Programming Real-Time Embedded Systems

Grégory Mermoud

School of Architecture, Civil and Environmental Engineering

EPFL, SS 2008-2009

http://disal.epfl.ch/teaching/embedded_systems/
Outline

- Embedded systems constraints (memory, computation, communication bandwidth, energy, e.g.).
- Real-time issues in embedded systems programming:
  - Perception-to-action loop
  - Interrupts
  - Scheduling
  - Operating system and device drivers
- Memory issues in embedded systems programming:
  - Data storage: primary, secondary and tertiary
  - Memory architectures and dsPIC architecture (e-puck)
  - Compression: using computation for saving memory
- Conclusion
Embedded systems

- Programming embedded systems is all about **resource management**, i.e. dealing with delays and time constraints imposed by the environment and limitations of the hardware (memory, computation, energy, communication bandwidth).

- When programming (or simply using) an embedded system, you must bear in mind **its limitations** and find appropriate techniques for dealing with them.

- **Energy, memory and computation** are often very limited and precious in real-time embedded system applications.

- Often, you need to make a **trade-off** between computation, memory and communication.

- **Fortunately,** computer and electronic engineers have developed a lot of useful techniques to perform these trade-offs.
Modern embedded systems have **increasing software complexity.**

Some applications may be **critical** and **improper software design** can cause real catastrophes!

**Ariane 5 explosion:** a 64 bit floating point number relating to the horizontal velocity of the rocket was converted to a 16 bit signed integer. The number was larger than 32,767, the largest integer storeable in a 16 bit signed integer, and thus the conversion failed:

```c
double h_v = horizontal_velocity();
short int h_v2 = (short int) h_v;
```

**Cost of these two lines of wrong code?** The destroyed rocket and its cargo were valued at **US $500 million.** The development cost of the rocket was **US $7 billion.**

Ariane 5 explosion on June 4, 1996. No human casualties (except for the software engineer who has been fired from ESA).
Real-Time Programming
Real-time: definition

- A system is said to be **real-time** if the total correctness of an operation depends not only upon its logical correctness, but also upon **the time** in which it is performed (Wikipedia).

- One can distinguish two types of real-time systems:
  - **Hard real-time systems**: the completion of an operation after its deadline is considered useless (ultimately, this may lead to a critical failure or the destruction of the system).
  - **Soft real-time systems**: the completion of an operation after its deadline decreases the service quality (e.g., dropping frames in a video streaming).

- **Warning**: a real-time system is not necessarily a high-performance system, and vice versa!
  - An e-puck can be a real-time system (if properly designed), but it will never be a high-performance system!
Real-time: it matters

- Read and digitalize instruments data
- Filter data
- Calculate updated roll, yaw and pitch
- Update flaps and throttle

This is an example of **hard** real-time embedded system
Perception-to-Action loop

Delays are everywhere!

sensors

analog-digital conversion time

processing time

actuator delay

actuators

sampling rate

Environment

propagation time
Perception-to-Action delay

**Sideairbag in a car:** perception-to-action delay < 10 mSec

**Wing vibration in a plane:** perception-to-action delay < 5 mSec

TIK Lectures - Embedded Systems, ETHZ
http://www.tik.ethz.ch/tik/education/lectures/ES/
“Super” loop software architecture

- The super loop or endless loop is often required because we have no operating system to return to at the end of the program!

  - Simple, efficient and portable!
  - Timing is inaccurate!
  - High power consumption!

- Assume that you want X() to be executed at precisely 18kHz (T = 55μs).
- You know that X() duration is about 10μs. Therefore, you wait 35μs (using a calibrated active loop).
- This type of software architecture does not guarantee accurate timing.

```c
void main() {
    // prepare for task X
    X_init();

    // super loop
    while (1) {
        X(); // perform task X
        wait(35); // active wait
    }
}
```
Solution: interrupts

- An interrupt is a special signal that triggers a change in execution, i.e. a call to a subroutine which is usually referred to as an **interrupt service routine** (ISR).

- The interrupt does **not** wait for the current program to finish. It is unconditional and immediate.

- Interrupts are very useful for interacting with **hardware devices**, but there are also **software interrupts**, which are triggered by a program.

- Interrupts are also very useful when coupled with a **timer**. They allow a function or subroutine to run periodically with **an accurate timing**.

