BENHA UNIVERSITY
FACULTY OF ENGINEERING AT SHOUBRA

ECE-508
Sensor Networks

Lecture #2
Single Node Architecture

Instructor:
Dr. Ahmad El-Banna

SPRING 2015

Agenda

- Sensor node architecture
- Energy supply and consumption
- Runtime environments for sensor nodes
- Case study: TinyOS

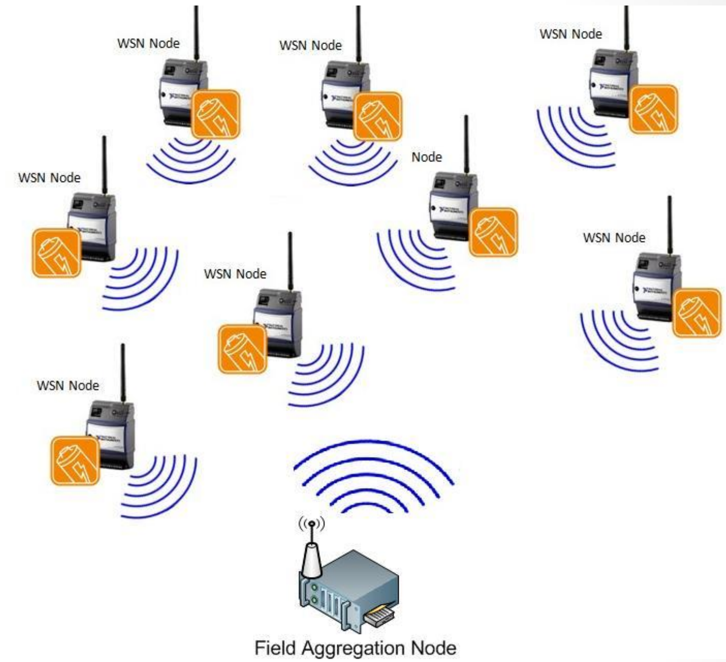
Goals of the chapter

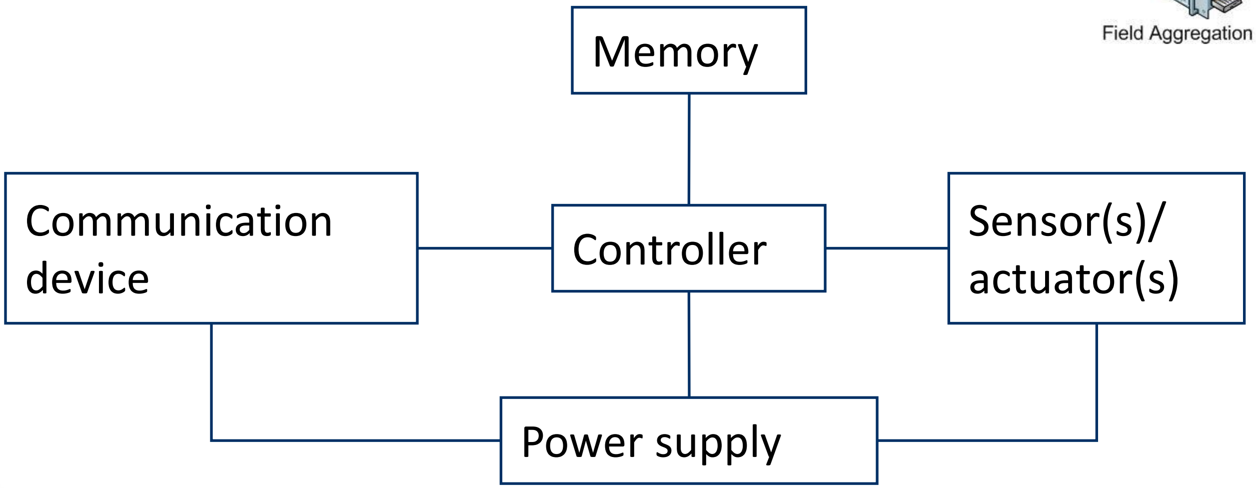
- Survey the main components of the composition of a node for a wireless sensor network
  - Controller, radio modem, sensors, batteries
- Understand energy consumption aspects for these components
  - Putting into perspective different operational modes and what different energy/power consumption means for protocol design
- Operating system support for sensor nodes
- Some example nodes

SENSOR NODE ARCHITECTURE

Sensor node architecture

- Main components of a WSN node
  - Controller
  - Communication device(s)
  - Sensors/actuators
  - Memory
  - Power supply





Ad hoc node architecture

- Core: essentially the same
- But: Much more additional equipment
  - Hard disk, display, keyboard, voice interface, camera, …

- Essentially: a laptop-class device

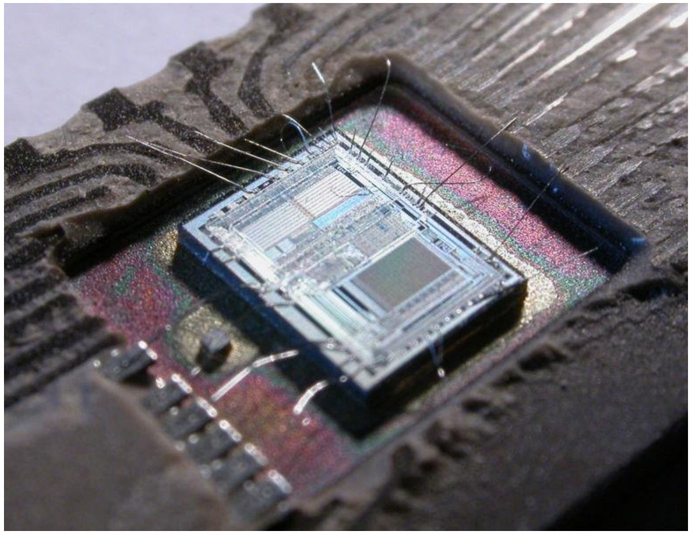
Controller

- Main options:
  - Microcontroller – general purpose processor, optimized for embedded applications, low power consumption
  - DSPs – optimized for signal processing tasks, not suitable here
  - FPGAs – may be good for testing
  - ASICs – only when peak performance is needed, no flexibility

- Example microcontrollers
  - Texas Instruments MSP430
    - 16-bit RISC core, up to 4 MHz, versions with 2-10 kbytes RAM, several DACs, RT clock, prices start at 0.49 US$
  - Atmel ATMega
    - 8-bit controller, larger memory than MSP430, slower

Microcontrollers !

- A **microcontroller** (sometimes abbreviated **µC, uC** or **MCU**) is a small computer on a single **integrated circuit** containing a **processor core**, **memory**, and programmable **input/output** peripherals.
- It can only perform simple/specific tasks.
- A microcontroller is often described as a '**computer-on-a-chip**'.



