Symbian OS Internals/06. Interrupts and Exceptions

by Dennis May

Einstein argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer.
Fred Brooks

When talking about microprocessors, we use the term exception to refer to any event, other than the execution of explicit branch or jump instructions, which causes the normal sequential execution of instructions to be modified. On processor architectures with multiple privilege levels, these events typically cause a transition from a less privileged execution level to a more privileged one.

On the types of processor that run Symbian OS, there are many events that cause exceptions. These events form six categories, which I will describe in the next six sections of this chapter. Although the words used to describe the categories are the same throughout the computer industry, their exact meanings tend to differ - so, even if you are familiar with this subject, please do skim these next few sections. In Symbian, we categorize the events from the perspective of the software event handlers, so our terms will probably not match the categories given by people who define them from a processor architecture perspective.

Whenever an exception occurs, the processor stops what it was doing, and begins execution in a defined state from a (new) defined location. But before doing so, it may save some information, such as the program counter, to allow the original program flow to be resumed. Whether this occurs, and how much information is saved, depends on the nature of the exception. I will say more on this later.

Exception types

Interrupts

An interrupt is an exception that is not caused directly by program execution. In most cases, hardware external to the processor core signals an interrupt - for example, peripheral devices indicate that they have data available or require data to be supplied. What differentiates an interrupt from other exceptions is that it occurs asynchronously to program execution; in other words, it is not caused by the sequential execution of an instruction. Usually an interrupt is serviced as soon as the currently executing instruction is complete, although on some processors interrupts may occur in the middle of an instruction, and on others not every instruction can be followed by an interrupt.

Whenever an interrupt occurs the processor must save enough state to allow the interrupted program to be resumed and to continue execution precisely as if the interrupt had never occurred. This state always includes the program counter, which specifies the address of the instruction that would have been executed next, had the interrupt not occurred. The saved state also includes the processor status and the condition code register, which contains the interrupt mask level in effect at the time the interrupt occurred. The processor will then disable interrupts whose priority is lower than or equal to that of the one accepted.

Interrupts may be maskable or non-maskable. Maskable interrupts can be disabled by software; they are then held pending until software enables them again. There is no way to turn off non-maskable interrupts - they are always recognized by the processor.

Although the way in which the processor handles interrupts and other exceptions may be similar, the asynchronous nature of interrupts means that their software handlers are very different. In this book, I will always use the term exception to mean any exception **other than** an interrupt, and I will always refer to interrupts explicitly by name.

Resets

A reset causes the processor to reinitialize its whole state, and to start execution from a known location in memory. For example the ARM processor will disable all interrupts, enter supervisor mode and fetch the first instruction to be executed from physical address 0.

Most processors do not save any information about the program that was executing when the reset happened - all of the processor's state is wiped clean. However, on some processors, this is not the case - the processor only initializes enough state to allow it to deterministically execute a boot sequence.

Resets may be caused by hardware, for example when power is first applied to the system, or by software, for example when an unrecoverable error is detected.

Aborts

An abort is an exception that occurs because of an unrecoverable error. The processor may save some information relating to previous program execution, but this can only be used for diagnostic purposes and does not allow the kernel to recover the

situation and continue execution of the program.

Aborts occur if the programmer accesses non-existent or unmapped memory. They also occur when there is inconsistent processor state due to a system programming error. An example of this happens on an IA32 processor if the programmer loads the ring 0 stack pointer with an invalid address. Any attempt to use the stack would then cause a stack fault exception. However, the processing of this exception requires that information be pushed onto the ring 0 stack, which is invalid. This is a system programming error.

Depending on the nature of the error, an abort may be handled by the processor as a reset or it may be handled in software. Aborts due to incorrect physical addresses will be handled in software, since it is possible that the fault could be localised to a single program. However aborts due to inconsistent processor state, such as the example given, cannot be recovered by software and so are handled as a reset. In fact IA32 processors will halt in the case of the stack fault example given and external logic will then reset the system. If the abort is handled in software, then the operating system may deal with it either by terminating the currently running program or by triggering a software reset of the system. Symbian OS deals with aborts by terminating the current thread of execution if it is a user thread or by rebooting the system if the abort occurs in kernel-side code.

Faults

A fault is an exception that occurs during instruction execution, signifying an error that may be recoverable - unlike an abort. The operating system can attempt to rectify the cause of the fault and then retry the instruction.

An example of this is a page fault. Suppose an instruction references a virtual memory address that is not currently mapped to any physical memory page. An operating system that supports demand paging (any of the common desktop operating systems) will first check that the virtual address accessed is valid; if it is, it will load the page containing the address referenced from backing store, and then retry the instruction.

Traps

A trap is an exception that replaces execution of a particular instruction. The usual recovery action ends with program execution resuming at the instruction after the one that caused the trap.

An example of this is an undefined instruction. This can be used to aid the emulation of coprocessors or other hardware blocks that are not physically present in the device. The instructions that normally access the coprocessor or hardware will be treated as undefined instructions and cause an exception; the exception handler emulates the missing hardware and then program execution resumes after the trapped instruction.

Programmed exceptions

Programmed exceptions result from the execution of specific instructions whose sole purpose is to cause an exception. These are of great importance to the functioning of an operating system. On a processor with multiple privilege levels, application code usually runs at the lowest privilege level, and the kernel of the operating system runs at the highest privilege level. How then can an application call the operating system to perform some task for it? The application cannot execute kernel code using a standard call mechanism - instead it must cause an exception, which executes at the highest privilege level. Programmed exceptions are the means by which application code gains access to operating system functionality, and for this reason they are also known as system calls.

When a system call is made, a call number is supplied - either as part of the instruction opcode that causes the exception (for example the ARM SWI instruction has 24 bits available for this), or in a register. The exception handler uses this to index into the kernel's system call table to locate the address of the required kernel function. Enough processor registers will be saved to make the system call appear as a standard C function call, although this register saving may be split between the kernel-side exception handler and a user-side shim (a small section of code which simply passes control to another piece of code without doing any work itself other than possibly some rearrangement of parameters) which contains the programmed exception instruction. For more details, please turn back to Chapter 5, *Kernel Services*.

Exceptions on real hardware

In the following sections, I describe in detail the types of exception that can occur on real hardware, and how they are handled on the ARM and Intel IA-32 (also known as X86) processor architectures.

ARM

The ARM architecture uses banked registers and a fixed-size, fixed address vector table to deal with exceptions. In ARM

terminology, there are seven execution modes:

Mode	Description
User (usr)	This is the only non-privileged mode - that is, certain instructions cannot be executed in user mode, and the MMU will block access to memory regions which are set up for privileged access only. User mode is entered explicitly by executing any of the instructions that write to the mode bits of the CPSR. Once the CPU is executing in user mode, only an exception can cause a transition to another mode. Under Symbian OS the majority of code (everything other than the kernel, device drivers and board support package code) executes in user mode.
System (sys)	This is the only privileged mode that is not entered by an exception. It can only be entered by executing an instruction that explicitly writes to the mode bits of the CPSR. It is exited either by writing to the mode bits of the CPSR or by an exception. Symbian OS does not use system mode.
Supervisor (svc)	This is a privileged mode entered whenever the CPU is reset or when a SWI instruction is executed. It is exited either by writing to the mode bits of the CPSR or by an exception other than reset or SWI. With the exception of interrupt service routines and most of the exception preambles, all Symbian OS kernel-side code executes in supervisor mode.
Abort (abt)	This is a privileged mode that is entered whenever a prefetch abort or data abort exception occurs. Symbian OS makes only minimal use of this mode - the exception preamble switches to supervisor mode after saving a small number of registers.
Undefined (und)	This is a privileged mode that is entered whenever an undefined instruction exception occurs. Symbian OS makes only minimal use of this mode - the exception preamble switches to supervisor mode after saving a small number of registers.
Interrupt (irq)	This is a privileged mode that is entered whenever the processor accepts an IRQ interrupt. Under Symbian OS, service routines for IRQ interrupts execute in this mode, unless nested IRQs are supported. IRQ mode is exited either by writing to the CPSR, for example when returning from the interrupt, or if another exception occurs, for example an FIQ interrupt preempting the IRQ service routine.
Fast Interrupt (fiq)	This is a privileged mode that is entered whenever the processor accepts an FIQ interrupt. Under Symbian OS, service routines for FIQ interrupts execute in this mode. FIQ mode is exited either by writing to the CPSR, for example when returning from the interrupt, or if another exception occurs. Under Symbian OS, this last case should never happen, because prefetch aborts, undefined instructions and SWIs are prohibited in interrupt service routines, and all interrupts are masked during an FIQ service routine.

The low 5 bits of the status register (CPSR) determine which mode is currently active. ARM supports two privilege levels - privileged and non-privileged. All execution modes except user mode are privileged. The ARM architecture supports 16 general purpose registers, labeled R0-R15. However, an instruction referring to one of these registers does not always access the same physical register. Accesses to registers R8-R14 refer to different physical registers depending upon the current execution mode.

Figure 6.1 illustrates which physical registers are accessed in each execution mode. You can see that R0-R7 are the same across all modes - the user mode registers are always used. We say that R0-R7 are never banked.

R13 and R14 are banked across all modes apart from system mode - each mode that can be entered because of an exception has its own R13 and R14. These registers generally hold the stack pointer and the return address from function calls respectively.

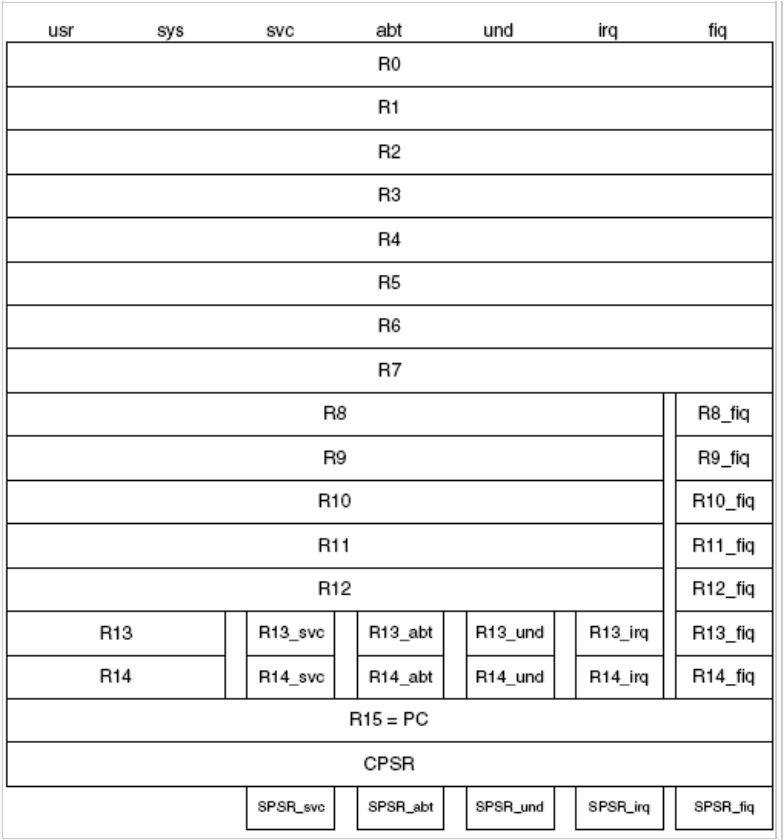

Figure 6.1 *ARM registers*

Also, each mode that can be entered by an exception has a SPSR (saved processor status register).

