Polling and Interrupts

Table of contents

The problem . . . 1
Polling . . . 2
Interrupts . . . 2
A fourth way to leave the sequence . . . 3
The three issues . . . 4
The sequence of actions . . . 5
Transparency . . . 6
Nesting and priority . . . 6
Traps . . . 8
Pitfalls . . . 9
See also . . . 9

How a CPU finds out that an I/O device needs attention: by **asking** (polling) or by being **told** (interrupts). This note covers what an interrupt does to the flow of control, the three problems any interrupt mechanism has to solve, the fourteen steps from a device's request to the resumed program, **transparency**, what to do about a second interrupt, and **traps**, the software counterpart. First met in 2026-09-21-interrupts[1]. How the AVR implements it (its vector table, the ISR in assembly and in C, and the registers that switch an interrupt on) is avr-interrupts[2].

The problem

The CPU is fast, and I/O is slow and **asynchronous**. At 8 MHz the AVR completes an instruction every one or two clock cycles, while a key is pressed, a character arrives or a timer matches on its own schedule. That schedule has no relation to the clock or to which instruction the program is on. The CPU needs to know two things:

[1] courses/csse2010/2026-09-21-interrupts.pdf
[2] courses/csse2010/avr-interrupts.pdf

- **when data is available from an input device**, and
- **when an output device is ready to accept data.**

There are two ways to find out: **polling-driven I/O** and **interrupt-driven I/O**.

Polling

The processor periodically asks each device whether it is ready, and waits for the answer. On the AVR, asking is reading a register: a pin in `PINx` (avr-io-ports[3]), or a flag in a timer's `TIFRn` (avr-timers-and-counters[4]). A polling program is a main loop that asks each device in turn:

```
while (1) {
    if (PINC & (1<<0)) {            /* is the button pressed?  */
        /* ... handle the button ... */
    }
    if (TIFR1 & (1<<OCF1A)) {       /* has the timer matched?  */
        TIFR1 = (1<<OCF1A);         /* clear the flag          */
        /* ... handle the tick ... */
    }
    /* ... everything else the program does ... */
}
```

It is **inefficient**, in two ways:

- **Time spent asking.** Most asks find nothing, and every one of them costs instructions.
- **Time spent waiting.** An event is noticed only at the next ask, so the delay before it is handled can be as long as one pass of the loop. Every piece of work added to the loop lengthens that pass, and so slows the response to *every* device.

But it may be fine if the processor has nothing else to do. A program whose only job is to wait for a button and then act gains nothing from anything cleverer. Polling is also simple: the program always knows exactly where it is, and nothing happens behind its back.

Interrupts

The device tells the CPU when it is ready. The three kinds of *ready*:

[3] courses/csse2010/avr-io-ports.pdf
[4] courses/csse2010/avr-timers-and-counters.pdf

Kind	Example
Device ready to accept more output	the serial transmit buffer is empty
Device has input ready	a key pressed on a keyboard
A regular interrupt	a timer generating an interrupt every, say, 20 ms

When the request arrives, the CPU finishes its current instruction, sets the running program aside, and runs a routine for that device, the **interrupt handler** or **interrupt service routine (ISR)**. When the handler completes, **control returns to where the program left off.** The interrupted program does not ask, wait, or even know.

