Programming Embedded Systems

Lecture 5

Interrupts, modes of multi-tasking

Wednesday Feb 1, 2012

Philipp Rümmer
Uppsala University
`Philipp.Ruemmer@it.uu.se`

Lecture outline

- Interrupts
    - Internal, external, software
    - Interrupt service routines
    - Deferred interrupt handlers
- Different kinds of multi-tasking

Interrupts

- Hardware feature of processor to react to events
  - Clock interrupts ("system tick"):
    drives scheduler, determines time slices
  - Other "internal" interrupts: timers, etc.
  - Software interrupts/traps/faults:
    raised by executing particular instructions
  - External interrupts:
    events from peripherals, pins, etc.

Interrupts (2)

- When interrupt occurs, MCU executes an **interrupt service routine** (ISR) at a pre-defined code location
- ISR locations are stored in “interrupt vector table”
- For complete list of possible interrupts on STM32F10x/CORTEX M3: see reference manual, http://www.st.com/stonline/products/literature/rm/13902.pdf

Clock interrupts

- Usually set up to occur regularly (period ≥1ms); frequency can be chosen
- In assignments/labs: every 1ms
- Driven by internal/external real-time clock
- ISR is the scheduler, which might decide to switch in another task when tick occurs

Internal interrupts

- Timers can be set up to raise interrupts;
  Most importantly: upon overflow

Software interrupts/traps

- Used to implement system calls if kernel is running in privileged mode (`SVCall` interrupt)
- Signal fault conditions:
  memory faults, bus faults, etc.

External interrupts

- Explicit external-interrupt lines (general-purpose I/O ports)
- Part of interface to peripherals and buses (signal packet arrival, finished transmission, etc):
  DMA, CAN, I2C, USB, SPI, UART, ...
- Reset: initialisation, invocation of `main`
- Non-maskable interrupt (NMI): highest-priority, used e.g. for watchdogs