The actions taken by ARM CPUs on recognizing an exception are:

1. For exceptions other than resets, the CPU saves the return address from the exception in the banked R14 for the respective exception mode
2. For exceptions other than resets, the CPU copies the current value of the CPSR to the SPSR for the respective exception mode
3. The CPU changes the execution mode to that appropriate for the type of exception
4. The CPU disables normal (IRQ) interrupts. If it is processing an FIQ (fast interrupt) it disables FIQs, otherwise it leaves them enabled
5. The CPU continues execution at the vector address for the exception concerned. It always starts execution in ARM (not Thumb) mode. This means that the first part of the exception handler must be written in 32-bit ARM instructions rather than 16-bit Thumb instructions. Of course the handler can change to Thumb mode if it wishes.

Figure 6.2 illustrates these actions.

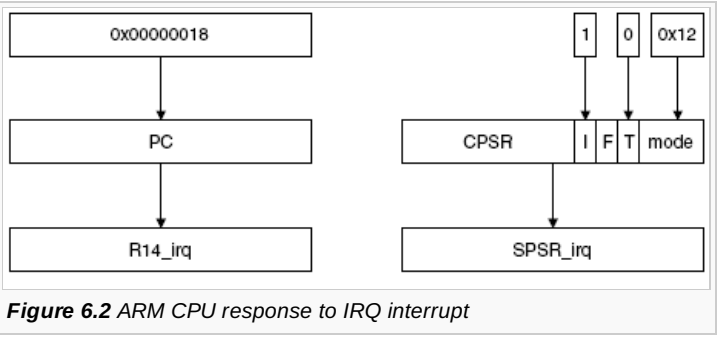

Figure 6.2 *ARM CPU response to IRQ interrupt*

When an exception is recognized, the processor only saves the return address and CPSR. Of course an exception handler will need to make use of some working registers too. The software handler must save these on the stack, and then restore them from the stack before returning from the exception. The banked R13 ensures that the exception handler has access to a valid stack area to which it can save its working registers.

The following table lists all the exceptions supported by the ARM architecture, along with the execution mode into which the exception puts the processor and the vector address for the exception:

Exception	Mode	Vector	Category
Reset	svc	0 × 00	Reset

Undefined Instruction	und	0 × 04	Fault, Trap or Abort
SWI	svc	0 × 08	Programmed Exception
Prefetch Abort	abt	0 × 0C	Fault or Abort
Data Abort	abt	0 × 10	Fault or Abort
IRQ	irq	0 × 18	Interrupt
FIQ	fiq	0 × 1C	Interrupt

As you can see in the previous table, the ARM core directly supports only two interrupt sources. External logic drives two signals, IRQ and FIQ, to signal these interrupts. FIQ has a higher priority than IRQ; if both are asserted simultaneously the FIQ is recognized first. What is more, IRQ interrupts are masked when an FIQ is recognized but FIQ interrupts are not masked when an IRQ is recognized. This means that FIQ interrupts can usually interrupt the service routine for an IRQ interrupt. Registers R8-R12 are banked for FIQ mode, which allows some FIQ interrupts to be serviced without the need to save working registers on the stack. This reduces the time taken to service the interrupt.

For most systems, and certainly for systems running Symbian OS, more than two interrupt sources are required. Because of this, we use an external interrupt controller. This accepts a number (typically 32 to 128) of interrupt signals from various peripherals.

The interrupt controller may provide the following services:

- Allow individual interrupt sources to be masked
- Allow the processor to look in one central place to discover which sources are currently pending
- Allow each source to be routed to either the IRQ or FIQ input to the processor
- Allow edge-triggered inputs to be latched before being fed to the processor.

The interrupt controller asserts the IRQ input to the processor if any interrupt source is:

1. Pending
2. Not masked
3. Routed to IRQ.

A similar rule applies to FIQ. On accepting an interrupt, the processor must check the interrupt controller to discover which sources are both pending and enabled. It then applies a software prioritization scheme to select one of these to be serviced. When the service routine completes, the procedure is repeated and another interrupt may be serviced. This continues until there are no more pending interrupts.

Intel IA-32

Most RISC processors use exception handling schemes similar to the one I described for ARM, in which special registers are used to hold return information from exceptions. The IA-32 architecture, coming from a CISC heritage, handles exceptions differently. The IA-32 architecture has an explicitly designated stack pointer register, ESP, along with special instructions that reference the stack (PUSH and POP). When it recognizes an exception, an IA-32 processor will push the return address and return status register onto the stack.

Before I go on to talk about IA-32 exception handling, it might be useful if I describe IA-32 memory addressing. Since Symbian OS runs in IA-32 protected mode, I will only cover that mode here. IA-32 protected mode uses a two-component memory address consisting of a segment selector and a 16- or 32-bit offset. The segment selector is specified by one of six 16-bit segment selector registers, as shown in the following table:

Register	Name	Description
CS	Code Segment	Specifies the segment for all instruction fetches. EIP specifies the offset component for instruction fetches.
SS	Stack Segment	Specifies the segment for all explicit stack instructions, including subroutine calls and returns and exception handling. ESP specifies the offset for explicit stack operations.
DS	Data Segment	Specifies the segment for data memory references other than those to the stack.
ES	Extra Segment	Specifies the segment for data memory references which explicitly indicate that ES is to be used.
FS	Second Extra Segment	Similar to ES.
GS	Third Extra Segment	Similar to ES.

The segment selectors are interpreted as follows:

- Bits 0 and 1 are known as the requestor privilege level (RPL) of the selector
- Bit 2 specifies whether the selector is local (1) or global (0). Symbian OS uses only global selectors
- Bits 3--15 form a 13-bit index into a descriptor table.

Bits 3--15 of the selector point to an 8-byte entry in the global descriptor table (GDT), which gives the base address of the segment, its size, privilege level and type (code, data, system information).

We find the effective memory address by adding the segment base address from the GDT entry to the 16- or 32-bit offset. This effective address is known as a linear address in IA-32 terminology. If paging is disabled, it is used directly as a physical address, but if it is not, then it is translated to a physical address using the page tables.

The IA-32 architecture supports four privilege levels (also known as rings). Level 0 is the most privileged; all instructions and resources are available at this level. Level 3 is the least privileged; application code usually runs at this level.

The RPL of the selector currently in CS is known as the current privilege level (CPL) and specifies the privilege level of the code that is currently executing. For segment registers other than CS, the RPL indicates the privilege level of the code that originated the selector - hence the name, requestor privilege level. So the RPL may not be the same as the CPL of the code currently executing - for example the selector may have been passed in as an argument from less privileged code.

The kumem functions use this method to ensure that user code is not allowed to write with kernel privileges; see Section 5.2.1.5 for more on this.

Symbian OS uses five segments and only privilege levels 0 and 3. We have one level 0 code segment and one level 0 data segment, both covering the entire 4GB linear address space. We also have one level 3 code segment and one level 3 data segment, each covering the lower 3GB of linear address space. Finally, we have a single task state segment, which I will describe later in this chapter.

Returning to IA-32 exception handling, each exception other than reset has an 8-bit vector number associated with it. Numbers 0 to 31 are reserved for standard exceptions such as interrupts, page faults and division-by-zero, as described in the following table of all supported exceptions on IA-32 architectures.

Vector	Description	Category	Error Code
	Reset	Reset	-
0	Division by zero	Abort	No
1	RESERVED		
2	Non-maskable Interrupt (NMI)	Interrupt	No
3	Breakpoint	Programmed Exception	No
4	Overflow	Abort	No
5	Out of bounds (BOUND instructions)	Abort	No
6	Invalid opcode	Trap or Abort	No
7	Device not available	Fault or Abort	No
8	Double Fault	Abort	Yes
9	RESERVED		
10	Invalid Task State Segment (TSS)	Abort	Yes
11	Segment not present	Fault or Abort	Yes
12	Stack segment error	Abort	Yes
13	General protection error	Abort	Yes
14	Page fault	Fault or Abort	Yes
15	RESERVED		
16	Floating point error	Trap or Abort	No
17	Alignment check error	Abort	Yes
18	Machine check error (Pentium and later)	Abort	No
19	SIMD Floating point exception (Pentium III and later)	Trap or Abort	No
20 - 31	RESERVED		
32 - 255	User defined exception; either hardware interrupt signaled via the INTR line or execution of INT instruction	Programmed Exception or Interrupt	No

When an exception is recognized, the processor uses the vector number to index the interrupt descriptor table (IDT). This is a table of 8-byte entries whose linear base is stored in the IDTR register. Each entry contains one of the following:

- A task gate; these are not used by Symbian OS

- A trap gate; this specifies a new CS and EIP indicating an address to which instruction execution should be transferred
- An interrupt gate; this is the same as a trap gate apart from the interrupt mask behavior.

Since a new CS selector is specified, a change of privilege level can occur when an exception is handled. The processor will not permit an exception to transfer control to a less privileged level. On Symbian OS, all exception handlers execute at level 0.

In the interests of security and robustness, it is a general principle that more privileged code must not rely on the validity of any data or addresses passed by less privileged code. So if an exception results in a change of CPL, the processor changes the stack from the current SS:ESP, which is accessible to less privileged code.