Microcomputer System and Microcontroller based System

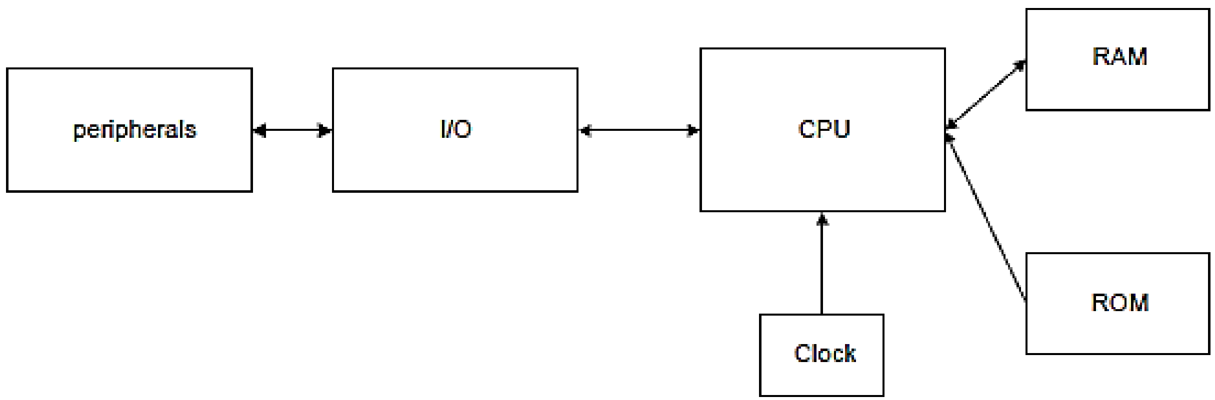


Figure 1: Basic building blocks of a computer

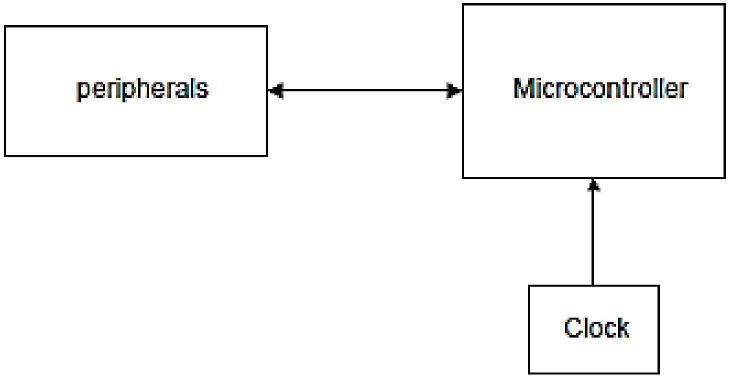
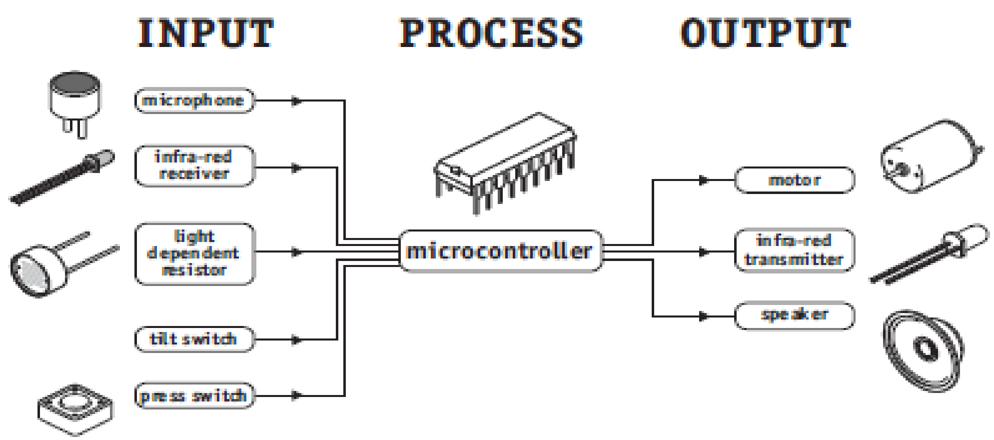


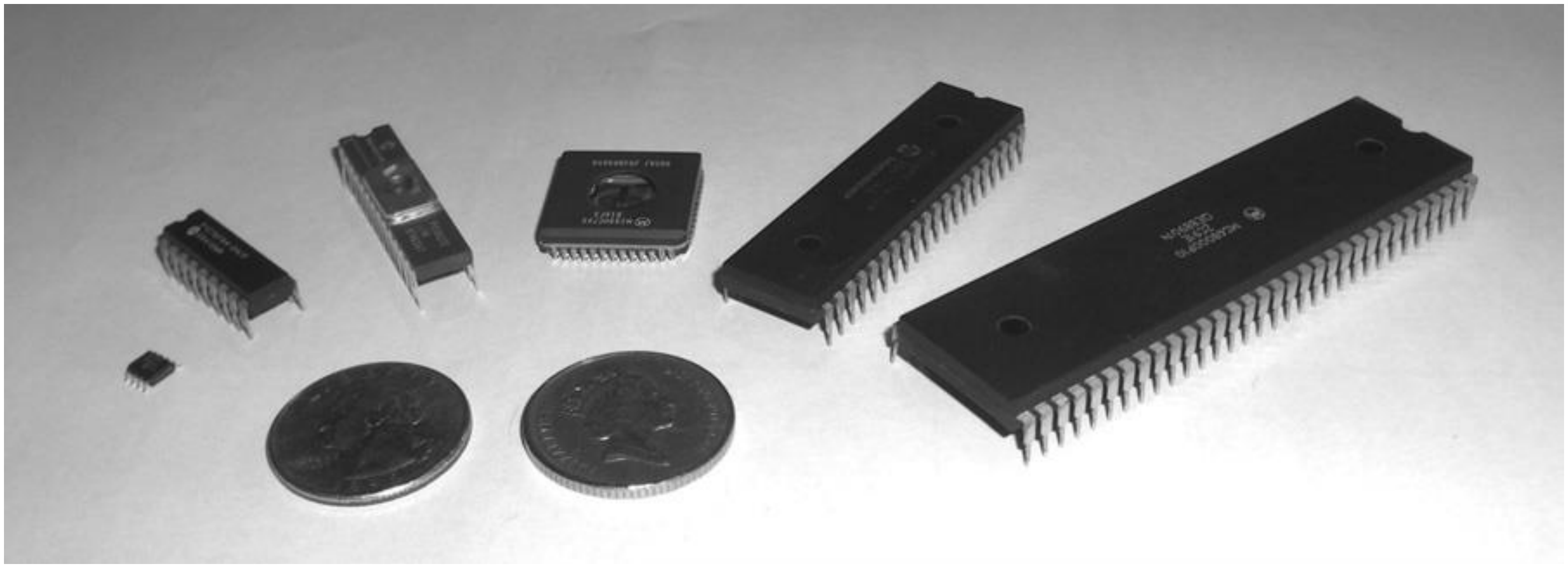
Figure 2: A microcontroller based system

Microcontrollers..

- Microcontrollers are purchased ‘blank’ and then programmed with a specific control program.
- Once programmed, the microcontroller is build into a product to make the product more intelligent and easier to use.
- A designer will use a Microcontroller to:
  - Gather input from various sensors
  - Process this input into a set of actions
  - Use the output mechanisms on the microcontroller to do something useful.



