

# 1. Overview of Symbian OS Hardware Interrupt Handling

John Pagonis

Revision 1.0, March 2004

## 1.1. Introduction

Symbian OS has a lightweight 32-bit pre-emptive kernel that follows a hybrid design combining characteristics from both micro-kernel and monolithic kernel architectures. Symbian OS design focuses on open mobile phones and thus is optimized for efficient resource constrained operation. In this article we will discuss hardware interrupt handling as exists in the kernel architecture for all versions of the operating system up to and including v7.0s. In Version 8.x, the architecture in this area is different and subject to future articles

From micro-kernel architectures, the Symbian OS kernel borrows the following characteristics:

- A message-passing framework designed for the benefit of user-side servers.
- Networking and telephony stacks hosted in user-side servers.
- File systems hosted in a user-side server.

Like in monolithic kernel architectures, similarly in the Symbian OS kernel:

- Device drivers are run kernel-side. They are not embedded in the kernel binary though, but are implemented as libraries that can be loaded and unloaded at runtime.
- The scheduler and scheduling policy is implemented in the kernel.
- Memory management and the memory manager are implemented in the kernel.

In general the kernel deals with core hardware resources such as:

- CPU and MMU.
- Memory management.
- Interrupt handling and management.
- DMA channel management.

For the benefit of user-side application development the kernel implements a set of abstractions that are offered through their associated APIs in a user library; such abstractions are for:

- Process control
- Thread control
- Memory chunks
- Device driver framework
- DLL manipulation
- Mutex control
- Semaphore control
- Client-server framework
- Asynchronous message passing
- Timers
- Date and time management
- Internationalization
- Event management
- Password management

Furthermore the kernel offers a set of APIs for kernel-side inclusion such as kernel module extensions. For example the kernel can be extended by the use of DLLs that can link dynamically against it like device drivers do.

In general, the main characteristics of device drivers, in Symbian OS, are that:

- User-side code can load and unload device drivers at runtime.
- User-side code can call synchronous and asynchronous functions in drivers.
- The device driver framework distinguishes between logical device drivers (LDD) and physical device drivers (PDD). LDDs are generally hardware-independent and can drive a whole class of PDDs.

## 1.2. Threading Model

Symbian OS is a pre-emptive multitasking operating system that runs (the kernel) in privileged (ARM SVC) mode while all other tasks run in non-privileged mode. Therefore in Symbian OS all access to memory and memory mapped hardware is protected, hence the use of the MMU by the operating system. Of course, the kernel can access all the memory belonging to any other program, whereas a non-privileged program can only access, directly, memory allocated to it.

In Symbian OS the unit of this memory protection is called the '*process*' while the unit of execution is the '*thread*'. A process may contain one or more threads, which in the case of the kernel, are the '*Kernel Server*' and '*Null*' thread. What gets scheduled in Symbian OS each time, are the threads (not processes, which are effectively memory protection containers).

In Symbian OS the Kernel Server exists to service requests (sent via messages) by user threads and is the highest priority thread in the system. In the same process the Null thread, which is the thread with the lowest priority, is responsible on startup for loading the file server and booting the kernel; then during normal execution for calling on to the power handling subsystem services in order to put the system into various power saving and sleeping modes.

As we have discussed previously [1] when user-side threads need to get privileged access or request kernel-side services, they go through a pre-programmed path, implemented in terms of a library.

Memory management makes sure that Symbian OS presents a *virtual machine model* to all running programs. This means that all programs are presented and make use of a linear virtual memory environment facilitated by the use of the MMU. Virtual memory in this context does not mean that Symbian OS makes use of program swapping (e.g., hard disks and demand paging), but that all programs during linking, locating and execution appear at the same virtual address; and that they cannot access each other's memory.

### **Symbian OS device driver architecture**

Only the kernel can access hardware directly and device drivers are used to provide user-side code with a mechanism to access hardware services.

