Symbian OS Internals/16. Boot Processes

- Symbian OS Internals Table of Contents

by Andrew Thoelke and Carlos Freitas
with Jon Coppeard

There is only one satisfying way to boot a computer.
J.H. Goldfuss

A description of how Symbian OS operates, manages hardware resources and provides services for application software is incomplete without an explanation of how it takes the hardware from an uninitialized, powered-off state to one in which the system is fully ready for action.

In this chapter I will walk through the boot process for mobile phones that run the operating system from execute-in-place (XIP) Flash memory, such as NOR Flash, and then describe the differences needed to support non-XIP media, such as NAND Flash.

The opposite process, shutdown, also deserves some attention, so I will go on to explain how Symbian OS responds to a request to switch off.

Operating system startup

The process of *bootstrapping* an operating system is a carefully managed operation. The different stages of initialization must be correctly sequenced to avoid services being initialized before the services they depend on are ready.

Figure 16.1 illustrates the main stages of OS initialization for an XIP Flash memory device. However, before I explain why the memory technology makes a difference to the boot process, we should start at the very beginning, or perhaps even a little earlier.

To successfully initialize the hardware and OS, it is important to know what state the hardware will be in immediately after it has been switched on or reset. For the most part, the OS has to assume that hardware is in an *unknown* state because the boot process may arise from several causes.

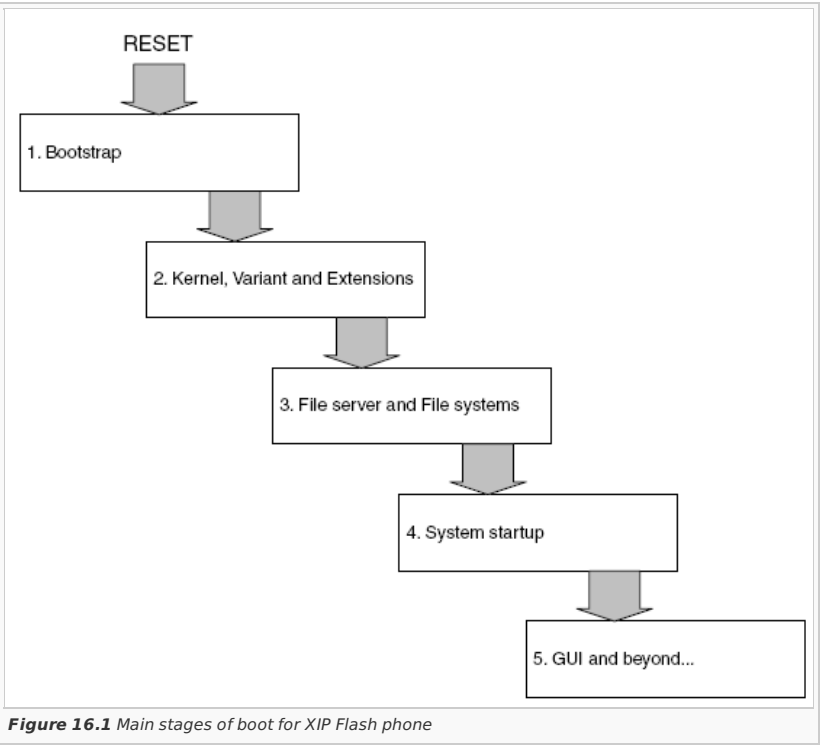

Figure 16.1 *Main stages of boot for XIP Flash phone*

For example, after a mobile phone is switched on, the CPU, MMU and memory controller are in the most primitive state: clocks are set to low frequencies, the MMU is disabled, only memory essential for reset is visible and RAM will contain garbage. On the other hand, following a software reset these components are typically already initialized. The initialization carried out during the boot process must be robust enough to handle any reason for reset.

Switching on the phone triggers the CPU and MMU to reset. This disables the MMU and causes the CPU to jump to a well-known location to execute the reset code. On ARM CPUs, this is address 0x00000000, which is usually referred to as the reset vector. Obviously there must be some code at physical address zero for this to work and hence some hardware - usually this will be some masked ROM or XIP Flash.

Mobile phones typically use some form of Flash memory to store the OS image and built-in software. Although this is significantly more expensive (and slower) than masked ROM, there are two substantial advantages:

- Mobile phones are complex products and often require an update during their lifetime. Flash memory enables the OS to be over-written or upgraded (*reflashed*)
- Masked ROM takes time to manufacture. This introduces a delay of several weeks between the software being ready and the production of the phone. *Several weeks* is a lot of phone sales.

Some types of non-volatile memory, such as NOR Flash, can be treated by the memory controller as directly accessed, read-only memory. This allows program code to execute directly from Flash memory, which makes initial system startup much simpler. It is interesting to compare this to desktop

systems where the operating system resides on hard disk and must be loaded into RAM before it is possible to start executing any of the code. There are some Flash technologies, such as NAND Flash, that cannot support execute-in-place and thus require a boot process that more closely resembles that of a desktop OS. In Section 16.2.1 I will look at how Symbian OS supports booting from NAND Flash, and discuss why you might choose to boot from NAND, given this inefficiency.

But now, let's return to the boot process.

High level process

As the OS starts up, the boot process runs through, broadly speaking, five stages. I say *broadly* because the later stages can be broken into more discrete steps. However, this book is primarily about the operation of the kernel and file server, so as the later boot stages are related to general OS services and user interface initialization I will not cover them in depth.

Figure 16.1 illustrates these main stages. The beginning of each stage is marked by the execution path entering a specific new process within the operating system, and each stage ends when the services provided by that process are ready for use by the OS.

Before examining each stage in more detail, I will provide an overview of each stage and what it achieves before it hands over to the next process in the chain.

1. Switching the phone on triggers a hardware reset. The first software that is executed after this is known as the bootstrap. On entry to the bootstrap, the execution environment is very primitive, and the bootstrap is tasked with providing a basic execution environment for the kernel process
2. On entry to the kernel, the CPU is now running at full speed and an execution stack allows typical C++ code to be run. However, there is still only a primitive memory environment and only one execution path. Static data is now initialized and interrupts are masked. Once kernel and base support package (BSP) initialization is complete, there is full management of CPU, memory, power and peripherals and the second OS process is started
3. At this stage in the boot there is a fully functional micro-kernel that supports multiple multi-threaded, protected user-mode processes. However, the OS has not yet provided the means to instantiate new processes, to extract file-based data from the Flash or to persist data in read/write Flash memory. The process of establishing all of these services falls to EFILE.EXE and its supporting process, ESTART.EXE
4. All of the kernel, user library and file server services are now fully initialized ready for the rest of the OS to begin its boot process - a job that is given to the system starter process. The system starter manages the initialization of the rest of the OS system services in an ordered manner, and can also provide monitoring and restart for those which terminate unexpectedly. The precise order in which the persistence, communications, multimedia, security and other services are started is controlled by a script and is phone-specific
5. Once enough of the system services are running, the primary GUI service, the window server, can be started. This now allows initialization of the other UI services and the applications that make up the aspect of the OS that is most evident to the user.