The processor uses the task state segment (TSS) in stack switching. The task register (TR) contains a segment selector that refers to a TSS descriptor in the GDT. The TSS descriptor specifies where in the linear address space the TSS resides. The TSS contains various fields related to the IA-32 hardware task switching mechanism but the only one relevant to Symbian OS is the privilege level 0 initial stack pointer, SS0:ESP0. When an exception occurs and control is transferred to privilege level 0 from a less privileged level, SS:ESP is loaded with the SS0:ESP0 value from the TSS. In Symbian OS this is always set to point to the top of the current thread's supervisor stack.

With this background information given, I can now describe in detail the response of an IA-32 processor to an exception:

1. The processor uses the exception vector number to index the IDT and obtain the CS:EIP of the exception handler
2. If the new CS specified in the IDT necessitates a transfer to a more privileged level, the processor loads SS and ESP from the currently active TSS and then pushes the original SS and ESP onto the new stack. All subsequent stack operations use the new stack
3. The processor pushes the EFLAGS register and then the current CS and EIP values
4. If the IDT entry contains an interrupt gate, the processor clears the IF flag so that maskable interrupts are disabled. A trap gate does no affect the IF flag
5. Depending on the type of exception, the processor may push an error code
6. The processor transfers control to the handler indicated by the CS and EIP values read from the IDT.

Like the ARM, the IA-32 architecture supports only two physical interrupt inputs to the processor. However, these inputs behave very differently to those on the ARM.

The NMI line always causes a vector 2 interrupt and may not be masked by software. It is of limited use in most IA-32 systems. Most hardware interrupts are directed to the INTR line. Interrupts signaled on this line are maskable by clearing the IF flags in the EFLAGS register.

IA-32 processors are used with external interrupt controller hardware, typically located in the motherboard chipset. The interrupt controller, which accepts inputs from a number of interrupt sources, allows the interrupt sources to be individually masked and prioritized. It also associates a vector number with each interrupt source.

When an enabled interrupt source becomes active, the interrupt controller signals an interrupt to the processor and passes the associated vector number. If interrupts are enabled (IF = 1) the processor calls the appropriate vector handler, as listed in the IDT. While the interrupt is in service the interrupt controller prevents lower priority interrupt sources from being signaled to the processor.

At the end of the software handler, the processor signals EOI (end of interrupt) to the interrupt controller. At this point lower priority interrupts can once more be signaled to the processor.

Interrupts

In this section, I will return to EKA2, and describe how it handles interrupts.

EKA2 interrupt handling

There are four phases in the handling of any interrupt. Figure 6.3 illustrates these four phases, and shows where in the kernel the code for each phase is located.

Interrupts occur very frequently during normal system operation - the system tick interrupts every millisecond and during some I/O operations, such as bulk data transfer over USB, an interrupt may occur every 50 µs. Because of this, the interrupt handling code is optimized for speed. The fewest possible registers are saved at each stage and the entire interrupt code path within the kernel and the dispatcher is written in hand-optimized assembler.

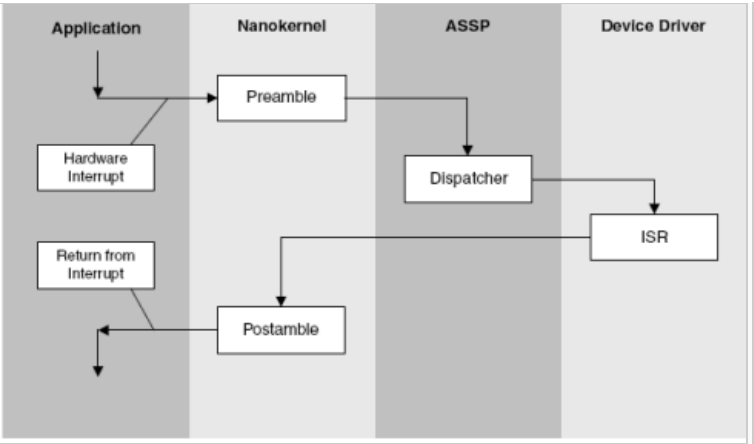

Figure 6.3 *Code flow during interrupt*

Preamble

The first code to execute following the processor response to an interrupt is the preamble, which is part of the nanokernel.

On ARM processors the preamble is entered directly from both the IRQ and the FIQ vectors.

On IA-32 processors, each IDT entry points to a preamble entry stub that saves the vector number, and then jumps to a common interrupt preamble. The preamble's job is to establish the correct processor state, by taking care of any non-automatic stack switching and register saving before the dispatcher and interrupt service routine run. Because interrupts are asynchronous to program execution, the preamble can make no assumptions about the processor state on entry.

EKA2 uses a separate stack for interrupts. This means that we don't need to reserve stack space for interrupts on each thread's supervisor stack, which reduces RAM usage. Depending on processor architecture, the switching of stacks may be performed automatically or by the software preamble:

- On ARM-based hardware, the stack is switched automatically, since an interrupt causes the processor to switch to mode irq or mode fiq and thus R13 irq or R13 fiq becomes the active stack pointer
- On IA-32 hardware, the processor interrupt response includes a switch to the current thread's supervisor stack if the interrupt occurs in user mode (CPL = 3). If the interrupt occurs in supervisor mode (CPL = 0), no stack switch occurs and so the active stack may be a thread supervisor stack or the interrupt stack, if another ISR was interrupted.

On the first of a nest of interrupts, the preamble switches to a separate interrupt stack and saves the original stack pointer on the new stack. This can be seen in lines 10-15 of the IA-32 preamble. The IrqNestCount variable is initialized to -1, so if incrementing it gives zero, this is the first of a nest of interrupts.

Symbian OS interrupt service routines are generally written in C++. Each processor has a calling convention that specifies which registers are preserved across function calls and which are not. On ARM, R0-R3, R12 and CPSR are destroyed by function calls; on IA-32, EAX, ECX, EDX and EFLAGS are destroyed by function calls. Lines 1 and 2 of the ARM interrupt preamble, and lines 1, 2, 3, 6 and 7 of the IA-32 interrupt preamble, save these registers.

ARM interrupt preamble

```
1 SUB LR, LR, #4
2 STMFD SP!, {R0-R3, R12, LR}
3 LDR R12, IrqHandler
4 LDR LR, ArmVectorIrq
5 LDR PC, [R12]
```

IA-32 interrupt preamble

```
1    PUSH DS
2    PUSH ES
3    PUSH EAX
4    MOV AX, SS
5    CLD
6    PUSH ECX
```

```
7    PUSH EDX
8    MOV DS, AX
9    MOV ES, AX
10   MOV EAX, ESP
11   INC DWORD PTR IrqNestCount
12   JNZ Nested
13   LEA ESP, IrqStackTop
14   PUSH EAX
15 Nested:
16   CALL [IrqHandler]
```

The preamble may need to set up more state, depending on the processor architecture. On IA-32, the hardware interrupt response sets up the SS and CS segment selectors, but not DS and ES. The preamble saves these selectors, then sets them both equal to SS, which is the privilege level 0 data segment covering the entire 4 GB linear address range. In addition, it clears the D flag, so that the IA-32 repeated string operations work in the forward direction, from low to high addresses.

On ARM, no further state setup is needed unless we want to support nested interrupts. In this case, ISRs cannot execute in mode irq. This is because a nested interrupt would corrupt R14 irq, which may hold the return address from a subroutine called during execution of the first ISR. So, if support for nested interrupts is required, a switch to mode sys will occur during the preamble. If the interrupt occurred in mode sys (as indicated by SPSR irq), it must be a nested interrupt, so R13 usr points to the interrupt stack. If the interrupt occurred in any other mode, the preamble saves R13 usr, then sets R13 usr to point to the top of the interrupt stack. It must also save R14 usr, since it will be corrupted by any subroutine calls made by the ISR.

Interrupt dispatcher

The role of the dispatcher is to determine the source of the interrupt and to call the registered service routine for that source. It is written as part of the base port for a given hardware platform. On platforms with an ASSP, the dispatcher is usually part of the ASSP extension; otherwise it is part of the variant.

On hardware without vectored interrupt support - which includes most current ARM-based hardware - the dispatcher interrogates the interrupt controller to establish which interrupts are both pending and enabled. It then selects one of these according to a fixed priority scheme, and invokes the corresponding service routine. Once the service routine is completed, the dispatcher will loop and interrogate the interrupt controller again; this continues until there are no more pending interrupts. In the code sample Example ARM IRQ dispatcher, lines 5-8 discover which interrupts are pending and select the one corresponding to the highest numbered bit in the hardware IRQ pending register. If no interrupts are pending, line 7 returns from the dispatcher.

Example ARM IRQ dispatcher

```
1    STMFD SP!, {R4-R6, LR}
2    LDR R4, InterruptControllerBase
3    LDR R5, Handlers
4  dispatch:
5    LDR R12, [R4, #IRQPendingRegOffset]
6    CMP R12, #0
7    LDMEQFD SP!, {R4-R6, PC}
8    CLZ R3, R12
9    ADD R0, R5, R3, LSL #3
10   ADR LR, dispatch
11   LDMIA R0, {R0, PC}
```

On hardware with vectored interrupt support, which includes IA-32 architectures, the dispatcher knows immediately from the vector number which interrupt source is involved. It immediately invokes the service routine and then returns.

EKA2 typically allows only one service routine for each interrupt - although since the dispatcher is outside the Symbian-supplied kernel, this decision is actually up to the base porter, who could easily change this code to allow several service routines to be associated with a single interrupt. It is fairly common among hardware designs to find several interrupts attached to the same line on the interrupt controller. This situation is normally handled by using a sub-dispatcher rather than by registering several handlers with the main dispatcher. A sub-dispatcher is a routine that is bound to a given interrupt source at system boot time, and which performs a similar function to the main dispatcher, but only for interrupts that are routed to a particular line. A common pattern is

that the main interrupt dispatcher is in the ASSP and this deals with the different interrupt sources recognized by the on-chip interrupt controller. Sub-dispatchers are in the variant, and these deal with interrupts from *companion* chips - peripheral devices external to the ASSP. These chips generally produce a single combined interrupt output, which is recognized as a single source by the main interrupt controller.

Interrupt service routines

The role of the interrupt service routine (ISR) is to perform whatever action is necessary to service the peripheral that generated the interrupt and to remove the condition that caused it to interrupt. This usually involves transferring data to or from the peripheral.

ISRs may be located in the ASSP, variant, an extension or a device driver. ISRs are typically written in C++, although some frequently used ones such as the system tick ISR are written in assembler.

