by this type of memory and reduce system power consumption, the framework must be able to:

1. Identify the geometry of the memory device - the number of power partitions and how they map to physical addresses
2. Track the utilization of pages on each power partition
3. Guarantee an optimal utilization of physical RAM, with the page frames in use arranged to reside on as few power banks as possible, and all at one end of the power partition list
4. Provide the functionality to enable partial self-refresh when necessary.

The geometry of the device is determined by the bootstrap and passed to the kernel via the super page. During the early phases of nanokernel initialization, the RAM allocator object that deals with the mapping of memory used by the OS to physical RAM is created and can map the number of power banks and the number of pages per power bank, thus creating a power partition address map.

Whenever physical memory is allocated or freed, the RAM allocator marks the pages as used or free. When memory is allocated, the algorithm that maps it to physical RAM should attempt to find the required number of pages at the lower end of the power partition address map. This may not always be guaranteed. Also, when physical RAM is freed, gaps will be left in the RAM address map which may or may not be fully re-used when an allocation of a number of contiguous pages less than or equal to the number of pages freed takes place.

Therefore the OS may need to implement a more aggressive strategy for rearranging physical RAM, such as ensuring that used pages are all at contiguous physical addresses at the low end of the power partition address. This defragmentation of physical RAM may be a time-consuming operation, and this must be taken in consideration when deciding when it needs to run. One option is to launch the operation when the CPU has no other threads in the ready list, that is, when the null thread is scheduled to run. In Section 15.3, I mentioned the risk of the defragmentation routine encroaching itself into CPU low power retention time - therefore there has to be some mechanism in place to allow the coordination of the triggering of the defragmentation task, its duration and the `CpuIdle()` routine.

A possible defragmentation algorithm investigates if the number of used pages at the higher address end of the power partition map is less than the number of free pages at the lower address end, and, if it is, starts copying those pages. The algorithm must also be able to determine if it is be able to copy all the pages within the allocated time. If it is interrupted before it completes the copy, it should abandon copying and relinquish control of the CPU to any other thread that needs to run as a result of the interruption. When it finally resumes, it needs to be able to determine which pages were copied and which were abandoned, as well as whether it should retry copying the pages that were abandoned, or if those have been invalidated as a result of the interruption.

Finally, the routines which prepare the CPU and platform to enter the standby or retention states must be able to obtain a list of what power banks are used and their locations, mark only these to be refreshed, and power down all others.

Summary

In this chapter, I have described the power management framework of Symbian OS in some detail. Next I shall look at how Symbian OS boots up.