

Lecture 5
**Interrupts,
 modes of multi-tasking**

Monday Jan 31, 2011

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

1/31

2/31

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.

3/31

4/31

Clock interrupts

- Usually set up to occur regularly (period $\geq 1\text{ms}$); 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

5/31

6/31

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.

7/31

Interrupts (2)

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

- 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>

Internal interrupts

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

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

8/31

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)

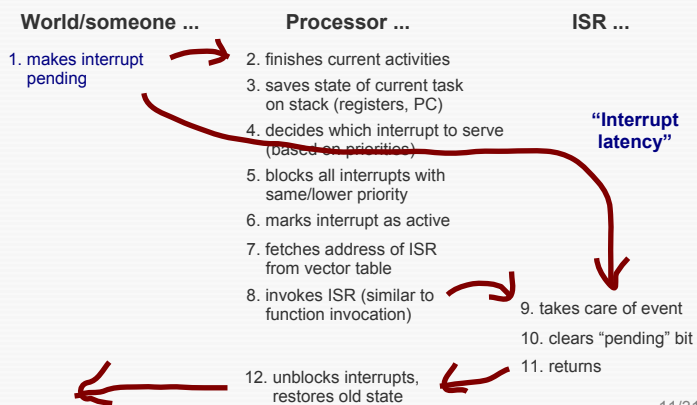
9/31

Further parameters

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

10/31

Interrupt handling



11/31

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

12/31

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

13/31

Interrupt tail-chaining

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

14/31

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"

15/31

16/31

Solutions

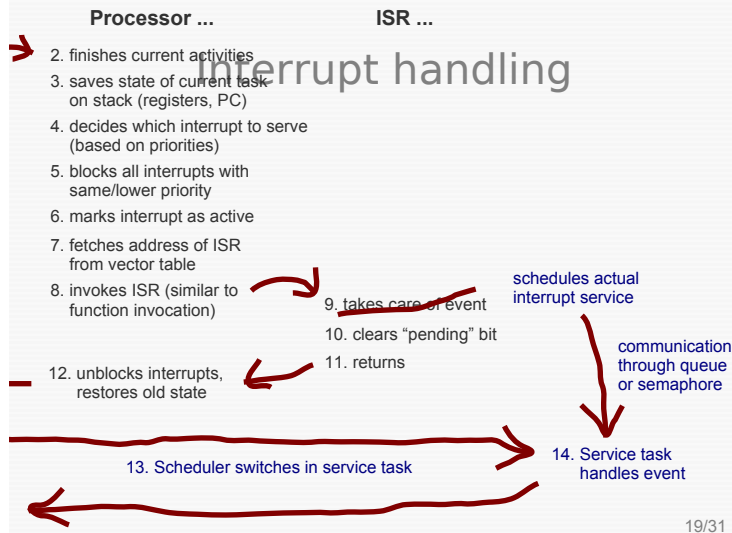
- Avoiding interrupts (*more later*)
- **Deferred/split-interrupt handling**
 - Common in most OSs, not only in real-time systems

17/31

Deferred interrupt handling

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

18/31



19/31

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

20/31

FreeRTOS example (2)

- External interrupt line, connected to `PORTA.0`

21/31

Why interrupts?

22/31

Why interrupts

- Keep system load low when nothing is happening
- Fast responses to events
- Legacy
- *Advantages in real-time systems?*

23/31

Event- vs. time-triggered syst.

Event-triggered

- Reacting to interrupts (e.g., when pin value changes)
- More dynamic
- More efficient
- Best-effort, might fail for high loads

Time-triggered

- Regular checking/polling (e.g., check pin every 10ms)
- More static
- Simpler scheduling, better predictability

Most systems mix both paradigms

24/31

Different System architectures/ Multi-tasking approaches

25/31

- main with single big loop containing all functionality of system

```
int main(void) {  
    // initialisation  
    while (1) {  
        // read inputs  
        // do computations  
        // write outputs  
    }  
}
```

- Standard method without OS
- Simple, but not very scalable
→ System with multiple tasks?

26/31

Fully pre-emptive multi-tasking

- Easy to handle continuous/long-running tasks
- Very responsive programs
- Care required when dealing with critical regions, mutual exclusion
- Standard setup with FreeRTOS
- Context switches can be “expensive” (include systick interrupt)

27/31

Cooperative multi-tasking

- Task-switching points have to be specified manually in long-running tasks (FreeRTOS: `taskYIELD`)
- Tasks are also switched out when they are blocked (say, delayed or waiting)
- Enabled in `FreeRTOSConfig.h`
`#define configUSE_PREEMPTION 0`

28/31

Cooperative multi-tasking (2)

- Simpler than pre-emptive multi-tasking
 - no trouble with critical regions, etc.
 - simple OS kernel
 - potentially more efficient than pre-emptive multi-tasking
- Less responsive than pre-emptive multi-tasking if too few `taskYIELD`

29/31

Hybrid multi-tasking

- Cooperative multi-tasking + “manual” pre-emption when necessary (by calling `taskYIELD` from ISR)
- Pre-emption no longer in fixed intervals, only on demand (when event occurs)
- Potentially efficient, but also error-prone

30/31

Co-routines (in FreeRTOS)

- Kind of cooperative multi-tasking
- Inactive tasks lose (most of) their state
 - Stack of inactive tasks is removed
 - Local variables lose their values
- Useful for devices with very little RAM, or if *many* tasks need to be created
- <http://www.freertos.org/croutine.html>
- Similar features in Erlang, Scala

31/31