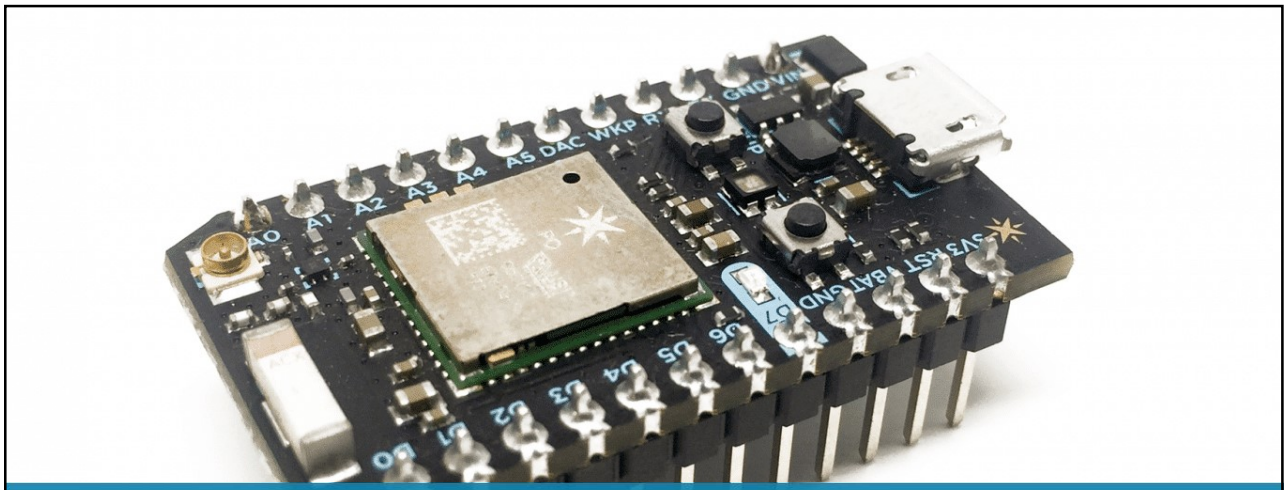


EMBEDDED PROGRAMMING AND CASE STUDIES

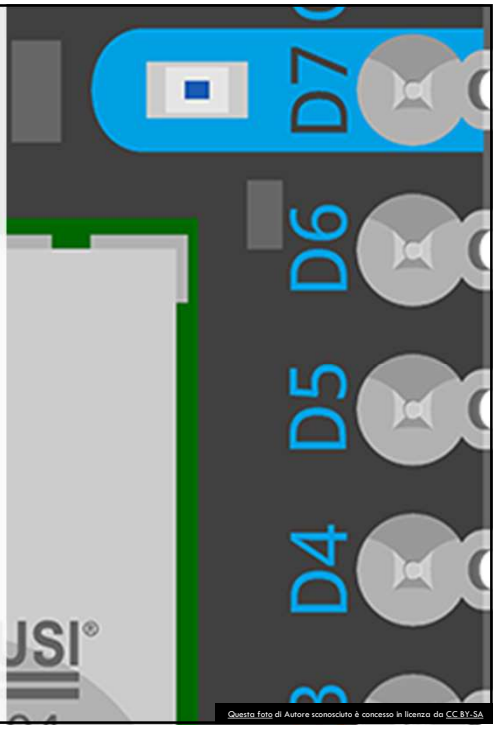
Mobile and Cyber-Physical Systems

1

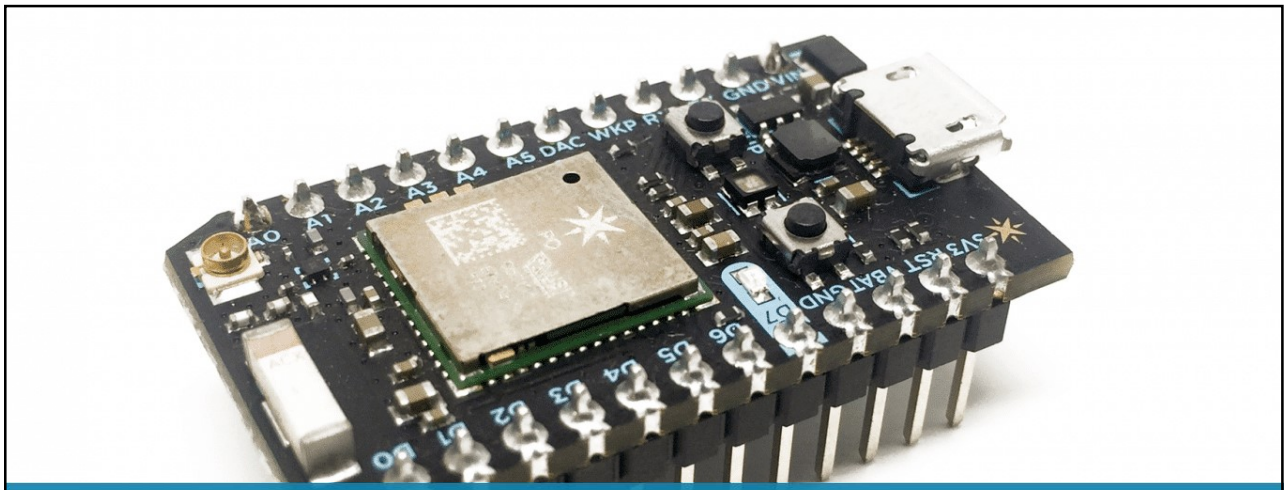
LEARNING OBJECTIVES

Embedded systems

Case study of Arduino



2



EMBEDDED PROGRAMMING AND CASE STUDIES

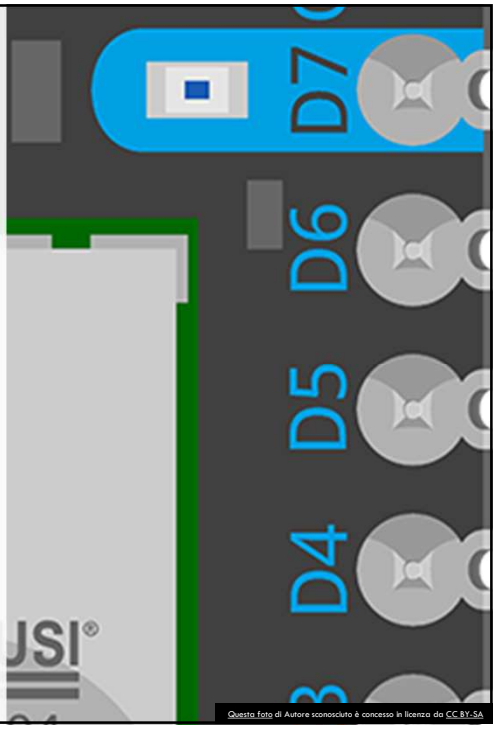
Mobile and Cyber-Physical Systems

1

LEARNING OBJECTIVES

Embedded systems

Case study of Arduino



2

• IoT devices are basically embedded devices: pc designed to 1 function (IoT, like termometer in plant measurement and send info)

EMBEDDED SYSTEMS

- Computer systems that are designed and built to perform a dedicated function (designed to 1 function)
- Different from PCs that are general purpose:
  - Just a specific function, not an arbitrary and variable set of functions as in a PC
  - It is built together with the software intended to run it
- The consequence is that hardware and software are designed together, in parallel (AKA hardware-software co-design)

=> IoT enables DC => energy efficiency => impact also programming

3

EMBEDDED SYSTEMS

Device

• Have to design HW & SW
is different from general purpose PC
=> shell, transducers, sensors...
=> process of design ≠ SW => you're completely free on how on classic PC
=> here SW impact

Embedded systems are a combination of hardware, software and (possibly) of mechanical components

Based on a microcontrollers: microprocessor designed for I/O
- A single chip that embeds a microprocessor, memory and I/O interfaces
- optimized for controlling I/O
  - comes with several ports
  - interact to the world => make it cps => implement functionalities of real time system

The microcontroller interacts with (is embedded into) a (complex) electro-mechanical devices to which it provides control

Given this controlling aim, it often has real time constraints

• Buy expensive sensor => easy SW
or
• Buy cheap sensor => SW reactive learning, software filters
no what you really need

=> HW & SW CO-DESIGN: develop pcs in parallel and affect each other

4

EMBEDDED SYSTEMS

Microcontroller

Many different types of microcontrollers on the market:

- Different processors, memory, performance
- Sometimes they are "general purpose", i.e. can be adapted to several embedded applications
- Sometimes they are designed for specialized purposes and for particular products, and they are often referred as Application-Specific Integrated Circuits (ASICs).
- Sometimes they are called System on a Chip (SoC) – but this term is vague and wider, may include also processors for smartphones

Used in many applications

- alarms, wearables, toys, industrial sensors, …

Many popular platforms:

- Arduino, Raspberry PI, … = more easy and more complex

5

PROGRAMMING ON EMBEDDED SYSTEMS

Many differences with traditional computing platforms:

- very small memory footprint (e.g., 16 KB of program memory based on flash memory technologies, and 8 KB of SRAM for data).
- non-existing or very simple user interface (e.g., a few buttons and just a few LEDs, or a simple LCD display).
- in most simple microcontroller-based systems there are no file systems
  - if an application requires a storage device, the developer needs to add the software to control it
- in some cases there is no Operating System (OS).
  - AKA "bare-metal" systems.
  - Just one application that directly manages interrupts
- in some cases there is a special OS for embedded systems
  - Small memory footprint, low overhead
  - Minimal functions
  - Real-time features
- need of an external PC/workstation to configure, program and debug

6

- When you co-design SW & HW for embedded systems => HAVE TO PROGRAM IT with different requirements ≠ general purpose devices (PC)

=> DEAL WITH PROGRAM MEMORY: program/code must have small FOOTPRINT to fit the memory available to end devices

PROGRAMMING ON EMBEDDED SYSTEMS

- GENERAL PC => RAM => reliable power source (volatile memory is ok)
- EMB SYS => turns off/on again ... => PROGRAM is NOT WRITTEN IN RAM, but in a PERMANENT MEMORY (flash memory usually)
  ≠
  RAM: contains data

Much different than conventional PCs…

- Not possible to program and compile code directly
  - Use cross-compilers to develop and compile on a general purpose PC
- Not possible to control the execution of the programs by a user interface => leds
  - The microcontroller embedded into a product, apart from any user interface
  - Much more difficult to debug the code

- When you turn-on embedded device:
  - no boot OS
  - load program
  - ...
- -> IMMEDIATLY IT LOADS THE PROGRAM, STARTS WHAT is supposed to do (program is in flash memory)

7

- ARDUINO flash is 16 kb ≃ including libraries
- Some cases there's not FS
- ARDUINO 3 types of memory: 1) program memory: 8/16/32 Kb
  2) RAM 4 kb
  3) flash ≃ 256 kb (since the disk)
- if OS is present, it should be load in program memory
  => OS informs # library
  => no FS when you write it in program memory (if it has)

PROGRAMMING CHALLENGES (I)

Timing correctness:

- it is as important as functional correctness.
- sometimes it even is an integral part of the functional requirements.
- for example, a GPS navigation system can always identify waypoints correctly (functional correctness), but it is useless if it is not able to report the waypoints before it is too late for the user to take actions.

High reliability is often required

- especially in control applications
- for example, if a system has a reliability requirement of four nines (i.e., 99.99% availability), it is not tolerable if the downtime is greater than 9 seconds per day.
- High reliability is not a trivial objective when operating in a hostile or an unexpected environment.
- Unpredictable event patterns from the environment may significantly change an embedded system's sequence of execution.

8

PROGRAMMING CHALLENGES (II)

Most embedded systems are "dumb" devices:

- cannot run a debugger, hard to detect and clear program defects.

Efficient use of memory:

- more memory means more costs
- making software is a creative activity; making software that can fit into the available memory space demands even more creativity.

Power management:

- it is critical to prolong the operating time of an embedded system.
- being able to switch to a low-power state when inactive is a must-have feature for many embedded systems.

9

OPERATING SYSTEMS

OS features may vary depending on the capability of the device

- In the simplest case (bare-metal) it may just be a minimal, real-time support
  - In form of libraries statically linked to the code
- In more powerful systems can be a modern OS (like an embedded version of Linux) or a real-time system (like ROS or others)
- We will see examples related to the first case. For the second case you can refer to the OS class.

• When you don't have a real full OS ; to : control, debug, etc.. need rely to remote PC
⇒ perform CROSS-compilation

- compile on host PC in embedded device and upload the code on it

10

CROSS-COMPILATION

- The code is written and compiled on a host (usually a PC) for the platform on which it will be executed (cross-compilation)
  - At compile time the programmer specifies the target platform (for example Iris, MicaZ, T-Mote, …)
- The code is statically linked to the libraries that implement the operating system
- After compilation the executable can be uploaded in the firmware of the device
  - It's also possible to upload the executable later and on other equivalent devices

11

THE EXCUTABLE

- The executable code comprises:
  - The programmer's code
  - Libraries of the OS
  - All linked into a unique executable program
  - The function `main` is provided by the OS
    - Invokes all initialization functions written by the programmer
- Whenever the device is turned on it starts executing the `main` :
  - First initializes the devices
  - Then activates the tasks that implement the functionalities of the device

12

CROSS COMPILATION ⟶ executable file: sort of static code that comprises:

1) the code
2) OS: usually in libraries to implement interconnection with HW

- embedded sys usually in C: easy control/manipulate HW:

1) play with registry
2) interface with assembly
3) Memory mngment at low level

⊕ efficient
⊖ errors/BUG

- usually starts with main() but in this context:

- is provided by the programming environment, like provided by OS
- initialize the device (boots the device):
  ⇒ cut part of HW may need more initialization, useless ask to programer

- after initialization, ⇒ pass the control to your functions ⇒

1) business logic AT runtime
2) add initial configs.

- includes the setup/configure device when is turned-on
- may also invoke your INITIALIZATION FUNCTIONS
- after that ⇒ your business logic

- code is cyclic:

1) READ some data from transducers
2) PROCESS them
3) CONTROL ACTUATORS (leds or display are kinds actuators)
4) STORE something on flash
5) SEND data
6) WAIT for sampling next time

RUNTIME BUSINESS LOGIC ⇒ repetition guided by RUNTIME SUPPORT = OS (loop already built in) ⇒ control loop

- sampling requires STRICTLY PERIODIC ACTIVITIES: reading at regular interval ⇒ low at regular interval
- how your code manages interaction with the HW (key point)

- PC'S OS: interaction with HW by OS ⇒ abstracts interaction with the HW:

- sending commands to HW & receive notifications when commands are executed
  ⇓
  INTERRUPTS

- INTERRUPTS: when you write a program you don't deal with interrupts & command with HW
  ⇒ abstraction by OS: that transforms commands/interrupt ⇒ SOMETHING ELSE:

① < 1) PROCESS/THREAD abstraction
2) LAYER OF LIBRARIES: to map functionality of HW in FUNCTIONS you can INVOKE

- in practice:

1) Commands by invoking high-level functions => send commands to HW
2) => 2) SYNCHRONISATION: what is achived by OS by interfacing interrupt level:
   - when HW completes an operation sends: ASYNCHRONOUS INTERRUPTS received by OS,
   - OS reacts async. to this request of the device

- So when you call a function => interaction with HW, if HW is slow => function have to wait operation is completed
  => implements SUCH SYNCHRONISATION: processes / threads

- KEY:
  - your program is executed in a thread
  - thread invokes a function
  - function realizes there's time to wait (before HW operation is completed)
    - rather that stopping and performing active waiting => function request to OS to SUSPEND THE THREAD.

- SUSPENSION: your activity IS FROZEN => state of activity is KEPT in dedicated DS (stack / thread descriptor)
  - microprocessor if free to do else (run other threads)
  - once INTERRUPT comes from HW => CAPTURED by OS
    - OS detects that an operation of a thread is completed
    - restore the STATE OF THAT THREAD => executed again
  - THREAD POV: no time passes, real world => TIMELAPSE between send a command and execution

- CONSEQUENCE:
  - high abstraction
  - you don't care about SYNCHRONIZATION
  - easy programming & use of HW
  - COST: - memory to store state of threads (thread descriptor)
    => space occupied on stack by threads

→ code is cyclic: read disk
- process
- control actuators (like display)
- may store something on flash (disk)
- may transmit/receive
→ wait until you sample again (sampling rate strictly paired activity at regular intervals)

repetition of this provide by OS / runtime ENV. => No write loops => control loop?

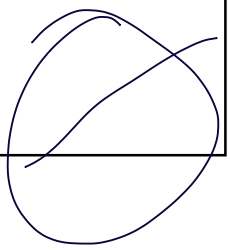
cuz new need initial configs => useless doing this bc supervisor => OS

neutrino Bosch logic

SW ORGANIZATION

- The job of an embedded system is cyclic
  - Natually defines a duty cycle
- Usually consists in:
  - Reading from transducers
  - Taking a decision
  - Controlling actuators
  - Optionally communicating with other devices
- All these steps executed in a control loop
  - A function invoked by the main after initialization

13



- How code interacts ⊗ => key point: CPS using sensor and actuator => virtual interaction with HW, mediated by OS:
low level: send command to HW and receive notification (interrupt)
e.g.: display

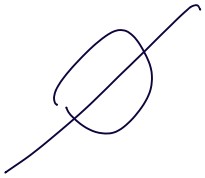
then transition after/except for something else:
1) Process/thread abstraction
2) layer of libraries

in practice: "commands" by invoking functions at high level, sending commands to HW

INTERACTION WITH THE HARDWARE

- At low-level, the code interacts with the device hardware by means of commands and interrupts
  - A command activates the hardware (for example, to read from a transducer)
  - An interrupt signals that the command has been executed
- In conventional O.S.:
  - The commands are available in terms of system calls
  - The user code invokes the system call
  - If there is to wait, the invoking thread is suspended
  - The interrupts are managed by the OS kernel that re-activates the suspended threads

