

# Symbian OS Internals/01. Introducing EKA2

– [Symbian OS Internals Table of Contents](#)

by Jane Sales with Martin Tasker

*The ability to quote is a serviceable substitute for wit.*

**W. Somerset Maugham**

---

## The history of EKA2

Kernel design is one of the most exciting opportunities in software engineering. EKA2 is the second iteration of Symbian's 32-bit kernel architecture, and this in turn follows 8- and 16-bit kernels designed in the 1980s for Psion's personal organizers and PDAs.

Psion's Organiser, launched in 1984, was based on an 8-bit processor and supported only built-in applications. For such a device, the only kernel needed was a bootstrap loader and a small collection of system services. There was no clear requirement to differentiate the 8-bit kernel from middleware or application software.

In 1986, Psion launched the Organiser II, an 8-bit machine offering expansion based on the interpreted OPL language. The demands on the OS were slightly greater - sufficiently good memory management, for example, to support an interpreted language.

A major evolution came when, beginning in 1990, Psion launched a range of machines including a laptop, a clamshell organizer and an industrial organizer, all based on a single OS. The 16-bit EPOC kernel was tied to the Intel 8086 architecture and supported expansion, with applications written not only in OPL, but also in the native C APIs of the EPOC OS - thus opening up the OS itself to any number of aftermarket application writers.

This openness placed massive new demands on the kernel. For one thing, it had to be documented and made accessible to aftermarket programmers. Perhaps some applications would be poorly written: the kernel had to provide memory protection so a bug in one program would not crash another - or even crash the whole OS. Applications demanded sophisticated memory management for their own working memory. A potentially limitless number of event-driven services had to execute efficiently on a highly resource-constrained machine. And all this had to be delivered on the platform of the 8086's segmented memory model, with challenges that PC programmers of the day will readily recall.

The 16-bit EPOC kernel thus had to address many of the requirements which are met by EKA2 today, because of its positioning between the embedded real-time operating systems and classic desktop operating systems such as Windows. Although it was similar to embedded RTOSes (it ran from ROM), it was bigger because it supported richer functionality and was open to aftermarket applications. Although it was similar to desktop OSes (it was open and used the 8086 architecture), it was smaller because the memory and power resources available were considerably less.

Two further evolutionary steps were necessary to arrive at EKA2. EPOC32, released in Psion's Series 5 PDA in 1997, began life in 1994. Its kernel, retrospectively dubbed EKA1, carried over the best features of the 16-bit EPOC kernel and fixed several significant issues. Firstly, EKA1 was thoroughly 32-bit - with no relics of the awkwardness in EPOC resulting from the 8086-segmented memory architecture. Secondly, the EKA1 kernel was designed from the beginning with hardware variety and evolution in mind - unlike 16-bit EPOC, which had been tied closely to a single 80186-based chipset. Many implementation details were changed as a result of these fundamentals, but EKA1 was otherwise surprisingly similar in spirit to 16-bit EPOC.

At that time, one of the proudest moments of my career took place - in my spare bedroom! The rest of the team were out of the office, so I worked at home for a week, frantically trying to achieve the first ever boot of the kernel before they got back. And late on the Friday afternoon, the null thread finally printed out its debug message - EKA1 was born.

But EKA1 was not destined to be the end of the story. The Symbian OS system for supporting event-driven programming was efficient overall, but provided no real-time guarantees. The kernel itself was designed with robustness - key for PDAs that hold a user's personal data - as the primary goal. As Symbian OS began to address the processing needs of mobile phones, it became apparent that an OS that could provide real-time guarantees was really needed.

There were other influences on EKA2 too. The experience gained from real hardware porting in the context of EKA1 was beginning to demonstrate that EKA1's module boundaries were not always drawn in the right place to make porting easy. Some ports, which should have required only a driver change, in practice required the kernel to be re-built.

So, a new kernel architecture was conceived and to distinguish it from the original 32-bit EPOC kernel, it was named EKA2 (EPOC Kernel Architecture 2), with the term EKA1 being invented for the original.