Microcontroller Packaging and Appearance



From left to right: PIC 12F508, PIC 16F84A, PIC 16C72, Motorola 68HC05B16, PIC 16F877, Motorola 68000

Features Example: PIC 16F877

Key Features	PIC16F877
MAX Operating Frequency	20MHz
FLASH Program Memory (14-bit words)	8K
Data Memory (bytes)	368
EEPROM Data Memory (bytes)	256
I/O Ports	RA0-5 (6) RB0-7 (8) RC0-7 (8) RD0-7 (8) RE0-2 (3)
Timers	3
CCP (Capture/Compare/PWM)	2
Serial Communications	MSSP, USART
Parallel Communications	PSP
10-bit Analog-to-Digital Module	8 Channels
Instruction Set	35 Instructions
Pins (DIP)	40 Pins

Communication device

- Which transmission medium?
  - Electromagnetic at radio frequencies?
  - Electromagnetic, light?
  - Ultrasound?

- Radio **transceivers** transmit a bit- or byte stream as radio wave
  - Receive it, convert it back into bit-/byte stream

Transceiver characteristics

- Capabilities
  - Interface: bit, byte, packet level?
  - Supported frequency range?
    - Typically, somewhere in 433 MHz – 2.4 GHz, ISM band
  - Multiple channels?
  - Data rates?
  - Range?
- Energy characteristics
  - Power consumption to send/receive data?
  - Time and energy consumption to change between different states?
  - Transmission power control?
  - Power efficiency (which percentage of consumed power is radiated?)
- Radio performance
  - Modulation? (ASK, FSK, …?)
  - Noise figure? $NF = SNR_I/SNR_O$
  - Gain? (signal amplification)
  - Receiver sensitivity? (minimum S to achieve a given E_b/N_0)
  - Blocking performance (achieved BER in presence of frequency-offset interferer)
  - Out of band emissions
  - Carrier sensing & RSSI characteristics
  - Frequency stability (e.g., towards temperature changes)
  - Voltage range

Transceiver states

- Transceivers can be put into different operational ***states***, typically:
  - ***Transmit***
  - ***Receive***
  - ***Idle*** – ready to receive, but not doing so
    - Some functions in hardware can be switched off, reducing energy consumption a little
  - ***Sleep*** – significant parts of the transceiver are switched off
    - Not able to immediately receive something
    - ***Recovery time*** and ***startup energy*** to leave sleep state can be significant
- Research issue: Wakeup receivers – can be woken via radio when in sleep state (seeming contradiction!)

Homework: Summarize a paper related to Wakeup receivers!
S15PG_SN_UrName_ASS02

Example radio transceivers

- Almost boundless variety available
- Some examples
  - RFM TR1000 family
    - 916 or 868 MHz
    - 400 kHz bandwidth
    - Up to 115,2 kbps
    - On/off keying or ASK
    - Dynamically tuneable output power
    - Maximum power about 1.4 mW
    - Low power consumption
  - Chipcon CC1000
    - Range 300 to 1000 MHz, programmable in 250 Hz steps
    - FSK modulation
    - Provides RSSI
- Chipcon CC 2400
  - Implements 802.15.4
  - 2.4 GHz, DSSS modem
  - 250 kbps
  - Higher power consumption than above transceivers
- Infineon TDA 525x family
  - E.g., 5250: 868 MHz
  - ASK or FSK modulation
  - RSSI, highly efficient power amplifier
  - Intelligent power down, “self-polling” mechanism
  - Excellent blocking performance

Example radio transceivers for ad hoc networks

- Ad hoc networks: Usually, higher data rates are required
- Typical: IEEE 802.11 b/g/a is considered
  - Up to 54 MBit/s
  - Relatively long distance (100s of meters possible, typical 10s of meters at higher data rates)
  - Works reasonably well (but certainly not perfect) in mobile environments
  - Problem: expensive equipment, quite power hungry

Wakeup receivers

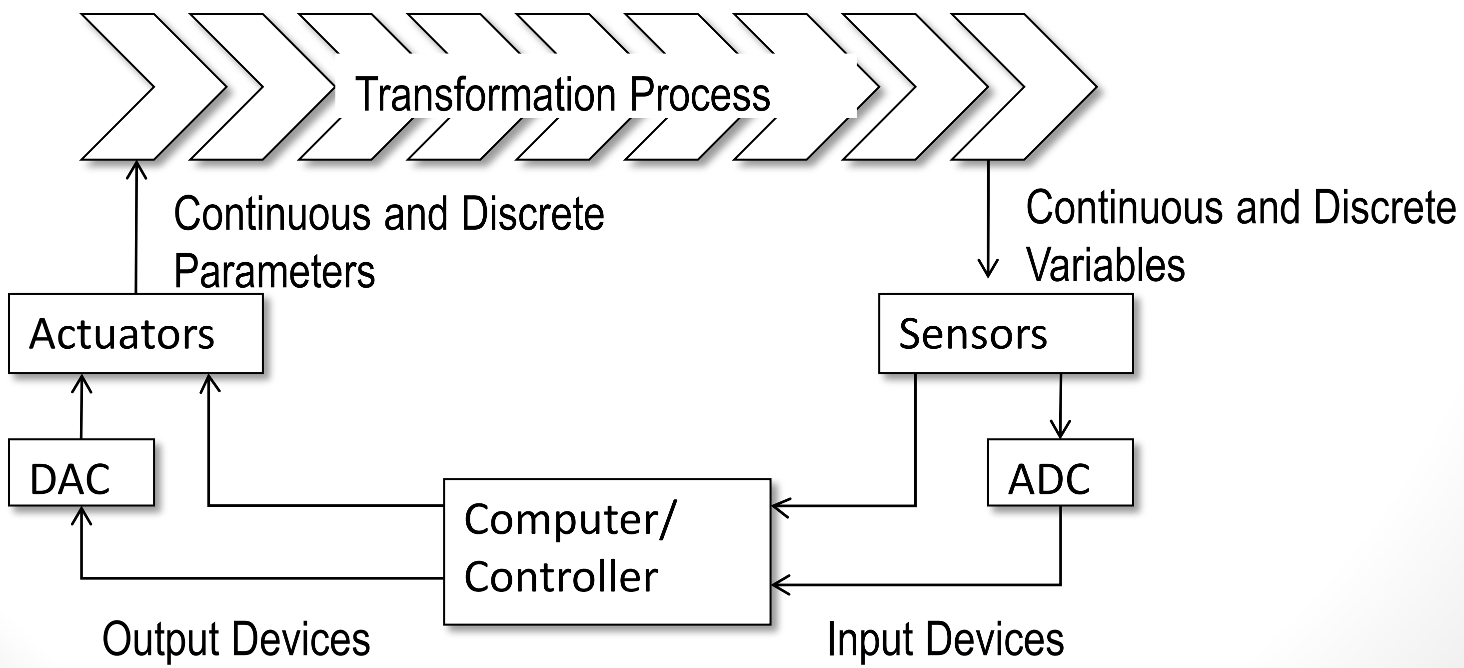
- Major energy problem: ***RECEIVING***
  - Idling and being ready to receive consumes considerable amounts of power

- When to switch on a receiver is not clear
  - Contention-based MAC protocols: Receiver is always on
  - TDMA-based MAC protocols: Synchronization overhead, inflexible