This is a standard sequence for booting the phone. According to this process, the OS services for displaying images or animations, or playing audio become available quite late in the boot. This does not reconcile with the typical experience of using a Symbian OS phone - in practice phones often use lower level graphics services to display images during boot without demanding that the full screen-sharing capabilities of the window server are available.

It is worth looking in more detail at some of the work done in the earlier stages of boot to understand how the OS brings the services online in a carefully orchestrated manner. Many of the interesting events or activities can be seen in Figure 16.2, which provides a graphical *history* of booting EKA2 on development hardware.

The horizontal lanes each show a particular related area of the OS or hardware that requires initialization during startup - for example, activities that are to do with initializing memory and memory services are in the central lane.

The four main vertical divisions show when execution passes from one executable to another during boot. The first two each represent one of the stages referred to earlier, while the last two together make up stage three of the startup. The total elapsed time for these stages of boot is less than one second.

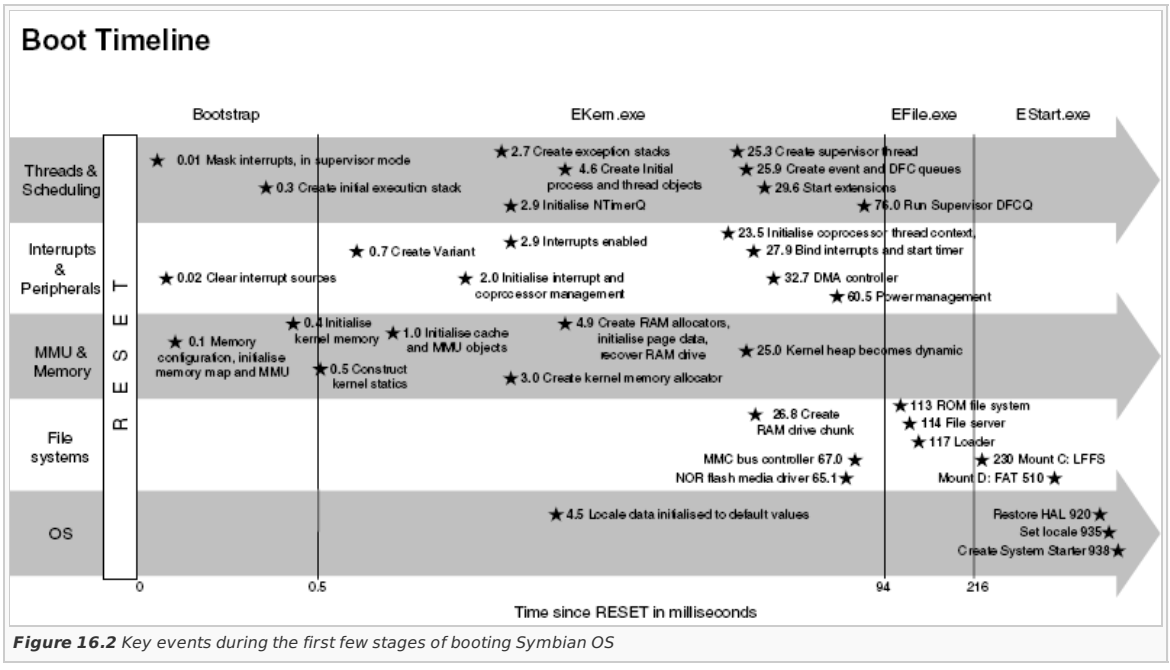

Figure 16.2 *Key events during the first few stages of booting Symbian OS*

The bootstrap

On entry to the bootstrap, bootrom.bin, very little is certain about the processor or hardware state. The CPU and memory will be set to run at safe, slow speeds, the MMU is disabled and the only CPU general register that is known to be good is the PC in supervisor mode (R15_SVC) - and that will be zero at the very start of the bootstrap.

Immediately, the bootstrap modifies the CPU mode to run in SVR mode with all interrupts masked. Any interrupt or exception at this stage would cause a fault, because the CPU registers for these states have not yet been initialized. The bootstrap then initializes hardware, starting with the CPU and MMU, and clears interrupts. The bootstrap creates an execution stack and points the supervisor stack pointer, R13_SVC, to this memory.

The superpage

From here, the bootstrap starts populating a structure called the superpage. This is a structure that is shared between the bootstrap and the kernel and is used for passing information determined by the bootstrap to the kernel. This is information such as addresses of key ROM locations (such as the root directory), location and size of memory banks, and various other values that are calculated by the bootstrap. The superpage is always allocated at a known virtual address, specific to each memory model - which is how the bootstrap and kernel know where to put it and find it. All other bootstrap data is lost after completion.

The bootstrap determines the available memory, both RAM and ROM. It does this using a combination of explicit information in a table provided by the BSP, and dynamic probing to determine memory address, size and width. At the end of this, the superpage will contain a record of all the blocks of memory in the system. With this information, the bootstrap creates and initializes the RAM page allocator. Next it allocates, clears and maps the primary page directory and first page table, then maps the ROM and other memory model-related data structures. It also maps hardware I/O in the page tables. At this stage, the MMU is switched to use virtual addressing.

Now the bootstrap is ready to prepare the kernel for execution. It allocates the initial thread stack and initializes it according to the size requested in the ROM. Similarly, it allocates and initializes the kernel static data.

After some final BSP-specific initialization, the bootstrap is ready to execute the kernel. The supervisor stack pointer is changed to point to the initial thread stack (the bootstrap stack is no longer used) and the CPU branches to the entry point of the primary executable in the ROM, usually EKERN.EXE.

The kernel

On entry to the kernel, the CPU is now running at full speed, the execution stack allows typical C++ code to be run, the memory hardware sub-system is tuned and the MMU provides virtual addressing. However, the memory environment is still primitive - the static data is initialized but there is no free pool allocator and there is only one execution path. Interrupts are still disabled.

The kernel starts by initializing all the CPU execution modes, although they will all share a common stack for now. This enables exceptions to be detected and diagnostics produced rather than generating errors that are difficult to debug. Next the kernel runs C++ constructors for static kernel objects, and we are ready to enter the C++ entry point for EKERN.EXE: `KernelMain()`.