EKA2 was conceived in 1998 and, little by little, brought from drawing board to market. By 2003, Symbian's lead licensees and semiconductor partners were committed to adopting EKA2 for future products.

This book was written to provide a detailed exposition on the new real-time kernel, providing the reader with the insights of the software engineers who designed and wrote it.

This chapter is designed as the foundations for that book and should give you a good understanding of the overall architecture of the new real-time kernel, and of the reasoning behind our design choices. I will also say a little about the design of the emulator, and then return to this subject in more detail a couple of times later in the book.

---

## Basic OS concepts

I'd like to start with a basic definition of an operating system (OS):

[http://developer.nokia.com/Community/Wiki/Symbian\\_OS\\_Internals/01.\\_Introducing\\_EKA2](http://developer.nokia.com/Community/Wiki/Symbian_OS_Internals/01._Introducing_EKA2)

(C) Copyright Nokia 2013. All rights reserved.

The operating system is the fundamental software that controls the overall operation of the computer it runs on. It is responsible for the management of hardware - controlling and integrating the various hardware components in the system. The OS is also responsible for the management of software - for example, the loading of applications such as email clients and spreadsheets.

The operating system is usually the first software that is loaded into a computer's memory when that computer boots. The OS then continues the start-up process by loading device drivers and applications. These, along with all the other software on the computer, depend on the operating system to provide them with services such as disk access, memory management, task scheduling, and interfacing with the user.

Symbian OS has a design that is more modular than many other operating systems. So, for example, disk services are in the main performed by the file server, and screen and user input services by the window server. However, there is one element that you can think of as the heart of the operating system - the element that is responsible for memory management, task management and task scheduling. That element is of course the kernel, EKA2.

There are many different flavors of operating system in the world, so let's apply some adjectives to Symbian OS, and EKA2 in particular:

Symbian OS and EKA2 are **modular**. As I've already said, operating system functionality is provided in separate building blocks, not one monolithic unit. Furthermore, EKA2 is modular too, as you can see in Figure 1.1.

EKA2 is **single user**. There is no concept of multiple logins to a Symbian OS smartphone, unlike Windows, Mac OS X, UNIX or traditional mainframe operating systems.

EKA2 is **multi-tasking**. It switches CPU time between multiple threads, giving the user of the mobile phone the impression that multiple applications are running at the same time.

