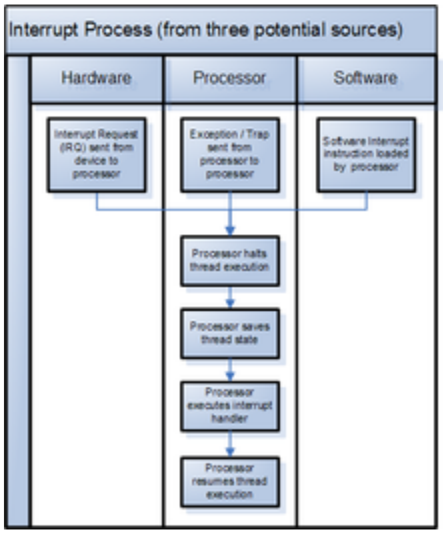
WIKIPEDIA

Interrupt

In system programming, an **interrupt** is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing. The processor responds by suspending its current activities, saving its state, and executing a function called an *interrupt handler* (or an interrupt service routine, ISR) to deal with the event. This interruption is temporary, and, after the interrupt handler finishes, unless handling the interrupt has emitted a fatal error, the processor resumes normal activities.[1] There are two types of interrupts: hardware interrupts and software interrupts (softirqs).


Interrupt Process (from three potential sources)
Hardware
Processor
Software
Interrupt Request (IRQ) sent from device to processor
Exception / Trap sent from processor to processor
Software Interrupt instruction loaded by processor
Processor halts thread execution
Processor saves thread state
Processor executes interrupt handler
Processor resumes thread execution

interrupt sources and processor handling

Hardware interrupts are used by devices to communicate that they require attention from the operating system.[2], or a bare-metal program running on the CPU if there are no OSes. Internally, hardware interrupts are implemented using electronic alerting signals that are sent to the processor from an external device, which is either a part of the computer itself, such as a disk controller, or an external peripheral. For example, pressing a key on the keyboard or moving the mouse triggers hardware interrupts that cause the processor to read the keystroke or mouse position. Unlike the software type (described below), hardware interrupts are asynchronous and can occur in the middle of instruction execution, requiring additional care in programming. The act of initiating a hardware interrupt is referred to as an interrupt request (IRQ). Different devices are usually associated with different interrupts using a unique value associated with each interrupt. This makes it possible to know which hardware device caused which interrupts. These interrupt values are often called IRQ lines, or just interrupt lines.

Software interrupts are also called exceptions. Unlike interrupts, exceptions occur synchronously with respect to the processor clock. That is why sometimes they are referred to as synchronous interrupts. A software interrupt is caused either by an exceptional condition in the processor itself, or a special instruction in the instruction set which causes an interrupt when it is executed. The former is often called a *trap* or *exception* and is used for errors or events occurring during program execution that is exceptional enough that they cannot be handled within the program itself. For example, a divide-by-zero exception will be thrown if the processor's arithmetic logic unit is commanded to divide a number by zero as this instruction is an error and impossible. The operating system will catch this exception, and can decide what to do about it: usually aborting the process and displaying an error message. Software interrupt instructions can function similarly to subroutine calls and are used for a variety of purposes, such as to request services from device drivers, like interrupts sent to and from a disk controller to request reading or writing of data to and from the disk.

Each interrupt has its own interrupt handler. The number of hardware interrupts is limited by the number of interrupt request (IRQ) lines to the processor, but there may be hundreds of different software interrupts. Interrupts are a commonly used technique for computer multitasking, especially in real-time computing. Such a system is said to be interrupt-driven.[3]

Interrupts are similar to signals, the difference being that signals are used for inter-process communication (IPC), mediated by the kernel (possibly via system calls) and handled by processes, while interrupts are mediated by the processor and handled by the kernel. The kernel may pass an interrupt as a signal to the process that caused it (typical examples are SIGSEGV, SIGBUS, SIGILL and SIGFPE).

Contents

Overview

Hardware interrupts were introduced as an optimization, eliminating unproductive waiting time in polling loops, waiting for external events. The first system to use this approach was the DYSEAC, completed in 1954, although earlier systems provided error trap functions.[4] Interrupts may be implemented in hardware as a distinct component with control lines, or they may be integrated into the memory subsystem.

If implemented in hardware as a distinct component, an interrupt controller circuit such as the IBM PC's Programmable Interrupt Controller (PIC) may be connected between the interrupting device and the processor's interrupt pin to multiplex several sources of interrupt onto the one or two CPU lines typically available. If implemented as part of the memory controller, interrupts are mapped into the system's memory address space.

