

# 08: Addressing, real-time systems

## Microcontrollers

Stefan Huber

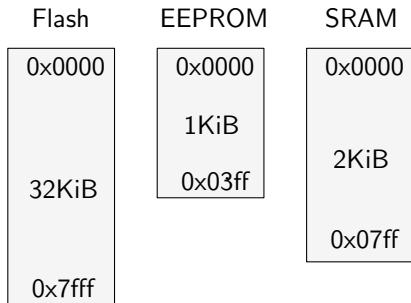
Dept. for Information Technologies and Digitalisation  
FH Salzburg

Summer 2025

# Addressing

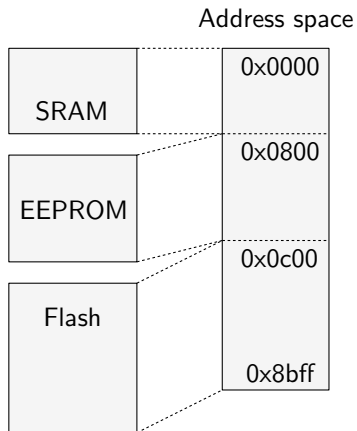
# Memory addressing

The ATmega32 uses **separate memory addressing** for its memory:



- ▶ Separate address spaces.
- ▶ Different access methods (instructions, registers) tell which is meant.

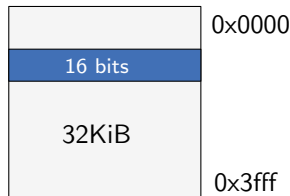
Other systems map them into a common **address space**:



# Byte- versus word-addressed

The ATmega32 flash memory is organized as  $16\text{ Ki} \times 16$  memory.

- ▶ The Flash is **word-addressed** with a word having 16 bits.
- ▶ The ATmega32 Flash memory addressing actually looks like this:



Rationale behind this design:

- ▶ Most AVR instructions are 16 bits wide: A word per instruction.
- ▶ The PC is 14 bits wide and can therefore address  $2^{14} = 16384 = 16\text{ Ki}$  program memory locations.
  - ▶ A PC of  $0x13$  translates to a byte address of  $0x26$ .
  - ▶ The PC cannot address an odd byte address by design!

# Byte order: Endianness

There are two possibilities for word-wise access to byte-addressed memory:

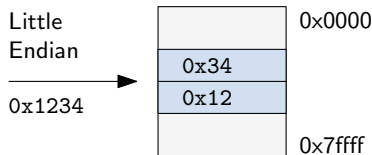
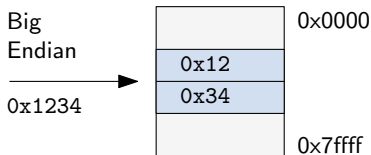
**Big Endian** High byte first (at the lower address).

**Little Endian** Low byte first (at the lower address).

Same for putting words on a network.

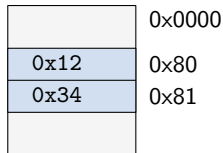
► “Lower address” means first in the byte stream.

Let us put the word `0x1234` in memory or on network:



# Byte order: Endianness

Memory (or network):



```
uint16_t* p = 0x80;  
// Prints 0x1234  
printf("%x", *p);
```

Big endian computer

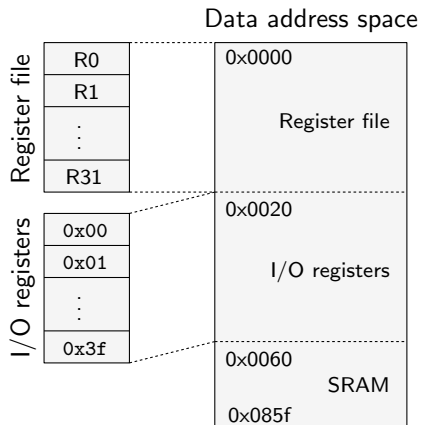
```
uint16_t* p = 0x80;  
// Prints 0x3412  
printf("%x", *p);
```

Little endian computer

## Examples

- ▶ Little Endian: Intel x86, ATmega32 with avr-gcc
- ▶ Big Endian: Motorola 68k, Internet protocol suite (see C-functions `htonl()`, ...)
- ▶ ARM (since version 3) support [bi-endianess](#), they can set endianness.