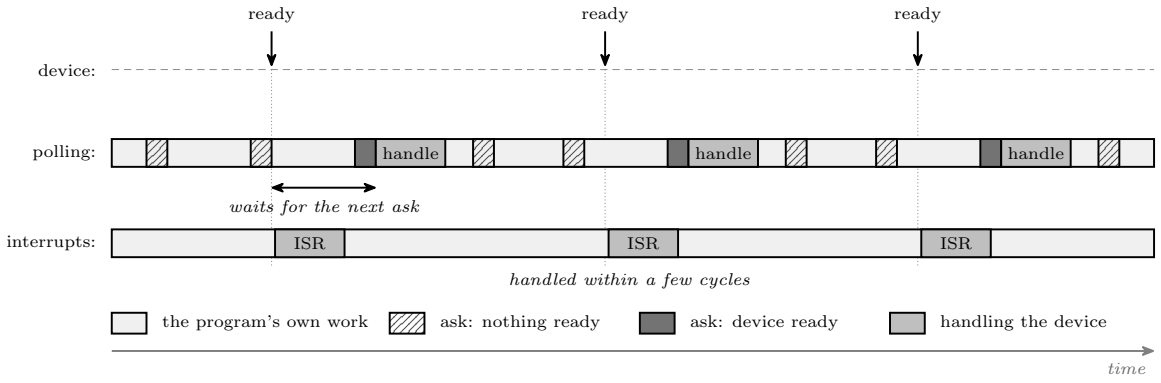


Figure 1: The same three device events handled both ways. Polling: the CPU breaks off its own work at fixed intervals to ask whether the device is ready. Most asks find nothing (hatched), and an event waits for the next ask before it is noticed, so how quickly it is handled depends on how often the program asks. Interrupts: the CPU does its work without asking, and each event diverts it into the handler within a few cycles. No time is spent asking, and the response does not depend on how long the main loop is. The work done is the same in both rows; what differs is the time wasted asking and the delay before each event is handled.

Interrupts are **more efficient for the CPU**, and they **change the flow of control**, which is where their cost lies.

A fourth way to leave the sequence

Normally the PC just increments. avr-branches-and-jumps[5] and procedures-and-the-stack[6] cover the instructions that change it. An interrupt changes it too, but without any instruction

[5] courses/csse2010/avr-branches-and-jumps.pdf
[6] courses/csse2010/procedures-and-the-stack.pdf

asking:

	Triggered by	Where the new PC comes from	Comes back?
Branch	an instruction, if a flag says so	an offset in the instruction	no
Jump	an instruction	the instruction	no
Procedure call	an instruction (`call`)	the instruction	yes: `ret` pops the return address `call` pushed
Interrupt	a device, between *any* two instructions	a table, indexed by which device	yes: the return-from-interrupt instruction pops the address the hardware pushed
Trap	an instruction that produces an exceptional result	a table, indexed by which error	not always

The useful way to see an interrupt is as **a procedure call that the hardware makes**. The mechanism is the same: push the return address, load the PC, and pop it back at the end. The difference is that the program never asked for it, and that one difference produces all three of the issues below.

The three issues

Any interrupt mechanism has to answer three questions.

1. **How does the CPU know where the ISR is?** A call names its target, but an interrupt does not, so the address must be found from the **source**. The CPU looks in an **interrupt vector table**, often at the beginning of memory. The table holds either the **address** of each handler, which the CPU loads into the PC, or a **jump instruction** to each handler, in which case the CPU loads the PC with the entry's own address and executes the jump. The AVR does the second. *Vector* is used both for the interrupt's number (its index in the table) and for the address of its handler, and it is worth checking which is meant.
2. **How are the registers preserved?** All of them, **including the status register**. The handler must save copies of any register it uses before using it, and restore them before it returns. Many CPUs save the status register automatically; the AVR does not, so the handler must save it itself if it would change it. The result is **transparency**, below.
3. **What happens if another interrupt arrives while one is being serviced?** Either it waits, or it interrupts the handler. See *Nesting and priority*, below.

The sequence of actions

From the device's request to the resumed program there are fourteen steps, shared between the hardware and the ISR:

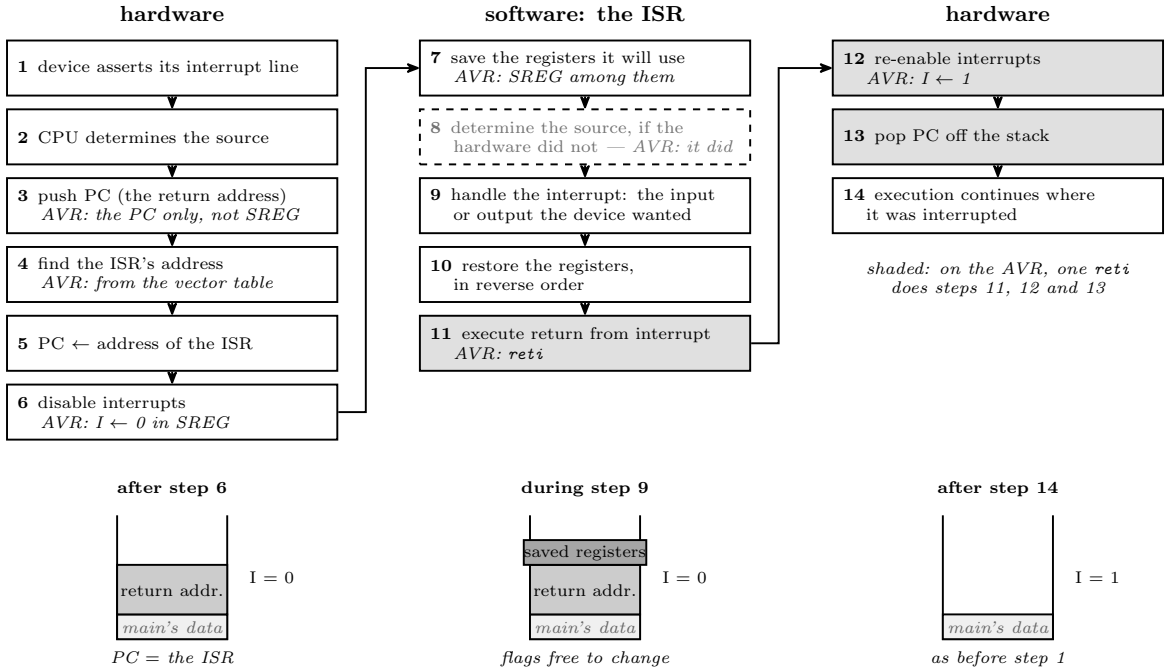


Figure 2: The fourteen steps of an interrupt, in the three phases the hardware and the ISR take turns at, with the state of the machine after each phase underneath. The hardware does the first six: it saves only the return address, finds and enters the handler, and turns interrupts off. The ISR saves what it will use, does the I/O and restores what it saved. The return-from-interrupt instruction then hands back to the hardware, which turns interrupts on and pops the return address. The notes in italics are the AVR's version of each step. The shaded steps 11 to 13 are a single AVR instruction, `reti`, and step 8 rarely arises on the AVR (dashed), because every source has its own vector and so the ISR knows which one fired; only a pin-change handler, whose vector covers a whole port, still has to find which pin changed. The stack after step 14 is exactly as it was before step 1, and that is what transparency requires.

Three things in the sequence are worth holding on to:

- **The hardware saves as little as it can.** It pushes only the return address, and everything else is the ISR's job. That keeps the entry fast, and it means an ISR that uses nothing need save nothing.

- **Interrupts are off from step 6 to step 12.** That is the default answer to issue 3: a second request simply waits until the handler has returned. On the AVR, *off* is the **I** bit of the status register being 0 (avr-architecture[7]).
- **Return from interrupt is not the same as return from procedure.** It has to undo step 6 as well as step 3. A handler that ends with a plain `ret` goes back to the right place but leaves interrupts disabled for good.

Transparency

When the ISR finishes, the computer is in the same state as it was before the interrupt. That is **transparency**, and it is the answer to issue 2. It matters more for an ISR than for a procedure because the interrupted code has no chance to prepare. A caller knows where its calls are and can agree with the procedure which registers it may use. An interrupt can land between any two instructions, including these two:

```
    cp   r16, r17        ; sets Z if r16 == r17
                         ; <- an interrupt here
    breq equal           ; reads Z
```

If the ISR does anything that sets flags — a `dec`, an `andi`, a `cp` of its own — and does not restore the status register, `breq` reads the ISR's Z instead of the one `cp` produced. The program then takes the wrong branch, but only on the rare occasions the interrupt lands in that one gap. A bug like that can survive a great deal of testing. So the status register is the one register a handler must always consider, even if it touches nothing else. Whether an instruction changes it is listed per instruction in the instruction set reference (avr-instructions[8]).