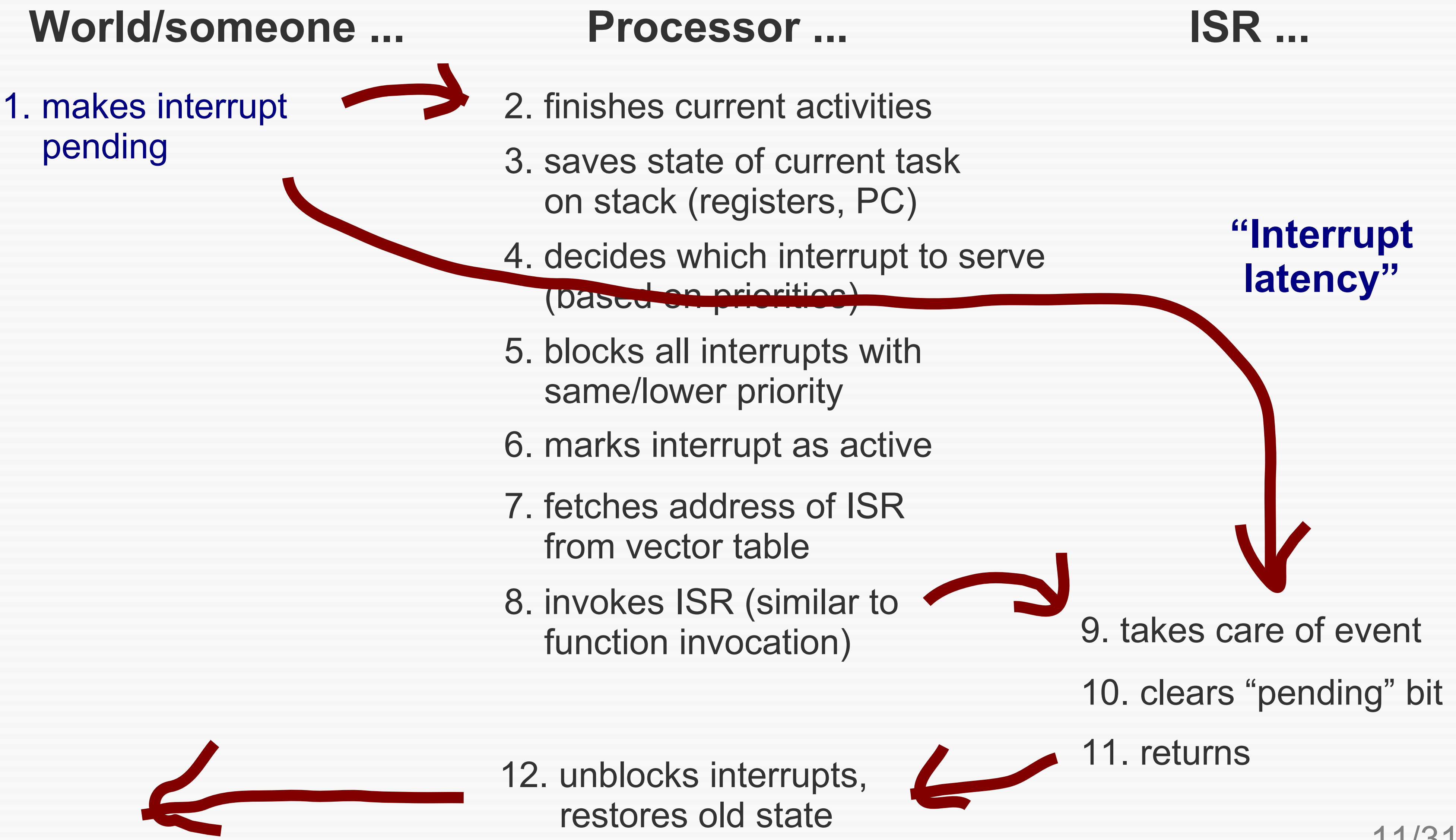
Setting up interrupts

- Typical parameters, chosen through special-purpose registers:
    - Enabled/disabled (unmasked/masked)
    - Priority (important when multiple interrupts occur simultaneously)
    - ISR address
    - Pulse/pending interrupts (cleared by itself/hardware or in ISR?)
    - Which events to observe (e.g., rising or falling edges of signals)

Further parameters

- **Pending:**
  Interrupt has been triggered, is waiting for being served or currently being served
- **Active:**
  ISR is executing

Interrupt handling



World/someone ...

1. makes interrupt pending

Processor ...

2. finishes current activities
3. saves state of current task on stack (registers, PC)
4. decides which interrupt to serve (based on priorities)
5. blocks all interrupts with same/lower priority
6. marks interrupt as active
7. fetches address of ISR from vector table
8. invokes ISR (similar to function invocation)

12. unblocks interrupts, restores old state

ISR ...

"Interrupt latency"

9. takes care of event
10. clears "pending" bit
11. returns

Interrupt latency

- Interrupt latency also depends on other, pending, higher-priority interrupts → *can vary*
- **Interrupt jitter**:
  amount of variation of latency
- Determines how fast system can react to events
- In the very best case, latency is
  12 cycles on CORTEX M3

Nested interrupts

- ISR can be interrupted itself by higher-priority interrupts
- Applies in particular to CORTEX M3 (**NVIC**, nested vectored interrupt controller)
  → used by default
- Can be prevented by disabling interrupts in ISR

Interrupt tail-chaining

- Directly execute a sequence of ISRs, without returning to normal program in between
- Safes some time (storing/restoring program state unnecessary)
- Also done by CORTEX M3 by default

Deferred interrupt handling

Motivation

- **Interrupts are generally problematic in real-time systems**
  - outside of normal scheduling, usually not pre-emptable for scheduler
  - can occur with high frequency, create high system loads
- With many OS kernels (e.g., FreeRTOS), certain/most functions must not be called from ISRs
- ISRs are normally not reentrant, or even a “critical section”

Solutions

- Avoiding interrupts *(more later)*
- **Deferred/split-interrupt handling**
  - Common in most OSs,
    not only in real-time systems

Deferred interrupt handling

- Keep ISR minimal
  (“Immediate Interrupt Service”)
- Actual handling of event done later in an ordinary task
  (“Scheduled Interrupt Service”)

Interrupt handling

World/someone...

1. makes interrupt pending

Processor ...

2. finishes current activities
3. saves state of current task on stack (registers, PC)
4. decides which interrupt to serve (based on priorities)
5. blocks all interrupts with same/lower priority
6. marks interrupt as active
7. fetches address of ISR from vector table
8. invokes ISR (similar to function invocation)

12. unblocks interrupts, restores old state

ISR ...

schedules actual interrupt service

communication through queue or semaphore

9. takes care of event
10. clears “pending” bit
11. returns

13. Scheduler switches in service task

14. Service task handles event

Processor ... **ISR ...**

2. finishes current activities
3. saves state of current task on stack (registers, PC)
4. decides which interrupt to serve (based on priorities)
5. blocks all interrupts with same/lower priority
6. marks interrupt as active
7. fetches address of ISR from vector table
8. invokes ISR (similar to function invocation)

~~9. takes care of event~~

10. clears “pending” bit

11. returns

12. unblocks interrupts, restores old state

FreeRTOS example

- Interrupt raised by timer
- ISR + scheduled interrupt task are C functions, communicate via binary semaphore
- ISR is specified in file `STM32F10x.s`
- STM32F10x uses 4bit priorities [0, 16) (split into *preemption-* and *sub-priority)*
- **Important:** most FreeRTOS functions (including semaphore functions) must not be used in ISR

FreeRTOS example (2)

- External interrupt line, connected to PORTA.0