- Another benefit of using interrupts is that in some microcontrollers you can use a **wake-from-sleep** interrupt. This allows the microcontroller to go into a low power mode, and be wakened later on by a hardware interrupt.
void _ISRFAST _T1Interrupt(void) {
    IFS0bits.T1IF = 0; // clear interrupt flag
    X(); // perform task X
}

void InitTMR1(void) {
    T1CON = 0;
    T1CONbits.TCKPS = 0; // prescaler = 256
    TMR1 = 0; // clear timer 1
    PR1 = (MILLISEC/18.00); // 18KHz interrupt (T = 55 us)
    IFS0bits.T1IF = 0; // clear interrupt flag
    IEC0bits.T1IE = 1; // set interrupt enable bit
    T1CONbits.TON = 1; // start Timer1
}

void main() {
    // initialize Timer 1
    InitTMR1();

    sleep(); // go to sleep, waiting for interrupt
}
The main loop gets executed, but the only thing it does is to put the microcontroller in sleep mode.

In sleep mode, timers and hardware interrupts are still active.

Each 55μs, Timer 1 triggers an interrupt, thus wakening the microcontroller.

The task X() gets executed during approximately 10μs. Then, the microcontroller can get back to sleep for 35μs.

Even if the duration X() is different, the timing is not affected.

Here, the microcontroller is in sleep mode during 64% of the time: very interesting from an energy point of view.
Interrupts: some definitions

- **Interrupt Service Routine (ISR):** simply another program function that gets executed upon trigger of its associated interrupt.
- **Interrupt vector:** a fixed address that contains the memory address of the ISR.
- **Interrupt flag:** one bit in a register that indicates whether the interrupt has been triggered or not.
- **Interrupt mask:** one bit in a register that controls whether the interrupt can be triggered or not.
- **Non Maskable Interrupt (NMI):** an interrupt that is always active.
- **Asynchronous event:** an event that could happen at any time (not necessarily synchronized with the clock).
- **Time-triggered interrupt:** an interrupt that is triggered by a timer in a periodic fashion.
- **Event-triggered interrupt:** an interrupt that is triggered by an (external) event (e.g., user input, A/D conversion, sensor measurement).
Microphone on the real e-puck

- First, you need a buffer to read the data from the microphone:

```c
#define SAMPLELEN 1840 // for buffer allocation
static int values[SAMPLELEN]; // buffer and index for storing
static int valuesw; // index
```

- Now, we will use an interrupt to sample periodically the measurements of the microphone.

- In our case, we want to achieve a sampling rate of 18 kHz (T = 55μs). Here we set up the timer so that it triggers an interrupt each 55μs:

```c
void InitTMR1(void) {
    T1CON = 0;
    T1CONbits.TCKPS = 0; // prescaler = 256
    TMR1 = 0; // clear timer 1
    PR1 = (MILLISEC/18.00); // 18KHz interrupt (T = 55 us)
    IFS0bits.T1IF = 0; // clear interrupt flag
    IEC0bits.T1IE = 1; // set interrupt enable bit
    T1CONbits.TON = 1; // start Timer1
}
```
Microphone on the real e-puck

- Now, we need to define an **Interrupt Service Routine (ISR)** that will be executed each 55μs.
- This function is very simple:
  - First, it clears the interrupt flag so that the interrupt can be triggered again.
  - Then, it checks whether the buffer is full. If it is the case, it just returns without doing anything.
  - Then, it calls the function `e_read_ad(int channel)` for initiating a new analog/digital conversion.
  - Finally, the new value is added to the buffer and the index is incremented.

```c
void _ISRFAST _T1Interrupt(void) {
    IFS0bits.T1IF = 0; // clear interrupt flag

    if (valuesw>(SAMPLELEN-1)) // stop writing when buffer is full
        {return;} // (will be reset in main loop)

    values[valuesw++] = e_read_ad(MIC3); // read one converted data
        // and add it to the buffer
}
```
Finally, in the main loop, the code becomes very simple:

```c
valuesw=0;  // reset index so that _T1Interrupt
while (valuesw<SAMPLELEN) {  // start to fill in the buffer
    __asm__ volatile("nop");  // and wait until it is full
}

for (int i = 0; i<size; i++) {
    printf("%d ",values[i]);
}
```

- Note that the following piece of code allows one to include assembler instructions in C code. In this case, the instruction is `nop`, which means “no operation”. It is useful to make the microcontroller “wait” for a while:

```c
__asm__ volatile("nop");
```

- Note that we use an active waiting loop, which is not really optimal from the energy/scheduling point of view.

- A better practice would consist in putting the microcontroller in sleep mode and using a software interrupt when the buffer is full to waken it and print the values in another ISR.
Multiple I/O = Multiple tasks
Multiple I/O = Multiple tasks

Sensing
- Read accelerometer
- Read IR sensors
- Read camera

Processing
- Crash detection
- Braitenberg algorithm
- Process image
- Object recognition and detection

Actuation
- Turn LEDs on
- Update wheel motors

Task A
- Dependency
- Maximal delay

Task B

Task C
Scheduling is the key!