A device driver is effectively an add-on to the kernel, implementing polymorphic interfaces defined by the kernel. It resides on the kernel side and therefore has the same access rights, uses the kernel heap and links to the kernel so that it can call kernel functions.

Due to Symbian OS pre-emptive scheduling, there is no known context when an interrupt occurs. In the device driver architecture, the Interrupt Service Routine (ISR) called at interrupt time can schedule a Delayed Function Call (DFC) that runs when the kernel is in a known state. The ISR/DFC mechanism

allows the device driver developer to choose where to perform specific tasks in order to minimize the thread latency response to hardware.

### Device driver structure

A device driver includes:

- A user-side API.
- Usually two kernel-side DLLs, each of which exports a single factory function at ordinal 1 to create the necessary kernel-side objects:
  - A logical device driver DLL (with filename extension .LDD), for use by the user-side part. This is hardware-independent. This does not need changing when porting the base.
  - A physical device driver DLL (with filename extension .PDD). This is hardware-dependent and is kept as thin as possible to minimize the base-porting effort. If no hardware-specific code is required, the whole kernel-side implementation can be supplied in the logical device driver (which may access the hardware directly despite its name).

Note that device manufacturers may choose to access hardware without the use of device drivers. Kernel extension DLLs can be provisionally linked to the kernel as needed per specific ROM. An example of this is the built-in keyboard driver. These are executed during kernel boot.

## 1.3. Hardware Interrupt Handling in Symbian OS

An interrupt is a signal that is asserted on a hardware interrupt pin available from the processor in order to notify the CPU, that an external device demands its attention. In general the number of available interrupt signals is determined by the CPU and hardware designs.

Systems that have asynchronous I/O make use of interrupt handling routines that exist to service each possible interrupt signal, such routines are usually referred to as ISRs or interrupt service routines. Addresses of ISRs are usually contained in an interrupt vector table so that the OS, following a principle similar to the one discussed previously [1], can determine the appropriate routine which is programmed to deal with a particular device.

Oftentimes there are more hardware interrupt sources than available CPU interrupt request lines [2]. This means that several interrupt sources need to share a single interrupt request signal line (processor pin) – in the case of the ARM processor there are two lines, the FIQ and IRQ line. A system therefore needs to determine which interrupt source caused the interrupt and dispatch the relevant handling routine accordingly.

Symbian OS has been designed to cater for such cases where a hardware design needs to reuse an interrupt line between multiple source devices.

To achieve this, Symbian OS makes use of *interrupt chaining* where the ISRs that correspond to the same interrupt signal are chained together to form a single linked list of ISRs. When an interrupt occurs, the interrupt service routine provided by each interrupt service object in the list is called in the order in which it was chained. Again, remember that, Symbian OS does not allow for nested interrupts, nor does it make use of interrupt priorities. Interrupt service routines are therefore called in the order in which they were linked. At the same time, the Symbian OS interrupt handling framework prevents from assigning an ISR to multiple signals (source devices).

An ISR runs on the kernel side, while the context of the system is unknown to the ISR.

At the point of the interrupt, the state of the kernel is undefined, which imposes restrictions on what can be done in the service routine. For example, the kernel may be half-way through a reschedule, in which case access to the process list would be invalid and dangerous.

In Symbian OS, because ISRs have no priorities and nested interrupts are not permitted, ISRs have to be very short in order to avoid blocking other interrupts for too long. For that reason, Deferred Function Calls (DFCs) are used instead to do any extended processing. Although it is important that DFCs are themselves quick, nevertheless they operate with both IRQs and FIQs enabled; which means that they can be interrupted by some other ISR.

### **What happens when an interrupt occurs?**

When the interrupt request signal is asserted, the ARM processor enters IRQ mode and branches to the interrupt handler (pointed from address 0x18 of the ARM exceptions vector table – see [2]). Then, the interrupt handler immediately does some house keeping and looks to discover the source of the interrupt by checking the specific hardware's interrupt controller register for pending interrupts that have not been masked out.