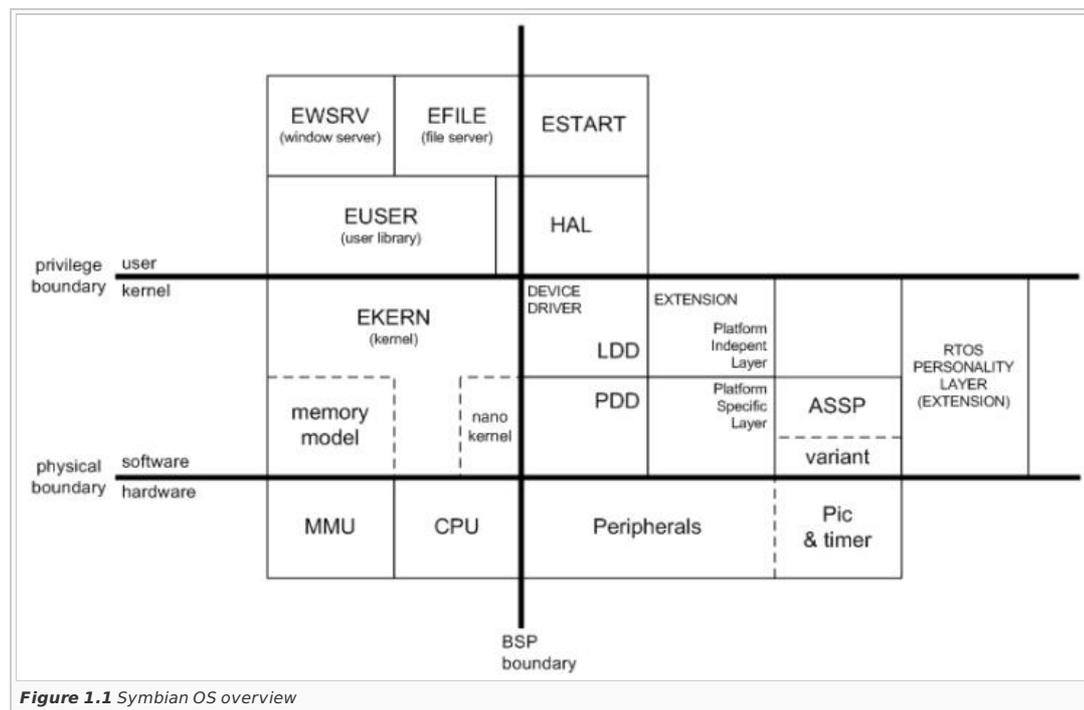


Figure 1.1 Symbian OS overview

EKA2 is a **preemptively** multi-tasking OS. EKA2 does not rely on one thread to relinquish CPU time to another, but reschedules threads perform, from a timer tick.

EKA2 is a **priority-based** multi-tasking OS with "priority inheritance".

EKA2 allocates CPU time based on a thread's priority and minimizes the delays to a high-priority thread when a low-priority thread holds a mutex it needs.

EKA2 is **real-time**. Its services are (mostly) bounded, that is it completes them in a known amount of time.

EKA2 can be a **ROM-based** OS.

EKA2 is suitable for **open but resource-constrained** environments. We designed it for mobile phones, and so it needs less of key resources such as memory, power and hard disk than open desktop operating systems such as Windows or Linux.

## Symbian OS design

### Design goals

When creating EKA2 we set ourselves a number of design constraints. We started by deciding what we didn't want to lose from EKA1. This meant that we wanted to ensure that the new kernel was still:

1. In the embedded OS tradition
2. Suitable for resource-constrained environments
3. Modular: consisting of microkernel and user-side servers
4. Portable to a range of evolving chipsets
5. Robust against badly written user code

6. Of high integrity, ensuring the safety of user data.

Then we decided on our new goals. The key goal was that the new kernel would be real-time and have enhanced overall performance. We decided that we would meet this if we could run a GSM protocol stack on our new operating system. A side benefit, and a worthy one, would be the ability to better support high-bandwidth activities such as comms and multimedia. This goal broke down into several sub-goals and requirements:

1. Latency = 1 ms from interrupt to user thread
2. Latency = 500  $\mu$ s from interrupt to kernel thread
3. Fast mutex operations
4. OS calls to be of determined length where possible
5. OS calls to be preemptible
6. Priority-order waiting on semaphores and mutexes
7. Timers with a finer resolution.

Then we considered how else we could improve the operating system, and we came up with the following list:

1. Ease porting - although EKA1 had been designed to be portable, we could go much further to make life easier for those porting the OS to new hardware
2. Be robust against malicious (rather than merely badly written) user code
3. Enable single-core solutions, in which embedded and user-application code run on the same processor core
4. Provide a better emulator for code development and debugging, that emulator being a closer match to real hardware
5. Simplify life for device driver writers.

And as we considered these design goals, we were aware that there was one over-riding constraint on our design. That constraint was to be backwards source compatibility with the EKA1's EUSER class library.

EUSER is the interface to the kernel for all Symbian OS applications, and there are a lot of them out there!

## Symbian OS kernel architecture

With those design goals in mind, we designed an operating system whose architecture, at the highest level, looked like that in Figure 1.1. You can see the major building blocks of the kernel. I've also included two other key system components that are usually considered to be part of the operating system, and that I will cover in this book: the file server and the window server. I'll cover each of these building blocks and give you an idea of its basic functionality.

### Nanokernel

The main function of the nanokernel is to provide simple, supervisor-mode threads, along with their scheduling and synchronization operations. We named the nanokernel as we did because the services it provides are even more primitive than those provided by most embedded real-time operating systems (RTOSes). However, we have carefully chosen those services to be sufficient to support a GSM signaling stack.