The kernel performs its initialization by running a number of initialization routines, init-0 to init-3. At each stage, core kernel initialization is performed and equivalent initialization routines are run in the BSP to allow the initialization of phone-specific hardware.

Init-0

From `KernelMain()` the initial thread first invokes `P::CreateVariant()`, which does phase zero of the initialization. That is, it initializes and constructs the static data objects for all BSP extensions, and then does the same for the variant. The variant must be initialized after the extensions, as the initialization of the variant may depend on one or more extensions having been through this phase already. Finally, the variant's exported `Initialise()` function is called to do early initialization and provide the kernel with a pointer to the singleton Asic object.

Init-1

Now that the BSP static objects are initialized and ready, the second action the initial thread takes is to call `K::InitialiseMicrokernel()`. This prepares various kernel and BSP objects in sequence:

- The Mmu and Cache management objects
- Coprocessor management
- Interrupt dispatchers
- Some ISRs
- Exception mode stacks for IRQ, FIQ, ABT and UND modes.

At this point, some three milliseconds into the boot, the kernel can unmask interrupts safely, although there are no interrupts enabled yet. The nanokernel timer period is now set from the variant information.

The kernel creates an initial free store allocator over the memory reserved by the bootstrap, but this heap is not yet able to grow dynamically or support multiple threads. However, the kernel can now support dynamic object creation:

- The kernel initializes locale data to default values
- The kernel creates a `DProcess` object to represent the EKERN process
- The kernel creates a `DThread` object to represent the currently running initial thread, which will become the *null* (or idle) thread.

The kernel now unlocks the scheduler and is ready to start scheduling threads. The kernel creates the supervisor DFC queue but does not yet create the thread to service the DFCs. This allows `TDfc` objects to be created, bound and even added to this DFC queue during early initialization - once the kernel creates the thread, it will process the queue and execute these DFCs.

Init-2

We now have a viable heap allocator and scheduler. The next step is to complete initialization of the memory manager. This creates all the virtual address region allocators and the RAM allocator mutex used to protect the manager. It also recovers any contents of the RAM drive that survived a warm reset.

The final coprocessor initialization creates the default thread context for saved coprocessor state.

Now the rest of the kernel resource management machinery can be brought into existence:

1. The object containers are created with their mutexes, and the initial process and thread are added to these
2. A chunk representing the already existing kernel heap is created and set up, and the heap's mutex is created. A second chunk to contain supervisor mode stacks is created. The kernel heap is now *mutated* into a dynamic heap - the kernel now has full memory allocation capabilities
3. The debugger interface is initialized
4. Publish and subscribe is initialized
5. The power model is initialized

6. The code management system is initialized.

The final act of the initial thread during boot is to create and resume the kernel supervisor thread - startup now continues in the context of that thread.

Eventually the null thread will run again, and will then enter a loop in `KernelMain()`, repeatedly running a low priority garbage collection process, and after that requesting the Asic object to take the processor into some form of idle state.

Init-3

The supervisor thread now initializes several more kernel services:

- The kernel side of the HAL system
- The event queue
- The default DFC queues
- The RAM drive chunk
- The tick- and second-timer system.

The supervisor then invokes the variant's init-3, which will at last enable the kernel timer services - and, now that the kernel is fully functional, initialize various other interrupt handlers and sources.

Finally, the supervisor now runs the export entry point for each extension in turn, in this way initializing the phone's BSP piece by piece. There are many more extensions than are shown in the timeline in Figure 16.2. I have only included a few of the key ones that provide services used elsewhere in this chapter.

The last such extension is always ExStart - this is dedicated to constructing the second process *by hand*, and so initializing EFILE.EXE and resuming this as the first user-mode process.

The file server

At this stage in the boot, we have a fully functional micro-kernel that supports multiple multi-threaded, protected user-mode processes. However, the OS has not yet provided the means to instantiate new processes, to extract file-based data from the Flash or to persist data in read/write Flash memory. The establishment of all of these services falls to EFILE.EXE, the designated secondary executable, and its supporting process ESTART.EXE.

EFILE starts by creating the infrastructure for the file server:

- The secondary thread for processing disconnect requests
- The object container classes for shared resources
- The `TDrive` objects providing the logical media interface to the file systems.

The file server mounts the first file system (the XIP ROM file system on drive *Z:*) manually, and this allows the file server service to be created.

The file server now creates the loader thread, which provides the executable loader service for the rest of the OS, and waits for it to signal that it is ready. The loader initializes itself by creating the server objects and initializing its filename cache.

The file server now completes initialization of the local drive collection and prepares its notification service before creating a final slave thread to continue startup. Finally it begins to service file server requests.

The OS can now service load requests against executable files found in the XIP ROM (as this is the only file system that is mounted!). The slave thread picks up the trail and now makes requests to the loader to install the local media sub-system drivers and physical media drivers, preparing the file server for mounting the full set of file systems. The last action of the slave thread is to create the second user process: ESTART.EXE.

ESTART does phone-specific initialization, which is why it is separated from the generic EFILE.EXE. ESTART initializes the local file systems one by one, installing and mounting the required file systems over each medium - for example LFFS on a NOR Flash memory. ESTART can also be configured to use error detection and repair tools (such as scandisk) on file systems that were not shutdown in an orderly way - or to format disks when the phone is booted for the first time.

Once the read/write file systems are available, ESTART locates the persistent HAL settings on the internal drive and restores them to the HAL. This is also where the current language and locale settings are identified and restored from disk.

When we reach this point, all of the kernel, user library and file server services are now fully initialized and ready for the rest of the OS to begin its startup process. ESTART has done its job, and it now creates the next process in the chain, the system starter.

The system starter

The system starter provides a framework for the mobile phone to start and maintain all of the services necessary for normal operation. This framework is driven by a configuration file that the phone manufacturer constructs. The system starter also allows for multiple startup scenarios (some of which are described in Section 16.2.2) by supporting the selection of different configuration files.

Just as with the kernel and file server, the order in which the system services are initialized is important. This is to respect dependencies between services and to ensure that vital services are started first. In particular, on a mobile phone one would like the telephone functionality to be ready as soon as possible.

During normal system boot, we would expect that this component would first start the various low level communications, audio and graphics services. One vital server to initialize before the user interface can appear is the window server (discussed in Chapter 11, *The Window Server*), which provides shared access to the UI hardware on the phone, such as the display, keypad and touch screen.

Once the window server is running then the rest of the application and UI framework can initialize, and finally the telephone application is run.

Alternative startup scenarios

Booting from NAND Flash memory

In Section 16.1, I briefly described NAND Flash memory, and particularly noted its inability to be used for XIP software. In Chapter 9, *The File Server*, I discussed this type of Flash and its use for both user data storage and for storing built-in software. That alone is not enough to enable the system to boot from such memory, because the ROFS (read-only file system) relies on the kernel, file server and loader to copy the executable code into memory and prepare it for execution. So how do we get these fundamental services running to support loading the main OS from a ROFS image?