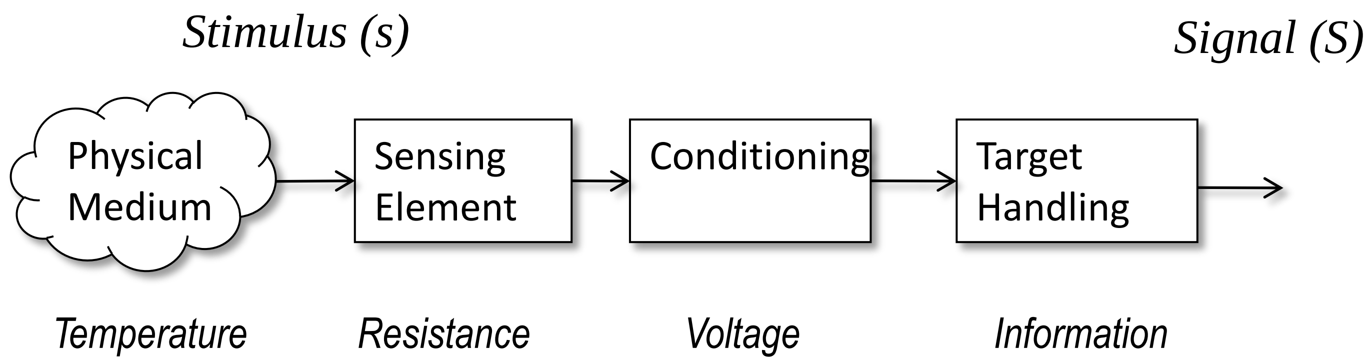
- Desirable: Receiver that can (only) check for incoming messages
  - When signal detected, wake up main receiver for actual reception
  - Ideally: ***Wakeup receiver*** can already process simple addresses
  - Not clear whether they can be actually built, however

Computer Process Control System

- To implement process control, the computer must collect data and transmit signals to the production process.
- Components required to implement the interface:
  - Sensors to measure continuous and discrete process variables
  - Actuators to drive continuous and discrete process parameters
  - Devices for ADC and DAC
  - I/O devices for discrete data

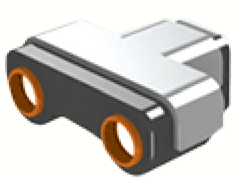


Sensors



- A sensor is a transducer that converts a physical stimulus from one form into a more useful form to measure the stimulus.
- Two basic categories:
  1. Analog
  2. Discrete
     - Binary
     - Digital (e.g., pulse counter)

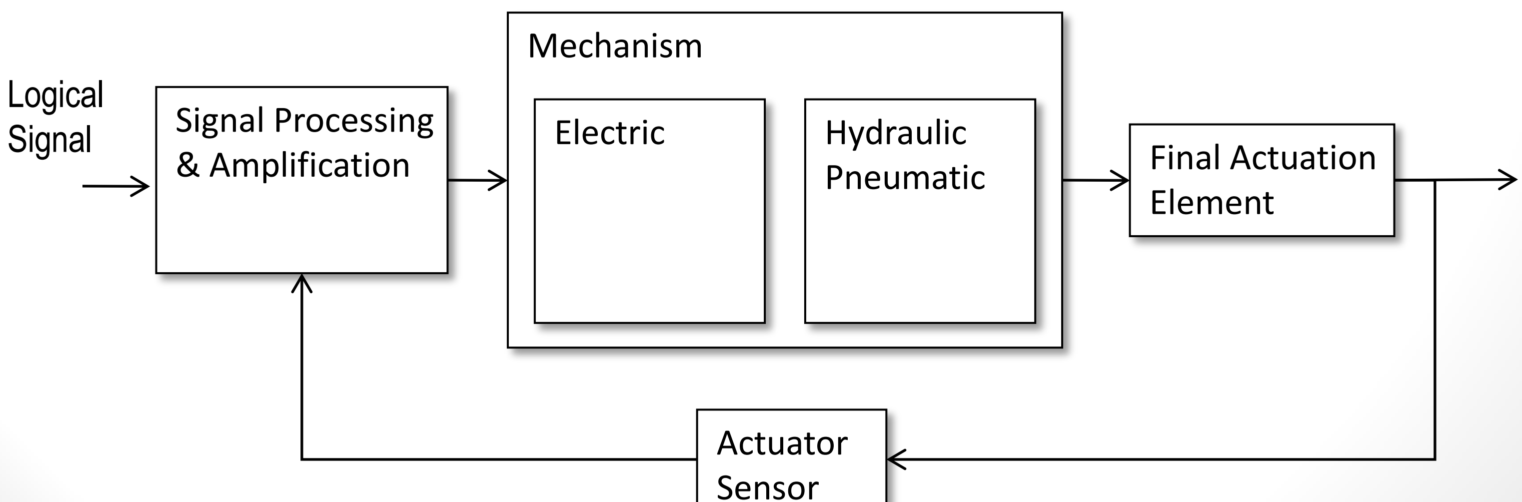
Sensors..



- Main categories
  - Any energy radiated? Passive vs. active sensors
  - Sense of direction? Omidirectional?
  - Passive, omnidirectional
    - Examples: light, thermometer, microphones, hygrometer, …
  - Passive, narrow-beam
    - Example: Camera
  - Active sensors
    - Example: Radar
- Important parameter: Area of coverage
  - Which region is adequately covered by a given sensor?

Actuators

- **Actuators** are hardware devices that convert a controller command signal into a change in a physical parameter
- The change is usually *mechanical* (e.g., position or velocity)
- An actuator is also a *transducer* because it changes one type of physical quantity into some alternative form
- An actuator is usually activated by a low-level command signal, so an *amplifier* may be required to provide sufficient power to drive the actuator



Types of Actuators

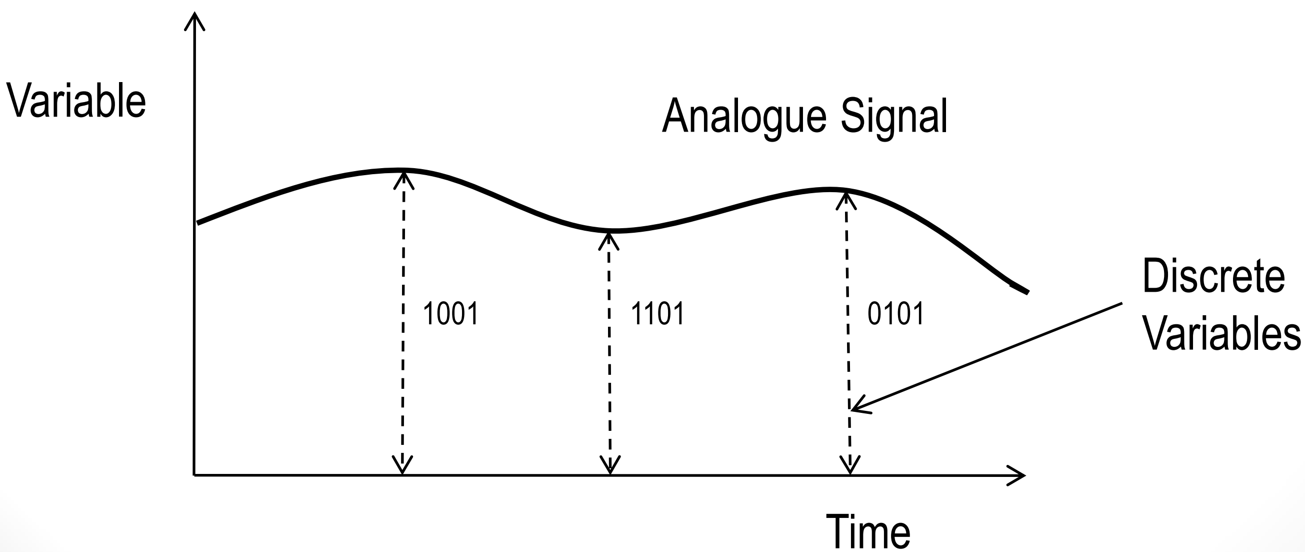
1. Electrical actuators
   - Electric motors
     - DC servomotors
     - AC motors
     - Stepper motors
   - Solenoids