Interrupts can be categorized into these different types:

- *Maskable interrupts*: hardware interrupts that may be ignored by setting a bit in an interrupt mask register's (IMR) bit-mask.
- *Non-maskable interrupts* (NMI): hardware interrupts that lack associated bit-masks, so that they can never be ignored. NMIs are used for the highest priority tasks such as timers, especially watchdog timers.
- *Inter-processor interrupts* (IPI): interrupts that are generated by one processor to interrupt another processor in a multiprocessor system.
- *Software interrupts*: interrupts generated within a processor by executing an instruction. Software interrupts are often used to implement system calls because they result in a subroutine call with a CPU ring level change.
- *Spurious interrupts*: hardware interrupts that are unwanted or unrecognizable. They are typically generated by system conditions such as electrical interference on an interrupt line or from unknown devices.

Processors typically have an internal *interrupt mask* which allows the software to ignore all external hardware interrupts while it is set. Setting or clearing this mask may be faster than accessing an interrupt mask register (IMR) in a PIC or disabling interrupts in the device itself. In some cases, such as the x86 architecture, disabling and enabling interrupts on the processor itself act as a memory barrier; however, it may actually be slower.

An interrupt that leaves the machine in a well-defined state is called a *precise interrupt*. Such an interrupt has four properties:

- The Program Counter (PC) is saved in a known place.
- All instructions before the one pointed to by the PC have fully executed.
- No instruction beyond the one pointed to by the PC has been executed, or any such instructions are undone before handling the interrupt.
- The execution state of the instruction pointed to by the PC is known.

An interrupt that does not meet these requirements is called an *imprecise interrupt*.

The phenomenon where the overall system performance is severely hindered by excessive amounts of processing time spent handling interrupts is called an interrupt storm.

History

The UNIVAC 1103 computer is generally credited with the earliest use of interrupts in 1953.[5] Earlier, on the UNIVAC I (1951) "Arithmetic overflow either triggered the execution a two-instruction fix-up routine at address 0, or, at the programmer's option, caused the computer to stop." The IBM 650 (1954) incorporated the first occurrence of interrupt masking. The National Bureau of Standards DYSEAC (1954) was the first to use interrupts for I/O. The IBM 704 was the first to use interrupts for debugging, with a "transfer trap", which could invoke a special routine when a branch instruction was encountered.The MIT Lincoln Laboratory TX-2 system (1957) was the first to provide multiple levels of priority interrupts.[6]

Types of interrupts

Level-triggered

A *level-triggered interrupt* is an interrupt signaled by maintaining the interrupt line at a high or low logic level. A device wishing to signal a level-triggered interrupt drives the interrupt request line to its active level (high or low) and then holds it at that level until it is serviced. It ceases asserting the line when the CPU commands it to or otherwise handles the condition that caused it to signal the interrupt.

Typically, the processor samples the interrupt input at predefined times during each bus cycle such as state T2 for the Z80 microprocessor. If the interrupt isn't active when the processor samples it, the CPU doesn't see it. One possible use for this type of interrupt is to minimize spurious signals from a noisy interrupt line: a spurious pulse will often be so short that it is not noticed.

Multiple devices may share a level-triggered interrupt line if they are designed to. The interrupt line must have a pull-down or pull-up resistor so that when not actively driven it settles to its inactive state. Devices actively assert the line to indicate an outstanding interrupt, but let the line float (do not actively drive it) when not signaling an interrupt. The line is then in its asserted state when any (one or more than one) of the sharing devices is signaling an outstanding interrupt.

Level-triggered interrupts are favored by some because it is easy to share the interrupt request line without losing the interrupts, when multiple shared devices interrupt at the same time. Upon detecting assertion of the interrupt line, the CPU must search through the devices sharing the interrupt request line until one who triggered the interrupt is detected. After servicing this device, the CPU may recheck the interrupt line status to determine whether any other devices also need service. If the line is now de-asserted, the CPU avoids checking the remaining devices on the line. Since some devices interrupt more frequently than others, and other device interrupts are particularly expensive, a careful ordering of device checks is employed to increase efficiency. The original PCI standard mandated level-triggered interrupts because of this advantage of sharing interrupts.

There are also serious problems with sharing level-triggered interrupts. As long as any device on the line has an outstanding request for service the line remains asserted, so it is not possible to detect a change in the status of any other device. Deferring servicing a low-priority device is not an option, because this would prevent detection of service requests from higher-priority devices. If there is a device on the line that the CPU does not know how to service, then any interrupt from that device permanently blocks all interrupts from the other devices.