# ATmega32 data memory map



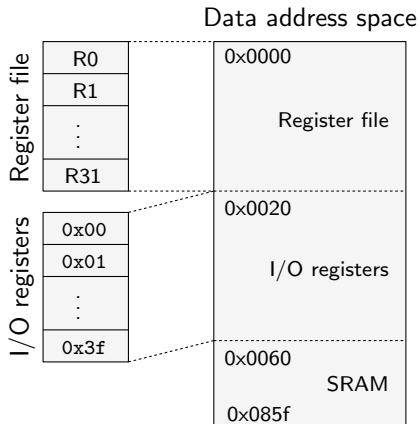
The register file and the I/O registers are mapped into the data address space.<sup>1</sup>

- ▶ The I/O registers encompass SREG, SPH, SPL and control registers like PORTA, OCR1, GICR and so on.<sup>2</sup>
- ▶ The 2 KiB SRAM starts at address 0x0060.

<sup>2</sup> [ATmega32, p. 17]

<sup>2</sup> [ATmega32, p. 327]

# ATmega32 data memory map



The register file and the I/O registers are mapped into the data address space.<sup>1</sup>

- ▶ The I/O registers encompass SREG, SPH, SPL and control registers like PORTA, OCR1, GICR and so on.<sup>2</sup>
- ▶ The 2 KiB SRAM starts at address 0x0060.

## Memory mapped I/O:

- ▶ Ordinary memory access instructions instead of special instructions or additional registers.
- ▶ For avr-gcc, PORTA is just a byte at address 0x003b, or literally: `*(volatile uint8_t *)((0x1B) + 0x20))`

<sup>2</sup> [ATmega32, p. 17]

<sup>2</sup> [ATmega32, p. 327]



# Real-time systems

# A real-time system

A real-time system is a computational system for which the **correct execution** not only depends on the logical correctness of the output, but also whether the output is **computed in time**.

Examples:

- ▶ Fly-by-wire in air planes or obstacle detection in autonomous driving.
- ▶ The control loop in a control system, the cyclic motion planing of a drive or robot.
- ▶ Live audio- and video-processing.

# Typical misconceptions

~~A real-time system would be a *fast* system.~~

- ▶ Real-time is about meeting **deadlines**, not about high throughput.
- ▶ Real-time is about computing **always as fast as required** (guaranteed timeliness) and not about as fast as possible on average (best effort).

Achieving **temporal determinism** in a real-time system comes at costs, which typically makes a real-time system achieve less average throughput.

Depending on the context, sometimes “real-time” refers to “live”, e.g., as fast as the physical time.<sup>3</sup>  
This may or may not imply real-time as in real-time systems.<sup>4</sup>

---

<sup>3</sup> E.g., playing a video or running a simulation live.

<sup>4</sup> And sometimes it is nothing more than advertisement.

# Rare events and peak load

A real-time system can be slow, but it must not miss a deadline even at a system's **peak load**.

- ▶ A **rare event** – e.g., failure of a component – can cause many related requests – e.g., an alarm shower – and lead to a peak load.
- ▶ For real-time systems we need to analyze the peak load, including fault scenarios.

