Embedded systems

#card
Computer systems (hw, sw, mechanical components) that are designed and built to perform a specific function

- hardware-software co-design to optimize the function
- **micro-controller**: a single chip that embeds a microprocessor, memory, and I/O interfaces
  - different flavors on the market, sometimes general purpose (Arduino) or specialized like Application Specific Integrated Circuits (ASICs)
    - a generic term like System on Chip (SoC) can be used

Programming

#card
The job is cyclic (intrinsic duty cycle) and executed in a control loop:

1. read sensors
2. analyze data
3. take decision
4. control actuators

Challenges

#card
Possible constraints are:

- small memory footprint (e.g 16KB)
  - trade-off amount/cost
- no user interface (sometimes leds, simple LCD) → debugging complicated
- no file system, if storage is needed then it must be installed
- no OS, namely "bare-metal" → application must be created to manage interrupts
- special OS that offers basic functionalities to abstract HW
- **timing correctness**
  - given the fact that they often used to implement control applications → real time features
- **high reliability**
  - hard to guarantee in an hostile, harsh or unexpected environment → testing is hard

- **power management**

OS

#card
Code it's written and compiled on a host (cross-compilation) and then libraries are statically linked to create the **executable code** that runs over the bare-metal OS.

- main function is started when device turned on
  - first initializes the device (e.g. interrupt vector, etc.) then execute main

HW interaction

#card

1. the code interacts with the HW by **commands**
2. then **interrupts** are raised and received asynchronously to show that it has been executed
3. returned values are managed by the **handler** that works with the result

Conventional OS

#card

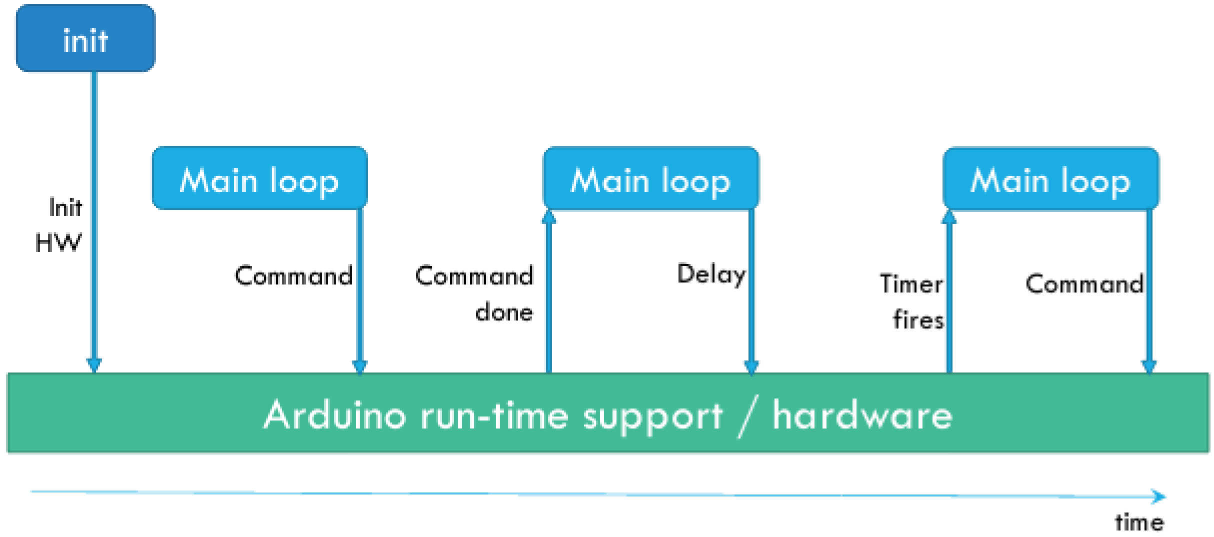
- Simple operation: e.g. printf → sys call → interacts with video card → return control to program that called printf
- Slow operation: e.g. I/O operation on disk → thread is suspended → context switch by saving the state → other stuff happening → when done interrupt is raised → handler is executed to resume the thread

Why it's not a good approach in embedded OSs?

#card
the memory usually is not sufficient to store states of thread and other handy functionalities

Arduino model

#card #exam
(Q21A)

