Signals, Instruments, and Systems – W13

More on Embedded systems and Field Instruments: Real-time Programming and Mobile Sensor Networks
Part I: Real-Time Programming for Embedded Systems
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 find the optimal **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 tracking the horizontal velocity of the rocket was converted to a 16 bit signed integer. The number was larger than 32,767, the largest integer storable 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: 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).

- **Note**: 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

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

- Perception
  - sensors
  - analog-digital conversion time
  - sampling rate

- Computation
  - processing time

- Environment
  - analog-digital conversion time
  - sampling rate

- Action
  - actuators
  - actuator delay
  - propagation time

*Delays are everywhere!*
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
“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 20µs. 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.
Interrupt software architecture

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 21µs. Then, the microcontroller can get back to sleep for 34µs.

What if the task \( X() \) lasts longer than expected?
Typical execution scheme

The timing remains unchanged: X() still runs every 55 µs!

**Bonus:** the microcontroller sleeps about 60% of the time (in other words, you can save a lot of energy, depending on the system).
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 (\(T = 55\mu s\)). Here we set up the timer so that it triggers an interrupt each 55\(\mu s\):

  ```c
  void InitTMR1(void) {
    T1CON = 0;
    T1CONbits.TCKPS = 0;     // prescaler = 256
    TMR1 = 0;                // clear timer 1
    PR1 = (MILLISEC/18.00);  // 18KHz interrupt (\(T = 55\mu s\))
    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; it performs only four operations:
  - First, it clears the interrupt flag so that the interrupt can be triggered again.
  - Second, it checks whether the buffer is full. If it is the case, it just returns without doing anything.
  - Third, 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;          // 1. clear interrupt flag

  if (valuesw>(SAMPLELEN-1))  // 2. stop writing if buffer is full
    return;                   // (will be reset in main loop)

  values[valuesw++] = e_read_ad(MIC3); // 3. convert and read data
                                  // 4. and add it to the buffer
}
```
Microphone on the real e-puck

- 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 you 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 (see slides before).

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

dsPIC microcontroller on e-puck

INPUTS
- IR proximity: 10-100Hz
- Accelerometer: 0.1-1kHz
- Microphones: 10-40kHz
- Camera: 10k-10MHz
- IR remote: 1kHz serial code

PROCESSING
- Generic, C prog.: PIC core
- Specific, ASM prog.: DSP core

OUTPUTS
- displacement
- display
- audio
- directed
- aesthetic
- Motors
- Lateral LEDs
- Speaker
- Front LED
- Body LED

COMMUNICATION
- radio
- cable
- Bluetooth
- RS232
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

Dependency
- Maximal delay

Task A
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!
- 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 **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 **split the execution of each task into multiple slices**.

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

- There are two types of time-sharing:
  - **Cooperative**: the procedure/task itself must yield and give time to other processes.
  - **Preemptive**: a scheduler preempts the active procedure/task to give time to 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 if 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 that it did not.
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.
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).
Compression

- **Data compression**: encoding information using fewer bits than the normal representation.
- Idea: Compress data before storing them in memory and/or sending them via a communication channel.
- Computationally expensive process.
- Trade-off between **computation** and **memory** and/or communication bandwidth.
- Classified in two types: **lossless** (e.g., zip, rar) vs **lossy** (e.g., jpeg, mp3).
Part II: Field Deployment of Mobile Sensor Networks: The OpenSense Project
OpenSense

Community-driven, large-scale air pollution measurement in urban environments

Mobile sensors (parasitic, uncontrolled mobility) on public transportation vehicles

Static wireless sensing and communication infrastructure

Sensorscope

Permasense
Motivation

Urban population will double in next decades

- > 50% of world population already lives in cities
- rural population expected to stagnate or drop

Urban air pollution

- 2% of all deaths (1.2 million people)
- 0.6% of burden of disease (DALY)


Global Health Risks, WHO 2009
Motivation

Air pollution is highly location-dependent
- traffic chokepoints
- urban canyons
- industrial installations

Fine resolution air quality data is needed!

