The essence of any computer system is to accept input data, process it, and produce output. How the system accepts input data is critical for real-time embedded systems. Because it needs to process the input data and output a response within a determined time interval. The most accurate way to achieve this would be to poll an input source constantly for data. This however puts a heavy burden on the CPU, limiting the system to just one task. Waiting for input data to arrive and react on it alleviates this burden and frees CPU resources. To be able to wait and react, we need hardware support for getting CPU attention away from its flow of processing.

Hardware triggered interrupts cause the CPU to stop the flow of execution and jump to a specific entry in the code that handles the interrupt. The arrival of an interrupt is not predictable; therefore it can occur at any time even when another interrupt is being handled. Once the interrupt was processed the CPU resumes from the point it stopped.

In this chapter

- Interrupt model
- Interrupt processing
- I/O memory mapping

Interrupt Model

I/O hardware triggers an interrupt whenever it needs the CPU’s attention, because its I/O registers changed state. Interrupts may be categorized into maskable, non-maskable, inter-processor, and software interrupts.

- Maskable interrupt – hardware interrupt that may be ignored if masked by setting a specific bit in the Interrupt Mask Register (IMR)
- Non-Maskable interrupt (NMI) – hardware interrupt that cannot be ignored
• Inter-Processor interrupt (IPI) – an interrupt triggered by one processor to another in a multiprocessor environment. Mainly used to implement cache coherence. In Windows Embedded Compact 7 it has become relevant as it supports multicore processors.

• Software interrupt – an exception triggered by some CPU instruction. Used as a trap to execute system calls since the CPU switches to kernel mode.

Hardware interrupts may be edge triggered or level triggered. As with everything in life there are pros and cons to each of these. Edge triggered interrupts occur when there is a level transition of a pulse on the interrupt request line, a rising edge trigger occurs when the transition is from low (0) to high (1), and falling edge trigger occurs when the transition is from high (1), to low (0). The major failing of edge triggered interrupts is that noise may be misinterpreted or interrupts may be missed altogether. Modern hardware really solves this problem using interrupt status registers. These should be scanned by well-implemented device drivers to check that there are no missed interrupts. Level triggered interrupts transition the interrupt request line to an active state (maybe high or low) and keep it until it receives a command to go inactive. The main failing for level triggered interrupts is handling of shared interrupt request line. If a lower level request is kept level active, it is a problem to detect a request from another I/O device even if it has a higher request level.

Windows Embedded Compact 7 and its predecessors need to provide OS services to handle these interrupts to allow the operating system itself and processes to perform their purpose. A physical interrupt is a signal arriving over a hardware line connected to the CPU, and the OEM Adaptation Layer (OAL) implementation maps hardware assigned interrupts to logical interrupts understood by the operating system to allow interrupt handling code to service correctly each interrupt.

**Interrupt Architecture**

The interrupt model for Windows Embedded Compact 7 is that all interrupts get vectored into the kernel. Windows Embedded Compact 7 interrupt architecture tries to balance simplicity with performance by splitting the interrupt processing into an extremely fast (and “lightweight”) Interrupt Service Routine (ISR) and a device driver provided thread that processes the bulk of the interrupt processing. Because extended processing in an ISR delays the servicing of other interrupts and may result in jitter the ISR does very little but acknowledges the interrupt and differs most of the interrupt processing to the IST. Because the IST is scheduled like any other thread in the system it is assigned a scheduling priority fitting the priority of the interrupt that it serves. This greatly improves real-time performance by reducing jitter and simplifies locking and scheduling by the kernel.

The kernel component that handles all interrupts is the exception handler. When an interrupt occurs, the CPU transfers control to the kernel exception handler. The exception handler in turn calls the ISR which is registered to handle the current interrupt. The ISR should return the interrupts logical interrupt identifier, which it passes to the kernel as its return value. The kernel sets an event associated with the logical interrupt, which causes an interrupt service thread (IST) to be scheduled. Code in the IST is responsible for servicing the device interrupt. The IST runs in the context of a thread in Device Manager and is essentially a typical thread running at a high priority.

**Interrupt Processing**

The diagram in Figure 8-1 shows the sequence of state transitions on a timeline going from right to left. Starting with the arrival of the hardware triggered interrupt all the way to completion of the interrupt