14



- Invoke f(x)
- Eventually receive output of f(x)

if f(x) needs interaction with HW: how visible low => sync =>

2) SYNC:
achieved by OS using
- interface at interrupt level: async nature of device, response async

INTERACTION WITH THE HARDWARE

- In practice, in conventional OS the programmer's code just see a library stub function that executes the command and returns a value
- The suspension of the thread is transparent to the user …
- … this requires a stack for each thread to store its context …
- … but what if memory is not much?
- Consider that in an Arduino 1, RAM (data) memory is only 4 KB, that serves for buffers, variables and now also stacks and context of threads? (ISRs for invoking functions)

=> with real time support, implement thread sw => NOWAY

INTERACTION OF THE SW WITH THE HW

- NO THREADS
- In the case the memory is not sufficient, OSs for (low-power) embedded systems use different strategies, SYNCR. IN 2 WAYS
- Two examples:
  1) Arduino ~~event~~ normal loop: 1 THREAD THAT IS SUSPENDED -> no memory consumed for that
  2) TinyOS & NesC event-based programming and tasks: ② use directly THE INTERRUPTS: -you do what os does

memories freezed until HW does its thing

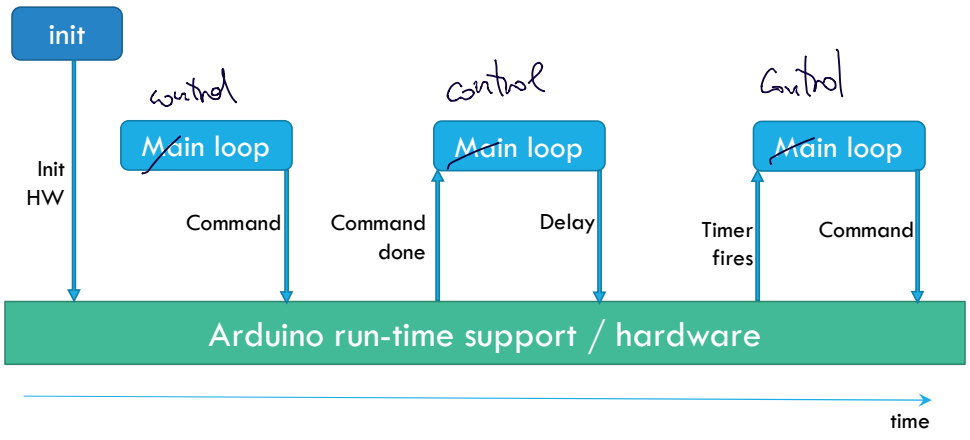
THE ARDUINO MODEL

- The job of Arduino is defined in a single loop function => control loop
  - It may invoke other functions anyway
- The loop is executed repeatedly by a single thread
- No suspension of the thread => ACTIVE WAIT
- If there is an I/O just wait until the I/O is complete
  - … but no other threads are scheduled!
- If you ~~want~~ need to delay the execution use explicit delays in the code
  - The thread sleeps for the desired time

• may need to implement a DELAY: (in the control loop)
=> can you implement periodic sample of the sensors => rigid interval of sampling
=> wait to perform next operation
=> in CONTROL LOOP => command is DELAY => active delay, doesn't freeze anything
=> NOT SAVES ENERGY, just waits

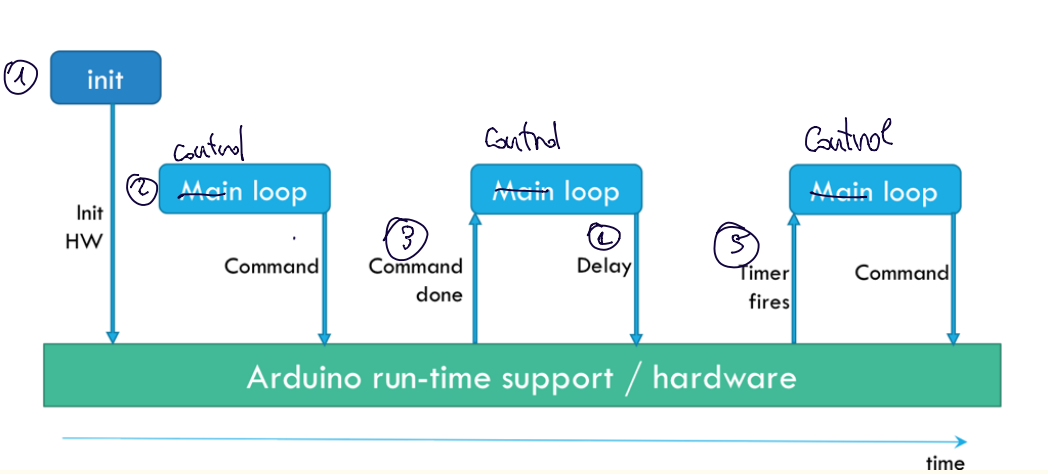
17

FLOW OF EXECUTION IN ARDUINO



18

ARDUINO'S FLOW EXECUTION



1) INITIALIZATION of the HW: executed by the main() provided by runtime-support and by programmer, after initialization, executes

2) CONTROL LOOP: - may send command to HW => implemented by invocation of functions of run-time support (invocation of libraries) => little abstraction of the HW
- libraries execute this command by sending a low level command to HW and wait until completed

3) COMMAND DONE: by returning => command is done (by lib.)

=> like printf / send

4) DELAY: if you're to wait, delay is just another command

5) TIMER FIRES: you're wakeen-up returning by libraries when the delay is concluded

=> this schema isn't good with something ASYNCHRONOUS with your code: id est: COMMUNICATION among devices (can't predict when a device will communicate with you)

- COMMUNICATE => ASYNCHRONOUS
- SENSING => SYNCHRONOUS

=> additional supports 4 asynchronous events => interface to deal with interrupts

TINY OS

- designed for low end-devices
- supports wirless/sensor networks

- to support networks of embedded device/communication => deal with managing ASYNCHRONOUS EVENTS
- abstractions of:
  1) COMMANDS: libraries that abstract functionalities of the HW => invoke commands than low-level programs HW
  2) EVENTS: basically interrupts:
     - at a low level, HW receives an interrupt
     - interrupt managed by INTERRUPT-HANDLER; sw provided by run-time support/library
       - rises up an UP-CALL: invokes a function/handler in the program
     - events managed ASYNCHRONOUSLY by event-handler (defined by the programmer)
       (like async programming with BUTTONS)

- what happens:
  1) program handlers
  2) link handlers to the event you want to intercept
  3) OS executes the handler

- in practice, when executing an interrupt handler:
  - is as if you're still in the interrupt handler (at low level)
  => BAD IDEA DO STUFF WITH INTERRUPT HANDLER: (CUT)
  1) SYSTEM IS NOT RESPONSIVE: "An interrupt cannot be interrupted by another INTERRUPT"

- When managing an interrupt, another may comes => what do you do? This interrupt may request to execute another handler and is possible -> depends on OS
  => easy way is to avoid the management of an interrupt when you're serving another interrupt

- PROBLEM IS: when you're managing an interrupt, you're dealing with variables, DS, in kernel OS.
  - if you're interrupted in between, you're leaving DS, var with a temporary inconsistently value,
  => the value isn't inconsistent by itself, you've not completed what you were doing,
  SO IS INCONSISTENT CUZ YOU'VE JUST BEEN INTERRUPTED in between, but as soon you're free to continue the interrupt handler => you'll leave everything inconsistent
  => another interrupt handler arises and uses same DS in kernel space => RACE CONDITION

=> so OS manages RACE CONDITIONS by disabling/enabling interrupts managing the concurrency || avoid them intently

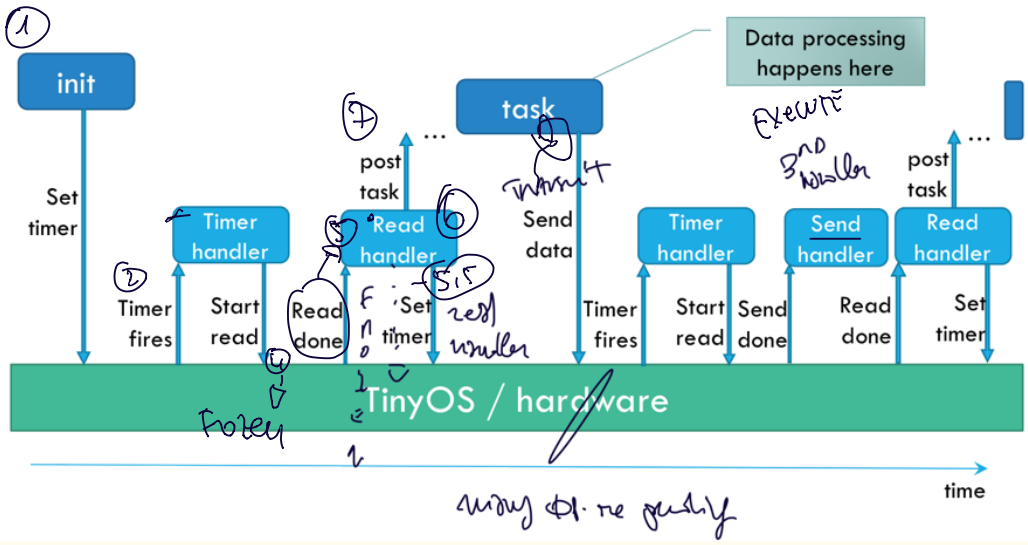
- PREVENTING THEM:
  - when you receive an interrupt => disable interrupts
  - execute the handler
  - enable interrupts at the end => STAY blocked and wait 1 hour