# Different types of real-time

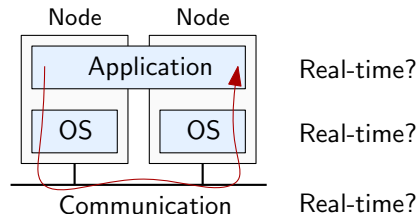
Three categories depending on the **consequence** of missing a deadline:

- Soft** If we miss the deadline the output can still be used. The user experience is degraded, e.g., video streaming. The deadline is a **soft deadline**.
- Firm** The output after the deadline is of no utility, e.g., too late object detection on a moving conveyor line. Infrequent deadline violations degrade quality of service but are tolerable.
- Hard** Missing a **hard deadline** is a system failure and can be catastrophic. Typical for safety critical systems and embedded systems.

# Distributed real-time systems

Many embedded systems are actually **distributed**, **cyber-physical**, **real-time systems**:

- ▶ **Distributed**: A collection of autonomous nodes together achieve a common goal, e.g., the ECUs in a car or the drives and controllers that automate an industrial machine.
- ▶ Interacting with the physical world often imposes real-time constraints. E.g., a closed-loop controller for an industrial process must not be late.



The entire signal path (e.g., for a syscall, for a RPC) must be real-time capable.

Literature:

- ▶ **Hermann Kopetz**. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 2nd. Springer Publishing Company, Incorporated, 2011. ISBN: 9781441982360

A distributed system requires (network) communication.

- ▶ Hence, a distributed real-time system requires **real-time communication**.
- ▶ In industry, time-triggered Ethernet-based networking protocols became standard, e.g., TTEthernet, Powerlink, Varan, Ethercat, TSN.
- ▶ On the host, we need network interfaces that are real-time capable. For instance, we need ethernet controllers<sup>5</sup> that allows for scheduled packet transmission. In Linux, there is the socket option `SO_TXTIME` which allows for scheduled packet transmission.

Not in the focus of this lecture.

---

<sup>5</sup> Like the Intel i210.

# Real-time operating systems

A real-time operating system has the following requirements:

- ▶ Predictable temporal behavior of system calls, e.g., concerning scheduling and memory management.
- ▶ Predictable response time upon events, e.g., when a GPIO toggles, a network packet arrived, a timer is due.
- ▶ Temporal isolation between processes, i.e., preemptive multitasking



# Worst-case execution time

For a given task, function or piece of code we define the **worst-case execution time** (WCET) as the maximum length of *physical* time required.

A deadline for a task can only be met if the WCET of the task is bound.

- ▶ Real-time algorithms are therefore  $O(1)$  algorithms (or the input size is bound).
- ▶ For instance, if we do dynamic memory allocation in real-time code then we need a constant-time memory allocator, such as TLSF [TLSF].

# How to find the WCET?

## Unpredictability of contemporary processors

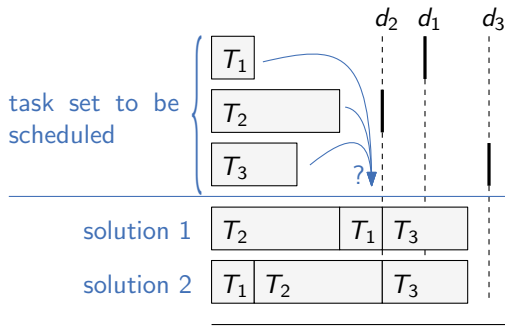
- ▶ Processors caches, including simultaneous access from multiple cores
- ▶ Virtual memory management, prefetching and speculative execution, instruction reordering
- ▶ Shared access to I/O and memory buses
- ▶ Power-saving strategies and system-management mode (SMM)

Hence, predicting the physical time spent on a LOC in C, or even a single machine instruction, is virtually impossible.

- ▶ Even if we could, WCET would be hopelessly pessimistic.
- ▶ This is an unsolved problem and makes engineering of real-time systems a delicate endeavor; a clean WCET analysis is typically not performed. Instead, margins – like 20 % – are added to measurements, although this is theoretically unjustified.
- ▶ **Cache locking** improves predictability and somewhat simplifies WCET analysis.