2. Hydraulic actuators
   - Use hydraulic *fluid* to amplify the controller command signal

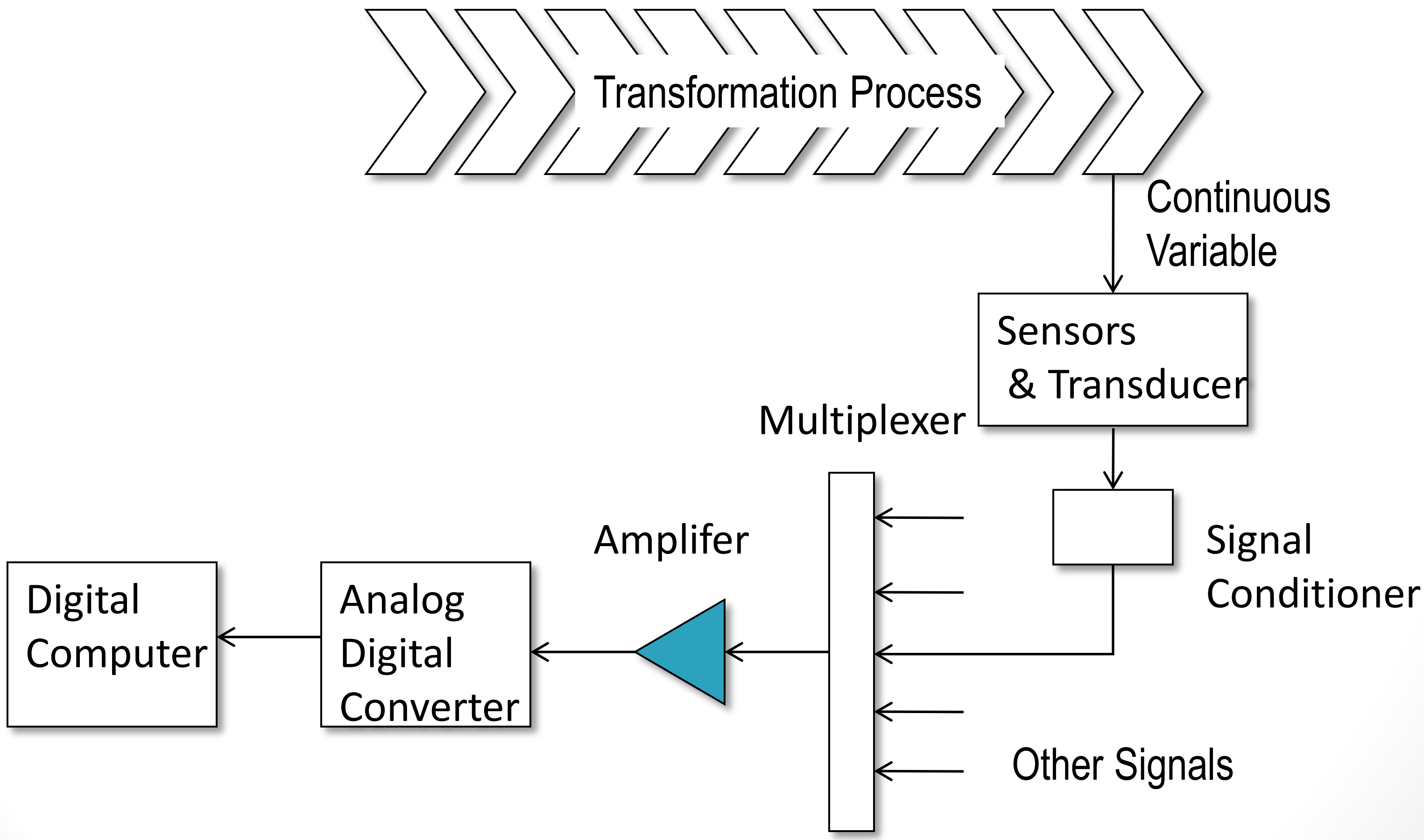
3. Pneumatic actuators
   - Use compressed *air* as the driving force

Analog-to-Digital Conversion (ADC)

- **Sampling** – converts the continuous signal into a series of discrete analog signals at periodic intervals
- **Quantization** – each discrete analog is converted into one of a finite number of (previously defined) discrete amplitude levels
- **Encoding** – discrete amplitude levels are converted into digital code



Hardware Devices in Analog-to-Digital Conversion



Features of an ADC

- Sampling rate – rate at which continuous analog signal is polled e.g. 1000 samples/sec
- Quantization – divide analog signal into discrete levels $N_q = 2^n$
  - where N_q = quantisation levels; and n is the number of bits.
- Resolution – depends on number of quantization levels

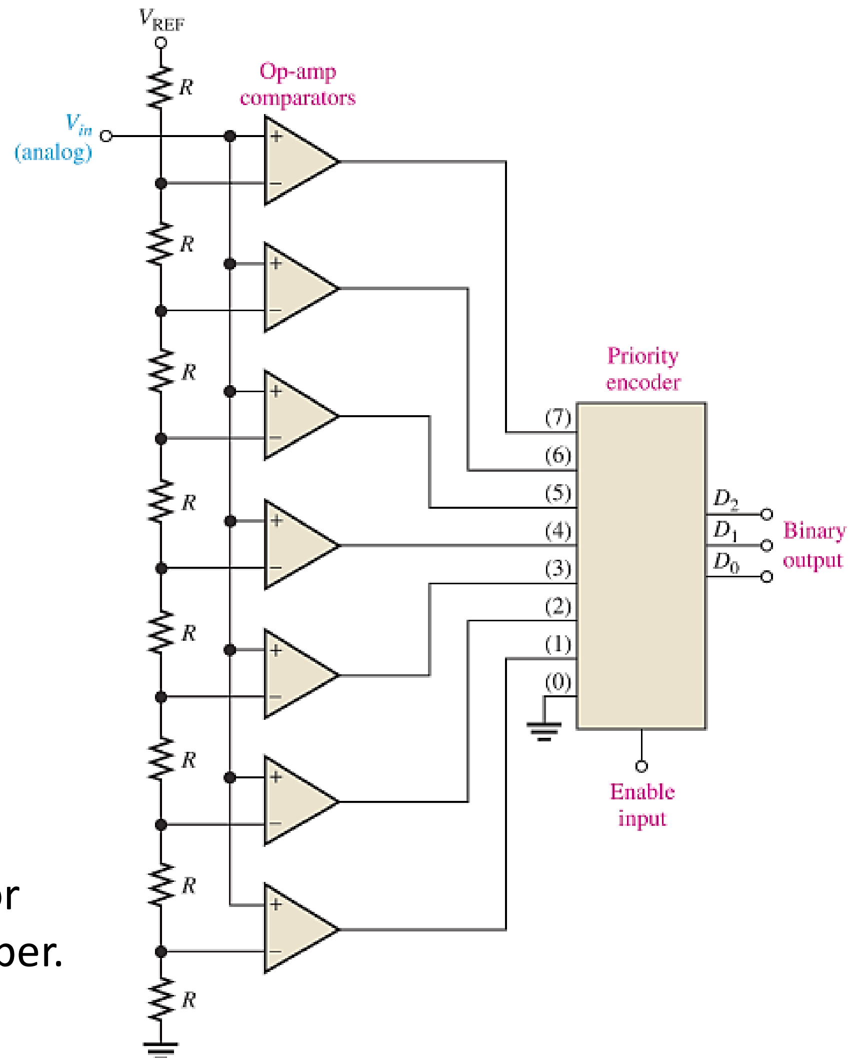
$$R_{ADC} = \frac{L}{N_q - 1} = \frac{L}{2^n - 1}$$