Figure 16.3 shows the modified startup stages for booting from NAND Flash. There are now two extra stages before the bootstrap is run:

1a. NAND Flash provides a very basic XIP service to allow a system to boot - the first 512-byte sector of the memory can be shadowed into some internal RAM and executed from there. This first sector must contain enough code to carry out the first step in loading the entire OS. The miniboot does this, by providing the essential CPU and memory setup, before loading the rest of the first Flash block (typically 16 KB) into RAM and continuing execution from there

1b. The program in this larger block is the *coreloader*. This understands the partitioning of the NAND Flash and the bad block algorithm, which I discussed in Chapter 9, *The File Server.* This allows it to locate and load the core image into RAM. The core image is an XIP image, which must at least contain all of the code required to initialize the kernel and file server and install the ROFS file system. The core image may be compressed, as this saves space in the Flash. Once the core image is loaded, the core loader executes the entry point in the XIP image (the bootstrap) and boot continues as for the XIP sequence for a while.

The XIP boot sequence is then modified once more - during file system initialization. The file system configuration for NAND Flash typically will combine both the XIP ROM file system and the ROFS file system under a single drive identifier, as I described in Chapter 9, *The File Server*. This allows built-in software that expects to be *in ROM* to be in the usual place - drive *Z:*.

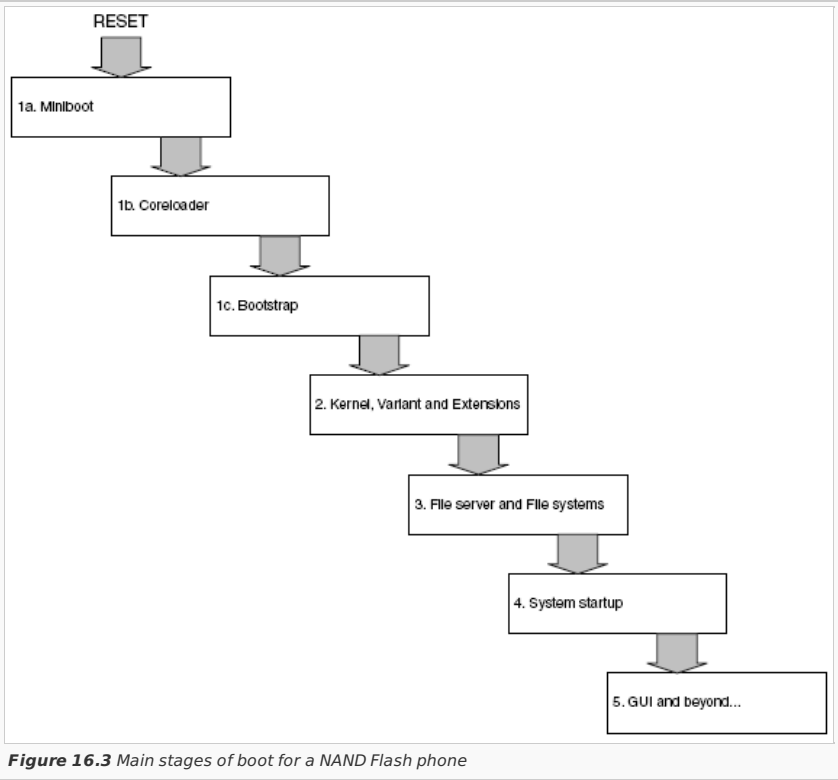

Figure 16.3 *Main stages of boot for a NAND Flash phone*

Given all this extra effort and delay during startup it is worth asking, what is the value of using NAND Flash for non-volatile storage in mobile phones? The answer is the same issue that prompted the question: performance. NAND Flash provides similar performance to NOR Flash for reading, but can write data three or four times faster. This will become very important for mobile phones that have multi-megapixel cameras or allow download of music tracks from a PC - NOR Flash is too slow for these use cases.

Other reasons for startup

The boot process usually starts because the user switched on the mobile phone, in which case the standard startup process is likely to be followed.

However, there are circumstances in which we follow alternative boot sequences, for example:

- A phone that is powered off displays its charging status when plugged into an electrical supply
- In the factory, when it is first switched on, the phone runs a series of diagnostic *self tests* to verify that the hardware and software are all functioning correctly
- While the phone's firmware (in other words, the Flash memory on which Symbian OS resides) is being updated, Symbian OS cannot safely be running. Depending on how the update software is implemented, this startup sequence may diverge inside the bootstrap before the kernel runs.

In the majority of Symbian phones, it is not the Symbian kernel that is initially aware of the reason for boot. The bootstrap provides a little information to the kernel through the superpage in the form of `iBootReason`, but this usually indicates only the difference between a cold boot (hardware power on) and a warm boot (software reset and reboot). Most often, the baseband software determines the reason for booting and provides this information to Symbian OS early on - for example in ESTART. The system starter uses this information to select the configuration file that corresponds to this startup scenario.

Booting in the emulator

The introduction of process emulation makes it possible for the emulator to *boot* in a way that is much closer to the real thing. The most significant differences are:

1. The object that replaces the bootstrap
2. Running different *types* of emulator.

16.2.3.1 Bootstrap

EPOC.EXE is the standard bootstrap program. It is a Win32 executable and its sole reason for existence is to call `BootEpoc()` in EUSER from its Win32 entry point. `BootEpoc()` takes a single Boolean parameter, ultimately used to determine if the emulator should automatically run a program after boot is completed.

BootEpoc() loads the kernel DLL dynamically and looks up the Symbian OS entry point, _E32Startup, by name. If it is successful, this entry point is then invoked, passing in BootEpoc()'s parameter. The kernel's _E32Startup() function first runs the kernel's static data constructors, then saves the parameter for later and calls the kernel's `BootEpoc()` function. This function does emulator specific initialization before calling the common kernel startup code in `KernelMain()`.

Variant and extensions

With no ROM, the kernel cannot immediately find the variant and extensions to load.

The variant is always called ECUST.DLL and `P::CreateVariant()` dynamically loads it - the DLL entrypoint is located as for other emulator DLLs, and called as normal for the variant. `A::CreateVariant()` then invokes the first ordinal in the variant, passing in the previously saved parameter that was originally passed to `BootEpoc()` by EPOC.EXE.

Variant initialization then proceeds as usual. The list of extensions to load is retrieved from the variant as the *extensions* property. These are dynamically loaded, in order, and their Symbian OS DLL entry points are called as normal for extensions. The last of these should be EXSTART, which creates a process based on the EFILE.EXE image - the file server.