# Real-time scheduling

- ▶ A **real-time scheduler** schedules real-time tasks  $T_i$  in a way such that they all meet their respective **deadlines**  $d_i$ .
- ▶ Multi-core real-time scheduling is *much* harder than single-core real-time scheduling.

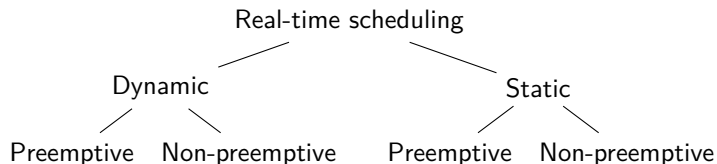


In this example,  $T_1$  and  $T_2$  cannot be scheduled last, because either of them will miss its deadline. Hence,  $T_3$  needs to be last, and we are left with two possibilities.

But how does an algorithm look like to solve the problem in general?

Figure: Two possible single-core schedules for tasks  $T_i$  with deadlines  $d_i$ .

# Classification of scheduling algorithms



**Dynamic** A dynamic scheduler is an online scheduler: Scheduling decisions are made during runtime. It is flexible, adaptable, but more complex. Making a scheduling decision can be computationally costly.

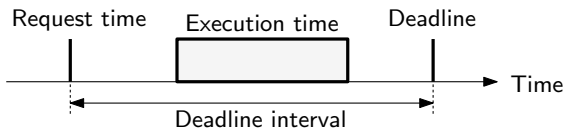
**Static** A static scheduler makes the schedule during compile time. It needs to know the task characteristics (WCET, interdependencies, mutex, deadlines) a priori. It is simple but not flexible. Typical for industrial control systems.

- ▶ Non-preemptive schedulers are simpler and might be used in constrained systems, but they do not provide temporal isolation.
- ▶ Linux provides a dynamic, preemptive real-time scheduler.

# Real-time tasks

The time when we request the execution of a task is called **task request time**.

- ▶ The difference between deadline and request time is called **deadline interval**.
- ▶ The **laxity** is the difference between deadline interval and the execution time. For schedulability the laxity must be non-negative.
- ▶ For analysis purposes, we have to assume that the execution time is the WCET.



Two types of tasks:

**Periodic** The initial request time determines all future request times a priori by adding multiples of a **period**.

**Sporadic** The task request times are not known a priori.

# Dynamic real-time scheduling

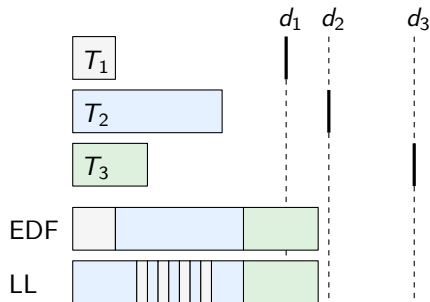
Both are based on **dynamic priorities**, i.e., scheduling priorities that change over time.

**EDF** The earliest deadline first (EDF) algorithm gives the task with the earliest deadline the highest priority. Note that we do not need to know the execution time for EDF.

**LL** The least laxity (LL) algorithm gives the task with the shortest laxity the highest priority.

Assumptions:

- ▶ Tasks are independent of each other; no precedence constraints.



- ▶ Laxity stays constant for the task that is scheduled and declines for those not being scheduled.
- ▶ Least laxity leads to frequent context switches when two tasks reach the same laxity, as they repeatedly relieve each other.
- ▶ This makes least laxity impractical.

## Optimality

A scheduler is called optimal when it can find a schedule (given one exists).<sup>6</sup>

One can prove that on uni-processor systems and with periodic tasks both, EDF and LL, are optimal.

- ▶ On multi-core systems they are both not optimal. But LL can find schedules where EDF fails.
- ▶ Real-time scheduling on multi-core systems is *very* hard.

Can we test whether a task set is schedulable?

