Understanding Interrupts

by Russell Massey

Normal execution of a given software application is contained within the bounds of one program, or instruction stream. Such execution is provable, as well as traceable. However, system designers and implementers also have to understand how breaks in program flow occur, and how they may affect the running program. Flow breaks fall into two general classes:

- *Exceptions* and *traps* are predictable, synchronous breaks in program flow. They are synchronous because they are caused by the execution of certain instructions (divide by zero; illegal memory access; software interrupt). Exceptions and traps trigger execution of instructions that are not part of the program, but perform work on behalf of it.
- *Interrupts* are asynchronous breaks in program flow that occur as a result of events outside the running program. They are usually hardware related, stemming from events such as a button press, timer expiration, or completion of a data transfer. We can see from these examples that interrupt conditions are independent of particular instructions; they can happen at any time. Interrupts trigger execution of instructions that perform work on behalf of the system, but not necessarily the current program.

This article explains how an interrupt is handled by the processor and software.

Why interrupt?

As a system's functional requirements and the size of the software grow, it becomes more difficult to ensure that time-critical items (such as capturing incoming data before it is overwritten by the hardware) are performed properly. We can approach this dilemma with a faster processor (more cost, more heat, and more radiated noise), or we can separate the time-critical functions from the others and execute them in a prioritized manner. Interrupts form the basis for this separation. The non-time-critical functions continue to execute as quickly as they can (within the main loop), but time-critical functions are executed on demand—in response to interrupts from the hardware.

Hardware

When a device asserts its interrupt request signal, it must be processed in an orderly fashion. All CPUs, and many devices, have some mechanism for enabling/disabling interrupt recognition and processing:

- At the device level, there is usually an interrupt control register with bits to enable or disable the interrupts that device can generate.
- At the CPU level, a global mechanism functions to inhibit/enable (often called the *global interrupt enable*) recognition of interrupts.
- Systems with multiple interrupt inputs provide the ability to mask (inhibit) interrupt requests individually and/or on a priority basis. This capability may be built into the CPU or provided by an external *interrupt controller*. Typically, there are one or more *interrupt mask registers*, with individual bits allowing or inhibiting individual interrupt sources.
- There is often also one *non-maskable interrupt* input to the CPU that is used to signal important conditions such as pending power fail, reset button pressed, or watchdog timer expiration.

Figure 1 shows an interrupt controller, two devices capable of producing interrupts, a processor, and the interrupt-related paths among them. The interrupt controller multiplexes multiple input requests into one output. It shows which inputs are active and allows individual inputs to be masked. Alternatively, it prioritizes the inputs, shows the highest active input, and provides a mask for inputs below a given level. The processor status register has a global interrupt enable flag bit. In addition, a watchdog timer is connected to the non-maskable interrupt input.

The interrupt software associated with a specific device is known as its *interrupt service routine* (ISR), or handler.

Software

Some older CPUs routed all interrupts to a single ISR. Upon recognizing an interrupt, the CPU saved some state