In Chapter 12, *Drivers and Extensions*, I will discuss ISRs in more detail, but this is for the most part irrelevant in understanding how Symbian OS handles interrupts. All we need to know now is that the ISR runs, and that it may make use of a small number of OS services, as follows:

- It may queue an IDFC or DFC (which I will describe in Sections 6.3.2.2 and 6.3.2.3) or, in the case of the millisecond tick ISR, trigger a reschedule directly if the current thread's timeslice has expired
- It may queue or cancel a nanokernel timer
- It may disable or enable interrupts using APIs provided by the ASSP or variant. I will talk about this more later.

Of these, the first is fundamental to the entire operation of the system. This is the way in which external events cause the appropriate kernel-side and user-side tasks to run.

Postamble

The postamble runs after the dispatcher returns, which it does when it has serviced all pending interrupts. The goal of the postamble is to restore the processor's state and permit the interrupted program to resume. However before this, it may need to schedule another thread to run.

The postamble performs the following actions:

1. It checks what code was running when the interrupt was triggered. If that code was not running as part of a thread, then the postamble returns from the interrupt immediately. This covers the case in which one interrupt occurs during servicing of another. It also covers any cases in which the processor is in a transitional state (a thread stack is not active). Such a state can occur in ARM processors just after an abort or undefined instruction exception. The processor enters mode abt or mode und and the active stack is a shared exception stack rather than a thread stack. This stack is only used briefly to save registers before switching to mode svc. However because a non-thread related stack is in use, rescheduling cannot occur. Lines 1, 4, 6 and 7 in the code sample ARM IRQ postamble check for this - if the interrupted mode is neither usr nor svc, we return from the interrupt immediately.

ARM IRQ postamble

```
1    MRS R0, SPSR
2    LDR R1, TheScheduler
3    ADD R12, SP, #24
4    AND R2, R0, #0x1F
5    LDR R3, [R1, #iKernCSLocked]
6    CMP R2, #0x10
7    CMPNE R2, #0x13
8    CMPEQ R3, #0
9    BNE IrqExit0
10   MOV R2, #0xD2
11   MSR CPSR, R2
12   LDR R2, [R1, #iRescheduleNeededFlag]
13   ADD R3, R3, #1
14   MOV LR, #0x13
15   CMP R2, #0
16   BEQ IrqExit0
17   STR R3, [R1, #iKernCSLocked]
18   MSR CPSR, LR
19   LDMDB R12!, {R1-R3}
```

```
20   STMFD SP!, {R1-R3}
21   LDMDB R12!, {R1-R3}
22   STMFD SP!, {R1-R3}
23   STMFD SP!, {R0, LR}
24   MOV R2, #0x13
25   MOV LR, #0x92
26   MSR CPSR, LR
27   ADD SP, R12, #24
28   MSR CPSR, R2
29   BL Reschedule
30   LDMFD SP!, {R1, LR}
31   ADD SP, SP, #24
32   MOV R12, SP
33   MOV R2, #0xD2
34   MSR CPSR, R2
35   MSR SPSR, R1
36   LDMDB R12, {R0-R3, R12, PC}
37 IrqExit0:
38   LDMFD SP!, {R0-R3, R12, PC}
```

2. It checks if preemption is disabled - if this is the case, it returns from the interrupt immediately (lines 5 and 8).

3. It checks if an IDFC (see Section 6.3.2.2) or a reschedule is pending; if not, it returns from the interrupt immediately. Lines 10, 11, 12, 15, 16 are responsible for this. Note that the postamble must perform this check with all interrupts disabled, not just those at the same hardware priority as the one just serviced. If this were not the case, a higher priority interrupt, such as an FIQ, could run and queue an IDFC just after the current interrupt (IRQ) performed the check. The FIQ postamble would not run the IDFC since it interrupted mode irq, and the IRQ postamble would not run the IDFC since it already decided there was no IDFC pending. The IDFC would then be subject to an unpredictable delay, until either another interrupt occurred or the current thread performed some action that resulted in rescheduling.

4. It disables preemption and re-enables interrupts (lines 13, 14, 17, 18). It transfers all saved state from the interrupt stack to the supervisor stack of the interrupted thread (lines 3 and 19-28). It calls the scheduler. This runs any pending IDFCs and then performs a context switch if one is needed. The call to the scheduler returns when the interrupted thread is next scheduled. Internally, the scheduler performs context switches by switching stacks. Any thread that is not currently executing has a call to the scheduler at the top of its call stack.

5. It restores the interrupted thread state from the supervisor stack and returns from the interrupt (lines 30-36).

Interaction with scheduling

The processing required for an external event generally splits into a number of stages with different response time requirements. For example, consider a PPP connection over a serial port. The UART receives data and stores it in its internal FIFO. When the FIFO is half-full, the UART raises an interrupt, which must be serviced before the FIFO becomes completely full to avoid data loss.

So, the first stage of processing is to move the data from the UART's receive FIFO to an internal memory buffer - and the deadline for this is the time taken to receive half a FIFO of data. Let's say that this is 8 characters at 115,200 bps, which gives us a time of 694 µs.

The second stage of processing is to perform PPP framing, verify the frame check sequence, transition the PPP state machine and transmit any acknowledgment required. The deadline for this is determined by the time that the peer PPP entity will wait before timing out the acknowledgment. This will be much longer than the first stage deadline, so second-stage processing can occur at lower priority than the receive interrupt, in a thread. In this way, further receive interrupts will preempt the second stage processing of earlier frames.

In a similar way, the PPP thread must preempt other activities with longer deadlines and long-running activities. The receive interrupt signals the PPP thread that data is available, which triggers the preemption. In general terms, processing for events with short deadlines should preempt processing for events with longer deadlines. This is done by using threads with differing priorities for the different types of event, with the most time critical events being handled directly by ISRs.

The kernel lock

In previous sections, it has become clear that there must be a method by which interrupts can cause the appropriate threads to run

so that an event can be processed. In this way, the response can occur in stages, with the most urgent part being handled by the ISR itself and less urgent parts being handled by threads of decreasing priority.

To ensure that response deadlines are met, the time between a hardware interrupt being signaled and the ISR running must be bounded (that is, it must have a maximum latency) and we want this latency to be as short as possible. This translates into a requirement that interrupts be enabled all the time apart from in sections of code whose execution time is bounded and as short as possible. To satisfy this requirement, most code, whether kernel- or user-side, executes with interrupts enabled. This includes code that manipulates global structures such as the thread ready list. To prevent such code from being re-entered and corrupting the global structure, a preemption lock (iKernCSLocked, usually known as the kernel lock) is employed.

The kernel lock is a simple counter that is normally zero. Sections of code that need to protect themselves against rescheduling increment the kernel lock at the beginning of the critical section and decrement it at the end. Then, when an interrupt occurs, the kernel will only attempt a reschedule if the kernel lock was zero at the time of the interrupt. This can be seen in step 2 of the interrupt postamble described in Section 6.3.1.4, and in lines 3 and 8 of the code sample ARM IRQ postamble.

Of course this method can only work if the ISR itself does not invoke any of these critical sections of code. We disable rescheduling in certain code sequences because they need to atomically manipulate structures such as the thread ready list. Disabling rescheduling prevents a second thread from running and modifying these structures while the first thread is still halfway through its modification. However disabling rescheduling does not disable interrupts, so an ISR that modified the thread ready list directly would still conflict with threads modifying it. Therefore ISRs may not add a thread to the ready list. In fact, they may not use any OS services other than those listed in Section 6.3.1.3.

IDFCs

Interrupts cause the scheduling of a thread by means of an Immediate Deferred Function Call (IDFC). IDFCs are objects that specify a function that will be called after the ISR, as soon as the system is in a suitable state. We call them *immediate* because they normally run before returning from the interrupt, not later on, in a kernel thread. The exception to this is if the kernel was locked when the interrupt occurred, in which case IDFCs are run immediately after the kernel is unlocked.

It works like this. First the ISR adds an IDFC to the IDFC pending queue, which is always accessed with interrupts disabled. When the scheduler next runs, it calls the function associated with the IDFC directly. (The function is not called by a thread.) This is how the state of the kernel lock governs when IDFCs are called. If an interrupt occurs when the kernel lock count is nonzero, the ISR runs but nothing else happens. If an interrupt occurs when the kernel lock count is zero, the ISR runs and afterwards, if IDFCs have been queued, the scheduler is called and it runs the IDFCs. The IDFCs may add one or more threads to the ready list, after which the scheduler may select a new thread to run.

IDFCs are called in the same order in which they were originally queued. They are called with interrupts enabled and with the kernel lock count equal to 1; this guarantees that they will not be re-entered or preempted, either by another IDFC or by a thread.

Most interrupts do not need further processing after the ISR has run. For example, the system tick ISR runs every millisecond but, unless a timer expires on this particular tick, no IDFCs need to run and no reschedule is required. In this common case, to save the time taken to run the scheduler (lines 17-36 in the code sample ARM IRQ postamble, as opposed to line 38) we use a flag, known as the DFC pending flag, to indicate that one or more IDFCs have been added to the pending queue. The interrupt postamble only needs to call the scheduler if this flag is set. The flag is reset when all pending IDFCs have been processed.

We use a similar flag, the reschedule needed flag, to indicate that changes to the ready list have occurred that may require a new thread to be scheduled. After IDFCs have been run, we will only select a new thread to run if this flag is set; the flag is cleared as part of the reschedule.

There are two places where the scheduler, and hence IDFCs, may run. The first is during the interrupt postamble (line 29 in the code sample ARM IRQ postamble) and the second is at the point where a thread releases the kernel lock.

In the first place, IDFCs will run if they were added by any of the ISRs that ran during interrupt processing. A new thread will then be selected to run if any of these IDFCs causes a thread with a priority greater than or equal to that of the interrupted thread to become ready.

In the second place, IDFCs will run if they have been added by the thread that held the kernel lock or by ISRs that ran while the kernel lock was held. A new thread may then be selected for three different reasons:

1. If any thread with a priority greater than or equal to that of the interrupted thread is made ready (either by the IDFCs or by the thread that held the kernel lock)
2. If the thread that held the kernel lock removes itself from the ready list
3. If thread priorities are changed by the thread that held the kernel lock.

Figure 6.4 illustrates the processing of an IDFC in the case where the kernel was originally locked and where no thread switch is

required following the IDFC.

