ANSWERS

A10.4 Interrupts

Operating systems, software and translators · A level · OCR H446 1.2.1, AQA 7517 4.6.1.4, Eduqas A500QS 2.6 · about 40 min

BugBotLab

What this lesson is about

Polling versus interrupts, priorities, the stack and interrupt service routines in the fetch-decode-execute cycle.

- Polling
- Interrupts
- Interrupts and the fetch-decode-execute cycle
- A worked trace
- Polling or interrupts?

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** Only when the program has finished
- ☐ **C** In the middle of executing each instruction
- ☐ **D** Once every second

> **Answer:** A. An instruction is never left half done; the check comes between cycles.

2. Put these steps in order for an interrupt of higher priority than the current task. [1 mark]

Number the lines 1 to 5 to put them in the right order.

___ `The saved register values are popped off the stack and the task resumes`
___ `The contents of the registers, including the PC, are pushed onto the stack`
___ `The processor finishes the current fetch-decode-execute cycle`
___ `The interrupt service routine runs`
___ `The address of the interrupt service routine is loaded into the PC`

> **Answer:**
>
> ```
> The processor finishes the current fetch-decode-execute cycle
> The contents of the registers, including the PC, are pushed onto the stack
> The address of the interrupt service routine is loaded into the PC
> The interrupt service routine runs
> The saved register values are popped off the stack and the task resumes
> ```
>
> Saving the registers is what lets the interrupted task continue exactly where it stopped.

3. Why are the saved register values kept on a stack? [1 mark]

- ☐ **A** An ISR can itself be interrupted, and the routines must be resumed in reverse order
- ☐ **B** A stack is the fastest kind of memory
- ☐ **C** The stack is stored in the BIOS
- ☐ **D** A stack lets any saved value be removed first

> **Answer:** A. Last in, first out matches nested interrupts.

4. A low-priority printer interrupt arrives while a high-priority power-failure ISR is running. What happens? [1 mark]

- ☐ **A** It waits until the power-failure ISR has finished
- ☐ **B** It interrupts the power-failure ISR
- ☐ **C** It is deleted
- ☐ **D** Both routines run at the same time on one core

> **Answer:** A. Only an interrupt of higher priority than the running routine is serviced straight away.

5. Which are advantages of interrupts over polling? [1 mark]

Tick every answer that is true.

- ☐ **A** No processor time is wasted checking devices that have nothing to report
- ☐ **B** Urgent events are dealt with at the end of the current cycle
- ☐ **C** They are simpler to program
- ☐ **D** They need no stack

> **Answer:** A, B. Interrupts are more efficient and responsive, but more complex, and need a stack to save state.

The task: nested interrupts

Simulate the processor from the worked trace, with nesting. The inputs are: - `MAIN_LENGTH`, the number of instructions in the main program (6); main has priority 0; - `ISR_LENGTH`, the number of instructions in every ISR (2); - `arrivals`, a dictionary from a cycle number to a list of `(name, priority)` interrupts that arrive during that cycle. Priorities are positive integers; higher is more important. Keep the running routine's name, priority and next instruction number (starting at `"main"`, 0, 0), a `stack` list and a `pending` list. Number cycles from 0. Each cycle: 1. print `cycle <c>: <routine> <instruction>` and move on to the next instruction; 2. if the routine has now run all its instructions: if it is main, stop the program; otherwise pop the routine it interrupted off the stack and print `return to <routine> at <instruction>`; 3. add this cycle's arrivals to `pending`; if the highest-priority pending interrupt has a priority greater than the running routine's, remove it from `pending`, push the running routine onto the stack, print `interrupt <name>: saved <routine> at <instruction>`, and start that ISR at instruction 0. Build every line from your variables. The robot stays still.

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

MAIN_LENGTH = 6
ISR_LENGTH = 2
arrivals = {1: [("printer", 1)], 2: [("power", 3)], 3: [("keyboard", 1)], 4: [("timer",
2)]}

stack = []
pending = []
```

The hint students can ask for: Keep three things for the running routine: its name, its priority and its next instruction. At the end of each cycle, first deal with a routine that has just finished, then add the new arrivals to the pending list and compare the best of them with the running priority. The stack only ever holds routines that were interrupted.

A solution

```
from bugbot import *
connect()

MAIN_LENGTH = 6
ISR_LENGTH = 2
arrivals = {1: [("printer", 1)], 2: [("power", 3)], 3: [("keyboard", 1)], 4: [("timer",
2)]}

stack = []
pending = []
name, priority, pc = "main", 0, 0
cycle = 0
while True:
    print(f"cycle {cycle}: {name} {pc}")
    pc = pc + 1
    length = MAIN_LENGTH if name == "main" else ISR_LENGTH
    if pc == length:
        if name == "main":
            break
        name, priority, pc = stack.pop()
        print(f"return to {name} at {pc}")
    pending.extend(arrivals.get(cycle, []))
```

```
    if pending:
        best = max(pending, key=lambda item: item[1])
        if best[1] > priority:
            pending.remove(best)
            stack.append((name, priority, pc))
            print(f"interrupt {best[0]}: saved {name} at {pc}")
            name, priority, pc = best[0], best[1], 0
    cycle = cycle + 1
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.