E32START

File server initialization proceeds as normal, the last action being to create a new process from the E32STRT.EXE image. This is the emulator equivalent of the ESTART.EXE that I describe in **Chapter 16**. It carries out the same initialization tasks, but provides some additional features for the emulator.

E32STRT determines which UI to run based on its configuration properties: it can run the graphical window server, the text shell or run without any UI at all.

E32STRT then checks to see how the emulator was bootstrapped. If it was started by EPOC.EXE, it exits leaving the UI to run and the emulator boot has completed. If not, it retrieves the program and command line to auto-execute from the *AutoRun* and *CommandLine* properties, creates that process, waits for it to exit and after that terminates the emulator. This latter course of action gives the same behavior that you see in the EKA1 emulator when you run an EXE from the Windows command line.

Operating system shutdown

Now that I have covered startup, I will move on to describe its opposite: shutdown. From the user's point of view, startup and shutdown are symmetrical activities. From the software perspective, there is little in common:

- Startup is a baton-passing exercise, which takes the mobile phone from a primordial state to one where all services are running, in a carefully sequenced procession. This sequence is a consequence of the system design and our main interest lies in how it is achieved
- Conversely, shutdown is an activity which must be orchestrated, bringing the running system to a state in which it is safe to *pull the plug* and remove power. I will discuss some design details here to help explain the way in which shutdown is managed.
- Shutdown - which involves shutting down the phone, closing all open applications and saving data that has changed in the current session - is normally initiated by a user action, such as pressing a power button or a power-off key. Shutdown also occurs in response to other user or network based activities:
- Restoring data from a backup can invalidate much of the OS state: it is easier to restart the entire system than to synchronize all system services with the new configuration data
- Firmware over the air (FOTA) update. In-place update of the Symbian OS firmware requires that Symbian OS is not executing at the time. Once the update is downloaded to the phone, the phone will need to restart in *update* mode.

In all of these scenarios, the shutdown can be managed in an orderly way to ensure that persistent data is saved correctly.

Sometimes, however, the cause of shutdown is less controlled. For example, some software faults in critical system services are unrecoverable and result in an immediate software reset. Or, a loss of power results in a very rapid shutdown of the system, even though a little residual power is available to complete critical disk activities. In such cases the startup process must make its best effort to recover the state of the data on the mobile phone, and repair any problems that it discovers. Some of this work is done in ESTART as it mounts the file systems - if the system did not shutdown cleanly, it can check for errors in the file system and may even reformat the file system if it cannot be repaired.

System shutdown may result in critical hardware components such as the CPU and most peripherals having their power removed (transition to the off state) - or it may leave these components in a standby state, from which it is possible to return to an operational (active) state without rebooting.

There are currently two architectures available that can be used to manage shutdown of the entire OS. The shutdown server is used in phones today, whereas the domain manager is a new component that will eventually replace the shutdown server in this role. I will look at the design of both these architectures.

The shutdown server

The shutdown server is the current architecture that is used to manage controlled shutdown in Symbian OS.

Shutdown is typically initiated by a dedicated kernel-level component such as a device driver or kernel extension, or by the variant detecting a power button press, a power hotkey tap, the closing of the phone's cover, or some other mechanism.

A user-side component may also initiate a shutdown sequence, by calling the UserSvr class exported API `HalFunction(TInt aGroup, TInt aFunction, TAny* a1, TAny* a2)` with `EHalGroupPower` as the identifier for the HAL group of functions and `EPowerHalSwitchOff` as the identifier for the function to execute. This could be used, for example, when the detection of the power key press is not done by a software component executing on the main application processor: instead the user-side component that is used as a communication channel to the baseband software will call the HalFunction API. The servicing of the UserSvrcall is done by the platform-specific part of the kernel power framework, as I explained in Chapter 15, *Power Management*.

As a result of detecting the hardware event corresponding to the physical action, or servicing the UserSvrcall as I have just described, a `TRawEvent` having one of the following types is added to the event queue:

- `ESwitchOff` - added by the servicing of the UserSvr call
- `EKeyOff` - added by the driver that detects the key presses
- `ECaseClose` - added by the platform-specific component that monitors the state of the phone's lid.

The window server receives the event, and translates it into a `TEventCode` type:

- `EEventSwitchOff` in place of `ESwitchOff`
- `EEventKeySwitchOff` in place of `EKeyOff`
- `EEventCaseClosed` in place of `ECaseClose`.

The window server will then either send the event to a registered component that requested it or, if no component is registered to receive off events, request the kernel to power down by calling `UserHal::SwitchOff()`.

All current Symbian OS mobile phone have a UI component - the Look And Feel, or LAF, shutdown manager - that would previously have registered with the window server to receive such events. Implementing this policy in the UI allows the shutdown behavior of the phone to be customized according to the UI model it employs.

The shutdown manager object should derive from `CLafShutdownManagerBase` and implement a policy on when to shutdown the mobile phone.

```
class CLafShutdownManagerBase : public CBase
    {
protected:
    inline CLafShutdownManagerBase(MShutdownEventObserver& aObserver);
protected:
    MShutdownEventObserver& iObserver;
    };
```

The policy can involve listening for events such as the ones I have just listed, monitoring user inactivity using inactivity timers, and more.

A typical implementation has the LAF shutdown manager opening a session with the window server at creation time and queuing a request for the notification of *off* events. When it does receive such an *off* event, the LAF shutdown manager decides if it should result in a power down or just the sending of state and data save notifications to applications.

The shutdown server is at the center of the shutdown architecture. It derives from `MShutdownEventObserver`, as follows:

```
class MShutdownEventObserver
    {
public:
    virtual void HandleShutdownEventL(MSaveObserver::TSaveType aAction,
    TBool aPowerOff)=0;
    virtual CArrayFix<TThreadId>* ClientArrayLC()=0;
    virtual TBool IsClientHung(TThreadId aId) const=0;
    virtual void GetShutdownState(TBool& aPowerOff,
    TBool& aAllSessionsHavePendingRequest) const=0;
    };
```

The implementation of the mandatory API by a shutdown server enables the LAF shutdown manager to:

- Request the sending of save notifications to registered components, possibly followed by a shutdown
- Return a list of registered clients
- Enquire if a registered client is still processing the save notification
- Obtain the shutdown status after a shutdown request has been issued.

When the UI creates the shutdown server, the shutdown manager is also created and its `iObserver` member is set to reference the shutdown server.