Nesting and priority

For an interrupt that arrives while an ISR is running, there are three policies:

- **Disable interrupts during an ISR.** The simplest: the second request waits (its flag stays set) and is serviced after the first handler returns. On CPUs that save the status register automatically, the global interrupt enable flag is among the bits the return-from-interrupt instruction restores, so it comes back on by itself. **This is the AVR's default.**
- **Prioritise.** Higher-priority interrupts may interrupt lower-priority handlers, but not the other way round and not at the same level. Urgent devices are never kept waiting behind unimportant ones, and an unimportant device can never delay an urgent one. **Many CPUs support this; the AVR does not.**

[7] courses/csse2010/avr-architecture.pdf
[8] courses/csse2010/avr-instructions.pdf

- **Re-enable interrupts inside the ISR.** Any interrupt may then interrupt the handler, *regardless of priority.* The AVR allows this if the handler executes `sei`.

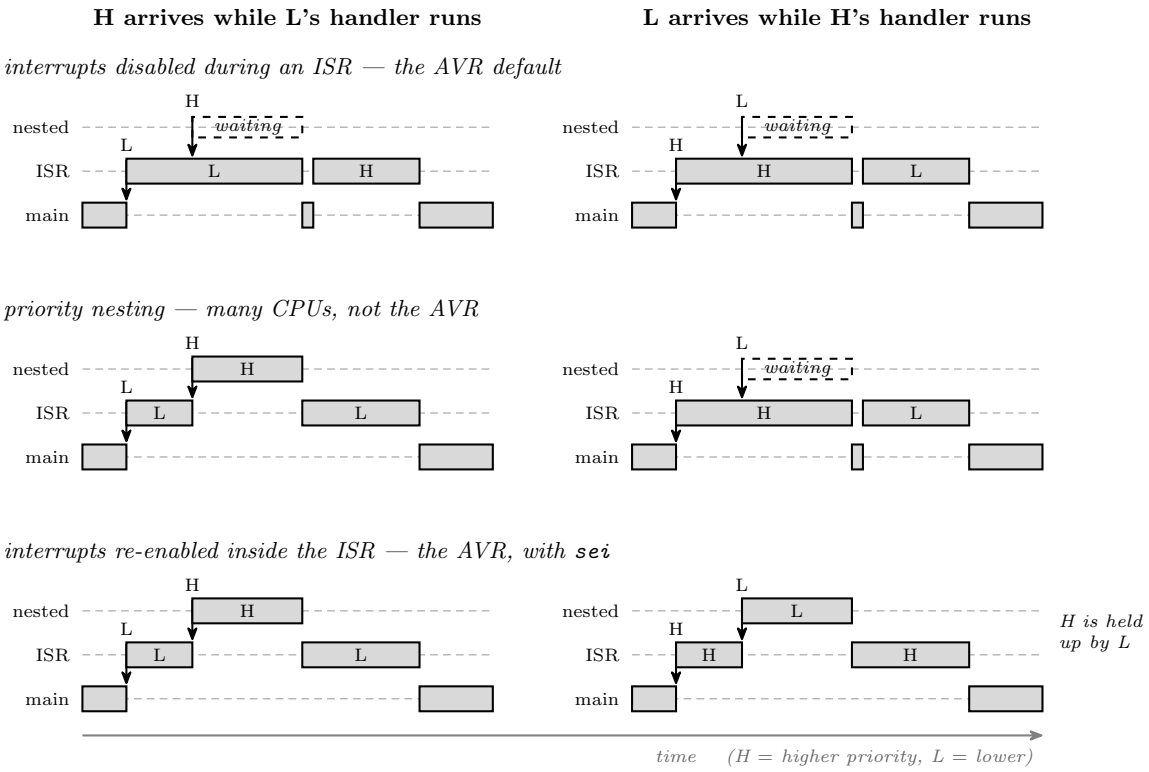


Figure 3: The three policies for an interrupt that arrives while another is being handled, in the two cases that tell them apart. In each panel, code on a higher lane has interrupted the code below it, and a dashed box is a request that is waiting. With interrupts disabled during an ISR, whichever handler started first finishes first, and the other waits for it in both cases. With priority nesting, the urgent request (H) cuts into the less urgent handler (L) but never the reverse, which is the behaviour one would want and the one the AVR does not provide. With interrupts re-enabled inside the handler, any request cuts in immediately, so in the right-hand case the urgent handler is itself held up by the less urgent one.