The nanokernel is the initial handler for all interrupts. It then passes the majority of them to the variant layer for dispatch. It also provides simple timing functions, such as the nanokernel timer (NTimer) API, which gives a callback after a specified number of ticks, and the sleep API (NKern::Sleep), which makes the current thread wait for a specified number of ticks.

The simple synchronization objects I mentioned earlier are the nanokernel mutex (NFastMutex) and the nanokernel semaphore (NFastSemaphore). Both of these forbid more than one thread from waiting on them.

Finally, the nanokernel provides deferred function calls (DFCs) and the oddly named immediate deferred function calls (IDFCs). If you want to find out more about these, then please go to [Chapter 6](#).

An important difference in EKA2 from EKA1 that should be noted is that neither the nanokernel nor the Symbian OS kernel link to the user library, EUSER. Instead, the nanokernel uses its own library of utility functions, and makes these available to the rest of the kernel, and device drivers too.

Another key difference from EKA1, somewhat related to the one I have just discussed, is that EKA2 does not support a kernel-side leaving mechanism. This means that errors are reported by returning an error code - or panicking the thread.

The majority of the time, the nanokernel is preemptible. Usually it runs unlocked and with interrupts enabled, but we do have to prevent other threads from running in a few sections of code, such as thread state changes and access to the ready list. We designed these critical sections to be as short as possible and to have bounded execution times, the goal being to maintain deterministic real-time performance. We protect the critical sections in the nanokernel by disabling preemption - this is possible because these sections are very short. In general, we use a mutex known as the system lock to protect critical code in the Symbian OS kernel and memory model, but the only place where the nanokernel uses this lock is to protect the scheduler's address space switch hook on the moving memory model.

What are the limitations on the nanokernel? The main one to note is that it does not do any dynamic memory allocation; that is, it can't allocate or free memory. In all of the nanokernel's operations, it assumes that memory has been preallocated by other parts of the operating system.

### Symbian OS kernel

The Symbian OS kernel provides the kernel functionality needed by Symbian OS, building on the simple threads and services provided by the nanokernel to provide more complex objects, such as user-mode threads, processes, reference-counted objects and handles, dynamically loaded libraries, inter-thread communication and more.

These objects also include a range of more sophisticated synchronization objects: Symbian OS semaphores and mutexes. Symbian OS semaphores are standard counting semaphores which support multiple waiting threads and which release waiting threads in priority order. Symbian OS mutexes are fully nestable (a thread can hold several mutexes at once, and can hold the same mutex multiple times). They also support priority inheritance: the holding thread inherits the priority of the highest priority waiting thread, if that is higher than its usual priority.

In contrast to the nanokernel, the Symbian OS kernel does allow dynamic memory allocation. It provides a kernel memory allocator - the kernel heap, which uses low-level memory services provided by an entity known as the memory model. The Symbian OS is completely MMU agnostic - we isolate all assumptions about memory to the memory model, which I describe in more detail in the next section.

The Symbian OS kernel is fully preemptible: an interrupt can cause it to reschedule at any point in its execution, even in the middle of a context switch.

This means that the Symbian OS kernel can have no effect whatsoever on thread latency.

We use system lock mutex, provided by the nanokernel, to protect the most fundamental parts of the Symbian OS kernel, such as:

1. The state of DThread objects. When Symbian OS threads interact with semaphores and mutexes, they undergo state transitions that are protected by the system lock
2. The state of most Symbian OS synchronization objects: IPC (servers and sessions), semaphores, mutexes, message queues, publish and subscribe properties
3. Handle arrays are valid for reading (but not writing) when the system lock is held. All the executive functions that take a handle hold the system lock while translating it - see [Chapter 5](#) for more on this subject.

## Memory model

In EKA2, we confine our assumptions about the memory architecture of the ASIC to one module, the memory model. Thus the memory model encapsulates significant MMU differences, such as whether a cache is virtually tagged or physically tagged, and indeed, whether there is an MMU at all. In EKA1, assumptions about memory and the MMU were spread throughout the operating system, making it difficult to produce a mobile phone based on an ASIC without an MMU, for example. This has become much easier with the advent of EKA2, since the memory model allows you to model memory in different ways, and to change that decision relatively easily.

