# Embedded systems

Bernard Boigelot

E-mail : Bernard.Boigelot@uliege.be
WWW : https://people.montefiore.uliege.be/boigelot/
https://people.montefiore.uliege.be/boigelot/courses/embedded/

*References* :

- *An Embedded Software Primer*, David E. Simon, Addison-Wesley, 1999.
- *Mastering the FreeRTOS Real-Time Kernel - A Hands-on Tutorial Guide*, version 1.1.0, Richard Barry, 2024.
- *Real-Time Systems*, Jane W. S. Liu, Prentice Hall, 2000.

# Chapter 1

# Introduction

# Embedded systems

Definition: An embedded system is a computer system used as a component of a more complex entity.

Typical applications:

- mobile phones, televisions, radios, multimedia systems, GPS receivers;
- cameras;
- wristwatches, calculators, smart cards, RFID tags, remote controls;
- home appliances;
- computer peripherals;
- measurement equipment, sensors;
- cash dispensers, self-service machines;
- medical devices, implants;

- elevators, intrusion-detection devices, domotic systems;
- telephone switches, network routers;
- automotive systems (ABS, ESP, injection controllers, adaptive cruise control, lane keeping assist, autonomous drive, . . . );
- avionics (fly-by-wire controls, glass cockpits, navigation aids, TCAS, autoland, . . . );
- industrial process controllers, robots;
- artificial satellites, spatial probes;
- . . .

# Advantages

- Moving some functionalities from hardware to software makes electronic circuits
  - simpler,
  - cheaper to build,
  - more powerful.
- Complex features can be implemented.
- Software components can easily be updated during the lifetime of a product, as well as reused in other projects.

# Developing embedded systems: Main difficulties

- Low-performance hardware: Low computing power, small amount of memory ...
- Specificity to a particular application.
- Concurrence: Several tasks operate in parallel.
- Reactivity: The system must constantly be able to answer solicitations.
- Real-time constraints.
- High level of quality expected: Reliability, robustness and efficiency are critical.
- Limited user interface.
- Adverse exploitation environment.
- Energy management is often necessary.

# Chapter 2

# Hardware

# Main components of an embedded system

- One or several processor(s) (CPU):
  - Microcontrollers (MCU): 8051 (Intel), PIC, AVR (Microchip), ...
  - Digital Signal Processors (DSP): TMS320 (Texas Instruments), SHARC (Analog Devices), MSC81xx (NXP), ...
  - Microprocessors dedicated to embedded applications: ARM [v7, v8, v9], RISC-V, ColdFire (NXP), ARMADA (Marvell), PowerPC (IBM, NXP, STMicroelectronics), x86, x86-64 (Intel, AMD), ...
  - Generic microprocessors: Snapdragon (Qualcomm), Xeon, Core, Atom (Intel), ...
  - Special architectures: Java Card, multicore processors, reconfigurable processors, ...

- Memory:
  - Static or dynamic RAM, ROM (EEPROM, FLASH, . . . ).
  - Either internal to the microcontroller, external, or integrated in a System on Chip (SoC).
  - Parallel or serial interface.
  - Possibility of addressing peripherals in memory.
- Internal or external peripherals:
  - Timers,
  - Converters,
  - Communication controllers,
  - . . .
- Communication buses.

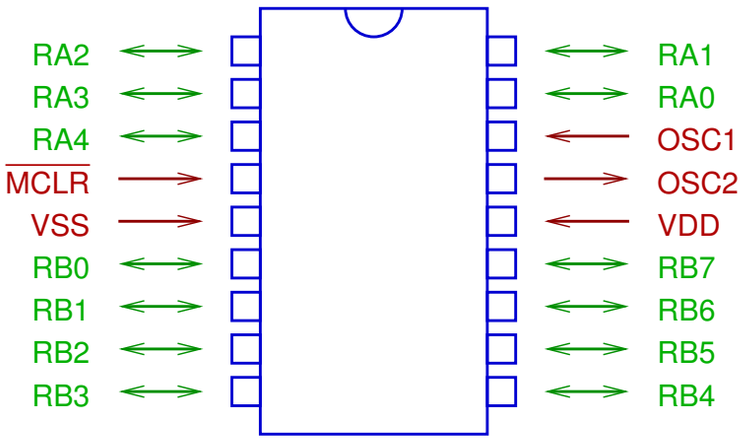
- Interfaces with the circuit environment:
  - Point to point: RS-232, IR, NFC, ...
  - Buses: $I^2C$, SPI, CAN, USB, JTAG, ...
  - Networks: Ethernet, Wi-Fi, ZigBee, Bluetooth,...
- Auxiliary components:
  - Power supply,
  - Clock generator,
  - Bus controllers,
  - ...

# Example of embedded microcontroller: Microchip PIC16F716

Main features:

- RISC architecture: Only 35 instructions.
- Harvard memory model: 2048 words of (FLASH) program memory, 128 bytes of (RAM) data memory, both internal.
- Reprogrammable via a serial interface.
- 13 dynamically configurable general-purpose input/output pins.
- Integrated peripherals: 3 timers, PWM controller, analog to digital converter, ...
- Computing power of $5.10^6$ instructions per second.
- Low power consumption: About $120\,\mu$A (under 2V) at $1$ MHz, $14\,\mu$A at $32$ kHz, $100$ nA in standby.

