

---

# Interrupts

# Interrupts and CPU Types

---

## **All CPU types provide some sort of interrupt**

- It usually is an electrical signal reaching the CPU
- Sometimes (e.g. 8086) there is one wire and one protocol
- More often there are multiplexing registers
  - ♦ You have a single interrupt entry point
  - ♦ Then a status register tells which one is active
  - ♦ Possibly a second level register details more
- Nowadays we have MSI too: Message Signalled Interrupts
  - ♦ They exist in PCI, to simplify hardware routing
  - ♦ They are the preferred IRQ form in PCI Express

## **Most simple firmware code bases use no interrupt at all**

- They rely on cooperative multitasking
- Or they are just a "while(1)" loop calling some action

# Interrupts and Traps

---

## **Interrupt, Trap, Abort, System Call, Breakpoint, ...**

- All of these are basically the same:
  - ♦ Normal program execution is interrupted,
  - ♦ Interrupts are disabled and CPU state changes
  - ♦ Execution resumes at a different location
  - ♦ You need some special instruction to "return"

## **Usually, we call "Interrupt Request" (IRQ) something external**

- A device driver requires attention
- A timer expired
- A network or USB frame arrived

## **And we call "Trap" sth that depends on the program flow**

- Illegal Instruction
- Division by zero
- Illegal pointer

## **Software interrupts are just the same**

- Special instructions that force a state change in the CPU

# Interrupts in ARM

---

## The ARM architecture has a special approach to traps

- The CPU jumps to a specific address (0..0x1c)
- The CPU changes its internal "mode"
- Some registers are swapped to a different "bank"
- The PC is set to the "vector" address (0x00..0x1c)

Thus, no automatic memory access is performed

```
b          reset
ldr       pc, _undefined_instruction
ldr       pc, _software_interrupt
ldr       pc, _prefetch_abort
ldr       pc, _data_abort
ldr       pc, _not_used
ldr       pc, _irq
ldr       pc, _fiq
```

**This approach simplifies hardware but makes software not trivial**

- Not a problem, usually, when you code this once only

# Interrupts in Cortex-M0 (LPC11)

---

## **The NVIC (Nested Interrupt Controller) is part of Cortex-M0**

- It is a peripheral device like times and I/O ports
- But ARM includes it in the processor definition
- The reset/interrupt mechanism is highly coupled with it

## **At address 0, we find the interrupt vectors**

- They are vectors (pointers), not instructions
  - ♦ The CPU pushes processor state to the stack
  - ♦ It then fetches the vector to the program counter
  - ♦ What happen is similar to a function call

## **16 Core vectors**

- 0: initial stack pointer
- 1: reset vector
- 2..15: more predefined traps

## **32 (or 64) SoC-specific vectors**

- LPC1135 and similar ones use 32 vectors
- They are internal timers and peripheral devices

**See chapters 6 and 24.3 of the uC manual (it's committed)**

# Interrupts and Tasks

---

## Usually, timer interrupts are used for preemption

- For example, both RM and EDF rely on preemption
- In the simplest implementation, you fire a periodic timer
  - ♦ Every time the timer ticks, you make a scheduling choice
  - ♦ The NVIC even includes it's own "system tick" IRQ
- Unfortunately, a periodic interrupt is a waste of CPU time
  - ♦ 20us every ms is 2% of CPU power
  - ♦ With a slower clock, it can even be much worse
  - ♦ You can't raise your HZ if you interrupt at every tick
- Linux deprecates the periodic interrupt since at at least 2011

**And preemption brings in semaphores, spin locks, and more**

## Device interrupts are used to restarted a stopped task

- An interactive shell waiting for the serial port
- An SPI data transfer waiting for a reply

**But this creates the need for timeouts and recovery**

# Bare-Metal support for Interrupts

---

**As usual, the designer faces a choice. Options are:**

- **No interrupts**
  - ♦ **Single-task or cooperative multi-tasking**
  - ♦ **Easiest and safe approach**
  - ♦ **Allows jitterless operation, if well done**
  - ♦ **Doesn't scale up to complex situations**
- **Single-interrupt system**
  - ♦ **The critical "task" is jitterless (the rest is not)**
  - ♦ **The rest of the system works as before**
  - ♦ **You can manage a lockless protocol for data-sharing**
- **Full peripheral interrupts**
  - ♦ **UART, SPI, timers, gpio, ...**
- **System tick alone**
  - ♦ **With polling I/O in each task**
- **Full preemption and interrupt-driven stuff**
  - ♦ **Like a desktop/server system**

# Some Considerations

---

## **No-interrupts systems are very simple to debug**

- We already have a time base without interrupts
  - ◆ If the uC allows: see the AVR port
- Don't underestimate the joy of lockless code

## **Single-interrupt systems allow precise data collection**

- You don't need to care much about printf/USB latencies
- Still, the overhead of irq enter/exit can be high

## **The scheduler can introduce substantial overhead**

- A few microseconds or more

## **Device interrupts can destroy your well-determined WCET**

## **When you introduce priorities, you face all kinds of problems**

- Deadlocks
- Priority inversion
- Stale tasks
- ...



# A lockless circular buffer

---

**Most, but not all, single-interrupt systems collect or emit data with a predefined rate**

**You usually need a circular buffer for your data items**

**The trivial implementation uses head+tail pointers or offsets**

- But when `head == tail`, is the buffer full or empty?
- And how can you detect overflows and underflows?

**Usually, people uses a mutex primitive to access head and tail**

**But we can do better. A lockless circular buffer is possible**

- Please think about the problem and offer a solution

# Splitting interrupt handlers

---

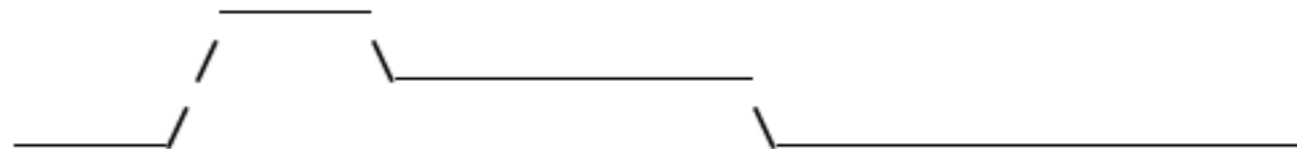
## "Real" operating system had to face the IRQ mess

- problem "S": the CPU can serve a single interrupt at a time
- problem "P": interrupts disrupt predictable (and RT) response

## The first approach is splitting the interrupt handling

- A "top half" handler does the very minimum stuff
  - ♦ It communicates with hardware, and acknowledges the interrupt
- A "bottom half" handler deals with OS data structures
  - ♦ This happens with interrupts enabled

IRQ (irq disabled in CPU)  
bottom half (irq enabled)  
process context (enabled)



## This mitigates problem S above, but not problem P

- And the bottom half still cannot sleep or schedule
- It runs in a privileged context, where the process is stalled

# Threaded Interrupts

---

## **To make RT predictable again we can downgrade interrupts**

- If the interrupt is a thread (a process) it can be scheduled
- And it can be prioritized above or below normal tasks

`https://lwn.net/Articles/302043/ (a.d. 2008)`

## **It's simpler done than said**

- The top half disables the interrupt and awakes the IRQ thread
- The IRQ thread (process) serves the interrupt and enables it again

## **This means your top-priority task as a predictable WCET**

- It is it's real (cpu-intensive) WCET
- Plus one interrupt time (1-2 us) for each device

**Network bursts or other massive I/O won't interfere any more**