Software components that need to be notified of an imminent shutdown create sessions with the shutdown server to receive save notifications. An example of such a component is a UI-specific save observer, which acts as a gateway for save/shutdown notifications on behalf of UI applications. Upon receiving a save notification, a save observer propagates the request to all running applications which will then save their data/status, close dialogs, exit, and so on.

The notification mechanism is based on an asynchronous request, which is placed on the shutdown server when a client creates a session with it. Clients of the shutdown server typically own a `CSaveNotifier` object that provides them with an interface to the shutdown server:

```
class CSaveNotifier : public CActive
    {
public:
    IMPORT_C static CSaveNotifier* NewL(MSaveObserver& aObserver);
    IMPORT_C ~CSaveNotifier();
    IMPORT_C void DelayRequeue();
    IMPORT_C void Queue();
    IMPORT_C void HandleError(TInt aError);
    ...
    };
```

The API allows for:

- Creating a session with the shutdown server that will queue an asynchronous request for shutdown notification
- Closing the session and canceling a pending request

- Delaying or stopping a shutdown sequence after the client received the notification, and resuming the sequence
- Notifying the shutdown server of an error in its internal save/shutdown sequence.

Clients of the shutdown server must also implement an `MSaveObserver` interface:

```
class MSaveObserver
    {
public:
    enum TSaveType
        {
        ESaveNone,
        ESaveData,
        ESaveAll,
        ESaveQuick,
        EReleaseRAM,
        EReleaseDisk,
        };
public:
    virtual void SaveL(TSaveType aSaveType)=0;
    };
```

Thus, when notified of an *off* event, the LAF shutdown manager calls the shutdown server's `HandleShutdownEventL()` API, specifying if a power down is required and specifying what save action is required from its clients (as a `TSaveType`). The LAF shutdown manager may also do this as a result of detecting a period of user inactivity.

The shutdown server manages the shutdown sequence. The servicing of `HandleShutdownEventL()` saves the locale and HAL settings that may have changed during the current session and, if a saving action of any type is required, notifies all registered clients that have pending save notification requests, by completing those requests. If a registered client does not yet have a pending request, then it will be notified immediately after it issues the request for notification.

After receiving a save notification, clients of the shutdown server call the `SaveL()` method (from `MSaveObserver`) which will perform client-specific status saving actions corresponding to the `TSaveType` argument passed.

Clients must then re-queue a request with the shutdown server. If a power down is required, the shutdown server will ask the kernel to shutdown only after all its clients have re-queued requests with it.

The shutdown server requests kernel shutdown by invoking the `User-Hal::SwitchOff()` API (an export from EUSER.DLL):

```
EXPORT_C TInt UserHal::SwitchOff()
    {
    TInt r = Power::EnableWakeupEvents(EPwStandby);
    if(r!=KErrNone)
        return r;
    TRequestStatus s;
    Power::RequestWakeupEventNotification(s);
    Power::PowerDown();
    User::WaitForRequest(s);
    return s.Int();
    }
```

This entire sequence is illustrated in Figure 16.4.

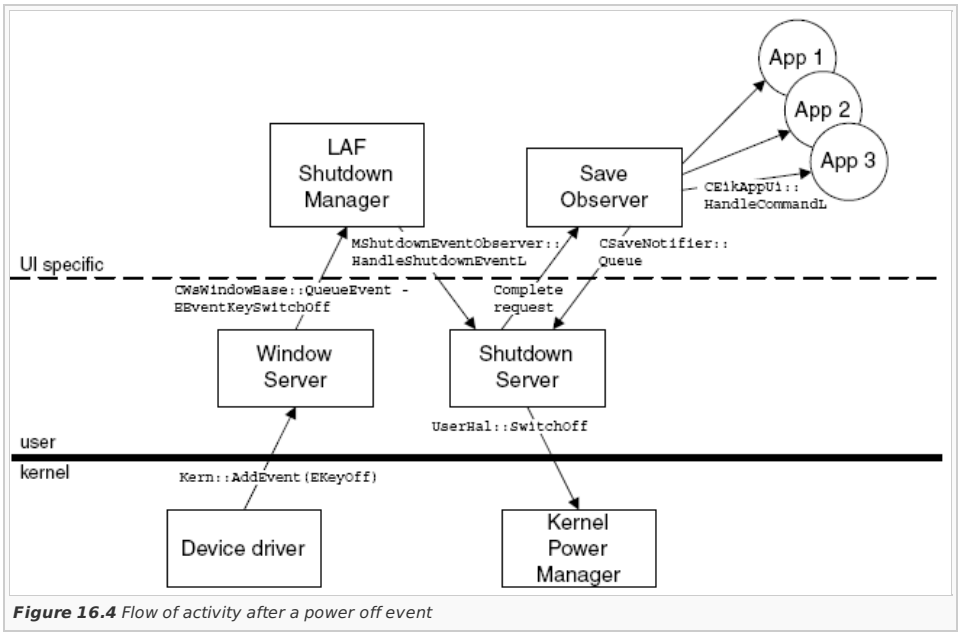

Figure 16.4 *Flow of activity after a power off event*

I have already discussed the Power class APIs invoked by this function in Chapter 15, *Power Management.* Be sure to note that:

- The target power state selected by `UserHal::SwitchOff()` is standby, as was the case with the EKA1 power management. However, as the implementation of `Power::PowerDown()` will call down to platform-specific code, phone manufacturers can interpret this request by powering down the hardware (CPU, peripherals) instead of transitioning them to a standby state
- The new shutdown architecture using the domain manager (discussed in the following section) leaves the choice of target power state to the LAF domain controller
- Wakeup events are enabled but, if they occur they have no impact on the transition once initiated.

16.3.2 The domain manager

As I mentioned earlier, in the future an architecture based upon domain management will be used as a replacement to the current scheme based on the shutdown server.

Overview

This alternative scheme is based on the concept of power domains, which may be populated with applications, and which are organized as dependency trees that represent the domain hierarchy (see Figure 16.5). Each node on the dependency tree represents a domain and is identified by a domain ID - an identifier number. Applications at the top of the tree, residing in parent domains, will usually provide services to those in child domains, or will manage the resources that they need to use. In this way the applications on one node of the domain tree have a dependency on the node above, and so on. This dependency is taken into consideration when a domain is activated, put into standby or shutdown.