- if you've devices is implemented with interrupts to make it responsive to async events => no place where to put processing activity => idea: create an istruction (TASK): like a "thread"

- a task in this OS is something you can post, so you tell to os: "I'd to operate this task"
  => OS will run the task as soon when microprocessor is free

- TASKS ARE NOT PRE-EMPTABLE: if you post 2 tasks, they will be EXECUTED IN SEQUENCE
  - in tasks you put complex processing, it you receive an interrupt during a task => INTERRUPT has privilege
  cuz is used to have async interaction with the world => high priority respect to the tasks
  => tasks not preemtable => don't needed save state, cuz you're not scheduling tasks => just run task, end, run another
  => when you receive an interrupt => PREEMPT the task to -> the handler -> CONSUME MEMORY (save state of the task)

SAMPLE FLOW OF EXECUTION IN TINYOS



- no control loop here:
if you wanna
=> implement by yourself

(1) INITIALIZATION: provided by runtime support AND implemented by your code
- set timer: to tell to device when wake-up to implement next sensing activity
- after INIT => turn off everything, don't do anything,
=> JUST WAIT to be waked-up by the INTERRUPT

(2) when TIMER FIRES: - execute interrupt handler: send command to HW to read data
(you're in an interrupt handler => it will stop shortly,
send the command, no wait command will be completed)

(4) Frozen until HW complete the operation

(5) when is completed, HW sends an interrupt (event => READ DONE) =>

(6) OS EXECUTE READ HANDLER (≠ timer handler): - related when read operation of the transducer is completed

- In the event handler of READ operation you've the value you'd => cannot process immediately in read handler coz you want read handler will be fast
=> you can just POST A TASK (7) => processing happens here

(5.1) in read handler you've been notified there's a data received from the sensor
=> have to program next read operation => must be performed to a given time
=> set the time to wake-up at next read operation

=> After this read handler can stop,
=> OS knows nothing else to do but there's a task posted => executes the task

processing data read before

=> after process, may transmit to other device
=> the task sends invokes a command to OS to implement a send

- Can't be immediately, takes time => OS will return ok, may can be successful
=> you can do within the task => data will be sent, complete the task and return to OS

=> now many operations are pending:
1) you've requested an interrupt for time to do next sampling, sent req. to OS to do the send by ≠ HW => wait interface
=> 2 ACTIVITIES undergoing by HW => 2 will arise interrupts soon or later

THE ARDUINO MODEL

Simple model:

- Fits well simple sensing and control activities
  - Includes several libraries for actuators
- Not borne for communications
  - Although recent versions include libraries (and HW) for this

… but it also offers asynchronous event management

- Necessary to manage communications through serial line

19

THE TINYOS MODEL: EVENTS, COMMANDS AND TASKS

- TinyOS offers functions (**commands**) to program and activate the hardware
- It abstracts interrupts in form of **events**
  - A sort of upcall, the programmer has to define a handler for each event
- It defines non-preemptive **tasks** that are executed sequentially
  - To manage different independent activities

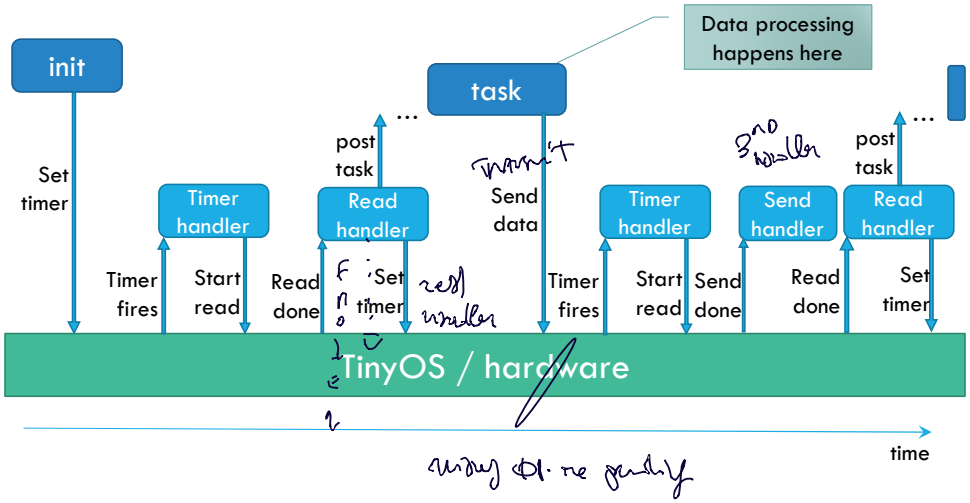
20

TINYOS: EVENTS, COMMANDS AND TASKS

- With this approach a task never waits
  - Saves memory to store its context
- However interrupts should be handled as soon as they arrive
  - By means of an event handler
  - Tasks can be pre-empted (only) by events
- The event handlers should be as short as possible
  - Can update data structures
  - can give commands to the HW
  - If their management requires more work ⇒ post a task

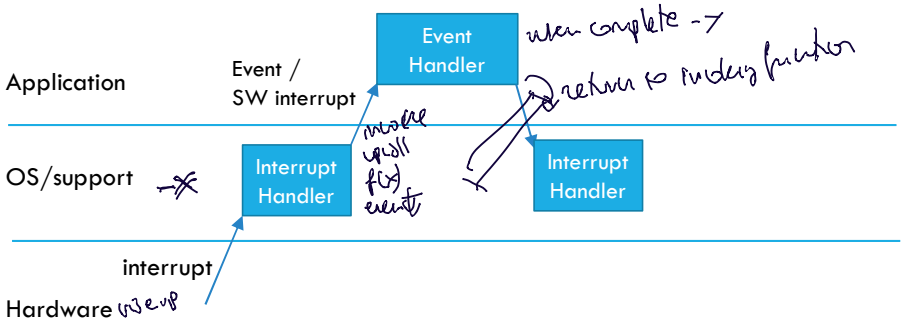
21

SAMPLE FLOW OF EXECUTION IN TINYOS



22

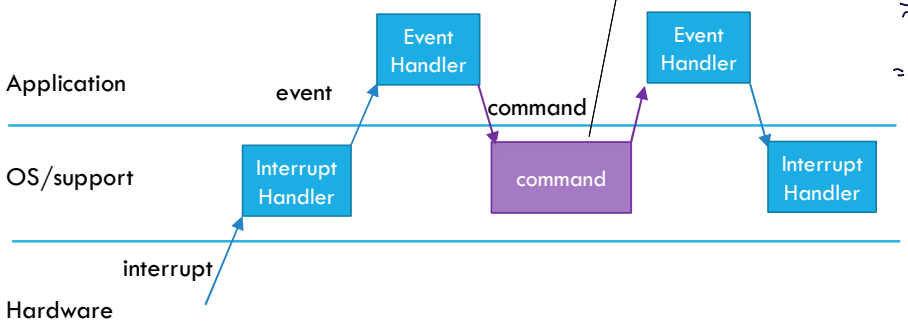
EVENTS EXECUTION MODEL (1)



23

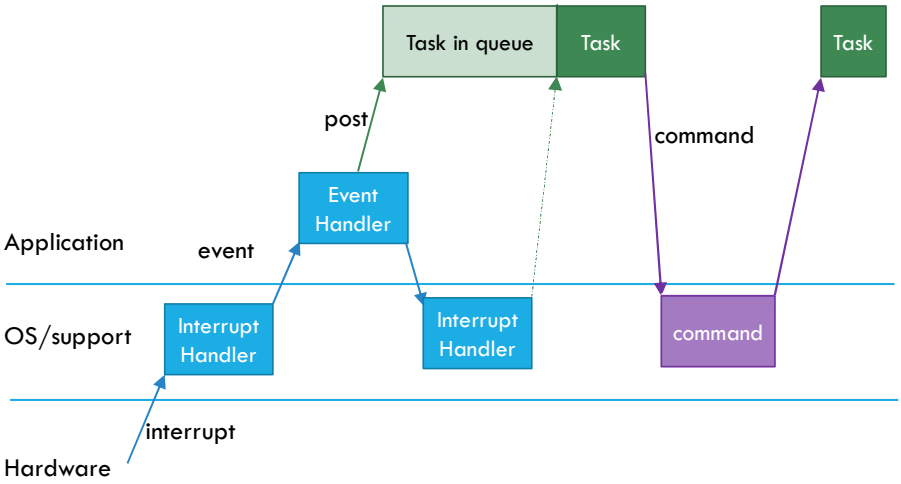
EVENTS EXECUTION MODEL (2)

Careful: risk of interference with OS/support data structures !