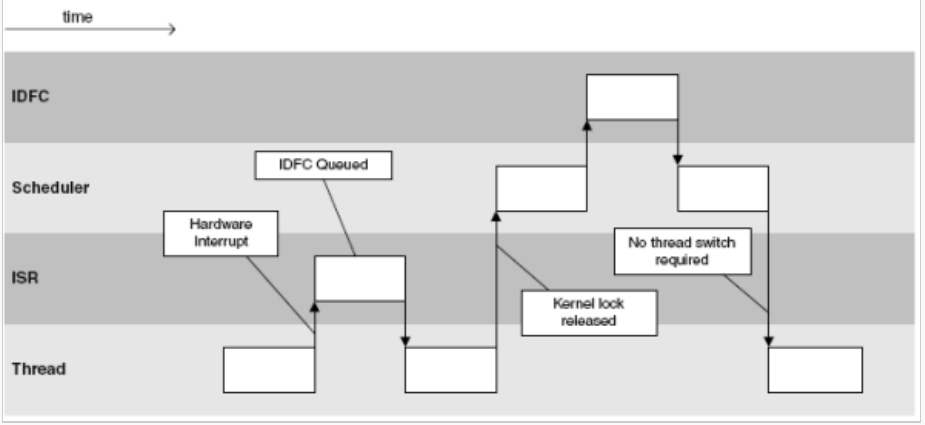

Figure 6.4 *IDFC processing*

DFCs

As I will explain in Section 6.3.2.4, IDFCs must be short and there are restrictions on which kernel services they may use. For these reasons IDFCs are rarely used directly except by RTOS personality layers. Instead, ISRs generally use Deferred Function Calls (known as DFCs) when they want to schedule a thread or perform other tasks not possible from within the ISR itself. DFCs make use of IDFCs in their implementation, so ISRs indirectly use IDFCs whenever they use DFCs.

A DFC is an object that specifies a function to be called in a particular kernel thread. DFCs are added to DFC queues. Exactly one kernel thread is associated with each DFC queue, but not all kernel threads are associated with a DFC queue.

DFCs have priorities that are between 0 and 7 inclusive. Within any given DFC queue, the associated kernel thread schedules DFCs cooperatively. It removes the highest priority DFC from the queue and calls its function. When the function returns, the kernel thread processes the next highest priority DFC; it processes DFCs with the same priority in the order that they were added to the queue. Once there are no DFCs remaining on the queue, the kernel thread blocks until another DFC is added to the queue. Each DFC must run to completion before any others on the same queue can run. However, since a different kernel thread services each DFC queue, a DFC running in a higher priority thread may preempt a DFC running in a lower priority thread.

A DFC may be queued from any context - from an ISR, IDFC or thread. However, the kernel handles these contexts a little differently. If an ISR queues a DFC, then the kernel adds it to the **IDFC** pending queue. (This is possible because IDFCs and DFCs are objects of the same type.) Then, when the scheduler runs, the nanokernel transfers the DFC to its final (DFC) queue and, if necessary, makes the corresponding kernel thread ready. Essentially, the DFC makes use of an IDFC with a callback function supplied by the kernel, which transfers the DFC to its final queue. This two-stage process is necessary because a DFC runs in a kernel thread and ISRs are not allowed to signal threads; however IDFCs are. Of course, if an IDFC or a thread queues a DFC, this two-stage procedure is not necessary; instead the kernel adds the DFC is directly to its final (DFC) queue.

Figure 6.5 illustrates the processing of a DFC queued by an ISR in the case where the kernel is unlocked when the interrupt occurs and where the DFC thread has higher priority than the interrupted thread.

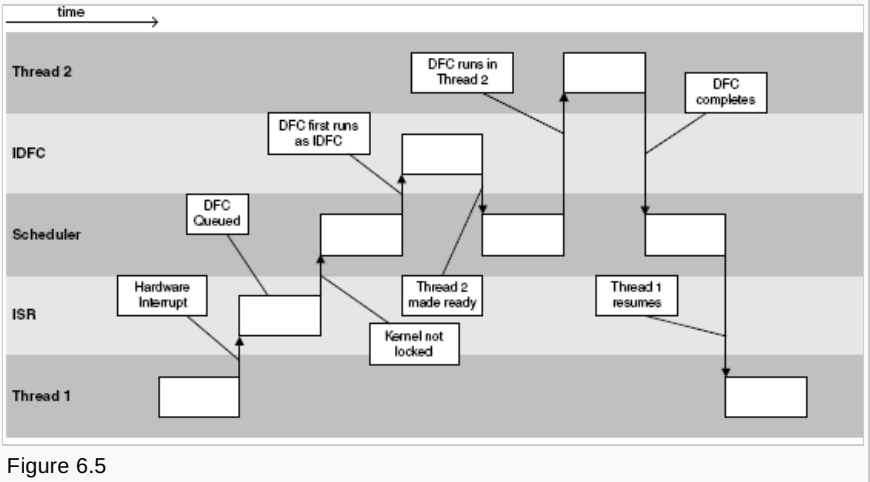

Figure 6.5

Kernel services in ISRs, IDFCs and DFCs

As I said earlier, to minimize interrupt latency we keep interrupts enabled most of the time - the only exceptions being a small number of short sections of code. Because of this, most operating system data structures might be in an inconsistent state during

an ISR. This means that, during ISR processing, we can make no assumptions about the state of any of the following:

- The thread ready list
- Nanokernel threads
- Fast mutexes and fast semaphores
- DFC queues
- Virtual memory mappings for most memory areas accessible from user mode.

ISRs cannot manipulate any Symbian OS thread or wait object, since the system lock fast mutex must be held while doing that. Nor can an ISR access user memory, such as the stacks and heaps of a user-mode thread.

In fact, the only services available to ISRs are:

- Queuing IDFCs and DFCs
- Queuing and canceling nanokernel timers
- Enabling and disabling interrupts.

Earlier, I said that IDFCs would only run after an ISR if the kernel were unlocked at the time the interrupt is serviced. This means that all nanokernel objects will be in a consistent state when IDFCs run, and so IDFCs can do a lot more than ISRs. They can:

- Make nanokernel threads ready
- Manipulate DFC final queues
- Signal fast semaphores
- Perform all operations available to ISRs.

However IDFCs may not block waiting for another thread to run. This is because IDFCs run with the kernel locked, which means that the kernel cannot reschedule. Since waiting on a fast semaphore or fast mutex is effectively blocking, IDFCs cannot do this either. And to take this argument to its conclusion, the prohibition on the use of fast mutexes in IDFCs also means that they may not perform any operations on Symbian OS threads or wait objects, since these are protected by the system lock fast mutex.

Similarly, the address space of a non-running user process is only guaranteed to be consistent when the system lock is held. Since it is not known which process is currently running when an IDFC runs, it may not access any user memory. Another reason for this prohibition is that exceptions are not tolerated during IDFCs.

Note that IDFCs run with preemption disabled, so they should be kept as short as possible.

DFCs run in the context of a kernel thread. In principle, this could be a bare nanokernel thread (NThread) but in practice, with the possible exception of code running in an RTOS personality layer, DFCs run in a Symbian OS kernel thread (DThread). This means that the full range of kernel services is available to DFCs, including but not limited to the following:

- Waiting on or signaling either nanokernel or Symbian OS wait objects
- Allocating or freeing memory on the kernel heap
- Accessing user memory
- Completing Symbian OS asynchronous requests (TRequestStatus).

Round-robin scheduling

The kernel schedules threads with equal priorities in a round-robin fashion. That is, each thread executes for a certain amount of time (its timeslice) and then the kernel schedules the next thread in cyclic order with the same priority. We implement this using the `iTime` field in the nanothread control block - this counts the number of nanokernel timer ticks remaining in the thread's time slice. The kernel decrements `iTime` on each nanokernel tick provided this field was initially positive. If `iTime` becomes zero, the kernel sets the reschedule needed flag, which causes the scheduler to run at the next opportunity. This is the only occasion where an interrupt triggers a reschedule directly rather than via an IDFC.

Using interrupts

The interrupt APIs

Device drivers and extensions gain access to interrupts via a generic interrupt management API, which I will describe later. The base porter implements this API in the ASSP module, if there is one, or in the variant, if there is not. In systems with an ASSP, parts of the implementation may remain in the variant - typically those parts dealing with interrupt sources outside the ASSP and that make use of a second-level dispatcher. However, the public functions themselves will be implemented in the ASSP.

The interrupt management API is encapsulated in a static class, Interrupt, defined as follows:

```
typedef void (*TIsr)(TAny*);
```

```
class Interrupt
  {
public:
  static TInt Bind(TInt aId, TIsr aIsr, TAny* aPtr);
  static TInt Unbind(TInt aId);
  static TInt Enable(TInt aId);
  static TInt Disable(TInt aId);
  static TInt Clear(TInt aId);
  static TInt SetPriority(TInt aId, TInt aPriority);
  };
```

The interrupt management API uses a 32-bit integer identifier to specify the interrupt source that is referred to. The mapping between interrupt sources and numeric identifiers is usually defined in a public header file exported from the ASSP module. A second public header file, exported from the variant, defines identifiers for interrupt sources external to the ASSP.

Next I shall describe each interrupt management method in turn.

```
TInt Interrupt::Bind(TInt aId, TIsr aIsr, TAny* aPtr);
```

This method associates an ISR with the interrupt source whose numeric identifier is specified by parameter aId. Parameter aIsr specifies the interrupt service routine. Following a successful return from `Interrupt::Bind()`, interrupts from the specified source cause function aIsr() to be called. The aPtr parameter is passed to aIsr as an argument. This will typically be a pointer to some data required by the ISR - for example the *physical channel* object for an interrupt in a physical device driver.