Edge-triggered

An *edge-triggered interrupt* is an interrupt signaled by a level transition on the interrupt line, either a falling edge (high to low) or a rising edge (low to high). A device wishing to signal an interrupt drives a pulse onto the line and then releases the line to its inactive state. If the pulse is too short to be detected by polled I/O then special hardware may be required to detect it.

Multiple devices may share an edge-triggered interrupt line if they are designed to. The interrupt line must have a pull-down or pull-up resistor so that when not actively driven it settles to its inactive state, which is the default sate of it. Devices signal an interrupt by briefly driving the line to its non-default state, and let the line float (do not actively drive it) when not signaling an interrupt. This type of connection is also referred to as open collector. The line then carries all the pulses generated by all the devices. (This is analogous to the pull cord on some buses and trolleys that any passenger can pull to signal the driver that they are requesting a stop.) However, interrupt pulses from different devices may merge if they occur close in time. To avoid losing interrupts the CPU must trigger on the trailing edge of the pulse (e.g. the rising edge if the line is pulled up and driven low). After detecting an interrupt the CPU must check all the devices for service requirements.

Edge-triggered interrupts do not suffer the problems that level-triggered interrupts have with sharing. Service of a low-priority device can be postponed arbitrarily, while interrupts from high-priority devices continue to be received and get serviced. If there is a device that the CPU does not know how to service, which may raise spurious interrupts, it won't interfere with interrupt signaling of other devices. However, it is easy for an edge-triggered interrupt to be missed - for example, when interrupts are masked for a period - and unless there is some type of hardware latch that records the event it is impossible to recover. This problem caused many "lockups" in early computer hardware because the processor did not know it was expected to do something. More modern hardware often has one or more interrupt status registers that latch interrupts requests; well-written edge-driven interrupt handling code can check these registers to ensure no events are missed.

The elderly Industry Standard Architecture (ISA) bus uses edge-triggered interrupts, without mandating that devices be able to share IRQ lines, but all mainstream ISA motherboards include pull-up resistors on their IRQ lines, so well-behaved ISA devices sharing IRQ lines should just work fine. The parallel port also uses edge-triggered interrupts. Many older devices assume that they have exclusive use of IRQ lines, making it electrically unsafe to share them.

Triggering

There are 3 ways multiple devices "sharing the same line" can be raised. First is by exclusive conduction (switching) or exclusive connection (to pins). Next is by bus (all connected to the same line listening): cards on a bus must know when they are to talk and not talk (ie, the ISA bus). Talking can be triggered in two ways: by accumulation latch or by logic gates. Logic gates expect a continual data flow that is monitored for key signals. Accumulators only trigger when the remote side excites the gate beyond a threshold, thus no negotiated speed is required. Each has its speed versus distance advantages. A trigger, generally, is the method in which excitation is detected: rising edge, falling edge, threshold (oscilloscope can trigger a wide variety of shapes and conditions).

Triggering for software interrupts must be built into the software (both in OS and app). A 'C' app has a trigger table (a table of functions) in its header, which both the app and OS know of and use appropriately that is not related to hardware. However do not confuse this with hardware interrupts which signal the CPU (the CPU enacts software from a table of functions, similarly to software interrupts).

Hybrid

Some systems use a hybrid of level-triggered and edge-triggered signaling. The hardware not only looks for an edge, but it also verifies that the interrupt signal stays active for a certain period of time.

A common use of a hybrid interrupt is for the NMI (non-maskable interrupt) input. Because NMIs generally signal major – or even catastrophic – system events, a good implementation of this signal tries to ensure that the interrupt is valid by verifying that it remains active for a period of time. This 2-step approach helps to eliminate false interrupts from affecting the system.

Message-signaled

A *message-signaled interrupt* does not use a physical interrupt line. Instead, a device signals its request for service by sending a short message over some communications medium, typically a computer bus. The message might be of a type reserved for interrupts, or it might be of some pre-existing type such as a memory write.

Message-signalled interrupts behave very much like edge-triggered interrupts, in that the interrupt is a momentary signal rather than a continuous condition. Interrupt-handling software treats the two in much the same manner. Typically, multiple pending message-signaled interrupts with the same message (the same virtual interrupt line) are allowed to merge, just as closely spaced edge-triggered interrupts can merge.

Message-signalled interrupt vectors can be shared, to the extent that the underlying communication medium can be shared. No additional effort is required.

Because the identity of the interrupt is indicated by a pattern of data bits, not requiring a separate physical conductor, many more distinct interrupts can be efficiently handled. This reduces the need for sharing. Interrupt messages can also be passed over a serial bus, not requiring any additional lines.

PCI Express, a serial computer bus, uses message-signaled interrupts exclusively.