Symbian currently provides four different memory models:

1. Direct (no MMU)
2. Moving (similar to EKA1)
3. Multiple (used for ASICs with physically tagged caches such as Intel X86 and later ARM cores)
4. Emulator (used by the Symbian OS emulator that runs on Windows).

The memory model provides low-level memory management services, such as a per-process address space and memory mapping. It performs the context switch when asked to do so by the scheduler and is involved in inter-process data transfer.

The memory model also helps in the creation of processes as an instantiation of an executable image loaded by the file server, and takes part in making inter-process data transfers.

If you are interested in finding out more about the memory model, turn to [Chapter 7](#).

## Personality layer

We designed the nanokernel to provide just enough functionality to run a GSM signaling stack. The idea behind this was to allow mobile phone manufacturers to run both their signaling stacks and their personal information management (PIM) software on a single processor, providing considerable cost savings over the usual two-processor solution.

Most mobile phone manufacturers have written their signaling stacks for existing RTOSes such as Nucleus or  $\mu$ ITRON. These signaling stacks represent a considerable investment in time and money, and it would be very time-consuming for the mobile phone manufacturers to port them to the nanokernel - not to mention the increase in defects that would probably ensue from such an exercise.

Because of this, we designed the nanokernel to allow third parties to write personality layers.

A personality layer is an emulation layer over the nanokernel that provides the RTOS API to client software. The personality layer would translate an RTOS call into a call (or calls) to the nanokernel to achieve the same ends. In this way, we allow source code written for that RTOS to run under Symbian OS with little or no modification.

For a more detailed description of personality layers, and the nanokernel design decisions that support them, turn to [Chapter 17](#).

## ASSP/variant extension

Typically, the CPU and the majority of hardware peripherals on mobile devices are implemented on a semiconductor device integrated circuit commonly referred to as an ASSP (Application-Specific Standard Product). To reduce both the bill of materials and the size of a phone, it is becoming common to add an increasing number of components to the ASSP. This might include stacking RAM and flash components on the same silicon package, or incorporating components into the silicon layout; for example, a DSP (digital signal processor) for audio/video processing, dedicated graphics processors and telephony baseband processors running GSM or CDMA communication stacks.

We refer to any hardware components outside the ASSP as variant-specific components. These typically include components such as flash and RAM storage technology, display devices, baseband and Bluetooth units. They are typically interfaced to the processor over semiconductor-vendor-specific buses and interconnect, or more standard communications lines such as USB and serial UARTs. ASSPs also tend to provide configurable GPIO (general purpose I/O) lines for custom functions such as MMC card detect and touch-screen pen down interrupt lines.

So, in Symbian OS, the ASSP/variant extension provides the hardware-dependent services required by the kernel - for example, timer tick interrupts and real-time clock access. In the days of EKA1, we built the ASSP into the kernel, and the separate variant layer described in the next section was mandatory. This made for unnecessary re-compilation of the kernel when porting to a new ASSP, so in EKA2 we have completely separated the ASSP from the kernel. Of course, this means that if you are porting EKA2, you no longer need to recompile the kernel every time you tweak your hardware.

## Variant

In EKA2, we don't insist that you make a division between the ASSP and the variant, as we do in EKA1. You may provide one single variant DLL if you wish. Nevertheless, if you were porting the OS to a family of similar ASICs, you would probably choose to split it, putting the generic code for the family of ASICs in the ASSP extension, and the code for a particular ASIC in the variant DLL. For example, within Symbian, the Intel SA1100 ASSP has two variants, Brutus and Assabet.

## Device drivers

On Symbian OS, you use device drivers to control peripherals: drivers provide the interface between those peripherals and the rest of Symbian OS. If you want, you may split your device driver in a similar way to the ASSP and variant, providing a hardware-independent logical device driver, or LDD, and a hardware-dependent physical device driver, or PDD.

Device drivers may run in the client thread or in a kernel thread: our new multi-threaded kernel design makes porting device drivers to Symbian OS from other operating systems much easier.