`Interrupt::Bind()` returns an error code if the specified interrupt identifier is invalid or if the requested interrupt source has already been bound.

```
TInt Interrupt::Unbind(TInt aId);
```

This method disables interrupts from the specified interrupt source and then removes any ISR that is currently bound to it. An error code is returned if the interrupt identifier is invalid or if there is no ISR associated with the interrupt source.

```
TInt Interrupt::Enable(TInt aId);
```

This method enables interrupts from the specified interrupt source. Note that this only enables the interrupt source in the main interrupt controller; further setup in the peripheral itself may be necessary before interrupts can actually occur. An error code is returned if the interrupt identifier is invalid or if there is no ISR associated with the interrupt source. An interrupt may not be enabled if it does not have an ISR associated with it.

```
TInt Interrupt::Disable(TInt aId);
```

This method disables interrupts from the specified interrupt source. Note that it only disables the interrupt source in the main interrupt controller; the peripheral itself is unaffected. An error code is returned if the interrupt identifier is invalid.

```
TInt Interrupt::Clear(TInt aId);
```

This method clears any pending interrupt from the specified source. It returns an error code if the interrupt identifier is invalid. This method is rarely used - most interrupts are cleared either implicitly during servicing or explicitly by writing to a hardware register. For example, a UART receive interrupt is usually cleared by reading all available data from the UART receive FIFO; a timer interrupt is cleared by writing to an acknowledgment register. The `Interrupt::Clear()` method is generally only used where the aId value is determined at runtime instead of being hard coded.

```
TInt Interrupt::SetPriority(TInt aId, TInt aPriority);
```

On hardware that supports multiple interrupt priority levels and that allows interrupt sources to have their priorities set dynamically, this call changes the hardware priority level of the specified interrupt source. The method returns an error code if this functionality is not supported or if the identifier is invalid. A typical use of this would be on ARM-based systems where the interrupt controller allows each interrupt source to be routed to either IRQ or FIQ. The board support package and device drivers for such a platform would call the `Interrupt::SetPriority()` API during initialization to configure the hardware to route each interrupt to either IRQ or FIQ as appropriate.

APIs for IDFCs and DFCs

IDFCs and DFCs are both represented by objects of the `TDfc` class. The public parts of this class are as follows:

```
typedef void (*TDfcFn)(TAny*);

class TDfc
  {
public:
  TDfc(TDfcFn aFn, TAny* aPtr);
  TDfc(TDfcFn aFn, TAny* aPtr, TInt aPri);
  TDfc(TDfcFn aFn, TAny* aPtr, TDfcQue* aQ, TInt aPri);
  void Add();
  void Cancel();
  void Enque();
  void Enque(NFastMutex* aMutex);
  void DoEnque();
  inline TBool Queued();
  inline TBool IsIDFC();
  inline void SetDfcQ(TDfcQue* aQ);
  inline void SetFunction(TDfcFn aFn);
  };
```

Now let's look at these public functions in more detail:

```
TDfc(TDfcFn aFn, TAny* aPtr);
```

Whether the `TDfc` represents an IDFC or a DFC depends on how it is constructed. This constructor initializes the `TDfc` object as an IDFC. Function aFn will be called when the IDFC runs, and aPtr will be supplied as the argument to aFn.

```
TDfc(TDfcFn aFn, TAny* aPtr, TInt aPri);
TDfc(TDfcFn aFn, TAny* aPtr, TDfcQue* aQ, TInt aPri);
```

These constructors initialize the `TDfc` object as a DFC. Function aFn will be called when the DFC runs and aPtr will be supplied as the argument to it. Parameter aPri specifies the priority of the DFC within its DFC queue and must be between zero and seven inclusive. Parameter aQ specifies which DFC queue the DFC will run on. The version of the constructor without the parameter aQ will initialize the queue pointer to NULL, and the DFC queue to be used must be set with SetDfcQ before the DFC can be queued.

```
void Add();
```

This method places an IDFC or a DFC on the IDFC pending queue. The method is idempotent - if the object is already queued, no action is taken. This method is almost always called from an ISR. It may also be called from an IDFC or a thread context with preemption disabled, but it must not be called from a thread context with pre-emption enabled.

```
void Cancel();
```

This method removes an IDFC or DFC from any queue it is currently on. If the object is not currently queued, no action is taken. You must only call this method from IDFCs or threads, not from ISRs.

```
void Enque();
void DoEnque();
```

These methods place a DFC directly on its final queue without going via the IDFC pending queue. They must not be called on a `TDfc` object representing an IDFC. Both methods take no action if the DFC is already queued. You can only call the `DoEnque()` from an IDFC or a thread context with preemption disabled; it is the recommended way to queue a DFC from inside an IDFC. You may also call `Enque()` from a thread context with preemption enabled; it is the recommended way to queue a DFC from a thread.

```
void Enque(NFastMutex* aMutex);
```

This method is equivalent to `Enque()` followed immediately and atomically by signaling the specified fast mutex. If aMutex is NULL, the system lock is signaled. The call is atomic in that no reschedule may occur between the DFC being queued and the fast mutex being released. This method may only be called from a thread context.

The method is useful when both the thread queuing a DFC and the DFC thread itself need to access a data structure protected by a fast mutex. Let's consider what would happen if it did not exist. The first thread would acquire the mutex, update the structure, queue the DFC and then release the mutex. If, as is commonly the case, the DFC thread had the higher priority, a reschedule would occur immediately after the DFC was queued, with the fast mutex still held. The DFC would immediately try to acquire the mutex and find it locked, causing another reschedule back to the first thread. Finally, the first thread would release the fast mutex and there would be yet another reschedule back to the DFC thread, which could then claim the mutex and proceed. The use of an atomic *queue DFC and release mutex* operation eliminates the last two of these reschedules, since the DFC thread does not run until the mutex has been released.

```
TBool IsQueued();
```

This method returns `ETrue` if the `TDfc` object is currently on the IDFC pending queue or a DFC final queue.

```
TBool IsIDFC();
```

This method returns `ETrue` if the `TDfc` object represents an IDFC, `EFalse` if it represents a DFC.

```
void SetDfcQ(TDfcQue* aQ);
```

This sets the DFC queue on which a DFC will run. It is intended for use in conjunction with the `TDfc` constructor that does not specify a DFC queue, which is used when the queue is not known at the time the DFC is constructed. For example, this can happen in logical device drivers where the `DLogicalChannelBase`-derived object contains embedded DFCs, which are therefore constructed when the logical channel object is instantiated. This occurs before the logical channel has been bound to any physical device driver, so if the DFC queue is determined by the physical device driver, the `TDfc::SetDfcQ` method must be used to complete initialization of the DFCs once the queue is known.

The `TDfc::SetDfcQ` method must only be used on unqueued DFCs; it will not move a queued DFC from one queue to another.

```
void SetFunction(TDfcFn aFn);
```

This sets the callback function to be used by an IDFC or DFC.

Aborts, traps and faults

In this section, I will describe how Symbian OS handles aborts, traps and faults, and the uses it makes of them. I will use the generic term *exceptions* from here onwards to cover *aborts, traps and faults*.

Response to exceptions

As with interrupts and system calls, the initial and final phases of exception handling occur in the nanokernel. The higher level processing occurs in a per-thread exception handler, which, for Symbian OS threads, is a standard handler in the Symbian OS kernel. Next I shall discuss each of these phases in more detail.

Preamble

The task of the exception preamble is similar to that of the interrupt preamble - to establish the correct context for the exception handlers to run, and to save the state of the system at the point where the exception occurred. However there are two main differences. The first is that the exception preamble saves the entire integer register set of the processor rather than just the minimum set required to restore the system state. The reasons for this are:

1. Exceptions are often indicative of programming errors, especially under Symbian OS, which doesn't support demand-paged virtual memory. Saving the entire register set allows us to generate more useful diagnostics
2. If an exception is intentional rather than the result of a programming error, we often need to modify the processor execution state before resuming the original program. For example, to run a user-side exception handler, we must modify the stack pointer and program counter of the running thread before resuming
3. Exceptions are quite rare events, so the performance penalty in saving and restoring all registers is acceptable.

The second difference from the interrupt preamble is that checks are made on the state of the system at the point where the exception occurred. Depending on the result of these checks, a kernel fault may be raised. This is because exceptions, unlike interrupts, are synchronized with program execution. Certain critical parts of kernel-side code are not expected to cause exceptions - if they do, a kernel fault is raised. This will immediately terminate normal system operation and either drop into the kernel's post-mortem debugger (if present) or (on a production device) cause the system to reboot.

Handling exceptions on ARM processors

On ARM processors, exceptions cause a transition to mode_abt or mode_und, and they disable IRQs but not FIQs. The exception modes have a single, small, shared stack between them - they do not have a per-thread stack. The preamble must switch the processor into mode_svc, so that the exception handler can run in the context of the thread causing the exception, and the kernel can reschedule correctly. The ARM exception preamble proceeds as follows:

1. Saves a small number of registers on the mode_abt or mode_und stack, including the address of the instruction that caused the exception
2. Checks the saved PSR to ensure that the exception occurred in either mode_usr or mode_svc. If not, the exception must have occurred either in an ISR or during the exception preamble itself. In either case, this is a fatal error, and a kernel fault will be raised
3. Enables IRQ interrupts. IDFCs queued by IRQs or FIQs will not run now, because the processor mode is abt or und, which is equivalent to the kernel lock being held (see Section 6.3.1.4)
4. Checks if the kernel lock is currently held. If so, this is a fatal error, as code holding the kernel lock is not allowed to fault. If not, locks the kernel and switches to mode_svc. The current thread's supervisor mode stack is now active
5. Checks that there is enough space remaining on the supervisor stack to store the full processor state. If not, the supervisor stack has overflowed, which is a fatal error
6. Transfers the registers saved on the mode_abt or mode_und stack to the mode_svc stack. Saves the rest of the processor registers on the mode_svc stack. Also saves the fault address register and the fault status register, which give additional information about the cause of an MMU-detected exception such as a page fault
7. Unlocks the kernel. Any deferred IDFCs and/or reschedules will run now
8. Calls the exception handler registered for the current nanothread, passing a pointer to the saved processor state on the stack.

Handling exceptions on IA-32 processors

On IA-32 processors, exceptions automatically switch to privilege level 0 and the current thread's supervisor stack becomes active, as I described in Section 6.2.2. The return address from the exception is either the address of the aborted instruction for an abort or the address of the following instruction for a trap, and the processor automatically saves this on the supervisor stack along with the flag's register. If the exception occurred in user mode, the processor also automatically saves the user-side stack pointer. The processor does not disable interrupts (because Symbian OS uses trap gates for exceptions) and the preamble can be preempted at any time. This is not a problem because the current thread's supervisor stack is active throughout the preamble. The

IA-32 exception preamble proceeds as follows:

1. If the exception does not push an error code onto the stack (see Section 6.2.2), the preamble pushes a zero error code
2. Saves all processor integer registers and segment selectors on the supervisor stack. Also pushes the fault address register; this is only valid for page faults but is nevertheless stored for all exceptions
3. Checks the interrupt nest count. If it is greater than -1, then the exception occurred during an ISR, which we treat as a fatal error
4. Checks if the kernel lock is currently held. If so, this is a fatal error
5. Calls the exception handler registered for the current nanothread, passing a pointer to the saved processor state on the stack

Postamble

If the exception was not fatal, then the postamble runs after the nanothread's exception handler has returned. The postamble restores all of the processor state from the stack, with the exception of the supervisor stack pointer. The processor will restore this automatically by popping the saved state when it returns from the exception. The last instruction of the exception postamble will be a *return from exception* instruction, which restores both the program counter and the status register. Execution then returns to the saved program counter value, which means that either the processor retries the aborted instruction or it executes the instruction after the faulting one. It is worth pointing out that the exception handler may previously have modified the saved state by changing values in the stack, but these modifications only take effect at this point.