- It is easy to carry out Task A (crash detection) in 5 ms.
- Problem: there is only 2.4ms left for tasks B and C!
Solution: time-sharing

- Execute only **subparts** of the other tasks by preempting them!
- Task B can be executed over 2 cycles of 5ms: split tasks into **2 subparts**!
- Task C can be executed over 60 cycles of 5ms: split tasks into **60 subparts**!
Time-sharing

- Time-sharing allows multiple tasks to be executed on a microcontroller or processor in a \textit{pseudo-parallel} manner, i.e. they appear to run in parallel for the external observer even though the execution is strictly sequential.

- The principle is to \textbf{split the execution of each task into multiple slices}.

- These slices can be re-organized in a different order that \textbf{complies with the dependencies between the tasks} in order to optimize the use of the microcontroller.

- There are two types of time-sharing:
  - \textbf{Cooperative}: the procedure/task itself must yield and give time to other processes.
  - \textbf{Preemptive}: a scheduler preempts the active procedure/task to give time another process.
Time-sharing and memory

- Time-sharing is a computation vs memory trade-off.
- Indeed, in order to allow multiple tasks to run in an interlaced manner, you need to as much memory as they were actually running in parallel.
- Also, you must take care of shared memory resources. Indeed, if multiple processes access the same memory location, you need to coordinate them in order to avoid unexpected behaviors.
- Assume that a process A reads a byte at memory address 0xF3A0 several times during its execution. However, it is preempted during its execution in favor of process B, which writes at memory address 0xF3A0. When process A becomes active again, the byte at 0xF3A0 has changed even though process A assumes.
Scheduler

- Preemptive time-sharing is very attractive, but it requires a smart and efficient scheduler.
- Upon preemption of the active process, the scheduler must decide which process must be executed next and how much processing time can be allocated to its execution before the next preemption.
- To make these decisions, the scheduler must take into account dependencies between the different processes, their respective deadlines and priorities.
- Generally, the scheduler is integrated into a more general software framework, which serves an interface between hardware and user applications, called an operating system (OS).
- Linux, Windows, MacOS X are typical operating systems for desktop computers, but they are generally too demanding in terms of computation and memory for embedded systems.
Operating system

- An operating system (OS) is an interface between hardware and user applications.
- It is responsible for the management and coordination of tasks and the sharing of the limited resources of the computer system.
- A typical OS can be decomposed into the following entities:
  - **Scheduler**, which is responsible for the sharing of the processing unit (microprocessor or microcontroller).
  - **Device drivers**, which are low-level programs that manage the various devices (sensors, actuators, secondary memory storage devices, etc.).
  - **Memory management unit**, which is responsible for the sharing of the memory (virtual memory).
  - **Optional: Graphical User Interface, File System, Security, etc.**

Most “OS” for embedded systems include these two entities only!
Device drivers

- A device driver or software driver is a computer program allowing higher-level computer programs to interact with a hardware device.
- Device drivers provide an API that allows the user application to access the device in a simplified and standardized manner.
- Reminder: an application programming interface (API) is a set of functions, procedures, methods or classes that an operating system, library or service provides to support requests made by computer programs.
- On the e-puck: no actual OS (no scheduler), but a standard library, which can be considered as a device driver API. The following features are available: initialisation of basic hardware features, management of proximity sensors, motor speed control, LEDs management, accelerometer reading, microphone sampling, sound emitting, image acquisition.
- Example: the functions `e_init_ad(void)` and `e_read_ad(int channel)` that are typical examples of functions that can be used by a higher-level program to interact with a hardware device (the A/D converter, and the microphone that is connected to it, in that case).
Embedded Systems and Memory
Data storage

- Embedded systems, and computers in general, process and generate a lot of data, but they also need to store these data.

- In general, we distinguish between three types of data storage:
  1. **Primary storage**, often known as memory simply, is the only one directly accessible to the microcontroller or the microprocessor. It is generally the **fastest** data storage (1-10 ns), but also the **smallest** (MB order), and it is generally **volatile**, i.e. it requires energy to maintain the stored information. Example: RAM, cache memory, registers, ROM (non-volatile)
  
  2. **Secondary storage** can be accessed through input/output channels. It is generally **non-volatile**, **large** (GB order), but quite **slow** (1-10 ms). It is often also **removable**. Example: hard disks, USB sticks, flash memory, floppy disks

  3. **Tertiary storage** are generally **non-volatile** storage devices of **very large** size (TB order), but with **very slow** access (5-60 s). It is generally **not** available on embedded systems.
Primary storage

- There are generally three levels of primary storage:
  1. Processor/microcontroller registers
  2. Process/microcontroller cache
  3. Main memory

- **Registers** are located inside the processor/microcontroller and they hold typically a word of data. Registers are the fastest type of computer data storage, but they are very limited (16x16 bit registers in dsPIC30F on the e-puck).