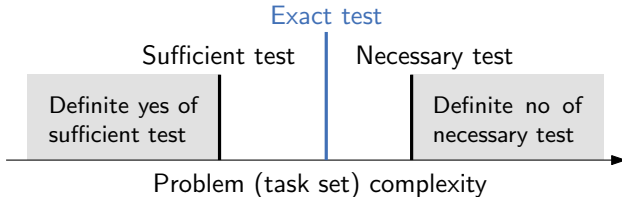
- ▶ We would like to test whether we can accept a new task to be scheduled.

<sup>6</sup>

To be precise, they find a schedule if a so-called clairvoyant scheduler – which knows all future request times – can find one.

# Schedulability tests

- ▶ An **exact schedulability test** answers “yes” if there exists a schedule and “no” if not. Depending on setups this problem can easily be NP-complete, so no polynomial time algorithms exist.<sup>7</sup>



- ▶ A **sufficient schedulability test** is simpler: It answers “yes” if there definitely exists a schedule. Otherwise a schedule may or may not exist.
- ▶ A **necessary schedulability test** is the opposite: It answers “no” if there definitely does not exist a schedule. Otherwise a schedule may or may not exist.

<sup>7</sup>

Unless the most famous open problem “P vs. NP” in computer science turns out to be  $P=NP$ .



# Utilization

We consider a set of **periodic tasks**  $T_1, \dots, T_n$ , where  $T_i$  has a period  $p_i$  and execution time  $c_i$ .

The **utilization**  $\mu$  is defined by

$$\mu = \sum_{i=1}^n \frac{c_i}{p_i}.$$

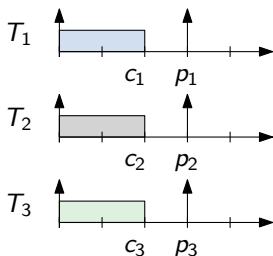
Interpretation: A single task  $T_i$  utilizes a single core for a fraction of  $\frac{c_i}{p_i}$ . All tasks together utilize a single core for a fraction of  $\mu$  of its time.

# A necessary schedulability test

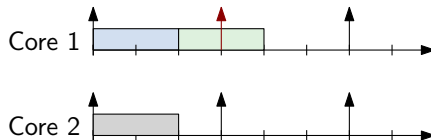
A **necessary schedulability test** for a  $m$ -core system is

$$\mu \leq m.$$

This test is **not sufficient**. Here we have tasks  $T_1, T_2, T_3$  with period  $p_i = 3$  and  $c_i = 2$  for  $1 \leq i \leq 3$ . Although  $\mu = 3 \cdot 2/3 = 2$ , we cannot schedule them on a 2-core machine:<sup>8</sup>



Some task violates its deadline.



<sup>8</sup>

If we could parallelize tasks, we could utilize all cores. But this would mean we split  $T_3$  into two tasks with execution time of 1 each, and we only changed our counter example to fix it.

# Dhall's effect

## Dhall's effect

Choose an arbitrarily large number  $m$  of cores. Then there is a set of tasks with  $\mu \approx 1$  that is non-schedulable with EDF. Even if deadlines equal periods.

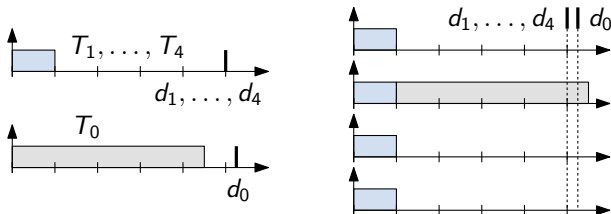
# Dhall's effect

## Dhall's effect

Choose an arbitrarily large number  $m$  of cores. Then there is a set of tasks with  $\mu \approx 1$  that is **non-schedulable with EDF**. Even if deadlines equal periods.