24

EVENTS EXECUTION MODEL (3)

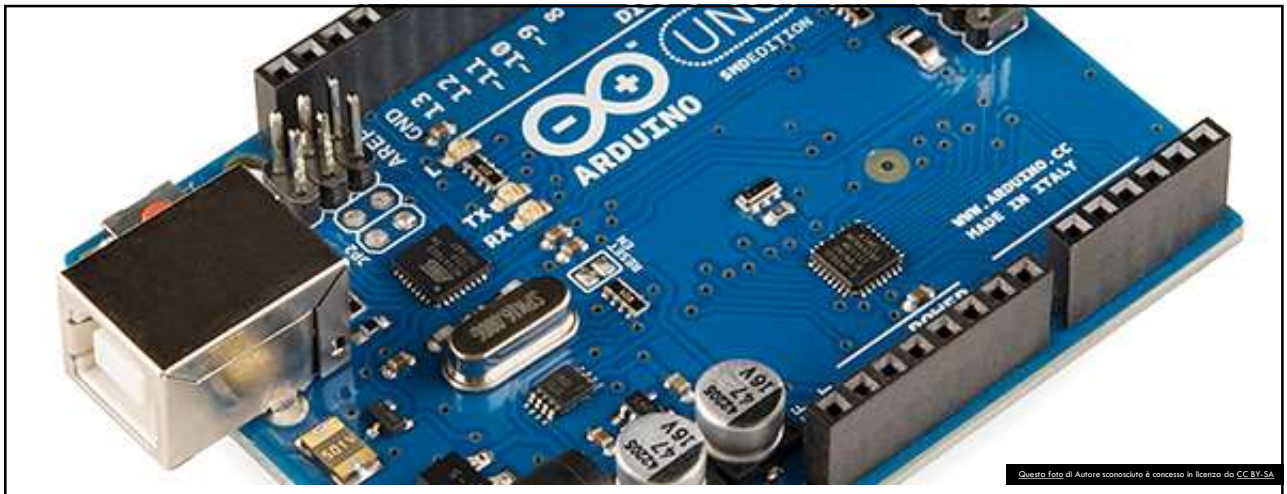


25

ARDUINO INTERRUPTS

- May be used to let Arduino go in low-power mode
  - Do not require to save state of the process
- Interrupts should be handled as soon as they arrive
  - By means of an event handler
- The event handlers should be as short as possible
  - Can update data structures
  - can give commands to the HW
  - Some commands are forbidden here…
- Make it similar to the TinyOS model

26



CASE STUDY: ARDUINO

27

THE IDEA OF ARDUINO

Arduino is an **open-source electronics prototyping**

platform based on flexible,

easy-to-use hardware and software.

It's intended for artists, designers, hobbyists and anyone interested in **creating interactive objects or environments.**

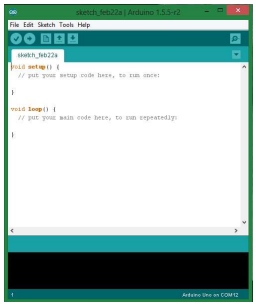
28

"ARDUINO"

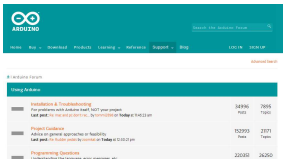
"Arduino" concept comprises:

- Device

- IDE



- Forum



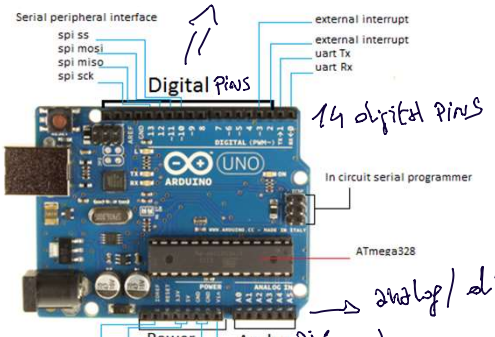
29

ARDUINO UNO

AVR Arduino microcontroller
Atmega328

- SRAM 2KB (data) code ≤ 2kB
- EEPROM 1KB
- Flash memory 32 KB (program)

I/O
Binary input / output



- circuit is powered with 5 volts
  => value 1 corresponds to
  $0{,}005 \simeq \frac{5V}{1024} = \frac{5V}{2^{10\text{ bit converter}}}$
- analog / digital converter: analog signal to 10-bit #
- $\frac{5V}{1024}$
- voltage to external stuff

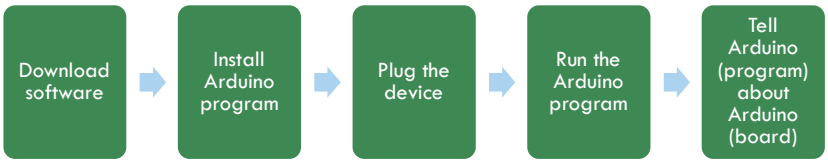
30

- BINARY I/O => digital pins
- ANALOG Input => analog pin => can't control a device with analog signal but solved cant some digital pins can emit SQUARE WAVES => complex control in output over digital pins

Software: preparing the environment

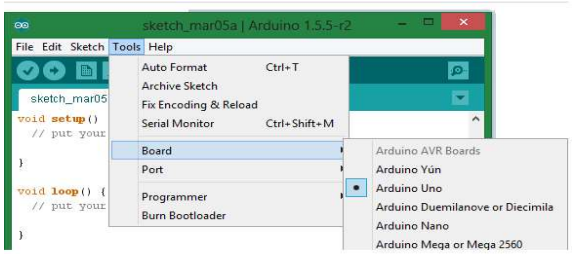
The open-source Arduino environment makes it easy to write code and upload it to the I/O board. It runs on Windows, Mac OS X, and Linux. The environment is written in Java and based on Processing, avr-gcc, and other open source software.

Arduino IDE can be downloaded at www.arduino.cc

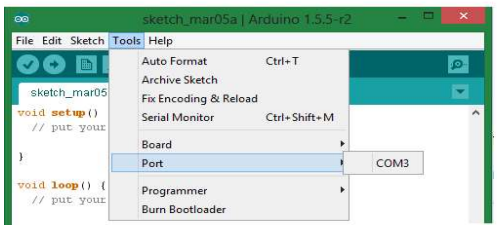


31

SELECT TYPE AND LOCATION



Select your arduino



Select the location of device

32

TERMINOLOGY

"*sketch*" – a program you write to run on an Arduino board

"*pin*" – an input or output connected to something.
- e.g. output to an LED, input from a knob.

"*digital*" – value is either HIGH or LOW.
- (aka on/off, one/zero) e.g. switch state

"*analog*" – value ranges, usually from 0-1023.
- e.g. LED brightness, motor speed, etc.

Always transformed to digital number 0-1023

33

IDE

Verify

Upload

New

Open

Save

Serial monitor

Toolbar buttons

Sketch editor

Console display

34

LANGUAGE

- The Arduino environment is based on Atmel Atmega microcontrollers (AVR family of microcontrollers).
- The language is "C"-like.
- The programs can be divided in three main parts (see figure)
- Many libraries for all Arduino shields and components available: http://arduino.cc/en/Reference/Libraries

- Variables
- Functions
- Sketch Structure :
  - setup } void
  - loop

35

SKETCH AND ITS STRUCTURE

SETUP => called automatically by mcu:
=> initial configuration (at boot)

```
void setup() {
  // put your setup code here, to run once:

}
```

loop invoked by mcu continuously after initialization

```
void loop() {
  // put your main code here, to run repeatedly:

}
```

Called when a sketch starts.
The setup function will only run once.

Does precisely what its name suggests, and loops consecutively.

36

OTHER FUNCTIONS & STRUCTURES

Control Structures: same as C;

Further Syntax: ;, {}, //, /**/, #include, #define;

Arithmetic Operators: +, -, =, /, *, %;

Comparison Operators: ==, !=, <, >, <=, >=;

Boolean Operators: &&, ||, !;

Pointer Access Operators: *, &;

Bitwise Operators: &, |, ^, >>, <<, ~;

Compound Operators: ++, --, ==, +=, -=, *=, /=, &=, |=;

37

VARIABLES

Constants: level of energy (HIGH; LOW); mode of pin (INPUT; OUTPUT; INPUT_PULLUP); led13(LED_BUILTIN);…;

Types: word; string;…;

Conversions: word();…;

Variable scope and qualifiers: volatile;…;

Usefulness: sizeof();

38

va messo nel setup(): configura i piedini digitali per renderli I/O

es.

```
void setup() {
  pinMode(3, OUTPUT);
  pinMode(4, OUTPUT);
  pinMode(2, INPUT);
}
```

HIGH = 1
LOW = 0

imposto un output 2 ora 5V (high) (low)