Symbian provides standard LDDs for a wide range of peripheral types (such as media devices, the USB controller and serial communications devices) -

nevertheless, phone manufacturers will often develop their own interfaces for custom hardware.

Device drivers have changed considerably from EKA1 to EKA2. See [Chapter 12](#), for more details.

## Extensions

Extensions are merely device drivers that the kernel automatically starts at boot-time, so you can think of them as a way to extend the kernel's functionality. For example, the crash debugger is a kernel extension, allowing you to include it or exclude it from a ROM as you wish, without having to recompile the kernel.

The variant and the ASSP that I discussed earlier are important extensions that the kernel loads quite early in the boot process. After this, the kernel continues to boot until it finally starts the scheduler and enters the supervisor thread, which initializes all remaining kernel extensions. Extensions loaded at this late stage are not critical to the operation of the kernel itself, but are typically used to perform early initialization of hardware components and to provide permanently available services for devices such as the LCD, DMA, I2C and peripheral bus controllers.

The final kernel extension to be initialized is the EXSTART extension, which is responsible for loading the file server. I discuss system boot in more detail in [Chapter 16](#).

## EUSER

The user library, EUSER, provides three main types of function to its clients:

1. Class library methods that execute entirely user-side, such as most methods in the array and descriptor classes (descriptors are the Symbian OS version of strings)
2. Access to kernel functions requiring the kernel to make privileged accesses on behalf of the user thread, such as checking the time or locale settings
3. Access to kernel functions requiring the kernel to manipulate its own memory on behalf of a user thread, such as process creation or loading a library.

Every Symbian OS thread gains its access to kernel services through the EUSER library. It is this interface that we have largely maintained between EKA1 and EKA2, resulting in minimal disruption to application authors.

## File server

The file server is a user-mode server that allows user-mode threads to manipulate drives, directories and files. Please turn to [Chapter 9](#) for more details.

## Window server

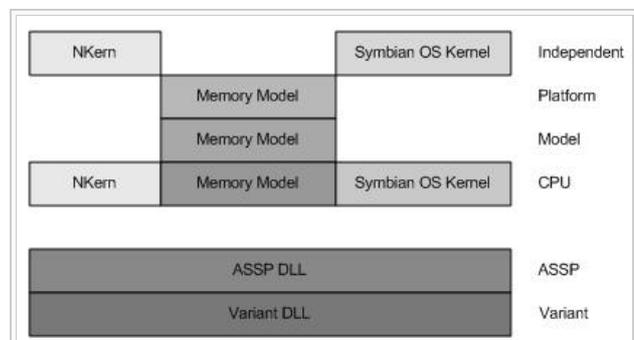
The window server is a user-mode server that shares the screen, keyboard and pointer between all Symbian OS applications. See [Chapter 11](#) for more details.

## Software layering

We can also consider the architecture of Symbian OS from a software layering perspective, as shown in Figure 1.2.

If you are familiar with EKA1, you will notice that the layering of EKA2 is a little different. Nonetheless, there are strong similarities, as we move down from the most generic, independent layer, in which code is shared between all platforms, to the most specific variant layer, in which code is written for a particular ASIC on a particular development board or in a particular mobile phone.

We call the top four software layers *the kernel layer*, and the bottom two, *the peripheral layer*. These last form a key part of the board support package that a phone manufacturer implements when porting Symbian OS to new hardware. This also comprises the bootstrap and device drivers and extensions.



**Figure 1.2** Kernel layering

The independent layer makes up about 60% of the kernel source code. It provides the basic building blocks of the nanokernel and the Symbian OS kernel - nanothreads, threads, processes, chunks, client-server and more. These base classes are derived in lower layers to provide implementations for the particular hardware on which Symbian OS is running.

The platform layer is concerned with executable images - whether Symbian OS is running on the emulator or real hardware - hence its alternative name of the image layer. Only the memory model has code at this level, and it provides two implementations, EPOC for device hardware and WIN32 for the emulator.

The model layer is all about the organization of per-process memory, and again only the memory model has code at this level. The memory model provides four different implementations - moving, multiple, direct and emulator. I will discuss these in more depth in [Chapter 7](#).