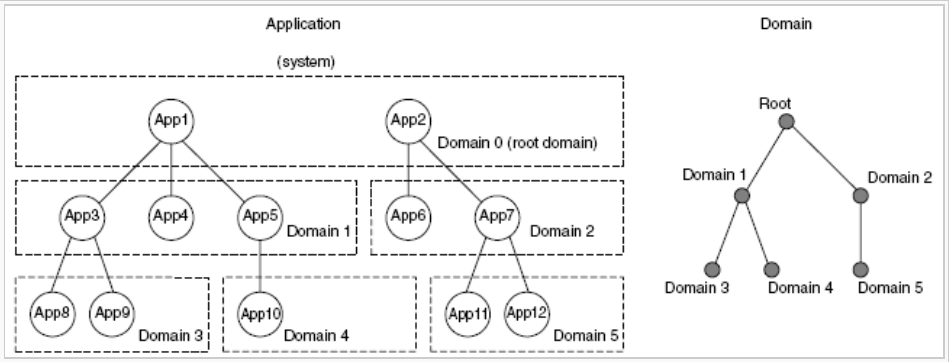

Figure 16.5 *Application dependencies and the domain hierarchy*

Domains can be in one of three possible power states: active, standby or off. Domains can be independently requested to transition between these different power states.

If a transition is applied to the root domain (the top of the tree) then the transition is a system transition. A system transition is one that is also applied to the kernel via the Power API, once the root domain has completed its transition. The system can transition from the active to the standby or off states - this is, of course, shutdown. The transition to active state is only allowed from the standby state and cannot be initiated by a call to the domain manager. This transition starts when the kernel wakes up from its standby state and reports the wakeup event to the domain manager, which then recovers from standby state by transitioning the domain tree back to active.

The transition of a domain to standby or off always starts with the transition of all the domain's child domains, followed by the transition of the target domain itself. A domain transition to the active state always starts with the transition of that domain, followed by the transition of its entire list of child domains.

Domain trees are static and are specified at system build time in the domain policy DLL. At system startup time, when the domain tree is loaded from this DLL, domains in the tree have no member applications. Applications will join the relevant domain as they startup.

We permit a maximum of 256 domains.

Applications are allowed to join domains, or disconnect from them, at any time. Once an application has joined a domain, it will remain a member of that domain until it explicitly relinquishes membership.

Membership of a domain gives an application the ability to request notification of the domain's power state changes. The application can decide on what action it should take on receiving this notification. Usually, on receiving a notification for a transition to a low power state, an application will save data related to its current state.

Domain state change notification is implemented using the publish and subscribe mechanism that I discussed in Chapter 4, *Inter-thread Communication*. State changes are published as property value changes. A state property identifier - a UID - has been reserved for this purpose, as have 256 sub-keys for the maximum possible number of domains. Applications that want to receive domain state notification simply subscribe to the state property.

Applications can join more than one domain and be notified of power state changes affecting all the different domains that they belong to. Applications may act upon the notification differently for each of the different domains they belong to, or for each type of state change they are notified about. For example, an application may save its data and state to persistent storage on domain shutdown, or save it to RAM on domain standby.

Design and APIs

The domain managed architecture is shown in Figure 16.6. The domain manager is a user-side system server, which manages application membership to domains as well as system-wide and domain-specific power state transitions. It owns the domain tree.

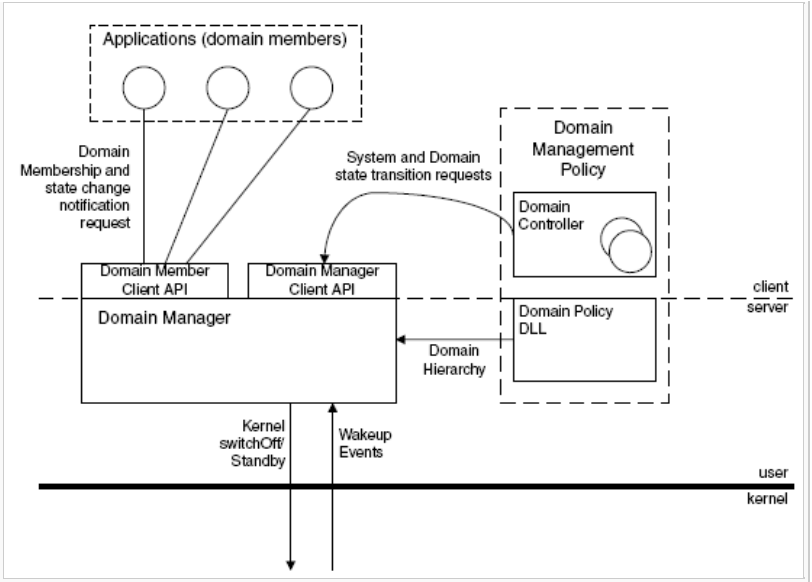

Figure 16.6 *Domain management architecture*

The domain manager gets the description of the domain tree from the domain policy DLL at system boot time when the domain manager is created; the domain policy DLL contains a *recipe* - or specification - to allow it to build the domain tree. Once loaded, the domain tree can never be modified.

The domain management policy is also responsible for triggering system-wide and domain-specific power state transitions. A component in the UI will provide a domain controller to implement these aspects of the policy, in a similar way to that in which the LAF shutdown server provides UI specific policy in the shutdown server architecture.

A domain controller has to connect to the domain manager, using the domain manager client API. The domain controller is required to have the PowerMgmt capability to request transitions, protecting the phone from malicious software. Once a connection is created, the controller can request a system or domain transition. After the transition has completed - which is signaled asynchronously - the controller should disconnect. Only one connection is allowed at any given time.

The domain manager client API is provided by the `RDmDomainManager` class:

```
class RDmDomainManager
    {
public:
    IMPORT_C static TInt WaitForInitialization();
public:
    IMPORT_C TInt Connect();
    IMPORT_C void Close();
    IMPORT_C void RequestSystemTransition(TPowerState, TRequestStatus& aStatus);
    IMPORT_C void RequestDomainTransition(TDmDomainId, TPowerState, TRequestStatus& aStatus);
    IMPORT_C void CancelTransition();
    IMPORT_C void SystemShutdown();
    };
```

This class offers methods to allow a domain controller to:

- Synchronize with the domain manager initialization before creating a session with it
- Create a controlling session with the domain manager
- Request system-wide transitions - to *standby* or *off*
- Request domain-specific transitions
- Cancel a system-wide or domain specific transition
- Disconnect from the domain manager.

Applications use the domain member client to interface to the domain manager. This client is a user-side DLL that encapsulates the domain membership and the state transition protocols. No platform security capabilities are required to use this interface to the domain manager.

The domain member client API is provided by the `RDmDomain` class:

```
class RDmDomain
    {
public:
    IMPORT_C TInt Connect(TDmDomainId aId);
    IMPORT_C void RequestTransitionNotification(TRequestStatus& aStatus);
    IMPORT_C void CancelTransitionNotification();
    IMPORT_C TPowerState GetPowerState();
    IMPORT_C void AcknowledgeLastState();
    IMPORT_C void Close();
    };
```

It offers methods to allow applications to:

- Request membership of a domain
- Request notification of domain state changes
- Cancel a request for notification of domain state changes
- Enquire the current state of a domain
- Acknowledge a state change notification
- Disconnect from a domain.