Symbian OS exception handler

All Symbian OS threads are provided with a standard exception handler. This implements several strategies for dealing with the exception. These strategies are tried one after another until either one of them successfully handles the exception or they all fail, in which case the thread causing the exception is terminated.

Magic handler

The first strategy is the so-called *magic* handler. This works slightly differently on the ARM and IA-32 versions of the kernel.

On ARM, the exception handler checks for a data abort occurring in mode svc where the address of the aborted instruction is one of a short list known to the exception handler. If these conditions are all satisfied, the magic handler is used; this simply resumes execution at the instruction after the aborted instruction with the Z (zero) flag set, and R12 set to the data address that caused the data abort.

On IA-32, the exception handler checks for an exception occurring at CPL = 0, that is, in supervisor mode, and checks the `iMagicExcHandler` field of the current thread. If the latter is non-null and the exception occurred at CPL = 0, the Symbian OS exception handler calls the function pointed to by `iMagicExcHandler,` passing a pointer to the processor state saved by the exception preamble. If this function returns zero, it means that the magic handler has handled the exception and modified the saved processor state appropriately (typically the saved EIP will be modified), so the Symbian OS exception handler simply returns and execution resumes according to the modified saved state. If the return value is nonzero, the Symbian OS exception handler proceeds to the `TExcTrap` strategy.

The magic handler is used to safely handle exceptions that are caused by dereferencing user-supplied pointers in frequently used kernel functions, such as `<tt style="font-family:monospace;">DThread::RequestComplete()</tt>`, which is used to complete Symbian OS asynchronous requests. The advantage of the magic handler is that it is very fast to set up - in fact on ARM it requires no setup at all and on IA-32 we only need to write a single pointer to the current thread control block. Fast set up is important, since the setup overhead is incurred in the normal case where no exception occurs. The disadvantage is that it can only be used in functions written in assembler because the magic handler must inspect and manipulate saved processor register values directly. In C++, we don't know what register the compiler is going to use for what.

TExcTrap handlers

The second strategy is the use of the `TExcTrap` class. This supports the catching of exceptions in C++ code and allows the handler to be written in C++. The price of this additional flexibility is that it takes longer to set up the `TExcTrap` before executing the code that might cause an exception.

A thread wishing to catch exceptions occurring in a section of code allocates a `TExcTrap` structure on its supervisor stack and initializes it with a pointer to the function that is to be called if an exception occurs. The initialization function saves the processor registers in the `TExcTrap` structure and attaches this structure to the thread. If an exception occurs, the Symbian OS exception handler sees that the thread has installed a `TExcTrap` handler and calls the nominated handler function, passing pointers to the `TExcTrap`, the current thread and to the processor context saved by the nanokernel exception preamble. The handler can then

inspect the saved context and any additional information passed in the `TExcTrap`, and decide to either retry the aborted instruction or to return an error code. In the latter case the processor registers saved in the `TExcTrap` are restored to allow C++ execution to continue correctly.

Coprocessor fault handler

The two strategies for handling exceptions that I have just described allow the catching of exceptions that occur with a fast mutex held. If neither of these methods successfully handles the exception, and the current thread holds a fast mutex, then the kernel will treat this as a fatal error. It does this because all of the exception-handling schemes that I am about to describe make use of fast mutexes.

The next check the kernel makes is for coprocessor faults. On ARM, the check is for *undefined instruction* exceptions on coprocessor instructions; on IA-32 the check is for *device not available* exceptions.

If the exception is of this type, and there is a context switch handler registered for the coprocessor involved, the registered handler is called. (We don't need to make the check on IA-32 as there is always a handler for the IA-32 FPU.) The return value indicates whether the exception has been handled successfully.

If the exception was handled successfully, the kernel restores the processor state and then either retries the coprocessor instruction or continues execution with the next instruction, depending on the reason for the exception. Coprocessor handlers can be used for two different purposes. One is to save and restore the coprocessor state as necessary to enable multiple threads to use the coprocessor. When a new thread attempts to use the coprocessor, an exception results; the exception handler saves the coprocessor state for the previous thread and restores the state for the current thread, and the instruction is then retried so it can execute correctly in the context of the current thread. This scheme is used for the IA-32 FPU and ARM VFP and I will describe it in more detail in Section 6.4.2.2. The other purpose for a coprocessor handler is to emulate a coprocessor that is not actually present. In this case execution will resume after the coprocessor instruction.

Kernel event handlers

The exception is then offered to all the kernel event handlers. The kernel calls each handler, passing a pointer to the processor context saved by the nanokernel exception preamble. The handler has the option to handle the exception, possibly modifying the saved processor context, and then resume the aborted program. Alternatively, it may ignore the exception, in which case it is offered to the next handler.

User-side exception handlers

If no kernel event handlers can deal with the exception, the last possibility for handling it is a user-side exception handler. If the exception occurred in user mode and a user-side exception handler is registered, then we modify the user-mode stack pointer and the return address from the exception to cause the current thread to run the user-side exception handler as soon as it returns to user mode.

If none of the previous methods are able to handle the exception, the current thread is terminated with the *KERN-EXEC 3* panic code.

Uses of exceptions

Trapping invalid pointers

Since Symbian OS does not support demand-paged virtual memory, any occurrence of a page fault must come from the use of an invalid memory pointer. In most cases this will result in the kernel terminating the thread that caused the page fault. Exceptions to this rule are:

- If the invalid pointer was passed in by other code, such as a server receiving a pointer from its client and using that pointer in an `<tt style="font-family:monospace;">RMessagePtr2::Read()</tt>` or `Write()` call. In this case the exception is caught within the kernel and an error code is returned to the server
- If the thread has set up a user-side exception handler to catch page faults.

Coprocessor lazy context switch

IA-32 and some ARM CPUs have floating point coprocessors that contain a substantial amount of extra register state. For example, the ARM vector floating point (VFP) processor contains 32 words of additional registers.

Naturally, these additional registers need to be part of the state of each thread so that more than one thread may use the coprocessor and each thread will behave as if it had exclusive access.

In practice, most threads do not use the coprocessor and so we want to avoid paying the penalty of saving the coprocessor registers on every context switch. We do this by using *lazy* context switching. This relies on there being a simple method of

disabling the coprocessor; any operation on a disabled coprocessor results in an exception. Both the IA-32 and ARM processor have such mechanisms:

- IA-32 has a flag (TS) in the CR0 control register which, when set, causes any FPU operations to raise a *Device Not Available* exception. The CR0 register is saved and restored as part of the normal thread context
- The ARM VFP has an enable bit in its FPEXC control register. When the enable bit is clear, any VFP operation causes an undefined instruction exception. The FPEXC register is saved and restored as part of the normal thread context
- Architecture 6 and some architecture 5 ARM devices also have a coprocessor access register (CAR). This register selectively enables and disables each of the 15 possible ARM coprocessors (other than CP15 which is always accessible). This allows the lazy context switch scheme to be used for all ARM coprocessors. If it exists, the CAR is saved and restored as part of the normal thread context.

The lazy context-switching scheme works as follows. Each thread starts off with no access to the coprocessor; that is, the coprocessor is disabled whenever the thread runs.

When a thread, HAIKU, attempts to use the coprocessor, an exception is raised. The exception handler checks if another thread, SONNET, currently has access to (*owns*) the coprocessor. If so, the handler saves the current coprocessor state in SONNET's control block and then modifies SONNET's saved state so that the coprocessor will be disabled when SONNET next runs. If there wasn't a thread using the coprocessor, then the handler doesn't need to save the state of the coprocessor.

Then coprocessor access is enabled for the current thread, HAIKU, and the handler restores the coprocessor state from HAIKU's control block - this is the state at the point when HAIKU last used the coprocessor. If this is the first time HAIKU has used the coprocessor, a standard initial coprocessor state will have been stored in HAIKU's control block when HAIKU was created, and this standard state will be loaded into the coprocessor. HAIKU now owns the coprocessor.

The exception handler then returns, and the processor retries the original coprocessor instruction. This now succeeds because the coprocessor is enabled.

If a thread terminates while owning the coprocessor, the kernel marks the coprocessor as no longer being owned by any thread.

This scheme ensures that the kernel only saves and restores the coprocessor state when necessary. If, as is quite likely, the coprocessor is only used by one thread, then its state is never saved. (Of course, if the coprocessor were to be placed into a low power mode that caused it to lose state, the state would have to be saved before doing so and restored when the coprocessor was placed back into normal operating mode. However at the time of writing no coprocessors have such a low-power mode.)

Debugging

Exceptions are used in debugging to set software breakpoints. The debugger replaces the instruction at which the user wants to place a breakpoint with an undefined instruction. When control flow reaches that point, an exception occurs and the debugger gains control. Registers may be inspected and/or modified and then execution resumes after the undefined instruction. The debugger must somehow arrange for the replaced instruction to be executed, possibly by software emulation, before resuming execution. There is more on this subject in Chapter 14, *Kernel-Side Debug*.

APIs for exceptions

In the following sections, I will describe the kernel exception APIs that are available to device drivers and extensions.

The XTRAP macro

This is a macro wrapper over the `TExcTrap` handlers that I described in Section 6.4.1.3. The macro is used as follows:

```
XTRAP(result, handler, statements);
XTRAPD(result, handler, statements);
```

The specified statements are executed under a `TExcTrap` harness. The parameter result is an integer variable that will contain the value `KErrNone` after execution if no exception occurred. The macro XTRAPD declares the variable result whereas XTRAP uses a preexisting variable. The parameter handler is a pointer to a function with signature:

```
void (*TExcTrapHandler)(TExcTrap* aX, DThread* aThread, TAny* aContext);
```

This function is called if an exception occurs during the execution of statements.

If XT_DEFAULT is specified as the handler parameter, a default handler is used that returns an error code `KErrBadDescriptor` on

any exception.