  - where R_{ADC} is the resolution of the ADC; L is the full-scale range of the ADC
- Conversion time – how long it takes to convert the sampled signal to digital code
- Conversion method – means by which analog signal is encoded into digital equivalent
  - Example – Successive approximation method & Flash

Flash ADC

- The **simultaneous**, or **flash**, method of A/D conversion uses **parallel comparators** to compare the linear input signal with **various reference** voltages developed by a voltage divider.
- When the **input** voltage **exceeds the reference** voltage for a given comparator, a **high level is produced** on that comparator's output.

→**2^n- 1** comparators are required for conversion to an **n-digit** binary number.



DAC

- Convert digital values into continuous analogue signal
  - Decoding digital value to an analogue value at discrete moments in time based on value within register

$$E_0 = E_{ref}\left\{0.5B_1 + 0.25B_2 + \cdots + \left(2^n\right)^{-1}B_n\right\}$$

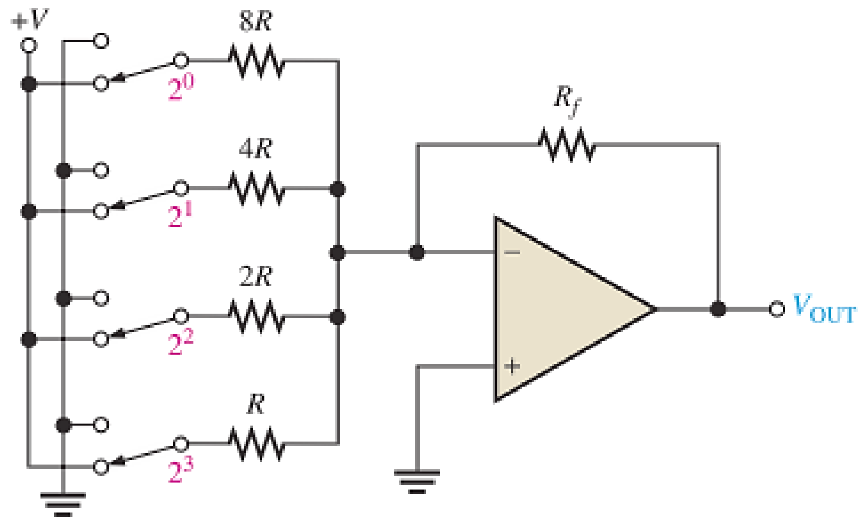
Where E_0 is output voltage; E_{ref} is reference voltage; B_n is status of successive bits in the binary register

DAC Examples

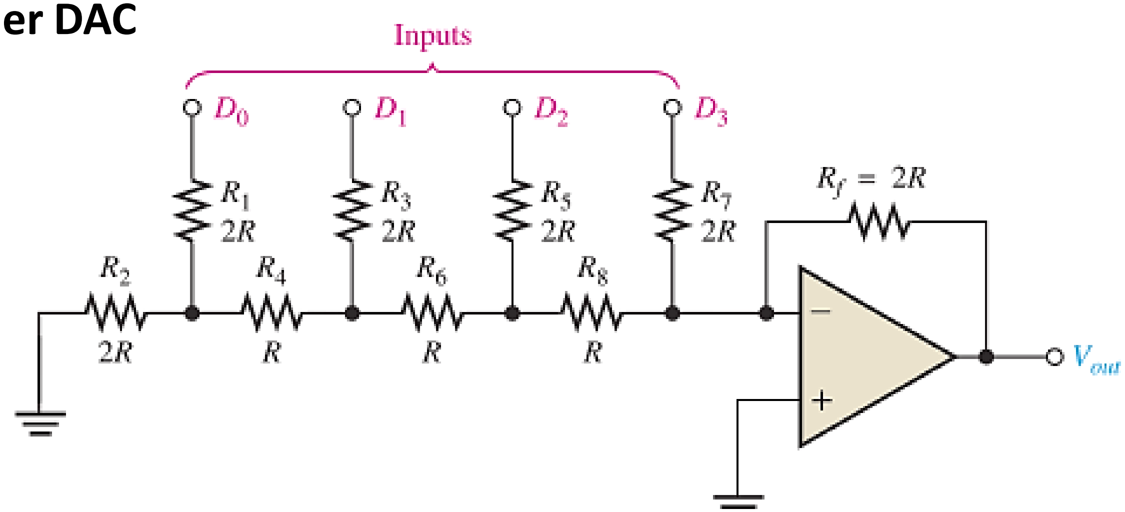
- **Scaling Adder as a four-digit DAC**

I_0=+V/8R
I_1=+V/4R
I_2=+V/2R
I_3=+V/R

$V_{out(D0)}$=-R_f I_0
$V_{out(D1)}$=-R_f I_1
$V_{out(D2)}$=-R_f I_2
$V_{out(D3)}$=-R_f I_3



- **An R/2R ladder DAC**



ENERGY SUPPLY AND CONSUMPTION

Energy supply of mobile/sensor nodes

- Goal: provide as much energy as possible at smallest cost/volume/weight/recharge time/longevity
  - In WSN, recharging may or may not be an option
- Options
  - Primary batteries – not rechargeable
  - Secondary batteries – rechargeable, only makes sense in combination with some form of energy harvesting
- Requirements include
  - Low self-discharge
  - Long shelf live
  - Capacity under load
  - Efficient recharging at low current
  - Good relaxation properties (seeming self-recharging)
  - Voltage stability (to avoid DC-DC conversion)

Battery examples

- Energy per volume (Joule per cubic centimeter):

Primary batteries			
Chemistry	Zinc-air	Lithium	Alkaline
Energy (J/cm^3)	3780	2880	1200
Secondary batteries			
Chemistry	Lithium	NiMHd	NiCd
Energy (J/cm^3)	1080	860	650

Energy Scavenging

- How to recharge a battery?
  - A laptop: easy, plug into wall socket in the evening
  - A sensor node? – Try to ***scavenge*** energy from environment
- Ambient energy sources
  - Light ! solar cells – between 10 μW/cm^2 and 15 mW/cm^2
  - Temperature gradients – 80 μ W/cm^2 @ 1 V from 5K difference
  - Vibrations – between 0.1 and 10000 μ W/cm^3
  - Pressure variation (piezo-electric) – 330 μ W/cm^2 from the heel of a shoe
  - Air/liquid flow (MEMS gas turbines)