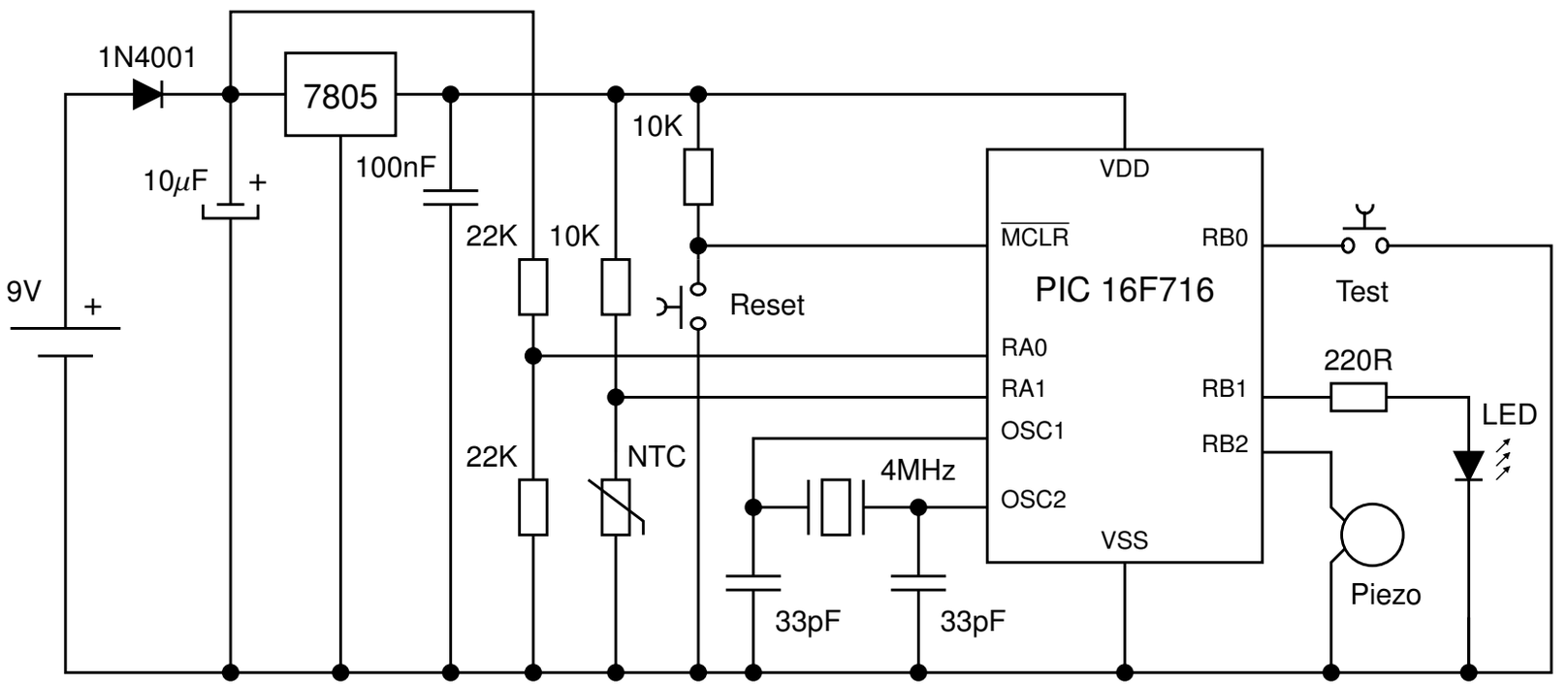
## Pinout:




## Description:

- VSS, VDD : Power supply (2.0–5.5 V).
- OSC1, OSC2 : Oscillator crystal or external clock source.
- $\overline{\text{MCLR}}$ : Operating mode selection (0 V: reset, VDD: program execution, 13 V: programming mode).
- RA0. . . RA4, RB0 . . . RB7 : General-purpose input/output pins (TTL/CMOS), dynamically configurable and multiplexed with some peripherals. RB6 and RB7 alternatively provide a serial interface in programming mode.

# Example of application: Temperature alarm



# Example of embedded bus

Problem: Managing data transfers between several devices (CPU, memory, sensors, peripherals, . . . ) using communication hardware that is as simple as possible.

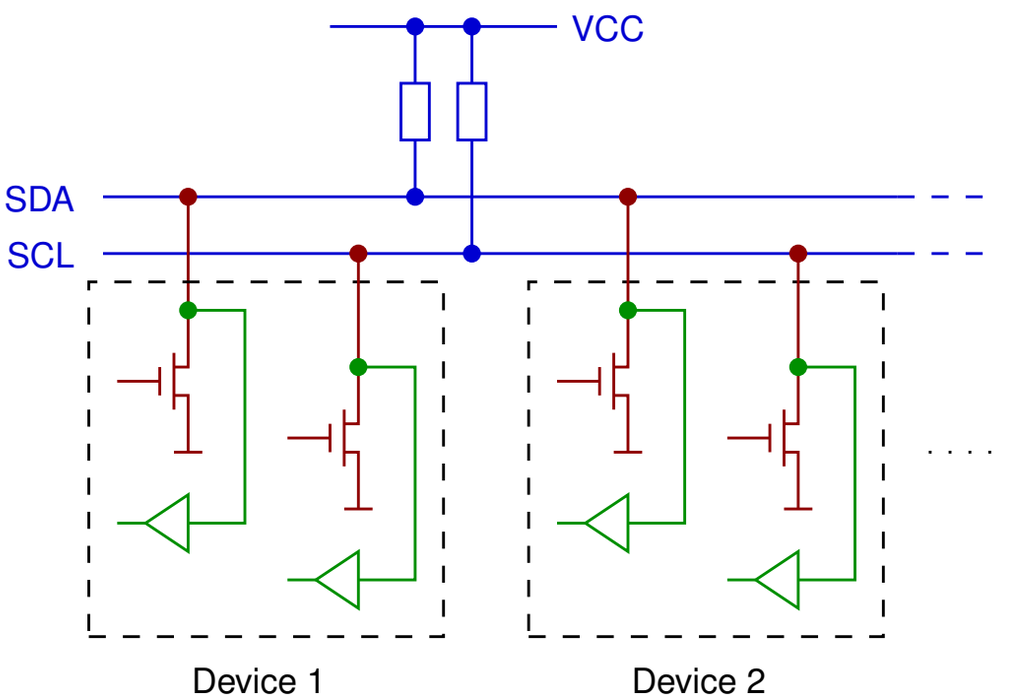
Requirements:

- Bus topology.
- Small number of communication lines.
- Flexible configuration.
- Mechanisms for addressing devices, managing transactions, for performing arbitration and flow control.

# Solution: $I^2C$ bus

Principles:

- The bus consists of a pair of two-way lines: SDA (Serial DAta) and SCL (Serial Clock).
- The value of each line stays high whenever it is unused.
- Each device connected to the bus can read the value of SDA and SCL, but is only able to force them down, i.e., to write a low value.



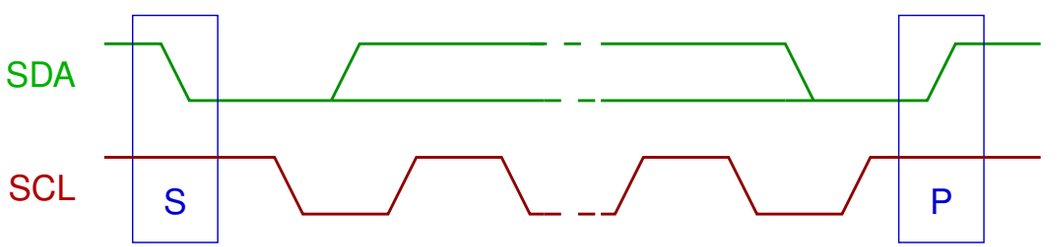
# $I^2C$: Transactions

The master of a transaction is responsible for

- generating a clock signal on SCL during the transaction.
- signaling the beginning (Start, S) and the end (Stop, P) of the transaction. The signals S and P correspond to the two possible transitions of SDA when SCL is high.