The CPU layer is for code that differs according to the processor that Symbian OS is running on; this is where assembler code belongs. The nanokernel, memory model and Symbian OS kernel all have code in this layer. At the time of writing, Symbian provides three possible CPU layers - X86 (a port to PC hardware), ARM (mobile phones) and Win32 (for the emulator).

The CPU layer of the memory model has code that is CPU- and MMU-specific, as well as specific to the type of memory model. The nanokernel's CPU layer contains most of the knowledge of the core CPU architecture - how exceptions and interrupts are handled, which registers need to be saved on a context switch and so on. A good proportion of the code in the CPU layer of the Symbian OS kernel is independent layer functionality that has been assembler-coded for improved performance.

The variant layer provides the hardware-specific implementation of the control functions expected by the nanokernel and the Symbian OS kernel. As I mentioned earlier, the phone manufacturer can choose whether to split this layer into an ASSP and a variant when porting to new hardware.

This variant layer can also provide hardware-specific implementations of hardware abstraction layer (HAL) functions, although these may equally be implemented in the kernel itself or in extensions.

In [Chapter 5](#) I will explain what services each layer exposes to the other layers.

## Design solutions

Now I'm going to talk about the design decisions that we took for EKA2, and how they helped us to achieve the goals that we had set ourselves.

### Multi-threaded preemptible kernel

To decrease thread latency, we chose to make EKA2 multi-threaded, allowing the preemption of low-priority kernel operations by high-priority ones.

EKA2 has five threads, and they are:

1. The null thread - idles the CPU, de-fragments RAM. This is also known as the idle thread
2. The supervisor thread - cleans up killed threads and processes, provides asynchronous object deletion
3. DFC thread 0 - runs DFCs for general device drivers, such as comms, keyboard and digitizer
4. DFC thread 1 - runs the nanokernel's timer queue
5. Timer thread - runs Symbian OS relative and absolute timers (`After()`, `At()`).

The purpose of these five threads is described in more detail in [Chapter 3](#).

The multi-threaded nature of the kernel also helped us to achieve another of our goals - making life easier for device driver writers. You often want to port a device driver from another operating system, but the single-threaded device driver model of EKA1 meant that porting a multi-threaded device driver was not a simple task - you usually had to redesign the driver from scratch. In EKA2, device drivers can make use of DFC thread 0, or can even create threads of their own if they wish. Device driver designs from other operating systems can be re-used and porting is now much simpler.

### Nanokernel

We chose to have a separate nanokernel, because it has several advantages:

1. Very low and predictable interrupt and thread latencies. This is because only the nanokernel disables either interrupts or rescheduling. (There are a handful of exceptions to this, but they are not important here.) The vast majority of the Symbian OS kernel, and the memory model, run with both interrupts and preemption enabled. Because the nanokernel provides only a small selection of primitives, it is easy to determine the longest period for which we disable interrupts or rescheduling
2. Simpler and better emulation. The Symbian OS emulator running under Windows has far more code in common with a real device, which means that the emulation is more faithful than that obtained with the EKA1 emulator
3. Support for single-core phones. The nanokernel allows you to run an RTOS and its GSM signaling stack alongside Symbian OS and its PIM software. For more detail see the section on Personality Layer.

### Modularity

The increased modularity of the new kernel makes porting the operating system to new ASSPs much easier. A large proportion of the processor-specific code is in the nanokernel, and differences in memory and MMU are confined to the memory model.

The memory model makes it easy for you to use the direct memory model in the early stages of a port to a new CPU architecture, changing to the moving or multiple models later on when you've done more debugging. It allows you to port the OS in smaller, simpler stages.

### Design limitations

Designing for real-time performance led to a couple of design limitations on EKA2:

1. To ensure deterministic interrupt latencies, we could not allow an unlimited number of interrupt service routines to bind to one interrupt source as was possible in EKA1. Now only one ISR may bind to an interrupt
2. To ensure bounded context switch times, we had to restrict the number of chunks in a process to a maximum of 16 - from an unlimited number in EKA1. (A chunk is the Symbian OS object that is fundamental to memory allocation - for more details see [Chapter 7](#).)