Enabling research in:
- Human exposure
- Air Pollution Engineering
- Urban Planning
- Environmental Justice
- Public Policy

Public service & education
- enable private users to make informed decisions
- raising popular awareness
Sparse networks of ground stations
Example: Switzerland’s NABEL (www.empa.ch/nabel)

- 16 stations
- specially selected sites
  - urban with traffic
  - urban residential
  - suburban
  - rural, etc.
- resolution:
  - high temporal
  - low spatial

Mission: monitor air pollution on national level & gauge impact of environmental policies

Public data access:
http://www.bafu.admin.ch/luft/luftbelastung/blick_zurueck/datenabfrage
Traditional Air Monitoring Systems

Satellite-based remote sensing

Examples:
• Measurements of Pollution in the Troposphere (MOPITT on Terra satellite)
• Ozone Measurement Instrument (OMI on Aura satellite)

Features:
• daily scans
• large coverage
• homogeneous quality
• sensitive to cloud coverage
• low resolution
OpenSense System

- mobile sensor network
- parasitic mobility: anchored to existing mobility sources

- public transport
  - vehicle energy supply
  - predictable mobility
  - single point maintenance

- low-cost, light-weight chemical (CO, CO$_2$, NO$_2$, O$_3$) & ultrafine particle (UFP) sensors
- intelligent integration & control to mitigate demanding constraints
Proposed System

SENSING SYSTEM
From many wireless, mobile, heterogeneous, unreliable raw measurements ...

INFORMATION SYSTEM
... to reliable, understandable and Web-accessible real-time information

sensor network control
optimization of data acquisition
information dissemination
Value of Dense Measurements

• Traditional approach
  ➢ Few stations
  ➢ Low resolution interpolated estimates of pollutant concentrations across massive regions

• Recent results
  ➢ Massive deployment of stations (150) at street-level (2008/2009 New York City Community Air Quality Survey)
  ➢ Pollutants of interest heavily concentrated along roads with high traffic densities
Challenges

Global questions:
• More data, more noise, but also more redundancy
  ➢ Can we produce better quality data?

Research directions:
• Wireless Sensor Network control
  ➢ When/Where to sample?
  ➢ What/To whom to transmit?
• Sensor Node design
  ➢ Sampling System
  ➢ Localization
  ➢ Software & hardware architecture
  ➢ Mechanical integration
• Community sensing
  ➢ privacy protection
  ➢ trustworthiness of data,
  ➢ relevance of data gathered and
  information produced
• Modeling
  ➢ sensor, device and mobility models
  ➢ air quality models
  ➢ privacy, trust & activity models
Gas Sampling System

Problem:
Chemical sensors have very slow dynamics (example: Telaire 6613 CO2 sensor step response <2min)

Open sampling
- sensors directly exposed to environmental measurand

Benefits:
- simple & “slim” solution
- continuous sampling

Drawbacks:
- no absolute concentration values
- noisy signal (sensitive to environment variations: pressure, humidity)

Typical response:

Closed sampling
- sensors exposed to measurand inside controlled chamber
- 3-phase strategy

Benefits:
- absolute measurements
- noise due to environment filtered

Drawbacks:
- complex & bulky
- non-continuous sampling

Typical response:

Idea: Combine these two approaches to get the benefits of both systems.

[Lochmatter 2010] [Trincavelli 2010]
Gas Sampling System

- Smart sampling module
  - possibly hybrid
  - single/multi-chamber
  - wind sensing

---

[Lochmatter et al. 2010]

[Lochmatter et al. 2010]

[Lochmatter et al. 2010]

[Lochmatter et al. 2010]

[Lochmatter et al. 2010]
Logging & Localization

**Logger**
- GPRS link to back-end server
- local storage on SD card

**Robust localization – prerequisite for pollution mapping**
- exploits commercial state of the art **u-blox LEA-6R**
  - GPS + dead reckoning (DR) module
- augmentation with other source of information
Localization

- large set of rich data:
  - location parameters (geographical coordinates, heading, odometer, speed, acceleration etc.)
  - vehicle context data

![Localization Diagram]

![Localization Diagram]
OpenSense Lausanne Node

**Particle sampling module**
- Ultrafine particle measurements using Naneos Partector
- Measures directly lung-deposited surface area

**Gas sampling module**
- CO, NO₂, O₃, CO₂, temperature & relative humidity
- Hybrid active sniffer/closed chamber sampling operation
- Enables absolute concentration mobile measurements

**Enhanced localization & logger**
- mounted inside bus
- Fused GPS, gyro and vehicle speedpulses
- Accurate sample geolocation even in difficult urban landscapes
- GPRS communication
Lausanne Deployment