When a transaction is in progress (i.e., between S and P), transitions of SDA are only allowed when SCL is low.

Illustration:



# $I^2C$: Data transfers

- During a transaction, the sender of data can either be the master or the slave.
- The value of each bit of data sent on the bus corresponds to the value of SDA during a low-to-high transition of SCL.
- Data is exchanged in 8-bit groups, the most significant bit (MSB) being sent first.
- Each group of 8 bits must be followed by an acknowledgment, represented by a low value placed on SDA by the receiver.

  If a group of bits is not acknowledged, then the master immediately aborts the transaction, and the slave stops sending or receiving data.

# $I^2C$: Addressing

When a transaction is initiated, the master has to specify which device is the other participant.

Principles:

- The first 8 bits exchanged in a transaction are always sent by the master.
- The first 7 bits of this group correspond to the address of the intended slave.
- The 8th bit then specifies the direction of the following data transfer:
  - 0 : The master is the sender;
  - 1 : The master is the receiver.

Remark: The first group of 8 bits must thus be acknowledged by the addressed slave, regardless of the data transfer direction.

# $I^2C$: Arbitration

It is possible to have several devices attempting to initiate transactions at the same time, by generating simultaneous Start signals.

For detecting potential conflicts, each master constantly monitors the value of SDA when it sends data. If the observed value differs from the sent one, then the master performing this observation immediately and silently withdraws from the transaction.

Remarks:

- A conflict can only be detected by the device that sends a high value.
- Transmitting simultaneously two exactly identical frames does not lead to a conflict!

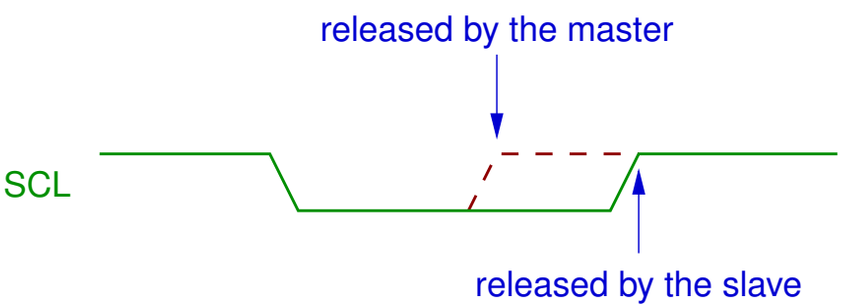
# $I^2C$: Flow control

In some cases, the frequency of the clock signal generated is too high to be followed by the slave.

In such situations, the slave can request the master to permanently or temporarily slow down the clock. This can be done by stretching the low value of SCL until the slave is ready again to send or receive data.

When the master releases SCL while the clock is stretched, it detects that the value of SCL stays low, and pauses its operations until this line is released by the slave.

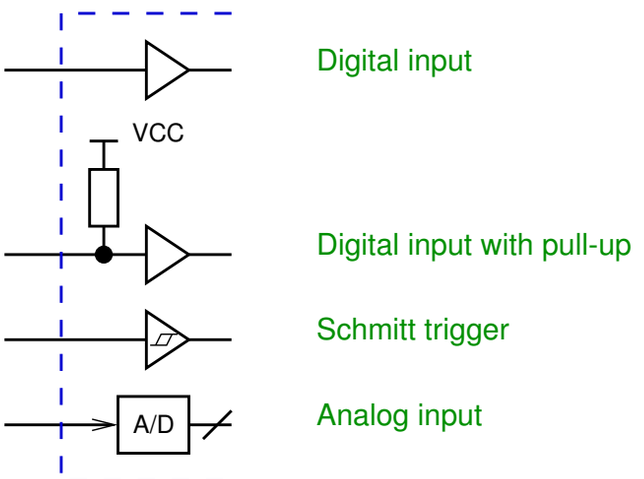
Illustration:



# Multiplexed input/output pins

Most microcontrollers allow to dynamically configure input/output pins in software.

## Examples of typical configurations:

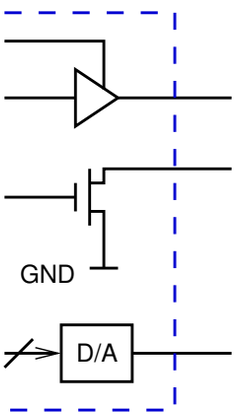



Digital input

Digital input with pull-up

Schmitt trigger

Analog input




Three-state output

Open-drain output

Analog output

This feature makes is possible to build simple circuits in which the processor can interact with a large number of peripherals.

## Example: Digital multimeter

The problem is to interface a microcontroller offering only 12 dynamically configurable input/output pins with:

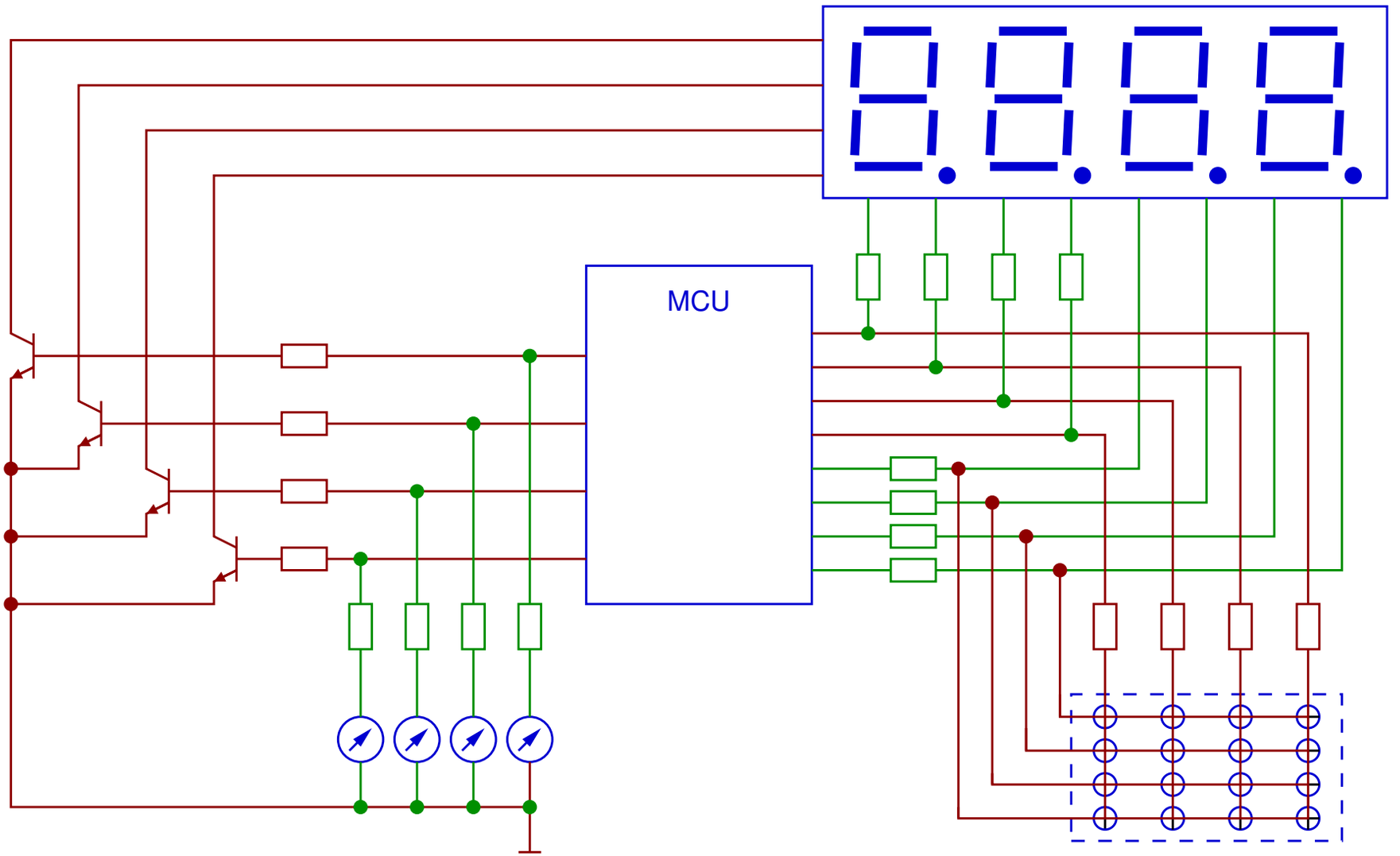
- a screen composed of four 7-segment displays,
- a keyboard organized as a $4 \times 4$ matrix,
- 4 analog input channels.

(*Source: Microchip application note AN557*)

Solution:

- The screen and the keyboard are scanned: At a given time, one can only display a single digit, or read a single column of keys.
- An additional phase is inserted for reading input channels.
- 4 pins are associated to both an input channel and a screen digit. They are alternatively configured as analog inputs (when reading channels) and digital outputs (when displaying a digit or reading the keyboard).
- The 8 remaining pins drive the screen segments during display and channel reading phases (8 digital outputs), and are also able to the scan the keyboard (4 digital outputs + 4 digital inputs with pull-up).

## Schematics:



# Chapter 3

# Interrupts

# Introduction

An interrupt is a signal that requests the processor to temporarily suspend program execution, in order to execute an interrupt routine (or Interrupt Service Routine, ISR).

Advantages:

- A very short response time to solicitations is achievable.
- Urgent operations can be programmed independently from the main code.

Interrupts can be triggered either by an exterior component:

- Interrupt ReQuest (IRQ), received from dedicated input pins,
- change of logic value at digital input pins,

or by the processor itself:

- timer expiration,
- arithmetic or instruction exception,
- software interrupt request,
- ...

# The interrupt mechanism

Upon receiving and accepting to service an interrupt request, the processor performs the following operations:

1. The execution of the current instruction terminates.
2. A pointer to the next instruction to be executed is stored on the runtime stack.
3. The address of the interrupt routine is read from the appropriate interrupt vector (according to the source of the interrupt request).
4. The interrupt routine is executed.
5. At the end of the interrupt routine, the processor resumes program execution, at the address retrieved from the stack.

# Interrupt control

Some critical operations can never be interrupted. It is then necessary to temporarily disable interrupts prior to their execution, and to enable them again afterwards.

Some processors allow to assign specific priorities to interrupts originating from different sources. Such architectures generally provide a mechanism for disabling the interrupts having a priority less than some specified threshold. Interrupt priorities are also used for resolving simultaneous interrupt requests.

Enabling and disabling interrupts is performed by executing specific instructions, or by setting the value of dedicated registers.

Notes:

- At power-on, interrupts are disabled by default, in order to allow correct initialization of the program.

- When an interrupt is triggered, some processors automatically disable all interrupts of less or equal priority. They have to be explicitly reenabled in the interrupt routine if needed.
- When an interrupt request is received, the processor sets interrupt flags, in order to trigger the interrupt as soon as it becomes enabled. Interrupt flags have to be cleared explicitly by the interrupt routine.
- Some architectures provide an interrupt source that cannot be disabled (Non Maskable Interrupt, NMI). Its usage is limited to exceptional situations (e.g., backing up critical data upon detecting an imminent power failure).

# Saving and restoring context

The correct operation of a program must not be influenced by interrupts triggered during its execution.

It is thus mandatory for interrupt routines to leave the processor state unchanged: values of registers and flags, interface configuration, status of peripherals, . . . , must not be modified.

This is achieved by saving the context at the beginning of interrupt routines, and restoring it at the end.

Notes:

- The context is either saved on the execution stack or in a specific memory area.
- Some processors automatically save the context (either totally or in part) when an interrupt is triggered.

- Context save and restore operations can sometimes be simplified by using dedicated instructions.

- The processors that automatically disable interrupts when branching to an interrupt routine enable them again as a side effect of context restoration.

# Programming interrupts

The compilers aimed at embedded applications provide language extension mechanisms for programming interrupts without going down to assembly language.

- Some functions can be designated as being interrupt routines (e.g., `interrupt` keyword, `__attribute((interrupt))` attribute or `#pragma interrupt` compilation directive in C).

  With some compilers, such mechanisms automatically insert context save and restore instructions to interrupt routines, and take care of setting interrupt vectors.

- Enabling and disabling interrupts is performed with the help of macros or specific compilation directives (e.g., `enable()`/`disable()` or `__enable_irq()`/`__disable_irq()` macros, `critical` keyword).

- It is sometimes necessary to inform the compiler than the value of a variable can be modified by interrupt routines, in order to prevent incorrect optimizations (e.g., `volatile` keyword in C).

# Communicating with interrupt routines

Interrupt requests are by nature unpredictable. This complicates data exchange operations between interrupt routines and the main code.

Example: Industrial controller. The alarm must sound if two temperature measurements made by an interrupt routine differ.