- **Processor cache** is an intermediate stage between ultra-fast registers and much slower main memory. It is introduced solely to increase performance of the computer. It serves to duplicate information that is frequently access from the main memory (“caching”).

- **Main memory** is directly or indirectly connected to the CPU via a memory bus. It can be either separated into two distinct entities that contain programs and data, respectively (Harvard architecture), or a single memory chip can contain both programs and data (Von Neumann architecture).

- **Data memory** must be random access, i.e. it takes the same time to access any location of the memory.

- **Program memory** can afford to be sequential access (because programs are read sequentially).
Throughput

These concepts can also be used to evaluate the performance of communication channels:

- The latency is the time it takes to send one single bit through the channel.
- The throughput is the rate (in bit per second) at which data can be transmitted through the channel.
Memory on the e-puck

- **Primary storage:** 8 KB of RAM *(random-access, volatile)* for data and 144 KB of Flash Memory *(sequential-access, non-volatile)* for program (Harvard Architecture)

- **Secondary and tertiary storage:** none (but one could develop an extension)

- **Permanent storage of data:** communication with the base station, which will take care of it

- Please note the **very small amount** of RAM (8 KB) available on the e-puck robot!

- Memory is a **very precious resource** in embedded systems!
Reminder: buffer mechanism

Image = 0x1

Write at 0x1

Read at Image

Robot memory

Buffer

Read and process 100ms

Write 33ms
Problem!

The microcontroller has not **enough memory** (and processing power) available for storing even a single image.

- To be able to acquire the camera information, the image rate has to be reduced and the resolution too.
Solution: image processing

- By using ** downsampling **, you can reduce the size of the image while retaining useful information.
- Typically, we can acquire a 40x40 downsampled color image at 4 frames per second.
- Without color, we can go up to 8 frames per second.
Another solution: compression

- Often, you can use **computation** to save memory and/or communication bandwidth.

- One typical example is **data compression**, i.e. the process of **encoding information** using fewer bits than an unencoded representation would use.

- Compression is often a computationally expensive process, but it can lead to large amount of saved memory.

- **Lossless compression** relies on statistical redundancy to represent data more concisely **without error** (e.g., ZIP, PNG).

- **Lossy compression** relies on approximations in order to represent data more concisely in such a fashion **that they remain useful in the context of the application** (e.g., MP3, JPEG, DivX).
Lossless compression

- A simple technique for lossless data compression is the Run-Length Encoding (RLE) approach.
- In RLE, long sequences of the same data value are stored as a single data value and a count.
- For instance, the sequence

  WWWWWWWWWWWWWBWWWWWWWWWWWWBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Lossy compression

- Lossy compression can often achieve much larger compression percentage than any known lossless compression.
- In some cases, the distortion induced by the compression is still compatible with the application.
- Since lossy compression schemes are strongly dependent on the type of information the data represent: using JPEG scheme for compressing a musical file will not yield good results.
- Lossless compression are also dependent on the nature of data, but rather from a statistical point of view.
Compression in Embedded Systems

- Often, you can use **compress** to save memory and/or communication bandwidth.
- You can **compress** data before storing them in memory and/or sending them via a communication channel.
- More generally, you can use **compression** (simple and complex schemes, lossless and lossy methods) as a trade-off between **computation** and **memory** and/or **communication bandwidth**.
- Also, do not forget **encryption** in the case of transmission or storage of **sensitive information**.
- **Encryption** is the process of transforming information (referred to as plaintext) using an algorithm (called cipher) to make it **unreadable to anyone except those possessing special knowledge**, usually referred to as a key.
Take Home Messages

- Programming embedded systems is all about **resource management**, i.e. dealing with delays and time constraints imposed by the environment and limitations of the hardware (memory, computation, energy, communication bandwidth).

- **Energy, memory and computation** are often very limited and precious in real-time embedded system applications.

- Often, you need to make a **trade-off** between computation, memory and communication using, for instance:
  - **Time-sharing** (for computation) and **sleep mode** (for energy)
  - **Compression** (for memory)

- Many other techniques exist, but the principle remains always the same: **find a trade-off between different resources in order to achieve your objectives!**
Reading and acknowledgements

- For each of the concepts described in this course, do not hesitate to use Google and Wikipedia for refining your understanding!

- **However**, each of these concepts (scheduling, real-time programming, operating systems, compression, memory architecture) could be the topic of a complete lecture, or even a whole course!

- We want you to **understand** the general principles of these different concepts, and why they might be useful to you as a **power user of embedded systems**.

- In case of problems with the e-puck robot, refer to the official website **www.e-puck.org**.

- Thanks to Francesco Mondada for his slides about the e-puck and the programming of embedded systems!