Energy scavenging – overview

Energy source	Energy density
Batteries (zinc-air)	$1050 - 1560\,\text{mWh/cm}^3$
Batteries (rechargable lithium)	$300\,\text{mWh/cm}^3$ (at $3 - 4$ V)
Energy source	**Power density**
Solar (outdoors)	$15\,\text{mW/cm}^2$ (direct sun)
	$0.15\,\text{mW/cm}^2$ (cloudy day)
Solar (indoors)	$0.006\,\text{mW/cm}^2$ (standard office desk)
	$0.57\,\text{mW/cm}^2$ ($<$ 60 W desk lamp)
Vibrations	$0.01 - 0.1\,\text{mW/cm}^3$
Acoustic noise	$3 \cdot 10^{-6}\text{mW/cm}^2$ at 75 Db
	$9,6 \cdot 10^{-4}\text{mW/cm}^2$ at 100 Db
Passive human-powered systems	$1.8\,\text{mW}$ (shoe inserts)
Nuclear reaction	$80\,\text{mW/cm}^3$, $10^6\,\text{mWh/cm}^3$

Energy Consumption

- A "back of the envelope" estimation
- Number of instructions
  - Energy per instruction: 1 nJ
  - Small battery ("smart dust"): 1 J = 1 Ws
  - Corresponds: 10^9 instructions!
- Lifetime
  - Or: Require a single day operational lifetime = 24*60*60 =86400 s
  - 1 Ws / 86400s ¼ ***11.5 μW*** as max. sustained power consumption!
- Not feasible!

Multiple power consumption modes

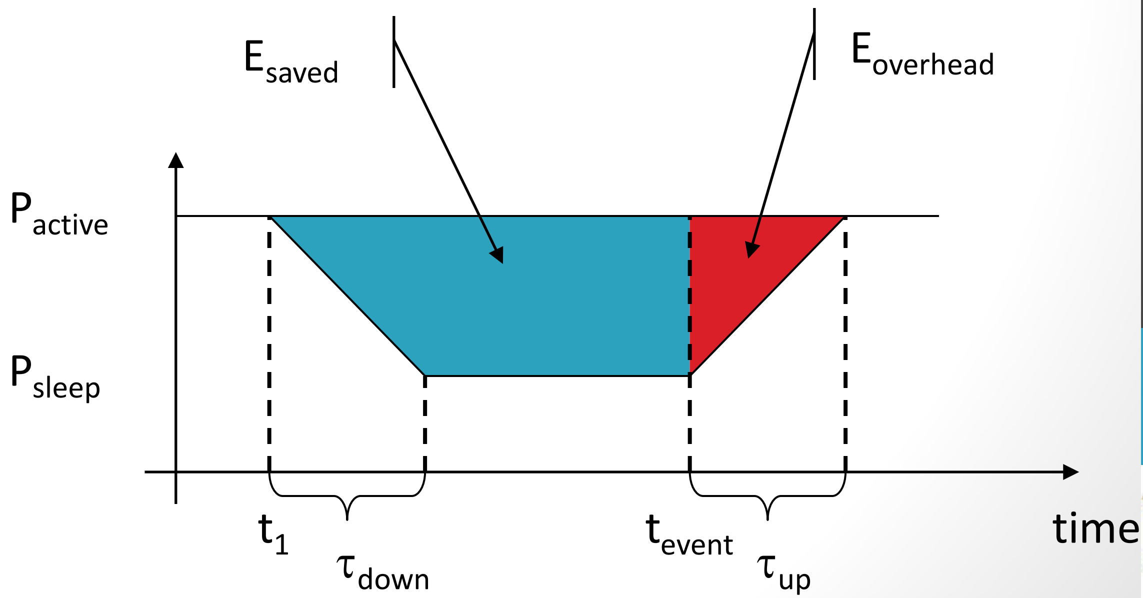
- Way out: Do not run sensor node at full operation all the time
  - If nothing to do, switch to ***power safe mode***
  - Question: When to throttle down? How to wake up again?
- Typical modes
  - Controller: Active, idle, sleep
  - Radio mode: Turn on/off transmitter/receiver, both
- Multiple modes possible, “deeper” sleep modes
  - Strongly depends on hardware
  - TI MSP 430, e.g.: four different sleep modes
  - Atmel ATMega: six different modes

Some energy consumption figures

- Microcontroller
  - TI MSP 430 (@ 1 MHz, 3V):
    - Fully operation 1.2 mW
    - Deepest sleep mode 0.3 μW – only woken up by external interrupts (not even timer is running any more)
  - Atmel ATMega
    - Operational mode: 15 mW active, 6 mW idle
    - Sleep mode: 75 μW

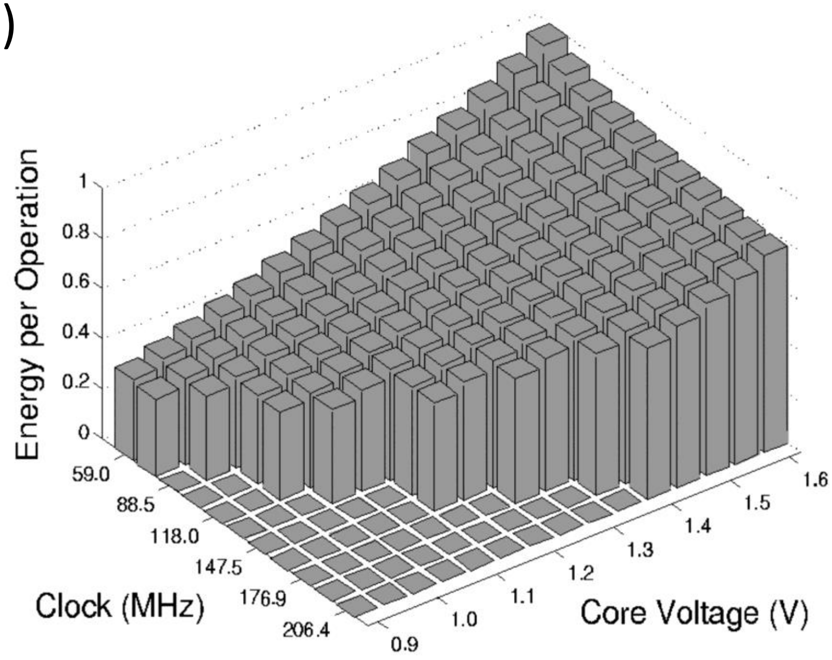
Switching between modes

- Simplest idea: Greedily switch to lower mode whenever possible
- Problem: Time and power consumption required to reach higher modes not negligible
  - Introduces overhead
  - Switching only pays off if $E_{saved} > E_{overhead}$
- Example: Event-triggered wake up from sleep mode
- Scheduling problem with uncertainty (exercise)



Alternative: Dynamic voltage scaling

- Switching modes complicated by uncertainty how long a sleep time is available
- Alternative: Low supply voltage & clock
  - ***Dynamic voltage scaling*** (DVS)
- Rationale:
  - Power consumption P depends on
    - Clock frequency
    - Square of supply voltage
    - *$P / f V^2$*
  - Lower clock allows lower supply voltage
  - Easy to switch to higher clock
  - But: execution takes longer



Memory power consumption

- Crucial part: FLASH memory
  - Power for RAM almost negligible

- FLASH writing/erasing is expensive
  - Example: FLASH on Mica motes
  - Reading: ¼ 1.1 nAh per byte
  - Writing: ¼ 83.3 nAh per byte