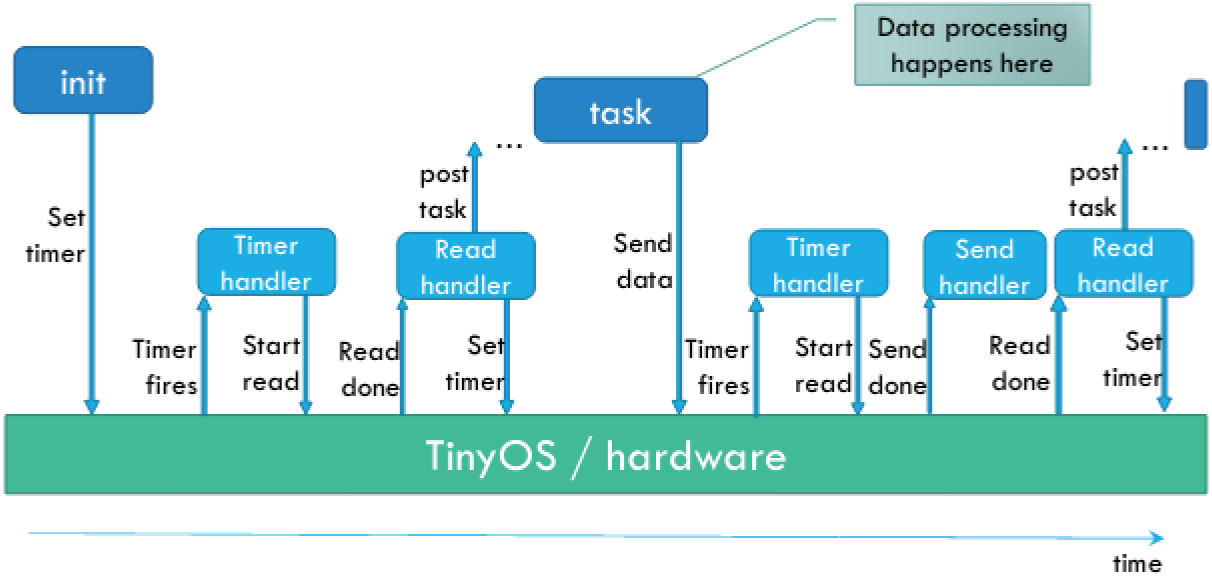


1. Device is **turned on**
2. **Init function** is executed to init interrupt vectors
3. **Main loop is executed repeatedly**, sometimes commands to read from transducers are executed and sent to run-time support
4. Run-time support **executes the reading** by using the libraries statically linked in the SW
   - in this time no other thread is executed → no context switch → no state to save
5. A **delay** command can be executed to specify when to do the next sensor reading
   - the micro-controller stays in a sleep mode
   - in order to match the sampling rate for sensors with the frequency of execution of the loop function
6. A timer wakes up the micro-controller to start again the main loop and is executed again
7. Loop until device off

It was not born for communications, but the need to mange asynchronous events born and this can be managed with interrupts

TinyOS model

#card #exam
(Q21B)



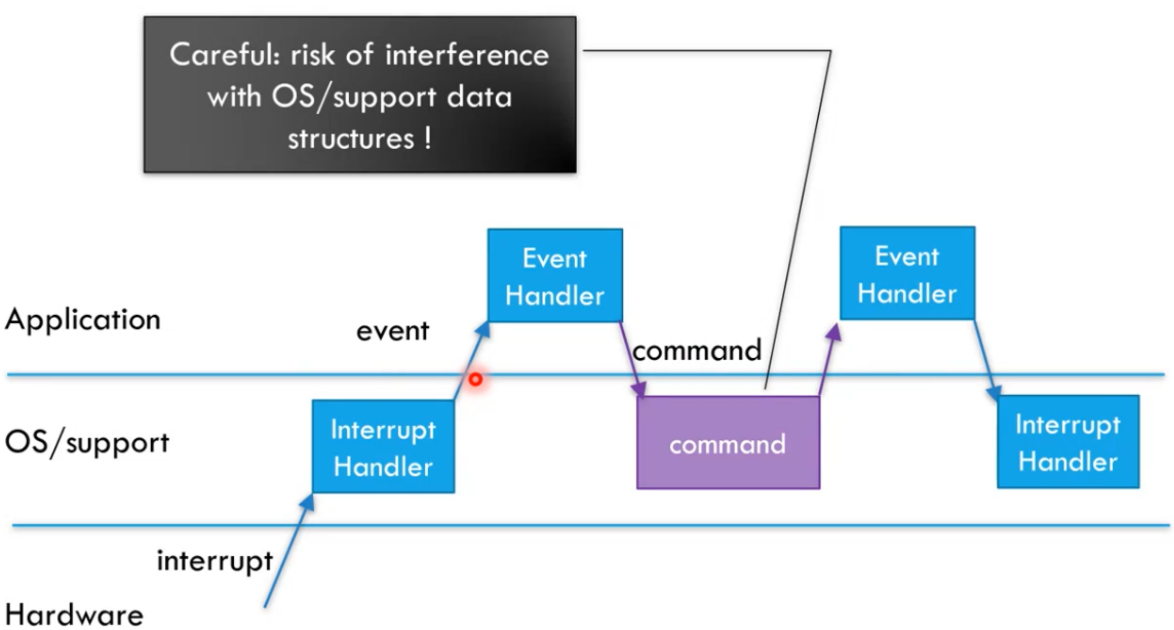
- Management of asynchronous interactions with the HW is done by:
  - **commands** to activate the HW
  - **events** that abstracts the interrupts
    - the programmer defines event handlers for the events (just minimal code to update data structures or commands to HW, otherwise task)
  - **tasks** to support large computations
    - decreases the chance of interruption of another event handler
    - the OS schedules it to run it later in a FIFO fashion
    - it can be interrupted by the event handler, but only the state of one task needs to be saved
    - preemptive (pre-rilascio) only for tasks, otherwise it's a non-preemptive approach
- Steps
  1. **Init function** is executed to init interrupt vectors and set up the first timer
     - here there is no loop, so the timer creates a period
  2. Micro-controller can go in **low power state**
  3. When **timer fires**, an interrupt is raised and the timer handler is executed. Usually contains a command to read from transducer
     - here the thread of the command start read is not stopped like in a normal OS.
  4. Start read immediately executed by TinyOS and **returns control** to the handler even if not completed (almost always)
  5. Timer handler completes other operations, returns and can go in sleep mode. The device is still performing the reading though
  6. When reading is done, device raises an interrupt, intercepted by TinyOS and transformed in an event read done that starts the execution of **read handler** that **receives the data**.