Doorbell

In a push button analogy applied to computer systems, the term *doorbell* or *doorbell interrupt* is often used to describe a mechanism whereby a software system can signal or notify a computer hardware device that there is some work to be done. Typically, the software system will place data in some well-known and mutually agreed upon memory location(s), and "ring the doorbell" by writing to a different memory location. This different memory location is often called the doorbell region, and there may even be multiple doorbells serving different purposes in this region. It is this act of writing to the doorbell region of memory that "rings the bell" and notifies the hardware device that the data are ready and waiting. The hardware device would now know that the data are valid and can be acted upon. It would typically write the data to a hard disk drive, or send them over a network, or encrypt them, etc.

The term *doorbell interrupt* is usually a misnomer. It is similar to an interrupt, because it causes some work to be done by the device; however, the doorbell region is sometimes implemented as a polled region, sometimes the doorbell region writes through to physical device registers, and sometimes the doorbell region is hardwired directly to physical device registers. When either writing through or directly to physical device registers, this may cause a real interrupt to occur at the device's central processor unit (CPU), if it has one.

Doorbell interrupts can be compared to Message Signaled Interrupts, as they have some similarities.

Difficulty with sharing interrupt lines

Multiple devices sharing an interrupt line (of any triggering style) all act as spurious interrupt sources with respect to each other. With many devices on one line, the workload in servicing interrupts grows in proportion to the square of the number of devices. It is therefore preferred to spread devices evenly across the available interrupt lines. Shortage of interrupt lines is a problem in older system designs where the interrupt lines are distinct physical conductors. Message-signaled interrupts, where the interrupt line is virtual, are favored in new system architectures (such as PCI Express) and relieve this problem to a considerable extent.

Some devices with a poorly designed programming interface provide no way to determine whether they have requested service. They may lock up or otherwise misbehave if serviced when they do not want it. Such devices cannot tolerate spurious interrupts, and so also cannot tolerate sharing an interrupt line. ISA cards, due to often cheap design and construction, are notorious for this problem. Such devices are becoming much rarer, as hardware logic becomes cheaper and new system architectures mandate shareable interrupts.

Performance issues

Interrupts provide low overhead and good latency at low load, but degrade significantly at high interrupt rate unless care is taken to prevent several pathologies. These are various forms of livelocks, when the system spends all of its time processing interrupts to the exclusion of other required tasks. Under extreme conditions, a large number of interrupts (like very high network traffic) may completely stall the system. To avoid such problems, an operating system must schedule network interrupt handling as carefully as it schedules process execution.[7]

With multi-core processors, additional performance improvements in interrupt handling can be achieved through receive-side scaling (RSS) when multiqueue NICs are used. Such NICs provide multiple receive queues associated to separate interrupts; by routing each of those interrupts to different cores, processing of the interrupt requests triggered by the network traffic received by a single NIC can be distributed among multiple cores. Distribution of the interrupts among cores can be performed automatically by the operating system, or the routing of interrupts (usually referred to as *IRQ affinity*) can be manually configured.[8][9]

A purely software-based implementation of the receiving traffic distribution, known as *receive packet steering* (RPS), distributes received traffic among cores later in the data path, as part of the interrupt handler functionality. Advantages of RPS over RSS include no requirements for specific hardware, more advanced traffic distribution filters, and reduced rate of interrupts produced by a NIC. As a downside, RPS increases the rate of inter-processor interrupts (IPIs). *Receive flow steering* (RFS) takes the software-based approach further by accounting for application locality; further performance improvements are achieved by processing interrupt requests by the same cores on which particular network packets will be consumed by the targeted application.[8][10][11]

Typical uses

Typical uses of interrupts include the following: system timers, disk I/O, power-off signals, and traps. Other interrupts exist to transfer data bytes using UARTs or Ethernet; sense key-presses; control motors; or anything else the equipment must do.

Another typical use is to generate periodic interrupts by dividing the output of a crystal oscillator and having an interrupt handler count the interrupts in order for a processor to keep time. These periodic interrupts are often used by the OS's task scheduler to reschedule the priorities of running processes. Some older computers generated periodic interrupts from the power line frequency because it was controlled by the utilities to eliminate long-term drift of electric clocks.

For example, a disk interrupt signals the completion of a data transfer from or to the disk peripheral; a process waiting to read or write a file starts up again. As another example, a power-off interrupt predicts or requests a loss of power, allowing the computer equipment to perform an orderly shut-down. Also, interrupts are used in typeahead features for buffering events like keystrokes.