Parameter aX points to the `TExcTrap` harness which caught the exception, aThread points to the control block of the executing thread and aContext points to a processor dependent structure which contains the values of all the processor registers at the point where the exception occurred. In fact this is simply the processor state information saved in the exception preamble by the nanokernel. The ARM version of this is:

```
struct TArmExcInfo
  {
  TArmReg iCpsr;
  TInt iExcCode;
  TArmReg iR13Svc; // supervisor stack pointer
  TArmReg iR4;
  TArmReg iR5;
  TArmReg iR6;
  TArmReg iR7;
  TArmReg iR8;
  TArmReg iR9;
  TArmReg iR10;
  TArmReg iR11;
  TArmReg iR14Svc; // supervisor mode LR
  TArmReg iFaultAddress; // value of MMU FAR
  TArmReg iFaultStatus; // value of MMU FSR
  TArmReg iSpsrSvc; // supervisor mode SPSR
  TArmReg iR13; // user stack pointer
  TArmReg iR14; // user mode LR
  TArmReg iR0;
  TArmReg iR1;
  TArmReg iR2;
  TArmReg iR3;
  TArmReg iR12;
  TArmReg iR15; // address of aborted instruction
  };
```

If the exception can be handled, the function should call:

```
aX->Exception(errorcode);
```

This will cause the execution of the XTRAP macro to terminate immediately without completing execution of statements; the results variable is set to the errorcode passed in to the call.

If the exception cannot be handled, the function should just return. The other exception handling strategies described in Section 6.4.1.3 will then be attempted.

The XTRAP macro is used to catch exceptions occurring in supervisor mode, typically in conjunction with the kumemget() and kumemput() functions (described in Section 5.2.1.5) to access user-side memory from places where it would not be acceptable to terminate the current thread on an exception. Examples of these are code that runs with a fast mutex held or inside a thread critical section. The XTRAP macro is the only way to catch exceptions that occur with a fast mutex held.

Kernel event handlers

XTRAP handlers can only catch supervisor-mode exceptions in one thread, and are normally used to catch exceptions within a single function call. We use kernel event handlers when we want to catch exceptions occurring in multiple threads or in user-mode over extended periods of time. We implement kernel event handlers using the class `DKernelEventHandler`, the public interface of which follows:

```
class DKernelEventHandler : public DBase
```

```
  {
public:
  // Values used to select where to insert the handler in the queue
  enum TAddPolicy
    {
    EAppend,
    };
  enum TReturnCode
    {
    // Run next handler if set,
    // ignore remaining handlers if cleared
    ERunNext = 1,
    // Available for EEventUserTrace only.
    // Ignore trace statement if set.
    ETraceHandled = 0x40000000,
    // Available for hardware exceptions only.
    // Do not panic thread if set.
    EExcHandled = 0x80000000,
    };

/* Pointer to C callback function called when an event occurs.
  aEvent designates what event is dispatched.
  a1 and a2 are event-specific.
  aPrivateData is specified when the handler is created,
  typically a pointer to the event handler.
  The function is always called in thread critical section. */

  typedef TUint (*TCallback)(TKernelEvent aEvent, TAny* a1, TAny* a2, TAny* aP);
public:
  // external interface
  IMPORT_C static TBool DebugSupportEnabled();
  IMPORT_C DKernelEventHandler(TCallback aCb, TAny* aP);
  IMPORT_C TInt Add(TAddPolicy aPolicy = EAppend);
  IMPORT_C TInt Close();
  inline TBool IsQueued() const;
  };
```

If you are writing an extension or a device driver and you want to use a kernel event handler, follow these steps. First instantiate the `DKernelEventHandler` class on the kernel heap. This requires two parameters - aCb is a pointer to a function to be called back when any notifiable event occurs, and aP is an arbitrary cookie which is supplied as an argument to aCb when notifying an event.

After instantiating the class, call `Add()` on it to start receiving event callbacks. From this point on, whenever a notifiable event occurs, the kernel will call the specified function, aCb. It calls the function in the context of the thread that caused the event, with the thread itself in a critical section. The aEvent parameter to the callback indicates the event the callback relates to; the value `EEventHwExc` indicates a processor exception. For processor exception callbacks, parameter a1 points to the saved processor state (for example, `TArmExcInfo` which I mentioned previously) and parameter a2 is not used. The return value from the handler function will take one of the following values:

- `DKernelEventHandler::EExcHandled` if the exception has been handled and normal program execution should be resumed
- `DKernelEventHandler::ERunNext` if the exception has not been handled and the next kernel event handler (if any) should be run.

ARM coprocessor handlers

On ARM-based hardware, Symbian OS provides additional APIs to support lazy context switching and software support for coprocessors. ARM systems may have several coprocessors; examples of these are the Vector Floating Point (VFP), DSP and motion estimation units.

Here is the coprocessor API:

```
const TInt KMaxCoprocessors=16;

enum TCpOperation
  {
  EArmCp_Exc, /* UNDEF exc executing a coproc instr */
  EArmCp_ThreadExit, /* Coproc current owning thread exited */
  EArmCp_ContextInit /* Initialise coprocessor */
  };

struct SCpInfo;
typedef TInt (*TCpHandler)(SCpInfo*, TInt, TAny*);

struct SCpInfo
  {
  TCpHandler iHandler; /*Hdler: context switch, init & thread exit*/
  NThread* iThread; /*Current owning thread, NULL if none */
  TUint16 iContextSize; /* context size for this coprocr */
  TInt8 iCpRemap; /* Coproc no to remap this one to if >=0 */
  TUint8 iSpare; TInt iContextOffset; /* Offset in thread extra context */
  };

class Arm
  {
  ...
public:
  static void SetCpInfo(TInt aCpNum, const SCpInfo* aInfo);
  static void SetStaticCpContextSize(TInt aSize);
  static void AllocExtraContext(TInt aRequiredSize);
  static TUint32 Car();
  static TUint32 ModifyCar(TUint32 aClearMask, TUint32 aSetMask);
  static TUint32 FpExc();
  static TUint32 ModifyFpExc(TUint32 aClearMask, TUint32 aSetMask);
  ...
  };
```

The code responsible for handling the coprocessor (which may be in the kernel or in the ASSP, the variant or an extension) should call the following function during the Init1 initialization phase (see Chapter 16, *Boot Processes*, for a detailed description of the system initialization phases):

```
Arm::SetCpInfo(TInt aCpNum, const SCpInfo* aInfo);
```

Parameter aCpNum specifies the number of the coprocessor whose handler is being defined. Parameter aInfo specifies information about the coprocessor, as follows:

- `SCpInfo::iHandler` specifies the function that should be called at system boot if a thread exits or if an undefined instruction exception occurs trying to access the coprocessor. The second parameter passed to the function specifies which of these events has occurred, specified by the `TCpOperation` enumeration
- `SCpInfo::iThread` specifies which thread owns the coprocessor. On initialization this should be NULL
- `SCpInfo::iContextSize` specifies the size in bytes of the per-thread context for this coprocessor
- `SCpInfo::iCpRemap` specifies whether this coprocessor is really part of another coprocessor. If set to zero or a positive value, this specifies the number of the primary coprocessor. All events for this coprocessor will be redirected to the primary coprocessor. A value of -1 indicates that no redirection is required
- `SCpInfo::iContextOffset` is calculated by the kernel and need not be set. When the kernel calls back the handler function specified in `SCpInfo::iHandler` it passes the following parameters:

```
TInt h(SCpInfo* aInfo, TInt aOp, TAny* aContext);
```

If aOp == `EArmCp_ContextInit`, the system is in the Init2 phase of boot. The handler should initialize the coprocessor and then save its state to the memory area pointed to by aContext. It should then disable access to the coprocessor.

If aOp == `EArmCp_ThreadExit`, a thread is exiting. The handler is called in the context of the exiting thread. If the exiting thread currently owns the coprocessor, the handler should mark the coprocessor as unowned, so that subsequent accesses do not try to save the state to a thread control block that no longer exists.

If aOp == `EArmCp_Exc`, an undefined instruction exception has occurred on an instruction targeted at this coprocessor. aContext points to the register state saved by the nanokernel exception preamble. The kernel calls the handler in the context of the thread attempting the instruction. If this is not the coprocessor's current owning thread, the handler should save the coprocessor state for the current owning thread and then disable the coprocessor for that thread. It should then enable the coprocessor for the current thread and restore the current thread's coprocessor state. If the exception is successfully handled the handler should return KErrNone, otherwise `KErrGeneral`.

The following functions are used to modify the access permissions for a coprocessor:

```
Arm::ModifyCar(TUint32 aClear, TUint32 aSet);
Arm::ModifyFpExc(TUint32 aClear, TUint32 aSet);
```

The functions return the original value of the register that they modify. Access to the VFP is controlled by the FPEXC register; access to other coprocessors is controlled by the CAR. Since these functions modify the hardware registers directly, they affect the current thread. To modify coprocessor access for another thread, the corresponding functions are used:

```
NThread::ModifyCar(TUint32 aClear, TUint32 aSet);
NThread::ModifyFpExc(TUint32 aClear, TUint32 aSet);
```

These do not modify the hardware registers - instead they modify the copy of those registers saved in the thread's control block.

Exceptions in the emulator

The emulator installs an exception handler on its Windows threads so that it can detect and handle exceptions occurring in Windows on that thread. This is similar to the data abort exception vector in the ARM code on a phone.

If the kernel is locked when an exception occurs the system halts, as this is a fatal error.

If the kernel is unlocked, the first action on taking an exception is to lock it. Next, we have to deal with the effect of a race condition between exception handling and forced exit - otherwise it is possible that a thread that has been killed will not actually die!

The exception handler then records the exception information and causes the thread to *return* to a second exception handler once the Windows exception mechanism has unwound. Running the second stage outside of the Windows exception handler allows the nanokernel to be in better control of the thread context; in particular it allows for the thread to be panicked and so on. The kernel remains locked through this process, so the saved exception data cannot be overwritten.

The second stage saves the full thread context including the original instruction pointer (this allows debuggers to display a complete call stack), then unlocks the kernel and invokes the nanokernel thread's exception handler. This handler can supply a final *user-mode* exception handler and parameters which is invoked before returning to the original exception location.

Summary

In this chapter, I've described interrupts and exceptions, and looked at their causes and the way in which processors react to them. I've shown how an operating system, in particular Symbian OS, makes use of exceptions, and I've gone on to examine how EKA2 handles them in detail. Finally, I have discussed the APIs that are available to allow you to make use of exceptions in base ports and device drivers.

The management of CPU resources is one of the key tasks of an operating system. The other is the management of memory, and it is this that I shall discuss in the next chapter, Memory Models.