This is done by linearly checking every bit in that register that corresponds to an interrupt signal source. For every pending IRQ source found, the interrupt handler dispatches to the ISR(s), by looking into an interrupt vector table. In fact, there usually lies a pointer to a service chain of ISRs; which are offered the id of the source of the IRQ in a FIFO manner. Prior to offering the IRQ id to the ISRs, the system checks if they are enabled at all. Finally upon completion, the end of the interrupt handling is signaled to the hardware.

### **What are the responsibilities and constraints of an ISR?**

Any ISR will be responsible to:

- ✓ Check whether the interrupt source that it is specifically responsible for servicing has, in fact, a pending interrupt. This is necessary when several interrupt sources share an interrupt signal.
- ✓ Clear the interrupt bit in the interrupt controller register.
- ✓ Acknowledge to the device that its interrupt request has been received (note: some devices need this, others don't).
- ✓ Do any necessary I/O.
- ✓ Queue a DFC to continue processing any data if necessary.

As already stated when an interrupt happens, the state of the kernel is undefined, which imposes restrictions on what can be done in the ISR. While a service routine can access the kernel heap and can, therefore, access certain kernel member variables and any memory areas previously allocated, it cannot:

- ❖ Allocate or free memory.
- ❖ Read from or write to user memory space.
- ❖ Signal a thread, this must be done by a DFC (because otherwise it would interfere with the task list of ready-to-run threads while the kernel could be in the process of rescheduling just before the interrupt).

## What are the responsibilities of a DFC?

Because nesting of interrupts is not permitted, any interrupt signal must be fully handled before other interrupt signals can be serviced and in order to perform processing that would otherwise be impossible or inappropriate inside the service routine, ISRs (in most cases) need to queue DFCs.

Because the kernel is guaranteed to be in a known state prior to scheduling any DFCs, it means that a DFC can:

- ✓ Call general kernel functions.
- ✓ Signal a thread (the kernel now is in a known state and the task list of ready-to-run threads can be indirectly altered).
- ✓ Access any previously allocated memory and existing data structures.

Again, like ISRs, DFCs cannot allocate or free memory (on the kernel heap).

As stated previously, during the execution of a DFC, interrupts (IRQs and FIQs) are enabled so that execution time is not as critical as the execution time of an ISR; and interrupt latency is kept to a minimal. Nevertheless, it is still important to keep processing time short because control cannot return to user threads until all DFCs have run. This is because DFCs are scheduled after all ISRs have been called, but just before the kernel reschedules any user threads.

## When do ISRs get installed and deleted?

Symbian OS interrupt architecture allows for multiple interrupt service routines to be bound to an interrupt signal. This makes shared interrupt lines easier to handle.

Interrupt service routines can be added and removed dynamically at runtime; this allows device drivers to add and remove ISRs when they are loaded or unloaded. Furthermore, interrupt service routines can be dynamically enabled and disabled.

Simply, if a service routine is disabled, it is not called when the interrupt signal to which it is bound, occurs.

DFCs are normally allocated early when a device driver is loaded. Adding them to the kernel's queue of DFCs or removing them from the queue, which simply involves manipulating pointers, requires no further memory allocation or de-allocation.

As with ISRs, Symbian OS imposes no limit to the number of DFCs which can be queued.

## References:

[1] Crossing the Userland, John Pagonis, March 2003.

[http://www.symbian.com/developer/techlib/papers/cpp\\_sysarch.html#userland](http://www.symbian.com/developer/techlib/papers/cpp_sysarch.html#userland)

[2] ARM System on Chip Architecture – 2<sup>nd</sup> ed., Steve Furber, Addison-Wesley. ISBN: 0201675196

Want to be kept informed of new articles being made available on the Symbian Developer Network?

[Subscribe to the Symbian Community Newsletter.](#)

The Symbian Community Newsletter brings you, every month, the latest news and resources for Symbian OS.