7. The processing of data is made through posting a task because it could take time. the task is prepared for execution, but not executed immediately because the execution is still in the **read handler**
8. The **read handler** setups timer for the next period for the new reading and it returns
9. A task is waiting for the execution, TinyOS starts the task
   once the data is done, in this case the device wants to **send data**, for example with a wireless interface to transmit data frame
   - this is done by the radio and takes time, TinyOS returns the control to the task that can complete other operations
10. Until the send handler, the device controlling the radio is operating, when send done interrupt is raised, intercepted by TinyOS produces **send done** to notify the program that the transmission is done
    - The send handler could for example check if ACK received
11. During that time it's possible that the timer may fire that has been set in the previous read handler
12. the activity is repeated and so on

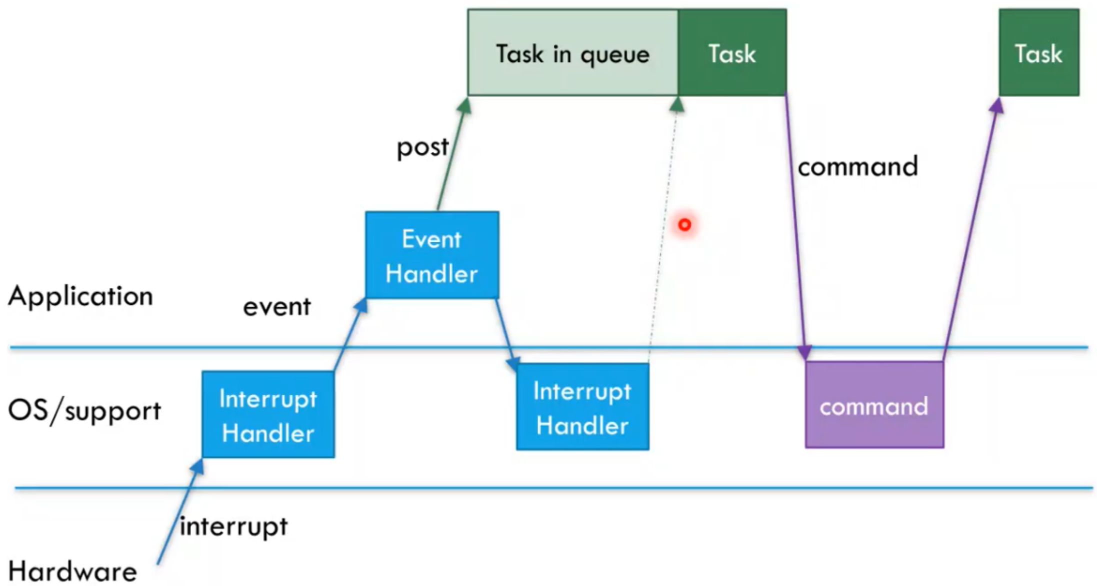
- **! no threads that may be suspended, the handler function state just takes the control again and no state needs to be saved because it's a function**
- **! the only context to keep is the one of the current task because it can be interrupted by an event**

Conflicts

#card

- Problem
  - 

- command for any reason interferes with the execution of interrupt handler, and the system could stop to work properly
- Solution
  1. Send a command to a device that is not the one from which the interrupt has been received
  2. Use the task queue to send commands

- 

Arduino

interrupts

#card
Interrupts **can be managed to have async interactions with external devices**

- can be used to put Arduino to low-power mode where it does not execute any code, so to wake it up an interrupt must be raised
- interrupt must be handled asap, by binding interrupts and functions for short events handler (update data structures, give command to HW)
- types
  - **external**
  - timer, managed by Arduino library
  - interrupt, from internal device managed by Arduino library

External

#card #exam
(Q21c)
External inputs two interrupts to two digital pins, that can be interpreted as a interrupt line

- `attachInterrupt`, associate and execute async function handler to a signal along a pin
- functions
  - `attachInterrupt(pin, handler, mode)`
  - other functions to enable/disable interrupts that it's useful in the handler
  - interrupt0 and interrupt1 mapped to pin 2 and pin 3, a mode needs to be defined something like (rising, falling, change, low)
  - handler as minimal as possible
  - volatile for write variables to avoid possible conflicts if the compiler decides to save in register

Energy management

#card
Power save mode. Microcontrollers offer fine grained control over the low power mode to let the programmer selectively keep enabled/disabled specific components (CPU clock, I/O, ...)