Wrong solution:

```
static volatile int temp[2];

interrupt void measure(void)
{
  temp[0] = !! first measurement;
  temp[1] = !! second measurement;
}

void controller(void)
{
  int temp0, temp1;
  for (;;)
    {
      temp0 = temp[0];
      temp1 = temp[1];
      if (temp0 != temp1) !! sound the alarm;
    }
}
```

Notes:

- Carrying out the comparison between the two measurements in a single C instruction does not solve the problem:

```
...
void controller(void)
{
  for (;;)
    if (temp[0] != temp[1]) !! sound the alarm;
}
...
```

  (Indeed, such an instruction is generally compiled into several machine instructions.)

- Even in programs written in assembly language, it is possible for the execution of individual instructions to be interrupted before their completion.

  This only happens with specific instructions, often repeatedly performing a simpler operation (e.g., block copy instructions).

- This type of bug can be very difficult to detect and to reproduce!

## Correct solution:

The instructions that read the measurements sent by the interrupt routine to the controller form a critical section, the execution of which cannot be interrupted.

```
static volatile int temp[2];

interrupt void measure(void)
{
  temp[0] = !! first measurement;
  temp[1] = !! second measurement;
}

void controller(void)
{
  int temp0, temp1;
  for (;;)
    {
      disable();   /* Disable interrupts */
      temp0 = temp[0];
      temp1 = temp[1];
      enable();    /* Reenable interrupts */

      if (temp0 != temp1) !! sound the alarm;
    }
}
```

## Other solution:

```
static volatile int temp_a[2], temp_b[2];
static int controller_uses_b = 0;

interrupt void measure(void)
{
  if (controller_uses_b)
    {
      temp_a[0] = !! first measurement;
      temp_a[1] = !! second measurement;
    }
  else
    {
      temp_b[0] = !! first measurement;
      temp_b[1] = !! second measurement;
    }
}

void controller(void)
{
  for (;; controller_uses_b = !controller_uses_b)
    if (controller_uses_b)
      {
        if (temp_b[0] != temp_b[1]) !! sound the alarm;
      }
    else
      if (temp_a[0] != temp_a[1]) !! sound the alarm;
}
```

Notes:

- This solution does not require to disable interrupts.
- The main code must sometimes perform one useless iteration before sounding the alarm.

## Improved solution:

```
#define MAX_FIFO 10     /* Must be even ! */
static volatile int temp_fifo[MAX_FIFO];
static volatile int first = 0;
static int last = 0;

interrupt void measure(void)
{
  /* If the buffer is not saturated */
  if (!((first + 2 == last)
      || (first == MAX_FIFO - 2 && last == 0)))
    {
      temp_fifo[first] = !! first measurement;
      temp_fifo[first + 1] = !! second measurement;
      first += 2;
      if (first == MAX_FIFO)
        first = 0;
    }
  else   !! discard measurements;
}

void controller(void)
{
  int temp0, temp1;

  for (;;)
    if (first != last)    /* If the buffer is not empty */
      {
        temp0 = temp_fifo[last];
        temp1 = temp_fifo[last + 1];
        last += 2;
        if (last == MAX_FIFO)
          last = 0;
        if (temp0 != temp1) !! sound the alarm;
      }
}
```

Note: For this solution to be correct, it is necessary that the instruction `last += 2` executes atomically.

This kind of solution is thus very sensitive to implementation details!

In practice, disabling interrupts during communications with interrupt routines is acceptable in most situations. The more complex solutions are used only when disabling interrupts is impossible or forbidden.

# Interrupt latency

The delay between an interrupt request $I$ and the end of execution of urgent operations in an interrupt routine $R_I$ is called the response time, or latency of the interrupt.

This latency is influenced by four parameters:

1. The longest interval during which interrupts of priority larger or equal to $I$ are disabled.
2. The time needed for executing the interrupt routines with a higher priority than $R_I$.
3. The maximum delay between an interrupt trigger and the branch to the corresponding interrupt routine.
4. The time spent in $R_I$ before having executed the urgent operations.

A good strategy is therefore to

- disable interrupts for the shortest possible time (parameter 1);
- make the interrupt routines quick and efficient (parameters 2 and 4).

Parameter 3 is a feature of the processor, and cannot be influenced by the programmer.

# Example

- A system implements the following interrupt routines, sharing the same priority.

| Name | Description | Execution time | Period |
|---|---|---|---|
| $I_1$ | Temperature measurement | 100 $\mu$s | 500 $\mu$s |
| $I_2$ | Timer expiration | 200 $\mu$s | 1000 $\mu$s |
| $I_3$ | Network I/0 | 300 $\mu$s | $> 1000$ $\mu$s |

- The main program disables interrupts during resp. 200 $\mu$s and 250 $\mu$s for exchanging data with $I_1$ and $I_2$.
- The time needed for triggering $I_3$ and executing the corresponding urgent operations is equal to 100 $\mu$s.

Question: Is the latency of $I_3$ smaller than 1000 $\mu$s ?

Answer:

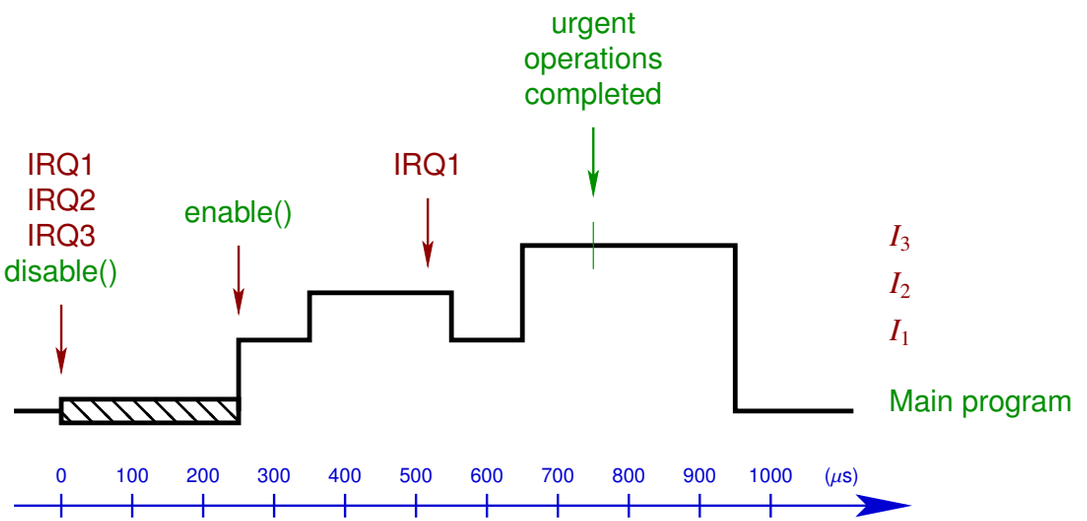
It is sufficient to study the system during an interval of length equal to $1000\ \mu$s. The highest possible latency is obtained with the following delays:

- Interrupts disable time : $250\ \mu$s.
- Executing $I_1$ : $2 \times 100\ \mu$s.
- Executing $I_2$ : $200\ \mu$s.
- Triggering and executing the urgent operations of $I_3$ : $100\ \mu$s.
- $\rightarrow$ Total: $750\ \mu$s.

