Handling External Events

Frequently a computer system has to respond to external events. A simple and crude mechanism for doing this is for the processor to execute a loop in which it continually tests for the occurrence of the event. This method – usually termed polling – is very inefficient since the processor does little other than wait for the external event. The alternative is for the processor to perform useful work inside the polling loop – but then there is a delay in responding to the event; if the event is a dangerous error condition such a delay could be undesirable.

The problems associated with handling an external event get even worse if that event is rather rare, which is often the case. A more elegant solution is needed other than polling.

1 The Basic Interrupt Mechanism

An interrupt facility is a hardware extension to the CPU which provides an external interrupt input (INT) to the processor. The external event is 'connected' to the INT input: when INT goes high the processor responds to the event[1]

In practice, the INT input is sampled at the end of every fetch/execute cycle. Recall that a processor continually cycles through the phases of fetching an instruction (op code), executing that instruction and then fetching the next instruction. At the point when the processor has completed execution of the current instruction, it examines the INT input which forms an input to the finite-state machine of the control path. If the INT input is not set, the processor carries on to fetch the next instruction. If INT is set then the processor executes an *interrupt service routine* (ISR) or handler which is a special subroutine to handle the interrupting event. ISRs will be discussed in greater detail below but two important points should be made about the timing of the basic interrupt mechanism:

- The processor handles the interrupt only at one precisely defined point in the fetch-execute cycle – after execution of the current instruction has been completed. This means the processor can safely branch-off to deal with interrupt and will not be left in some indeterminate state.
- Following on from the above, processing of an interrupt is not completely asynchronous; in fact, it could be considered as a refined version of polling. In naive polling, the programmer writes an explicit loop which tests the state of some external event. But in the interrupt mechanism, the INT input is tested (at the end of) every single instruction. We are thus freed from the dilemma of trying to do useful work inside the testing loop, and if so how much? The processor's latency in responding to an interrupt is bounded from above by the duration of the longest instruction: even if an interrupt arrives just after the start of the execution of the longest possible instruction, we know the interrupt will be handled before the next instruction is executed.

2 The Interrupt Service Routine (ISR)

When a processor handles an interrupt it calls an interrupt service routine (ISR) which is a special kind of subroutine which is located as some fixed point in the processor's memory set aside for the ISR. As with a conventional subroutine `CALL`, the contents of the program counter (PC) will be saved to the stack in some minimal prolog (although there is no explicit `CALL` instruction). The microinstructions to push the PC and transfer control to the ISR are embedded in the control path.

On completion, the ISR will typically execute a `RETI` (return from interrupt) instruction which only occurs in an ISR. `RETI` will pop the old PC from the stack allowing the processor to resume execution at the point where the interrupt was called. The thing that distinguishes `RETI` from the return instruction found at the end of a normal subroutine is that `RETI` carries out additional housekeeping such as clearing the latch at the INT

[1] In practice, the INT input is latched so the timing constraints on the interrupt can be quite relaxed and the processor will not 'miss' an event.

input, thus preparing the processor to deal with another interrupt.

3 Dealing with Multiple Interrupts

The basic interrupt mechanism described above works well for single, isolated interrupt events. In practice, we may face the situation where either several interrupts occur at the same time or an interrupt may occur when the processor is in the middle of handling an earlier interrupt. The key to dealing with these complications is to recognise that some interrupts will be more important than others and so should be given a higher priority.

If multiple interrupts occur simultaneously then we need to sort them, deal with the highest priority interrupt first and queue the rest for processing at a later time.

An interrupt occurring when we are already processing an earlier interrupt is a little more complicated. Clearly if the new interrupt has a lower priority than the one we are already handling then we should just queue the new interrupt. If, on the other hand, the new interrupt is of higher priority then we should handle it without delay – that is, is interrupt the currently executing ISR.

If we are going to allow different types of interrupt, then the basic idea of a single ISR needs modification – really we need each interrupt to have its own ISR which complicates matters considerably. The additional hardware needed to deal with multiple interrupts is usually provided by another chip, a *programmable interrupt controller* (PIC[2]) – the details of such devices are outside the scope of this introductory course.

4 Non-maskable Interrupts

One of the instructions provided in most processors is the ability to disable interrupts[3]. This is sometimes desirable. There are, however, events which it is undesirable to ignore – such as impending power failure or other catastrophic errors – and to deal with these processors often have separate non-maskable interrupt (NMI) input in addition to the normal INT input. An NMI usually has its own distinct ISR at some other location from the main ISR.

5 Software Interrupts

Many processors offer the facility to call an ISR from software by executing an instruction of the form:

```
INT m
```

where `m` is a reference to the entry point of the relevant ISR. The reasons for being able to call an ISR from software are twofold: Firstly, some actions need to be taken both in response to either an external event or as part of normal program flow. Being able to call an ISR directly from software avoids code duplication and gives greater flexibility over a `CALL` instruction.

Secondly, the mechanism gives a means of testing/debugging ISRs without the need to generate external hardware events.

6 Problems with Interrupts

Although interrupts are a very powerful way of dealing with events and are very widely used, particularly in operating systems, they are not without their problems.

[2]Not to be confused with the popular PIC family of microcontrollers!
[3]Together, obviously, with an instruction to re-enable them again!

Interrupt *latency* is a particular issue in real-time systems, especially the fact that the time needed to respond to multiple, prioritised interrupts is often impossible to predict.

Further to this, testing all the possible combinations and timings of interrupts in a system is usually impractical. There is always the concern that a timing difference of a few nanoseconds may result in undesired or unintended behaviour.

Despite all their drawbacks, interrupts are very useful although many people regard them as inherently unsafe.