Shutdown sequence

When the domain controller requests a system-wide transition by calling the domain manager client exported API `RequestSystemTransition()` specifying standby or off as the power state, or by calling `SystemShutdown()` (which is a convenient wrapper for a PowerOff transition request), the domain manager will perform the following sequence of actions:

1. Enable wakeup events for the target state (by calling `Power::EnableWakeupEvents(...)` and passing the target power state)
2. Request notification of the occurrence of any wakeup events by calling `Power::RequestWakeupEventNotification()`
3. Notify all applications in the domain tree of the imminent transition, and wait for them to complete their transition. Notifications are issued taking into consideration the dependency between these components: applications that depend on resources offered by others are notified before those they depend on, and are shutdown first. In this way, resources are not released prematurely and are always available for applications that need them during the shutdown sequence. This orderly issuing of notifications also guarantees the speediest shutdown
4. If no wakeup event has been notified, the domain manager creates a session with the file server and calls the `RFs` class's API `FinaliseDrives()`. This iterates over every local or internal drive (that has a file system already mounted and has a media mounted that is not write protected) calling a file system-specific mount control block method (FinaliseMountL()). This may simply mark the drive as having been successfully finalized and not needing to be scanned the next time the phone is rebooted. This signals the completion of the user-side preparations to shutdown: at this point all user-side components that needed to save data or state must have already done so. The method can therefore mark the mount as read-only to prevent any spurious writes
5. At this point, if no wakeup event has been notified, the domain manager asks the kernel framework to power down the CPU and peripherals: `Power::PowerDown()`.

You might have noticed that steps 4 and 5 are just special cases of the domain transitions - and that one could view the file server and kernel as being nodes above the root domain in the hierarchy. Although it is possible to implement the file server shutdown as a domain transition, it is simpler at present to implement this transition as a special case. This is true in particular due to the need to support both shutdown architectures at present.

It may happen that halfway through the system shutdown, the domain controller decides to interrupt the transition (for example, if part of the system cannot be transitioned at this time). It may do so by calling the domain manager client exported API `CancelTransition()`, which results in the following sequence of actions being performed:

1. Wakeup events for the current transition are disabled and the power framework's target state is reset to active by calling `Power::DisableWakeupEvents()`
2. Wakeup event notification is canceled by calling `Power::CancelWakeupEventNotification()`
3. The shutdown notification is canceled for registered applications.

Migration from the current architecture

The domain manager and its domain member client interface can replace the shutdown server and the save notification mechanism by implementing save observers as nodes in the domain tree (that is, as power domains) and using the LAF shutdown manager in the role of domain controller.

Domain management offers some advantages over the current shutdown server architecture:

1. Shutdown is *orderly*: it takes into consideration the dependencies between components
2. Shutdown of the file server is controlled
3. Wakeup events are monitored during shutdown (see Section 16.4).

Operating system sleep and wakeup events

Within the current shutdown framework, some events may not lead to fully shutting down the phone via the `UserHal::SwitchOff()` API. Instead, moving it to a silent running mode, characterized by switching off the user interface, may be a more appropriate action, leaving the phone able to respond to incoming calls and quickly return to a running state. A clamshell phone may switch to silent running mode while the lid is closed and awake again when it is opened.

Such an operational mode may also be used at the beginning of a full shutdown operation, to provide feedback to the user that the mobile phone is switching off.

In this case, the UI observer of the `EEventCaseClosed` can simply disable appropriate hardware through the HAL or other device drivers:

```
 HAL::Set(HAL::EDisplayState, 0);
HAL::Set(HAL::EKeyboardState, 0);
HAL::Set(HAL::EPenState, 0);
```

Recovering from such a state usually happens because of a wake up event, generated by events such as the user pressing the on key, the phone being opened or an alarm going off.

I have already introduced the concept of wakeup events in Chapter 15, *Power Management*. In that chapter, I defined wakeup events as hardware events that, occurring while the platform is in low power state, may be capable of initiating a transition to an operational state. The kernel can also track them while the OS is preparing the platform transition to a low power state. If they are reported during that stage, they will lead to the canceling or even the reversing of the transition. Wakeup events can therefore play an important role during system shutdown.

Suppose the system has transitioned to a low power state such as the standby state, which does not require a full system reboot and leaves some systems operational (namely those involved in the detection of wakeup events). If a wakeup event then occurs, it is initially handled by the kernel power framework, which will restore the hardware platform to the state prior to the transition.

The current shutdown framework, based upon the shutdown server and LAF shutdown manager, relies on the kernel power framework sending an `ESwitchOn` event (or an `ECaseOpen` event if waking up was triggered by opening the phone lid) when waking up from the low power state.

This may be done by the BSP part of the kernel framework.

The window server captures the events, translates them into `EEventSwitchOn` or `EEventCaseOpened` and sends them to a component that registered for *on* events (typically the LAF shutdown manager). The LAF shutdown manager will then perform whatever actions are required to restore the UI and applications to a state corresponding to the phone operational mode. Typically:

```
HAL::Set(HAL::EDisplayState, 1);
HAL::Set(HAL::EKeyboardState, 1);
```

```
HAL::Set(HAL::EPenState, 1);
```

In a shutdown architecture based upon the domain manager, the domain controller requests a system transition to standby by calling the domain manager client asynchronous API `RequestSystemTransition()` and passing a `TRequestStatus`. The domain manager asks the kernel framework for notification of wakeup events before the transition to standby.

After waking up the platform, the kernel power framework completes the domain manager notification, which will then complete the domain controller's request, thus notifying it of a system wakeup.

On a domain-manager-controlled shutdown, the domain manager also monitors wakeup events during the OS shutdown sequence. At any point in that sequence before the kernel is requested to transition, a wakeup event may occur. For example, the user of the phone may change her mind and press the power button again. Or an alarm from a user-side software component may go off.

The kernel power framework completes the domain manager wakeup notification request, thus notifying it of the occurrence of the wakeup event. Upon receiving the notification, the domain manager performs the following actions:

1. Disables wakeup events and resets the kernel power framework target state to active by calling `Power::DisableWakeupEvents()`
2. Cancels the shutdown notification for applications that have not yet shutdown
3. Sends notifications to applications to transition back to the active state.

Summary

In this chapter I have demonstrated what happens during the first and the last few seconds of execution of Symbian OS - how the operating system bootstraps itself into existence and how it safely shuts itself down again.

In the next chapter, I will consider performance, and discuss how you can get the most out of EKA2.