FUNCTIONS

- Digital functions: pinMode(); digitalRead(); digitalWrite();
- Analog functions: analogReference(); analogRead(); analogWrite();
- Advanced I/O: tone(); noTone(); shiftOut(); shiftIn(); pulseIn();
- Time: millis(); micros(); delay(); delayMicroseconds();
- Math/trigonometry: min(); max(); abs(); sin(); cos(); …
- Random Numbers: randomSeed(); random();
- Bits and Bytes: lowByte(); highByte(); bitRead(); bitWrite(); bitSet(); bitClear(); bit();
- External Interrupts: attachInterrupt() detachInterrupt();
- Interrupts: interrupts(); noInterrupts();
- Communication: Serial; Stream;

delay → in millisecond
1 sec = 1000 msec

39

AN EXAMPLE: ANALOG & DIGITAL READ

```
// Reads an analog input on pin 0 and converts it to voltage;
// Reads a digital input on pin 2;
// When input on pin 2 is high prints voltage in the serial and   switches
the led.

const int buttonPin = 2;    // the number of the pushbutton pin
const int ledPin =  13;     // the number of the LED pin
int buttonState = 0;        // variable to read the pushbutton status

void setup() {              // runs once when you press reset
  Serial.begin(9600);       // init serial lineat 9600 bps (bit per sec)
  pinMode(ledPin, OUTPUT);  // init. the LED pin as an output
  pinMode(buttonPin, INPUT);  // init. pushbutton pin as an input
}
```

configura SERIAL LINE TO MONITOR WITH PC

NECESSARIO serial line receives the speed (have to match with speed of desktop site)

PIN MODE: pin # led pin = 13 will be used in output other in input

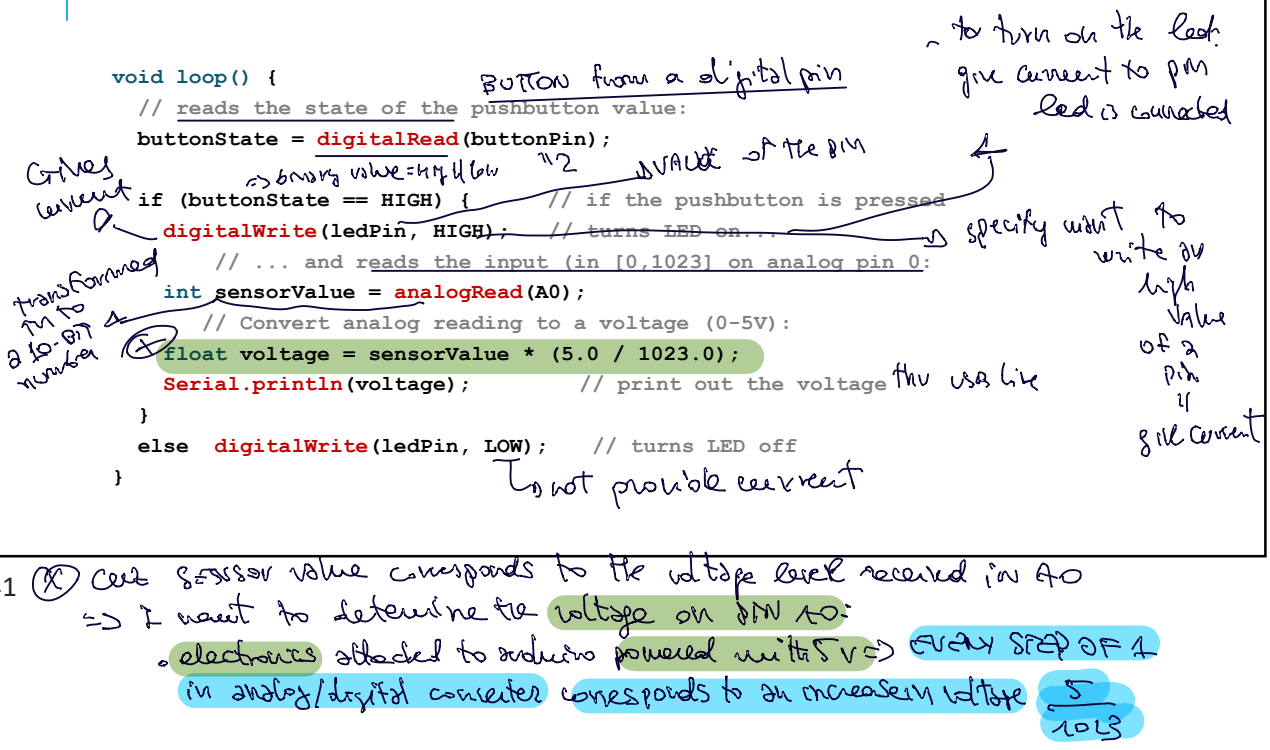
40

AN EXAMPLE: ANALOG & DIGITAL READ

```
void loop() {
  // reads the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  if (buttonState == HIGH) {     // if the pushbutton is pressed
    digitalWrite(ledPin, HIGH);  // turns LED on...
    // ... and reads the input (in [0,1023] on analog pin 0:
    int sensorValue = analogRead(A0);
    // Convert analog reading to a voltage (0-5V):
    float voltage = sensorValue * (5.0 / 1023.0);
    Serial.println(voltage);          // print out the voltage
  }
  else  digitalWrite(ledPin, LOW);    // turns LED off
}
```

41



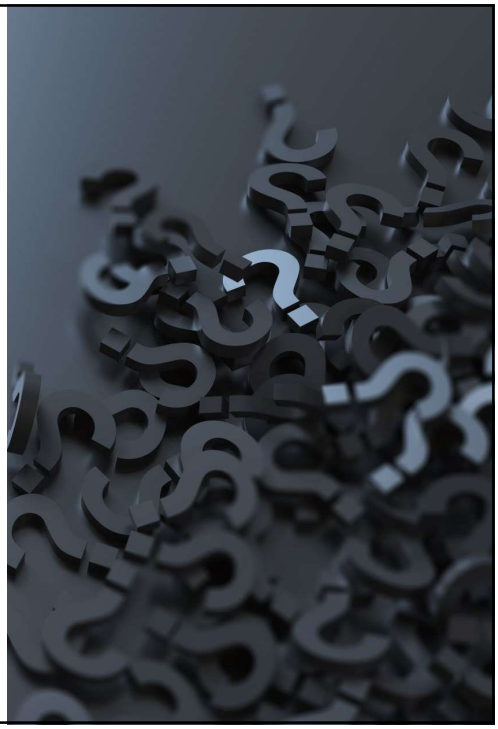
EXERCISE

Consider the following fragment of an Arduino code:

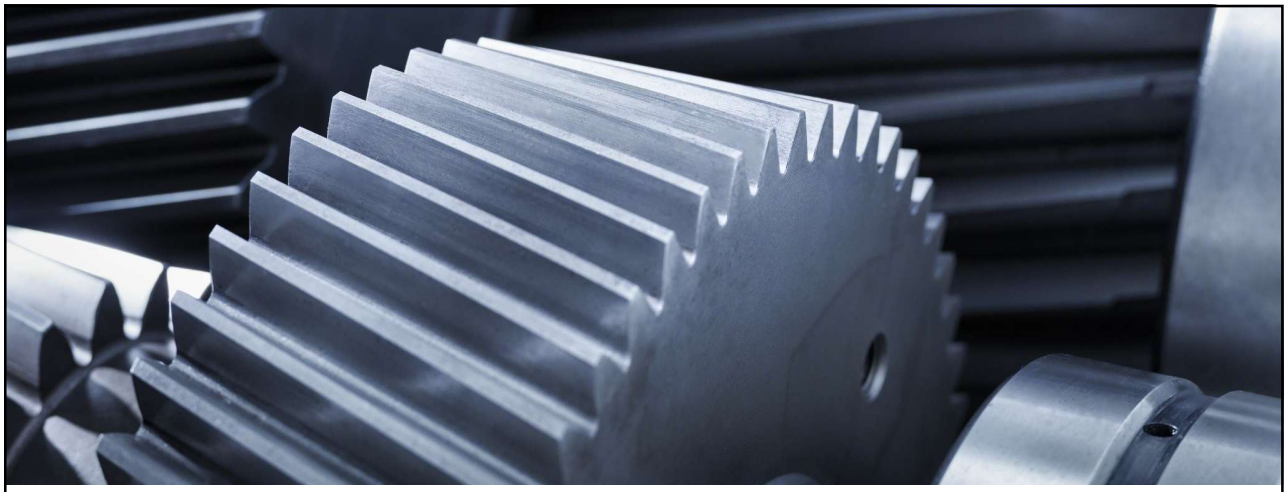
```
void loop() {
   int sensorValue = analogRead(A0);
   Serial.println(sensorValue);
   delay(100);
}
```

Compute its duty cycle assuming that:

- Reading an analog value takes 2 milliseconds
- Transmitting along the serial line takes 5 milliseconds



42



INTERRUPTS

43

- If connected to arduino wifi/bluethood cards: transmit ~~communicate~~ via Serial.println();

- Interrupts to manage ASYNC interaction with external environment

- Similar to Arduino's schema:
  1) Interrupt arrives to microcontroller
  2) microc. executes INTERRUPT HANDLER of the real-time suppor
  3) Runtime support invokes your handler

ARDUINO: INTERRUPTS

Go directly to your code

managed internal runtime support

with delay you're using timer

Although the basic schema of Arduino sketches is the synchronous reading of sensors, Arduino also provides an interface to interrupts

- This enables an asynchronous access to sensor data and to actuators

On Arduino there are three types of interrupts:

- External: a signal outside Arduino, connected to a pin
- Timer: internal to Arduino
- Device: an internal signal coming from a device (ADC, serial line etc.)