Example:

- ▶ A task  $T_0$  with deadline  $d_0 = 1 + \epsilon$ , with  $\epsilon > 0$  arbitrarily small and an execution time  $c_0 = 1 - 2\epsilon$ .
- ▶ Tasks  $T_1, \dots, T_m$  with deadline  $d_i = 1$  and  $c_i = 4\epsilon$ . So we have  $\mu = 4\epsilon m + \frac{1-2\epsilon}{1+\epsilon} \approx 1$ .



- ▶ EDF schedules  $T_1, \dots, T_m$  first. Hence,  $T_0$  ends at time  $1 + 2\epsilon$ , after its deadline.

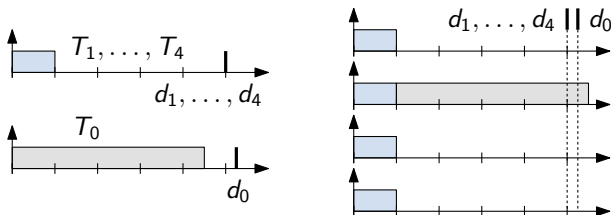
# Dhall's effect

## Dhall's effect

Choose an arbitrarily large number  $m$  of cores. Then there is a set of tasks with  $\mu \approx 1$  that is **non-schedulable with EDF**. Even if deadlines equal periods.

Example:

- ▶ A task  $T_0$  with deadline  $d_0 = 1 + \epsilon$ , with  $\epsilon > 0$  arbitrarily small and an execution time  $c_0 = 1 - 2\epsilon$ .
- ▶ Tasks  $T_1, \dots, T_m$  with deadline  $d_i = 1$  and  $c_i = 4\epsilon$ . So we have  $\mu = 4\epsilon m + \frac{1-2\epsilon}{1+\epsilon} \approx 1$ .



- ▶ EDF schedules  $T_1, \dots, T_m$  first. Hence,  $T_0$  ends at time  $1 + 2\epsilon$ , after its deadline. **And LL?**

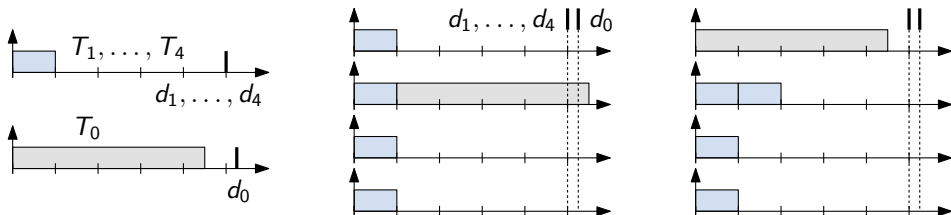
# Dhall's effect

## Dhall's effect

Choose an arbitrarily large number  $m$  of cores. Then there is a set of tasks with  $\mu \approx 1$  that is **non-schedulable with EDF**. Even if deadlines equal periods.

Example:

- ▶ A task  $T_0$  with deadline  $d_0 = 1 + \epsilon$ , with  $\epsilon > 0$  arbitrarily small and an execution time  $c_0 = 1 - 2\epsilon$ .
- ▶ Tasks  $T_1, \dots, T_m$  with deadline  $d_i = 1$  and  $c_i = 4\epsilon$ . So we have  $\mu = 4\epsilon m + \frac{1-2\epsilon}{1+\epsilon} \approx 1$ .



- ▶ EDF schedules  $T_1, \dots, T_m$  first. Hence,  $T_0$  ends at time  $1 + 2\epsilon$ , after its deadline. **And LL?**

- [ATmega32] *ATmega32: 8-bit AVR Microcontroller with 32KBytes In-System Programmable Flash.* Atmel Corporation. Feb. 2011.
- [Kop11] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* 2nd. Springer Publishing Company, Incorporated, 2011. ISBN: 9781441982360.
- [TLSF] *Two Level Segregate Fit.* URL: <http://www.gii.upv.es/tlsf/>.