Programming Embedded Systems

Lecture 5
Interrupts,
modes of multi-tasking

Monday Jan 31, 2011

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

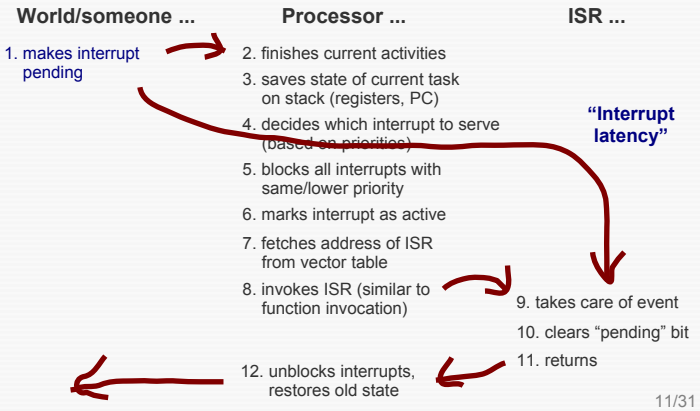
Setting up interrupts

- 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



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"

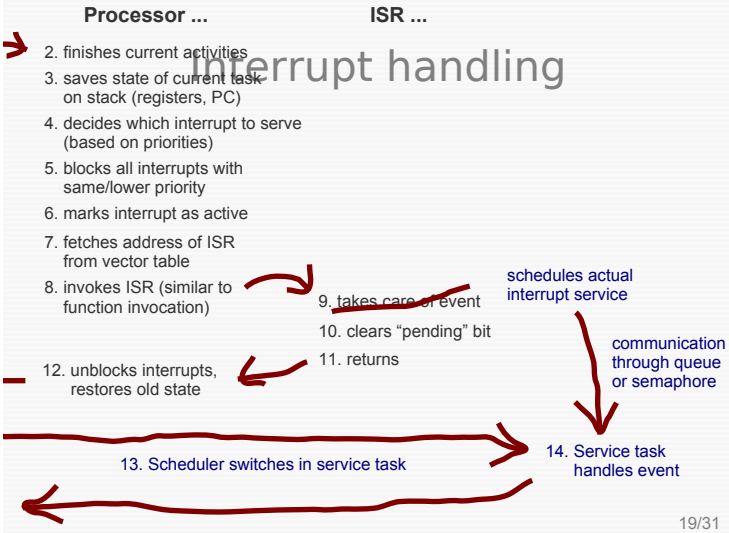
Solutions

- 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



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

Why interrupts?

Why interrupts

- Keep system load low when nothing is happening
- Fast responses to events
- Legacy

- *Advantages in real-time systems?*

Event- vs. time-triggered syst.

Event-triggered	Time-triggered
• Reacting to interrupts (e.g., when pin value changes)	• Regular checking/polling (e.g., check pin every 10ms)
• More dynamic	• More static
• More efficient	• Simpler scheduling, better predictability
• Best-effort, might fail for high loads	

Most systems mix both paradigms

Different System architectures/ Multi-tasking approaches

Super-loop

- `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?

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)

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
```


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`

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

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