Interrupts are used to allow emulation of instructions which are unimplemented on certain models in a computer line.[12] For example floating point instructions may be implemented in hardware on some systems and emulated on lower-cost systems. Execution of an unimplemented instruction will cause an interrupt. The operating system interrupt handler will recognize the occurrence on an unimplemented instruction, interpret the instruction in a software routine, and then return to the interrupting program as if the instruction had been executed.[13] This provides application software portability across the entire line.

See also

- Advanced Programmable Interrupt Controller (APIC)
- BIOS interrupt call
- Event-driven programming
- Exception handling
- INT (x86 instruction)
- Interrupt coalescing
- Interrupt handler
- Interrupt latency
- Interrupts in 65xx processors
- Ralf Brown's Interrupt List
- Interrupts on IBM System/360 architecture
- Time-triggered system
- Autonomous peripheral operation

References

1. Jonathan Corbet; Alessandro Rubini; Greg Kroah-Hartman (2005). "*Linux Device Drivers*, Third Edition, Chapter 10. Interrupt Handling" (https://lwn.net/images/pdf/LDD3/ch10.pdf) (PDF). O'Reilly Media. p. 269. Retrieved December 25, 2014. "Then it's just a matter of cleaning up, running software interrupts, and getting back to regular work. The "regular work" may well have changed as a result of an interrupt (the handler could wake_up a

process, for example), so the last thing that happens on return from an interrupt is a possible rescheduling of the processor."

2. "Hardware interrupts" (https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_MRG/1.3/html/Realtime_Reference_Guide/chap-Realtime_Reference_Guide-Hardware_interrupts.html). Retrieved 2014-02-09.
3. Rosenthal, Scott (May 1995). "Basics of Interrupts" (http://web.archive.org/web/20160426144654/http://www.sltf.com/articles/pein/pein9505.htm). Archived from the original (http://www.sltf.com/articles/pein/pein9505.htm) on 2016-04-26. Retrieved 2010-11-11.
4. Codd, Edgar F. "Multiprogramming". *Advances in Computers*. **3**: 82.
5. Bell, C. Gordon; Newell, Allen (1971). *Computer structures: readings and examples* (https://books.google.com/books?id=e39TAAAAMAAJ). McGraw-Hill. p. 46. ISBN 9780070043572. Retrieved Feb 18, 2019.
6. Smotherman, Mark. "Interrupts" (https://people.cs.clemson.edu/~mark/interrupts.html). Retrieved Feb 18, 2019.
7. "Eliminating receive livelock in an interrupt-driven kernel" (http://portal.acm.org/citation.cfm?id=263335). doi:10.1145/263326.263335 (https://doi.org/10.1145%2F263326.263335). Retrieved 2010-11-11.
8. Tom Herbert; Willem de Bruijn (May 9, 2014). "Documentation/networking/scaling.txt" (https://www.kernel.org/doc/Documentation/networking/scaling.txt). *Linux kernel documentation*. kernel.org. Retrieved November 16, 2014.
9. "Intel 82574 Gigabit Ethernet Controller Family Datasheet" (http://www.intel.com/content/dam/doc/datasheet/82574l-gbe-controller-datasheet.pdf) (PDF). Intel. June 2014. p. 1. Retrieved November 16, 2014.
10. Jonathan Corbet (November 17, 2009). "Receive packet steering" (https://lwn.net/Articles/362339/). LWN.net. Retrieved November 16, 2014.
11. Jake Edge (April 7, 2010). "Receive flow steering" (https://lwn.net/Articles/382428/). LWN.net. Retrieved November 16, 2014.
12. Thusoo, Shalesh; et al. "Patent US 5632028 A" (https://www.google.com/patents/US5632028). *Google Patents*. Retrieved Aug 13, 2017.
13. Altera Corporation (2009). *Nios II Processor Reference* (https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/nios2/n2cpu_nii51002.pdf) (PDF). p. 4. Retrieved Aug 13, 2017.

External links

- Interrupts Made Easy (http://www.atarimagazines.com/compute/issue149/60_Interrupts_made_easy.php)
- Interrupts for Microchip PIC Microcontroller (http://www.microautomate.com/PIC/pic-interrupts.php)
- IBM PC Interrupt Table (http://stanislavs.org/helppc/int_table.html)
- University of Alberta CMPUT 296 Concrete Computing Notes on Interrupts (https://web.archive.org/web/20120313195304/http://ugweb.cs.ualberta.ca/~c296/Arduino-

UofA/ConcreteComputing/section/interrupts.htm), archived from the original on March 13, 2012

Retrieved from "https://en.wikipedia.org/w/index.php?title=Interrupt&oldid=905761457"

This page was last edited on 11 July 2019, at 07:42 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.