Notes:

- Only the largest interval in which interrupts are disabled has to be taken into account!

- Example of scenario in which the maximum latency is reached:




- The execution of $I_3$ always terminates before $1000$ $\mu$s.

# Chapter 4

# Software architectures

# The round-robin architecture

Principles:

- Interrupts are not used.
- Tasks are invoked in turn, and run until their completion.

Illustration:

```
void main(void)
{
  for (;;)
    {
      if (!! task 1 is ready)
        {
          !! operations of task 1;
        }
      if (!! task 2 is ready)
        {
          !! operations of task 2;
        }

      ⋮

      if (!! task n is ready)
        {
          !! operations of task n;
        }
    }
}
```

Advantages:

- Simple solution, but sufficient for some applications.
- Exchanging data between tasks is easy.

Drawbacks:

- The worst-case latency of an external request is equal to the execution time of the entire main loop.
- Implementing additional features can adversely affect the correctness of a system, by increasing latencies beyond acceptable bounds.

## Example (multimeter):

```
#include "types.h"
#include "multimeter.h"

static UINT1 phase = 0; /* 0–3: display, 4: keyboard, 5: channels */
static UINT1 display_content[4];
static SINT4 measures[4];

static keyboard_state   keys;
static multimeter_state parameters;

void main(void)
{
  !! initialize global data;

  for (;;)
    {
      switch (phase)
        {
          case 4:
            handle_keyboard();
            if (keys.new_keypress)
              {
                keypress_action();
                keys.new_keypress= 0;
              }
            break;

          case 5:
            handle_channels();
            update_display_content();
            break;

          default:
            handle_display();
        }
      if (++phase > 5)
        phase = 0;
    }
}
```

```
void handle_display(void)

{
  UINT1 digit, segments;

  !! PORTA: 4 digital outputs;
  !! PORTB: 8 digital outputs;

  digit    = !! compute the digit to be displayed, from the
             !! values of display_content and phase;
  segments = !! pattern corresponding to digit;

  out(PORTA, 1 << phase);
  out(PORTB, segments);

  delay(DISPLAY_DELAY);
}

void handle_channels()
{
  !! PORTA: 4 analog inputs;
  !! PORTB: 8 digital outputs;

  out(PORTB, 0);
  delay(CHANNELS_DELAY);

  !! read PORTA, and place the result in measures;
}

void handle_keyboard()
{
  static UINT1 column = 0;
  UINT1 row;

  !! PORTA: 4 digital outputs;
  !! PORTB: 4 digital outputs (low nibble),
  !!         4 digital inputs with pull-ups (high nibble);
```

```
  out(PORTA, 0);
  out(PORTB, 1 << column);
  row = in(PORTB) >> 4;

  !! update keys according to the content of row;

  if (++column >= 4)
    column = 0;
}

void keypress_action()
{
  !! update parameters according to the key that has
  !! been pressed (specified in keys);
}

void update_display_content()
{
  !! update display_content according to the values in
  !! measures and parameters;
}
```

Notes: The parameters DISPLAY_DELAY and CHANNELS_DELAY must be chosen

- large enough to ensure an accurate conversion of analog samples, and a good illumination of display segments.
- small enough to avoid display flickering, as well as missing key presses.

# The round-robin with interrupts architecture

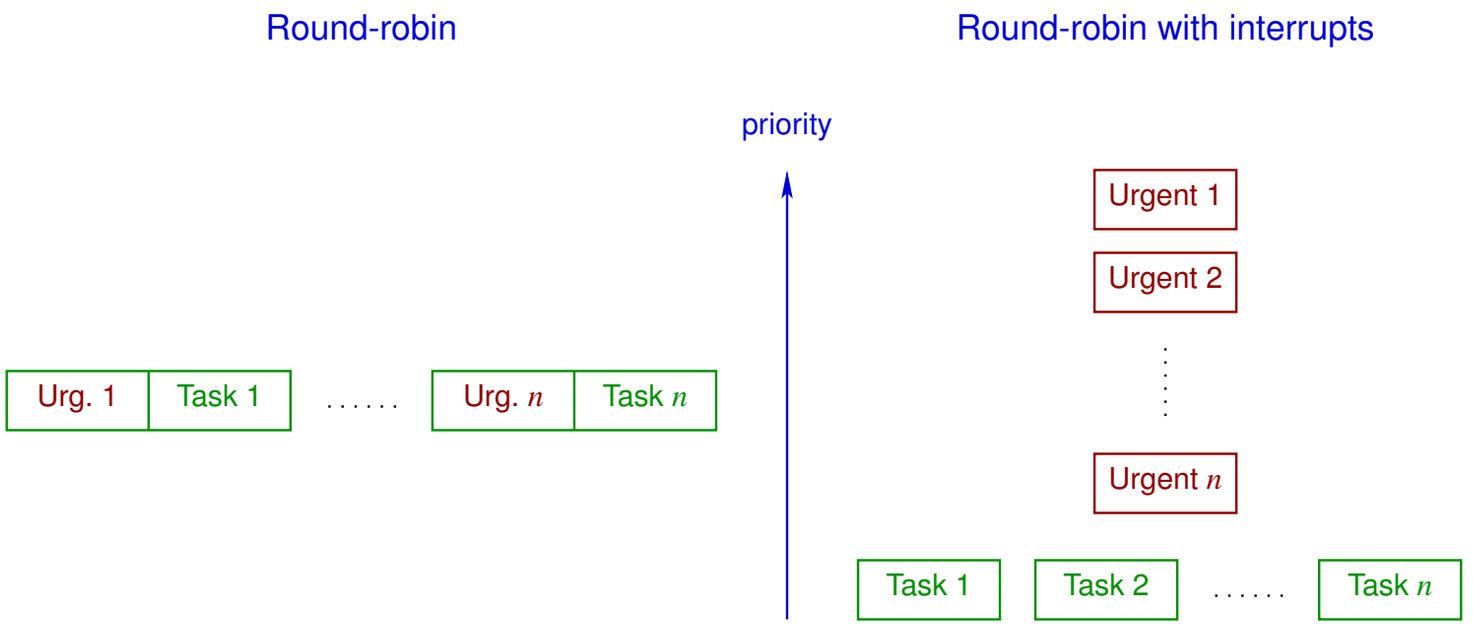
Principles: Tasks are invoked in round-robin fashion, but interrupt routines take care of urgent operations.

