



A9.7 Interrupts

Computer architecture · A level · OCR H446 1.2.1, AQA 7517 4.7.3.6 ·
about 20 min

Name _____

Class _____

Date _____

What this lesson is about

Sources of interrupts, polling, the check at the end of each cycle, saving the volatile environment on a stack, and priorities.

- What an interrupt is
- Polling or interrupts?
- Interrupts and the fetch-decode-execute cycle
- Why save the registers?
- Priorities and nesting
- A timer interrupt for a robot

Questions

5 marks in all

1. When does the processor check for an interrupt? [1 mark]

- ☐ A At the end of each fetch-decode-execute cycle
- ☐ B Halfway through executing an instruction
- ☐ C Only when the program has finished
- ☐ D Once a second

2. Why is the volatile environment saved before an ISR runs? [1 mark]

- ☐ A The ISR uses the same registers, so the interrupted program's values would otherwise be lost
- ☐ B To free up main memory
- ☐ C So the ISR can run faster
- ☐ D To stop further interrupts

3. Put the handling of an interrupt in order. [1 mark]

Number the lines 1 to 6 to put them in the right order.

- ___ The address of the ISR is loaded into the PC
- ___ The registers are pushed onto the stack
- ___ The registers are popped off the stack and the program resumes
- ___ The current instruction finishes executing
- ___ The processor checks for interrupts and finds one of higher priority
- ___ The ISR runs

4. Why is a stack the right structure for saving registers when interrupts are nested? [1 mark]

- ☐ A The last registers saved are the first restored, so nested interrupts unwind in the right order
- ☐ B A stack is faster than a queue
- ☐ C A stack can hold any amount of data
- ☐ D A stack is kept inside the ALU

5. Which is a disadvantage of polling compared with interrupts?

[1 mark]

- ☐ A Processor time is wasted checking devices that have nothing to report
- ☐ B It needs extra hardware
- ☐ C Devices cannot send data
- ☐ D It makes the stack overflow

The task: a timer interrupt

Simulate a processor that is interrupted by a timer. The main program runs 12 cycles. In each cycle it drives forward 3 cm, adds 3 to `acc`, and adds 1 to `pc`, in that order. At the **end** of any cycle after which `pc` is a multiple of 4 (so after cycles 4, 8 and 12), the timer interrupts. Handle it like this: 1. Save the volatile environment by pushing `pc`, then `acc`, onto the list `stack`, and print `saved PC=4 ACC=12` (with the real values). 2. Run the ISR. It uses the accumulator as its working register: set `acc` to 200 plus 100 times the number of interrupts so far including this one (300, then 400, then 500), and play `acc` as a tone for 0.2 seconds. 3. Restore the registers by popping them off `stack` in the reverse order, and print `restored PC=4 ACC=12`. After the 12 cycles, print `done PC=12 ACC=36`, using the values in the registers.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

pc = 0
acc = 0
stack = []

while pc < 12:
    forward(50, distance=3)
    acc = acc + 3
    pc = pc + 1

    if pc % 4 == 0:
        # Save environment
        stack.append(pc)
        stack.append(acc)
        print(f"saved PC={pc} ACC={acc}")

        # Run ISR
        acc = 200 + 100 * len(stack) // 2
        play(acc, 0.2)

        # Restore registers
        acc = stack.pop()
        pc = stack.pop()
        print(f"restored PC={pc} ACC={acc}")

print(f"done PC={pc} ACC={acc}")
```

Plan your program here, then type it in and press Run.



Do it on the robot

www.bugbotlab.com/learn/a9-7-interrupts/

The simulator checks it and tells you when it passes. Nothing to install, no account.

Challenges

1. Leave out the restore step. What does the program print at the end, and why is that wrong?

2. Add a second interrupt source, a "bump" with higher priority, that can arrive while the timer ISR is running. Show the stack depth reach 4.
3. Explain the difference between polling a sensor and a sensor that raises an interrupt, using a BugBot example.