On the AVR, then, **priority only decides the order in which waiting interrupts are serviced**. It matters only when two or more are waiting at once, for instance because they arrived during another handler. The highest-priority one, the one with the lowest vector, goes first. It never lets an urgent interrupt cut into a running handler. The one way to get nesting is to re-enable interrupts inside the handler, and then priority plays no part at all. Both cases

are traced, with the I bit, in 2026-09-21-interrupts[9].

The practical consequence is a rule: **keep ISRs short.** While any handler runs, every other interrupt waits. A handler that busy-waits for 10 ms delays every other device by 10 ms, and a timer tick that arrives twice during that time is serviced only once, because a flag can record *that* an event happened but not how many times.

Traps

Traps are software interrupts: interrupts caused by events in the software rather than in the hardware, and specifically by **exceptional** events, such as

- **overflow**,
- **divide by zero**, and
- **an undefined opcode**.

They relate to error checking the way interrupts relate to polling. Without traps, a program that must not overflow has to test after every arithmetic instruction, which is polling for errors. With them, the hardware watches, and the program is diverted only when the error actually happens. **Traps save continually checking for errors.**

Trap handlers don't always return to the original program. After a division by zero or an undefined opcode, there is often no sensible instruction to continue with. Under an operating system the usual outcome is that the program is ended.

	Interrupt	Trap
Caused by	a device, outside the program	the instruction being executed
When	at an unpredictable point in the program, *asynchronously*	at the same instruction every time, given the same data
Returns?	always, to the interrupted instruction's successor	not always

The AVR does not provide traps. Overflow sets V in the status register and it is up to the program to branch on it (avr-branches-and-jumps[10]), and the AVR has no divide instruction to fail. **But it is possible to generate hardware interrupts from software**, by writing to an external interrupt pin configured as an output (avr-interrupts[11]).

[9] courses/csse2010/2026-09-21-interrupts.pdf
[10] courses/csse2010/avr-branches-and-jumps.pdf
[11] courses/csse2010/avr-interrupts.pdf

Pitfalls

- **Assuming priority means pre-emption.** On the AVR it does not: a higher-priority interrupt waits for a lower-priority handler to finish, like any other.
- **Not saving the status register** in a handler that changes flags. The symptom is an interrupted comparison going the wrong way, rarely and irreproducibly.
- **Long handlers.** Every other interrupt waits while one runs, and repeated events during the wait are merged into one.
- **Variables shared between a handler and the main program** that the compiler is allowed to cache. In C they must be `volatile` (avr-interrupts[12]).
- **Returning with `ret` instead of the return-from-interrupt instruction**: the program continues, and interrupts never come back on.

See also

- avr-interrupts[13] — the ATmega324A's vector table, ISRs in assembly and C, enabling an interrupt, `volatile`, a complete timer-interrupt program
- 2026-09-21-interrupts[14] — Lecture 17, with the priority cases traced and worked questions
- procedures-and-the-stack[15] — `call`, `ret` and the stack the return address goes on
- avr-branches-and-jumps[16] — the other ways the PC leaves its sequence
- avr-timers-and-counters[17] — polling a timer flag, the thing interrupts replace
- software-timers[18] — a tick interrupt as the program's clock, and why busy waiting is the thing to avoid
- avr-usart[19] · avr-spi[20] · analog-to-digital-conversion[21] — three devices that can be polled or left to interrupt
- fetch-decode-execute-cycle[22] — the PC's normal behaviour, increment and fetch

[12] courses/csse2010/avr-interrupts.pdf
[13] courses/csse2010/avr-interrupts.pdf
[14] courses/csse2010/2026-09-21-interrupts.pdf
[15] courses/csse2010/procedures-and-the-stack.pdf
[16] courses/csse2010/avr-branches-and-jumps.pdf
[17] courses/csse2010/avr-timers-and-counters.pdf
[18] courses/csse2010/software-timers.pdf
[19] courses/csse2010/avr-usart.pdf
[20] courses/csse2010/avr-spi.pdf
[21] courses/csse2010/analog-to-digital-conversion.pdf
[22] courses/csse2010/fetch-decode-execute-cycle.pdf