Illustration:

```
volatile BOOL ready1 = 0, ready2 = 0, ...,
              readyn = 0;

interrupt void urgent1(void)
{
   !! urgent operations of task 1;
   ready1 = 1;
}

interrupt void urgent2(void)
{
   !! urgent operations of task 2;
   ready2 = 1;
}

⋮

interrupt void urgentn(void)
{
   !! urgent operations of task n;
   readyn = 1;
}
```

```
void main(void)
{
  for (;;)
    {
      if (ready1)
        {
          !! non-urgent operations of task 1;
          ready1 = 0;
        }
      if (ready2)
        {
          !! non-urgent operations of task 2;
          ready2 = 0;
        }

      :

      if (readyn)
        {
          !! non-urgent operations of task n;
          readyn = 0;
        }
    }
}
```

Advantage: The urgent operations take priority over the non-urgent ones.




Drawbacks:

- The non-urgent tasks share the same effective priority. This yields high latencies when at least one task has a large execution time (e.g., raster generation in laser printers).

Important note: Moving non-urgent operations from tasks to interrupt routines is not a good solution!

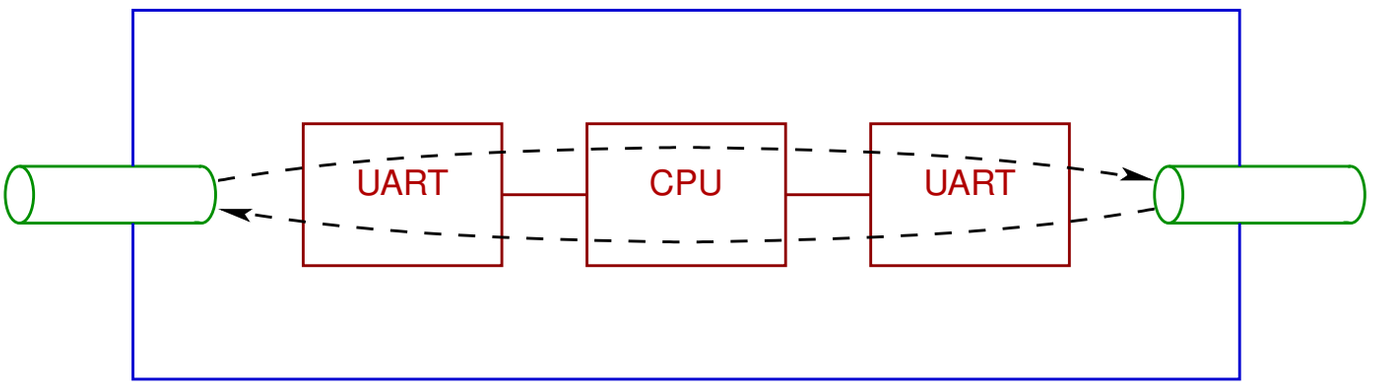
Indeed,

  - performing non-urgent operations in an interrupt routine increases the latency of interrupts with a lower or equal priority;
  - interrupts do not offer flexible synchronization mechanisms.

- Data exchange operations between interrupt routines and tasks have to be correctly implemented (cf. Chapter 3).

## Example: Serial filter

The goal is to develop a two-way filter connecting two serial lines.




## Principles:

- Incoming bytes are signaled by interrupt requests, which must be answered as soon as possible (before the next received byte).
- When a UART is ready to send a byte on its output line, it requests an interrupt. The processor is then free to wait for an arbitrarily long time before providing this byte.

Solution:

```
#include "types.h"
#include "fifo.h"
#include "filter.h"

static volatile BOOL uart1_ready, uart2_ready;
static volatile fifo rx1, tx1,
                     rx2, tx2;

interrupt void uart1_rx(void)
{
  char byte;

  byte = !! reception from UART1;
  fifo_put(rx1, byte);
}

interrupt void uart2_rx(void)
{
  char byte;

  byte = !! reception from UART2;
  fifo_put(rx2, byte);
}

interrupt void uart1_ready_to_send(void)
{
  uart1_ready = 1;
}

interrupt void uart2_ready_to_send(void)
{
  uart2_ready = 1;
}
```

```
void main(void)
{
   !! initialize global data;
   !! initialize interrupt vectors;

   enable();

   for (;;)
     {
       if (fifo_content_size(rx1) >= FILTER_THRESHOLD)
         {
           !! remove data from rx1;
           !! filter;
           !! add the result to tx2;
         }

       if (fifo_content_size(rx2) >= FILTER_THRESHOLD)
         {
           !! remove data from rx2;
           !! filter;
           !! add the result to tx1;
         }

      if (uart1_ready && !fifo_is_empty(tx1))
         {
           char byte;

           byte = fifo_get(tx1);
           disable();
           !! send byte to UART1;
           uart1_ready = 0;
           enable();
         }
```

```
      if (uart2_ready && !fifo_is_empty(tx2))
         {
           char byte;

           byte = fifo_get(tx2);
           disable();
           !! send byte to UART2;
           uart2_ready = 0;
           enable();
         }
     }
 }
```

Notes:

- Attempting to add data to a saturated FIFO buffer cannot be a blocking operation (i.e., it must instead discard data).

- The functions for handling FIFO buffers must execute correctly both in the interrupt routines and in the main code.

  Example of implementation:

```
void fifo_put(fifo q, char c)
{
  BOOL intr_enabled;

  ...

  intr_enabled = disable();
  !! critical section;
  if (intr_enabled)
    enable();

  ...
}
```

# The waiting-queue architecture

Principles:

- In the same way as the round-robin with interrupts architecture, the operations are partitioned into urgent and non-urgent tasks.
- Interrupt routines perform urgent operations, and then place in a waiting queue requests for executing non-urgent tasks.
- The main program retrieves execution requests from the queue and calls the corresponding functions. These requests are not necessarily processed in FIFO order. (For instance, different selection priorities can be assigned to non-urgent tasks.)