It's important to note that not all EKA2 services are bounded in time: for example, the allocation and freeing of memory are potentially unbounded. This is discussed in [Chapter 17](#).

## The Symbian OS emulator

### Design decisions

The emulator has two main uses - developing Symbian OS software and demonstrating that software.

The first of these use cases makes more demands on kernel services, so we concentrated on it when we drew up our requirements. At the highest level, it gave us just a couple of key requirements for the emulator:

1. It needs to support development and debugging using standard tools on the host platform
2. It should provide as faithful an emulation as possible of Symbian OS on target hardware.

These requirements seem to conflict, because the first requires the use of entities in the hosted platform (until now, always Windows) that do not exist in the same form in the real Symbian OS. For example:

1. Source-level debugging requires that the object code is stored in standard Windows executable files that the Windows debugger can recognize and that are loaded via the standard Windows loader
2. Debugging multi-threaded software requires that the Windows debugger recognize those threads. This means that we should implement emulated threads as Windows threads.

In the end, we decided to write the EKA2 emulator as a port of the EKA2 kernel, rather than trying to make the Symbian OS kernel API work over Win32 APIs. We used Windows as little as possible so as to share the maximum amount of Symbian OS code (and hence behavior) between the emulator and real mobile phones.

in common. Both systems contain the same core kernel code, from the Symbian OS kernel and the nanokernel. At the lower, architecture-specific, levels of the nanokernel, we have an emulated Win32 CPU rather than an ARM CPU or an X86 CPU. This means that the emulator is effectively a port to a different processor. For example, the emulator has processes and scheduling that are almost identical to those on a real device.

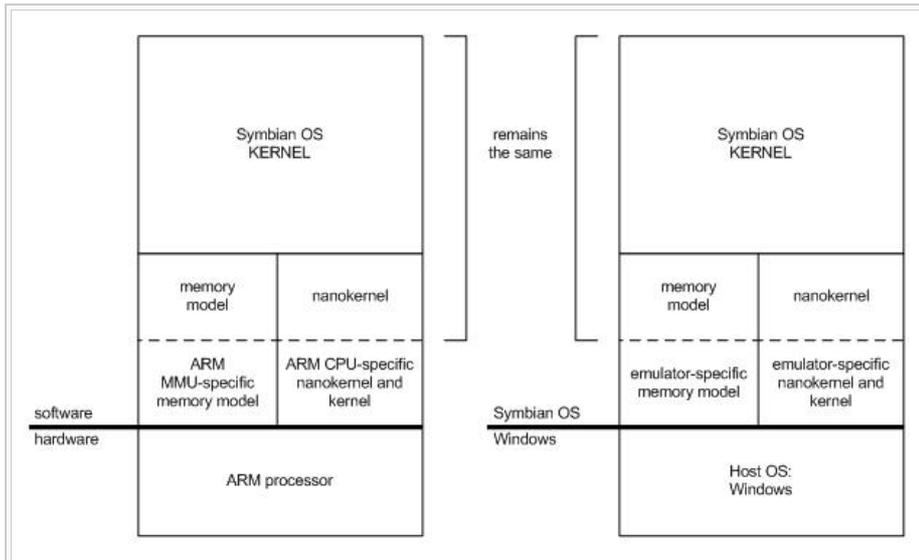


Figure 1.3 Emulator code re-use

The memory model, however, is completely different on the emulator and a real mobile phone. On the emulator, it is always the special emulator memory model, which has knowledge of the different image files that are loaded to create processes. These are standard Win32 PE EXE files, and so we satisfy our earlier requirement for source-level debugging. In theory, this approach could make it easier for us to implement an emulator on platforms other than Windows.



© 2010 Symbian Foundation Limited. This document is licensed under the [Creative Commons Attribution-Share Alike 2.0 license](http://creativecommons.org/licenses/by-sa/2.0/legalcode). See <http://creativecommons.org/licenses/by-sa/2.0/legalcode> for the full terms of the license.

Note that this content was originally hosted on the Symbian Foundation developer wiki.