44

- DECLARE AN INTERRUPT: attachInterrupt() : → used in initialization of the device

ARDUINO: INTERRUPTS

attachInterrupt(interrupt#, function-name, mode)

- interrupt#: # of interrupt you want to connect
- function-name: function in your code
- mode: sort of HW support to interpreter a signal at low-level on pin to take a decision to rise up the interrupt

interrupt connected to pins
=> detect async events that happen in the world

Internal interrupts (either Timer or Device) are managed by the run time support of Arduino

We focus instead on the external interrupts

- Supported by Arduino the run-time:
  `attachInterrupt(interrupt#,func-name,mode);`
- Note that some versions of Arduino also support an additional type of interrupts, called «pin change»

external interrupt

attachInterrupt(
interrupt #,
func_name
mode;)

ARDUINO: EXTERNAL INTERRUPTS

- Only 2 external interrupts connected to: PIN 2 e PIN 3 → corresponds to → interrupt 0, interrupt 1
- MODE: 1 to many
- es.: rising:
  - attachInterrupt(0, ..., rising)
  - => int 0 connected to PIN 2
  - if signal of PIN 2 passes from voltage 0 to 5 => signal is RISING and rises up an interrupt that invokes your function

There are only two external interrupt pins in Arduino:

- INT0 and INT1 (rising/CHANGE)
- They are mapped to pins 2 and 3 (I2, I3) external
- They can be set to trigger on RISING or FALLING signal edges, on CHANGE or on LOW level. (, HIGH)
- The triggers are interpreted by hardware, and the interrupt is very fast.

Can send an interrupt every time the signal is high

ARDUINO: EXTERNAL INTERRUPTS

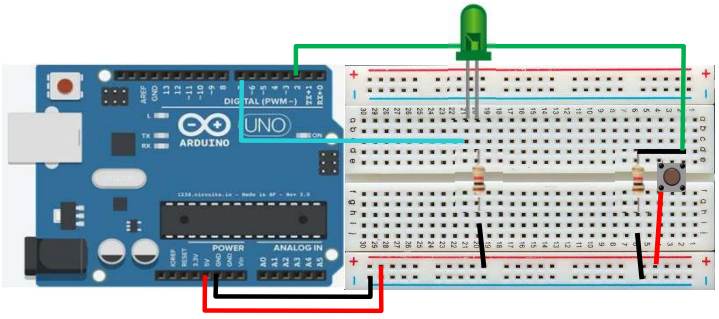
- RISING:
  the interrupt occurs when the pin passes from LOW to HIGH state
- FALLING:
  the interrupt occurs when the pin passes from HIGH to LOWstate
- CHANGE:
  the interrupt occurs when the pin switches state
- LOW:
  the interrupt occurs whenever the pin has LOW state
  - Not necessary a change of state. If it remains LOW the interrupt occurs again
- HIGH:
  the interrupt occurs whenever the pin has HIGH state
  - Not necessary a change of state. If it remains HIGH the interrupt occurs again

47

INTERRUPTS – EXAMPLE

Connections:

- A button to digital input 2
  - With 10KΩ resistor
- A led to digital output 7
  - With 220Ω resistor



48

```
/**
 * test interrupt
 *
 */

volatile int greenLed=7;
volatile int count=0;

void setup()
{
  Serial.begin(9600);
  pinMode(greenLed, OUTPUT);
  digitalWrite(greenLed, LOW);

  attachInterrupt(0,
      interruptSwitchGreen,RISING);
  // 0 because it is pin 2
}

void loop() {
  count++;
  delay(1000);//WAIT 1record
  // interrupts are received
  // also within delay!
  Serial.print("waiting:");
  Serial.println(count);
  if ( count == 10 )
  {
    count = 0;//reset count
    digitalWrite(greenLed, LOW);//turn off the led
    Serial.println("now off");
  }
}

void interruptSwitchGreen()
{
  digitalWrite(greenLed, HIGH);
  count=0;
  Serial.print("now on");
}
```


ARDUINO: EXTERNAL INTERRUPTS

Note:

- delay() does not work within the interrupt handler
- millis() is not incremented in the interrupt handler
- Count is declared as «volatile» bacause it is modified in the interrupt handler
  - This is a compiler directive that directs the compiler to load the variable from RAM and not from a storage register
  - In this way, if it is modified somewhere, we always get the correct value
  - To be used for all variables shared between loop and interrupt handler!
- greenLed is never modified, we can avoid «volatile» here

1) POTENTIAL RACE CONDITION:

- cvt variable count used in interrupt handler and in the loop
  => any time can be modified either by control loop or the handler (R. C.)
- to keep consistent value of count => manage RC between interrupt and control loop

  => SOLVED cvt interruptSwitchGreen() is always executed disabling interrupts (not be interrupted while executing interruptSwitchGreen())

2) Compiler may allocate count in register for efficiency:

- var count always kept in REGISTER
  => if count=7 and interrupt arrives => micro controller store all registers on STACK (including count=7)
  => executes interruptSwitchGreen() and set count = 0; => end interrupt
  => have to resume loop => microcontroller fetches registers value saved in the stack
  => copy them in register, write count = 7 => doesn't work

- AVOID THIS DECLARING count as VOLATILE, => force compiler to keep variable count always in MEMORY (at any time)
  => if want to read/write variable count => always access to the memory
- GOOD PRACTICE: declare volatile all variables in interruptHandler and in the loop
- INTERRUPT HANDLER always void and NO PARAMS:
  - 1) cvt you don't know when the interrupt procedure will be executed
  - 2) this procedure will be invoked implicitly by reception of the interrupt
    - interrupt: just a signal on a pin (no params associated to interrupt)
    - general interrupt
  - 3) no invocation of interruptSwitchGreen() => no point in the code where recive any return value

=> 4) Handlers cannot also do complex operations, cvt one executed with disabled interrupt, if you do lot of work => interrupt disable a long time => bad idea

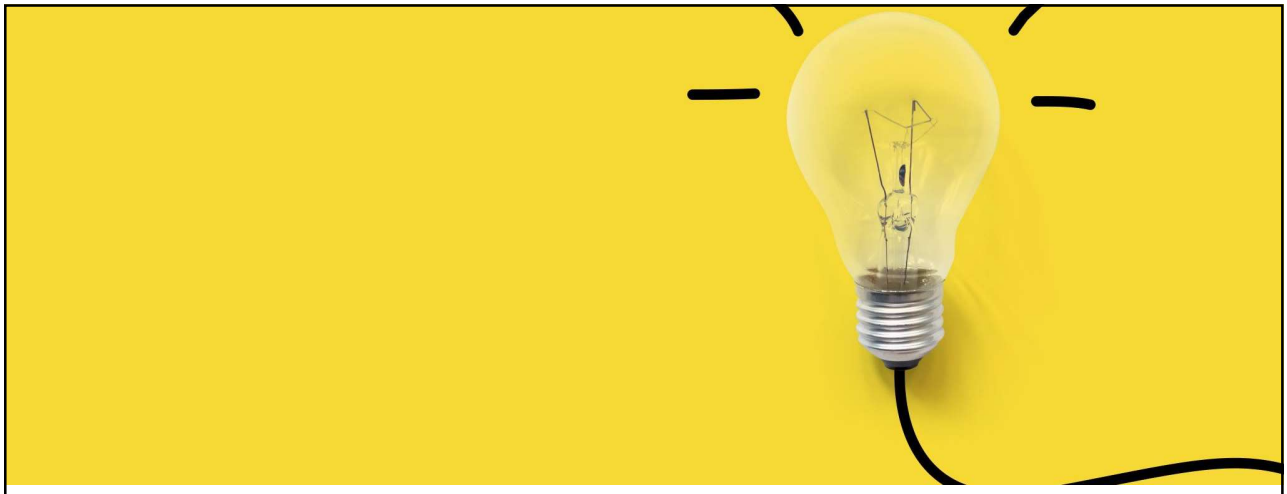
- Save reason, in interrupt_handler NO DELAY
  (L) when esp. a delay, freeze micro controller and waken up by an interrupt (interrupt are disabled and you never wake up)

ARDUINO: EXTERNAL INTERRUPTS

- Other functions:
  - `detachInterrupt()` disconnects the interrupt – cancels a previous attachInterrupt()
  - `interrupts()` enable interrupts
  - `noInterrupts()` disable interrupts

51

ENERGY MANAGEMENT IN ARDUINO



52

. # states in microcontroller : # micros may have # STATES

• STANDARD LIBs of arduino don't save energy, => delay() just keep waiting, microcontroller is "frozen" but not put in an idle state, still active and powered

OF ARDUINO BOARDS

• # microcontroller supports # STATES : core dipende

=> LIBRARIES to activate energy efficiency modes on microcontroller, board, whether, all components

ENERGY MANAGEMENT IN ARDUINO

Arduino Sleep Modes to limit power consumption (AKA Arduino Power Save mode)

Can be activated by means of the Low power library (https://github.com/rocketscream/Low-Power)

- Provide calls to switch the processor and the system components in low-power mode → to implement low power state of microcontroller
- Wake-up mechanisms either on internal or on external interrupts (or on system reset)

The actual sleep mode mechanism depends on the version of the device and of the processor (microcontroller)

• common ways:
- micro
- memory
- digital/analog conv.