## Illustration:

```
#include "queue.h"

static volatile queue waiting_queue;

interrupt void urgent1(void)
{
   !! urgent operations of task 1;
   !! add task1 to waiting_queue;
}

interrupt void urgent2(void)
{
   !! urgent operations of task 2;
   !! add task2 to waiting_queue;
}

⋮

interrupt void urgentn(void)
{
   !! urgent operations of task n;
   !! add taskn to waiting_queue;
}
```

```
void main(void)
{
   !! initialize waiting_queue with an empty content;

   for (;;)
     {
       while (!queue_is_empty(waiting_queue))
         {
           !! extract a function from waiting_queue;
           !! execute this function;
         }
     }
}

void task1(void)
{
   !! non-urgent operations of task 1;
}

void task2(void)
{
   !! non-urgent operations of task 2;
}

⋮

void taskn(void)
{
   !! non-urgent operations of task n;
}
```

Advantage: The latency of a non-urgent high-priority task can become smaller than the execution time of all the non-urgent operations.

Drawbacks:

- The maximum latency of a non-urgent task is still at least as large as the execution time of the slowest task.
- Implementing the waiting-queue data structure can be tricky.

Example of application: A system monitors an industrial process by receiving data from an array of sensors, processing this data, and displaying summarized results.

With the queue architecture, it is possible to ensure that the values produced by critical sensors are always taken into account, even in the case of data saturation caused by a malfunctioning low-priority sensor.

# The real-time operating system architecture

Principles:

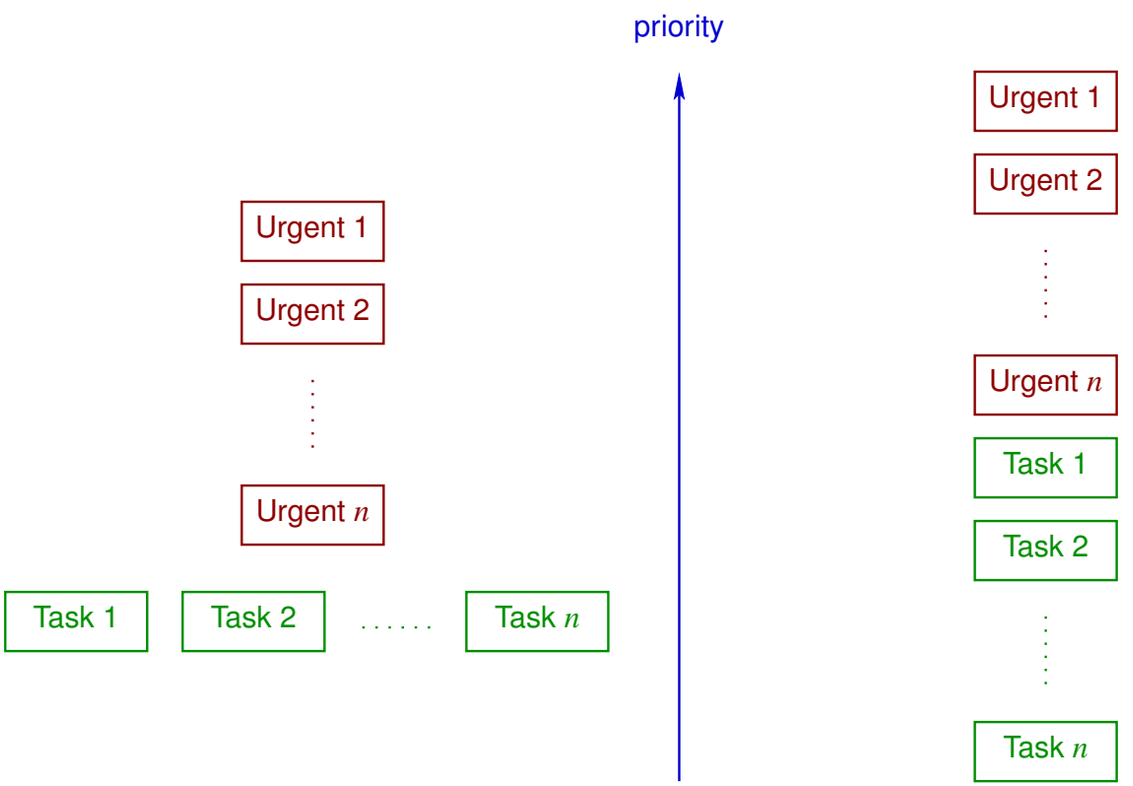
- Urgent operations are performed by interrupt routines. Those are able to signal to other tasks that non-urgent operations are ready to be carried out.
- The non-urgent tasks are invoked dynamically rather than in a predefined order. The responsibility of calling tasks is assigned to the operating system, implemented as an additional software component.
- The operating system is able to suspend the execution of a task before its completion, in order to transfer the processor to another task.
- The signals exchanged between tasks are handled by the operating system, instead of being implemented with shared variables.

## Illustration:

```
#include "signal.h"

interrupt void urgent1(void)
{
   !! urgent operations of task 1;
   !! send signal 1;
}

interrupt void urgent2(void)
{
   !! urgent operations of task 2;
   !! send signal 2;
}

⋮

void task1(void)
{
   !! wait for signal 1;
   !! non-urgent operations of task 1;
}

void task2(void)
{
   !! wait for signal 2;
   !! non-urgent operations of task 2;
}

⋮

void main(void)
{
   !! initialize the operating system;
   !! create and enable tasks;
   !! start task sequencing;
}
```

## Advantages:

- One can easily combine low-latency operations together with long computations.



- The system is efficient: When a non-urgent task is waiting for a signal, the processor remains available for other computations.
- The structure of the system is robust: New features can easily be added without affecting the latencies of urgent operations or of high-priority tasks.
- Operating systems tailored to embedded applications are commercially available.

Drawbacks:

- The system is complex (but this complexity is mainly located in the operating system, which can be reused over many projects).
- Data exchange operations have to be coordinated between a task and an interrupt routine, but also between tasks.
- The operating system consumes some amount of system resources (a typical figure is 2 to 4 % of the instructions executed by the processor).

# Summary

## Task priorities and latencies:

| Architecture | Available priorities | Maximum latency |
|---|---|---|
| round-robin | none | total execution time of all tasks |
| round-robin with interrupts | interrupt routines; all tasks share the same priority | total execution time of all tasks + interrupt routines |
| waiting queue | interrupt routines, then tasks | execution time of the longest task + interrupt routines |
| operating system | interrupt routines, then tasks | execution time of interrupt routines |

## Robustness and simplicity:

| Architecture | Robustness against modifications | Complexity |
|---|---|---|
| round-robin | poor | very simple |
| round-robin with interrupts | good for interrupt routines, poor for the tasks | must handle data exchanges between tasks and interrupt routines |
| waiting queue | fair | must handle data exchanges, and implement the waiting queue |
| operating system | very good | quite complex |