On top of 10 buses in Lausanne
• CO, NO₂, O₃, CO₂, UFP, temperature, humidity
• Active sniffing & closed sampling system
• Localization: Augmented GPS; communication: GPRS
• Prototypes deployed in multiple stages since June 2011
• Full deployment: Since November 2013

At NABEL roadside station in Lausanne
• Calibration and sensor drift evaluation
• Testing new sensors
• Since June 2010

On top of C-Zero electric vehicle
• 100% electric, clean platform
• flexible mobility
• system test bed
• targeted investigation tool
• intelligent network servicing
OpenSense Zurich Node

Inside the OpenSense Zurich node

Installation on top of VBZ Cobra tram

OpenSense Zurich node
Zürich Deployment

On top of 10 streetcars in Zurich
- $\text{O}_3$, CO, ultrafine particles, temperature, humidity
- Localization: GPS; Communication: WLAN and GSM
- Since September 2011

At NABEL station in Dübendorf
- Long-term sensor testing (e.g., $\text{O}_3$)
- Testing new sensors (combined CO/NO$_2$)
- Since April 2011

On top of “LuftiBus”
- $\text{O}_3$, ultrafine particles, temperature, humidity
- Localization: GPS; Communication: GSM
- Since March 2013, covers whole Switzerland
Pollution Data – Zurich Deployment

<table>
<thead>
<tr>
<th>Pollutant</th>
<th># of Measurements</th>
<th>Sampling rate</th>
<th>Time Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>UFP</td>
<td>56.000.000</td>
<td>5s</td>
<td>22 months</td>
</tr>
<tr>
<td>Ozone</td>
<td>8.900.000</td>
<td>20s</td>
<td>22 months</td>
</tr>
<tr>
<td>CO</td>
<td>5.300.000</td>
<td>20s</td>
<td>22 months</td>
</tr>
</tbody>
</table>

[Keller et al., SenseApp 2012]
Air Quality Data

<table>
<thead>
<tr>
<th>Pollutant</th>
<th># of Measurements</th>
<th>Sampling rate</th>
<th>Time Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>UFP</td>
<td>55,139,480</td>
<td>1s</td>
<td>11 months</td>
</tr>
<tr>
<td>Ozone</td>
<td>8,653,384</td>
<td>5s (15s first 6 months)</td>
<td>12 months</td>
</tr>
<tr>
<td>[CO, NO2, CO2]</td>
<td>27,538,704</td>
<td>5s</td>
<td>12 months</td>
</tr>
</tbody>
</table>

[Arfire, unpublished, December 2014]
Mobility Data

Gyro yaw rate

X-axis acceleration & vehicle context

Coverage of Lausanne region

<table>
<thead>
<tr>
<th>Measurement</th>
<th># of Measurements</th>
<th>Sampling rate</th>
<th>Time Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>[GPS, gyroscope]</td>
<td>117,456,000</td>
<td>1s</td>
<td>12 months</td>
</tr>
<tr>
<td>[odometer, accelerometer]</td>
<td>469,824,000</td>
<td>0.25s</td>
<td>12 months</td>
</tr>
<tr>
<td>vehicle context info</td>
<td>1,737,000</td>
<td>event-driven</td>
<td>12 months</td>
</tr>
</tbody>
</table>
Phase 2: Crowdsourcing, dispersion modeling & “closing the loop”

- Martinoli, Thiele – Stations and mobility
- Aberer, Faltings – Data, Models, Trust, Privacy
- Krause – CrowdSensing for quakes
- Emmenegger – Air quality measurement and modeling
- Bochud, Riediker – Studies on health impact of air quality
Conclusion
Take Home Messages (Part I)

• 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!**
Take Home Messages (Part II)

• Mobile sensor networks can increase coverage and spatial resolution of measured data.
• Increasing the resolution of air pollution data is necessary for understanding health impact.
• Whether data extracted from poor quality measurements can be processed to obtain useful data on air pollution is an important research question.
• Other questions: How to design the node? How to control the network?
• Using existing mobility sources holds important benefits, but achieving a reliable system integration is non-trivial.
Real-time Programming

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

Mobile sensor networks

• OpenSense: http://opensense.epfl.ch


• More pointers: see week 14