53

• LowPower.h : LIBRARY

low power state

to send bits to controller, turns off

Mode	Features
Idle	Stops the CPU but SPI, 2-wire serial interface, USART, Watchdog, counters, analog comparator remain active (it basically stops the CLK_{CPU} and CLK_{FLASH}). Arduino waked up any time by using external or internal interrupt.
ADC Noise Reduction Mode	Stops the CPU but ADC, external interrupt, USART, 2-wire serial interface, Watchdog, and counters operate (it basically stops the CLK_{CPU}, $CLK_{I/O}$ and CLK_{FLASH}). Can wake up by external reset, serial interface and specific kinds of interrupts.
Power-Down Mode	Stops all the generated clocks and allows only the operation of asynchronous modules. Turns off the external oscillator, but the 2-wire serial interface, watchdog and external interrupt operate. Can wake up by external reset, serial interface and specific kinds of interrupts (no timer interrupt for example)
Power-Save Mode	Similar to power-down, but if the timer/counter is enabled, it keeps running even during sleep time. In addition to the other methods, the device wakes up also on timer overflow. If time/counter is not used: recommended to use Power-down mode instead of power-save mode.
Standby Mode	It is similar to Power-Down mode, but the external oscillator remains active
Extended Standby Mode	It is similar to the power-save mode only with one exception that the oscillator keeps running.

if not interrupt from anywhere else

SLEEP MODES IN ATMEGA328P ARDUINO

54

3 interrupt: clock, interne comp/ external component

USING IDLE SLEEP MODE

Need to include the library (this for all sleep modes):

```
#include "LowPower.h"  //Note: no setup is required
```

Example to switch to idle sleep mode:

```
LowPower.idle(SLEEP_8S, ADC_OFF, TIMER2_OFF, TIMER1_OFF,
TIMER0_OFF, SPI_OFF, USART0_OFF, TWI_OFF);
```

Note:

- sleeps for 8 seconds and then wakes up automatically the device
- it turns off all the timers, SPI, USART, and TWI (2-wire interface).

55

USING IDLE SLEEP MODE

Note that the following sleep periods are defined:

```
SLEEP_15MS,   SLEEP_30MS,
SLEEP_60MS,   SLEEP_120MS,
SLEEP_250MS,  SLEEP_500MS,
SLEEP_1S,     SLEEP_2S,
SLEEP_4S,     SLEEP_8S,
SLEEP_FOREVER
```

You don't need to turn off everything…

```
ADC_OFF / ADC_ON
TIMER0_OFF/ TIMER0_ON
```

Etc…

56

USING POWER-DOWN MODE

Need to include the library (this for all sleep modes):

```
#include "LowPower.h"  //Note: no setup is required
```

Example to switch to power-down mode:

```
LowPower.powerDown(SLEEP_8S, ADC_OFF, BOD_OFF);
```

Note:

- sleeps for 8 seconds and then wakes up automatically the device
- but you can also set Arduino to wake up when an interrupt is received…

57

USING POWER-DOWN MODE

…but you can also set Arduino to wake up when an interrupt is received:

```
void loop()
{
    // Allow wake up pin to trigger interrupt on low.
    attachInterrupt(0, wakeUp, LOW);

    LowPower.powerDown(SLEEP_FOREVER, ADC_OFF, BOD_OFF);
    // Disable external pin interrupt on wake up pin.

    detachInterrupt(0);

    // Do something here
}
```

58

ARDUINO FORUM & SUPPORT

Support for arduino programmer: http://forum.arduino.cc

Tutorial of Arduino Owner:

Arduino Tutorial

Starter projects with Arduino:

Starter Projects

Projecthub

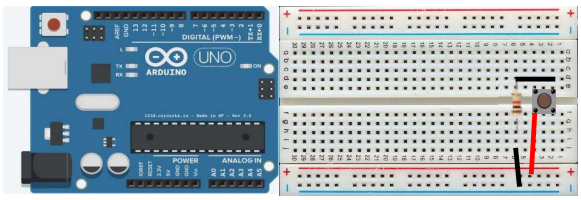
Tutorial for AdaFruit component:

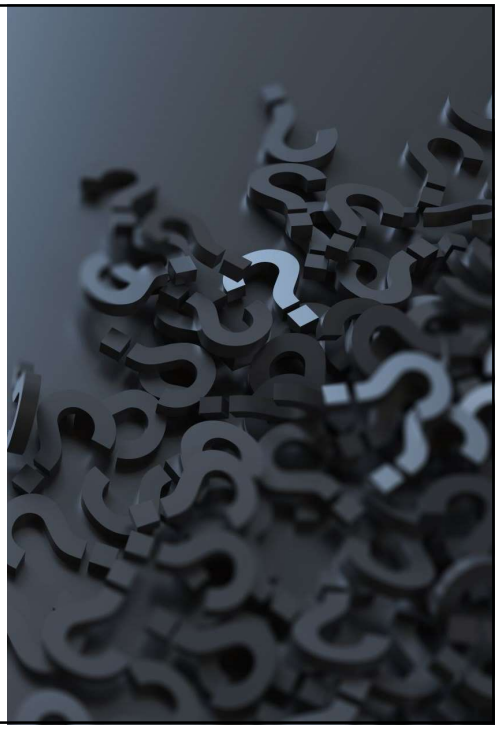
- GSM and GPS
- Adafruit products

59

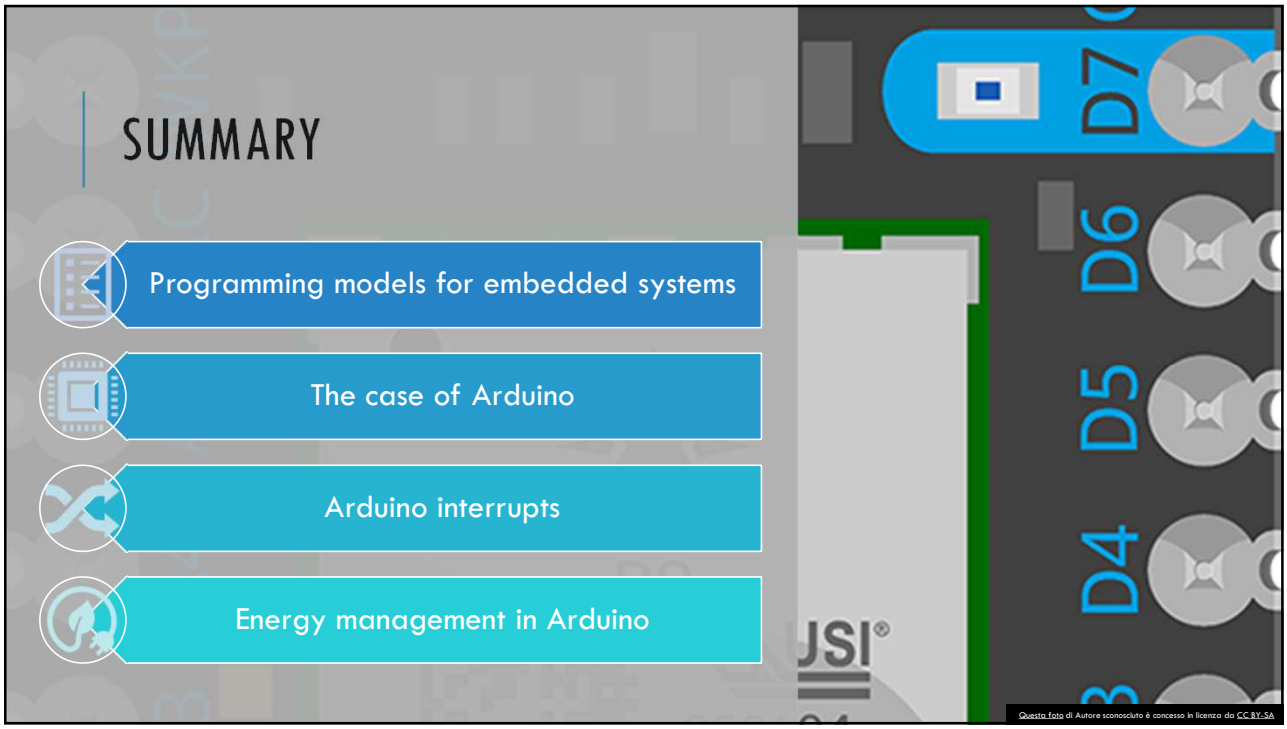
EXERCISE

Write a short fragment of Arduino code that configures the interrupt INT 0 to occur whenever a button switches state and complete the corresponding hardware configuration.





60


SUMMARY
Programming models for embedded systems
The case of Arduino
Arduino interrupts
Energy management in Arduino
D7
D6
D5
D4
Questa foto di Autore sconosciuto è concesso in licenza da CC BY-SA

62