CUT HERE

FIGURE 1 Interrupt hardware model

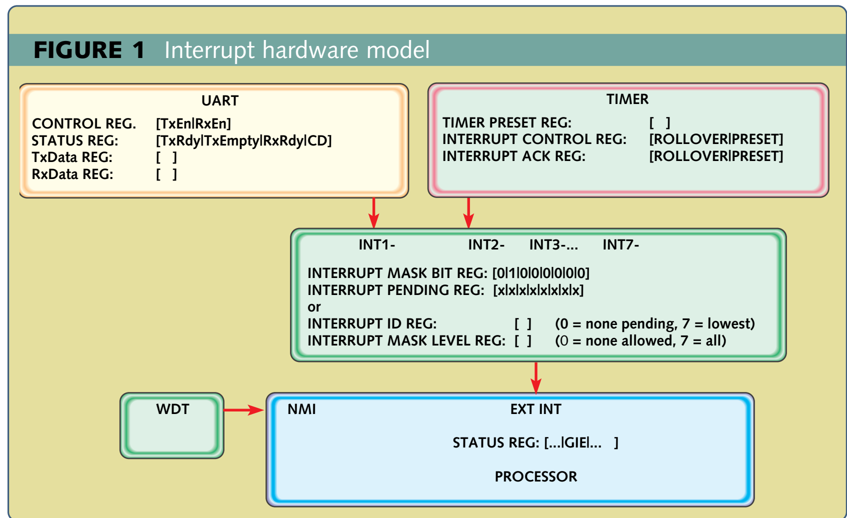
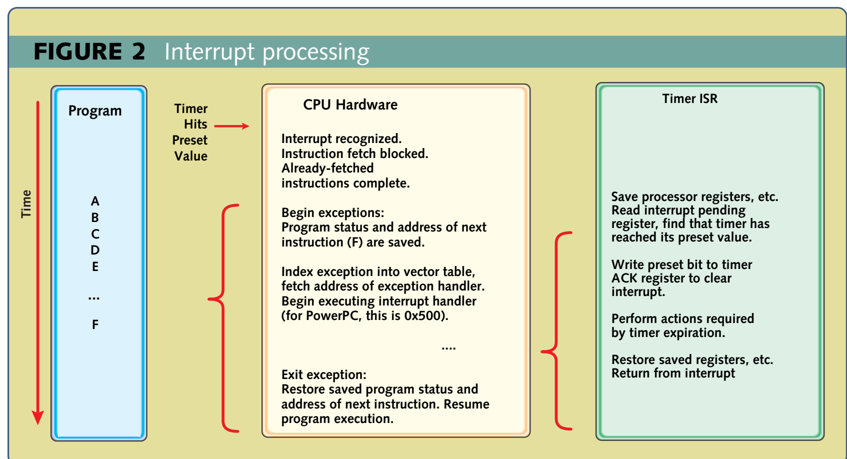


FIGURE 2 Interrupt processing



information and started execution at a fixed location. The ISR at that location had to poll the devices in priority order to determine which one required service. However, the basic process of interrupt handling is the same as in the more complex case.

Most modern CPUs use the same general mechanism for processing exceptions, traps, and interrupts: an interrupt vector table. Some CPU vector tables contain only the address of the code to be executed. In most cases, a specific ISR is responsible for servicing each interrupting device and acknowledging, clearing, and rearming its interrupt; in some cases, servicing the device (for example, reading data from a serial port) automatically clears and rearms the interrupt.

Interrupts may occur at any time, but the CPU does not instantly recognize and process them immediately. First, the CPU will not recognize a new interrupt while interrupts are disabled. Second, the CPU must, upon recognition, stop fetching new instructions and complete those still in progress. Because the interrupt is totally unrelated to the running program it interrupts, the CPU and ISR work together to save and restore the full state of the interrupted program (stack, flags, registers, and so on). The running program is not affected by the interruption, although it takes longer to execute. The hardware and software flow for a timer interrupt is shown in Figure 2.

Many interrupt controllers provide a means of prioritizing interrupt sources, so that, in the event of multiple interrupts occurring at (approximately) the same time, the more time-critical ones are processed first. These same systems usually also provide for prioritized interrupt handling, a means by which a higher-priority interrupt can interrupt the processing of a lower-priority interrupt. This is called *interrupt nesting*.

In general, the ISR should only take care of the time-critical portion of the processing, then, depending on the complexity of the system, it may set a flag for the main loop, or use an operating system call to awaken a task to perform the non-time-critical portion.

Latency

The *interrupt latency* is the interval of time measured from the instant an interrupt is asserted until the corresponding ISR begins to execute. The worst-case latency for any given interrupt is a sum of many things, from longest to shortest:

- The longest period global interrupt recognition is inhibited
- The time it would take to execute all higher priority interrupts if they occurred simultaneously
- The time it takes the specific ISR to service all of its interrupt requests (if multiple are possible)
- The time it takes to finish the program instructions in progress and save the current program state and begin the ISR

We can see how higher-priority interrupts can have much lower latencies. In simple cases, latency can be calculated from instruction times, but many modern systems with 32-bit CPUs, caches, and multiple interrupt sources, are far too complex for exact latency calculations. Interrupt latency must be considered at design time, whenever responsiveness matters. **esp**

Russell Massey is a staff engineer with Honeywell Industrial Control. He has worked in the computer field for over 25 years. His e-mail address is russell.massey@honeywell.com.

CUT HERE