Transmitter power/energy consumption for n bits

- Amplifier power: $P_{amp} = \alpha_{amp} + \beta_{amp} P_{tx}$
  - P_{tx} ***radiated power***
  - α_{amp}, β_{amp} constants depending on model
  - Highest efficiency ($\eta = P_{tx} / P_{amp}$) at maximum output power
- In addition: transmitter electronics needs power P_{txElec}
- Time to transmit n bits: $n / (R * R_{code})$
  - R nomial data rate, R_{code} coding rate
- To leave sleep mode
  - Time T_{start}, average power P_{start}

! $\mathbf{E_{tx} = T_{start} P_{start} + n / (R * R_{code}) (P_{txElec} + \alpha_{amp} + \beta_{amp} P_{tx})}$

- Simplification: Modulation not considered

Receiver power/energy consumption for n bits

- Receiver also has startup costs
  - Time T_{start}, average power P_{start}
- Time for n bits is the same $n / (R * R_{code})$
- Receiver electronics needs P_{rxElec}
- Plus: energy to decode n bits $E_{decBits}$

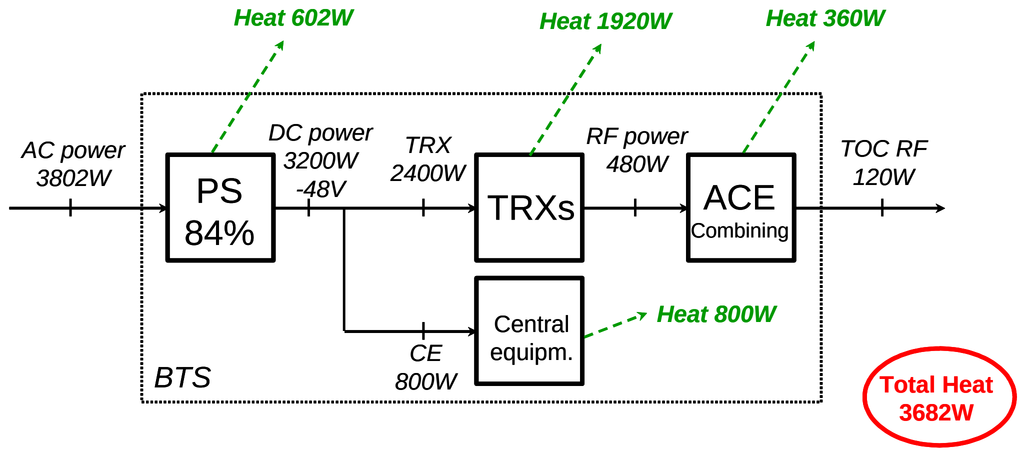
! $E_{rx} = T_{start} P_{start} + n / (R * R_{code}) P_{rxElec} + E_{decBits} (R)$

Some Transceiver Numbers

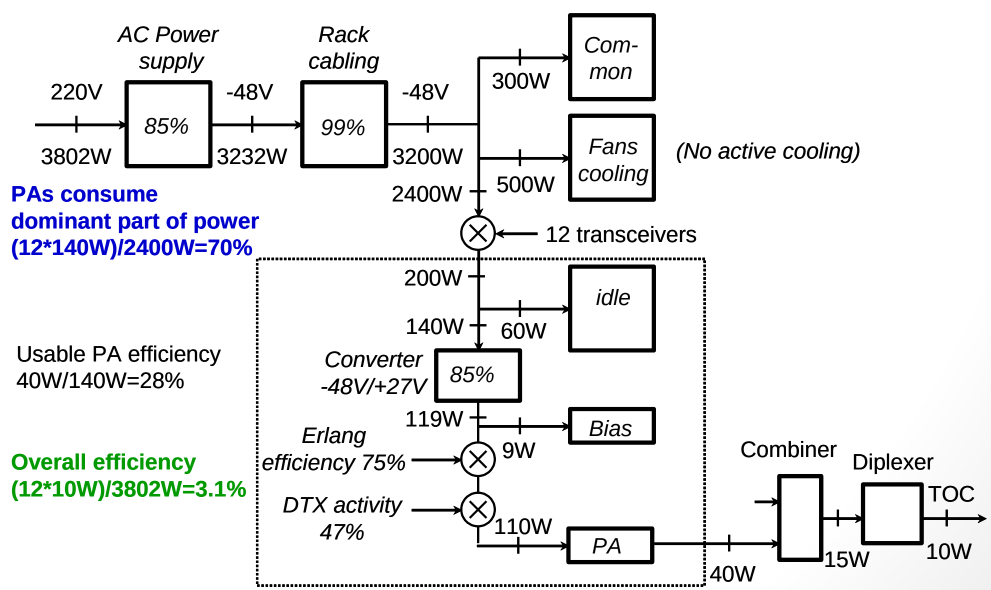
Symbol	Description	Example transceiver		
		μAMPS-1 [559]	WINS [670]	MEDUSA-II [670]
α_{amp}	Eq. (2.4)	174 mW	N/A	N/A
β_{amp}	Eq. (2.4)	5.0	8.9	7.43
P_{amp}	Amplifier pwr.	179 – 674 mW	N/A	N/A
P_{rxElec}	Reception pwr.	279 mW	368.3 mW	12.48 mW
P_{rxIdle}	Receive idle	N/A	344.2 mW	12.34 mW
P_{start}	Startup pwr.	58.7 mW	N/A	N/A
P_{txElec}	Transmit pwr.	151 mW	$\approx$ 386 mW	11.61 mW
R	Transmission rate	1 Mbps	100 kbps	OOK 30 kbps ASK 115.2 kbps
T_{start}	Startup time	466 μs	N/A	N/A

Comparison: GSM base station power consumption

- Overview



- Details



- (just to put things into perspective)

Controlling Transceivers

- Similar to controller, low duty cycle is necessary
  - Easy to do for transmitter – similar problem to controller: when is it worthwhile to switch off
  - Difficult for receiver: Not only time when to wake up not known, it also depends on ***remote*** partners

  ! Dependence between MAC protocols and power consumption is strong!

- Only limited applicability of techniques analogue to DVS
  - Dynamic Modulation Scaling (DSM): Switch to modulation best suited to communication – depends on channel gain
  - Dynamic Coding Scaling – vary coding rate according to channel gain
  - Combinations

Computation vs. communication energy cost

- Tradeoff?
  - Directly comparing computation/communication energy cost not possible
  - But: put them into perspective!
  - Energy ratio of “sending one bit” vs. “computing one instruction”: Anything between 220 and 2900 in the literature
  - To communicate (send & receive) one kilobyte = computing three million instructions!
- Hence: try to compute instead of communicate whenever possible
- Key technique in WSN – ***in-network processing!***
  - Exploit compression schemes, intelligent coding schemes, …

RUNTIME ENVIRONMENTS FOR SENSOR NODES

Operating system challenges in WSN

- Usual operating system goals
  - Make access to device resources abstract (virtualization)
  - Protect resources from concurrent access
- Usual means
  - Protected operation modes of the CPU – hardware access only in these modes
  - Process with separate address spaces
  - Support by a memory management unit
- Problem: These are not available in microcontrollers
  - No separate protection modes, no memory management unit
  - Would make devices more expensive, more power-hungry

! ???

Operating system challenges in WSN

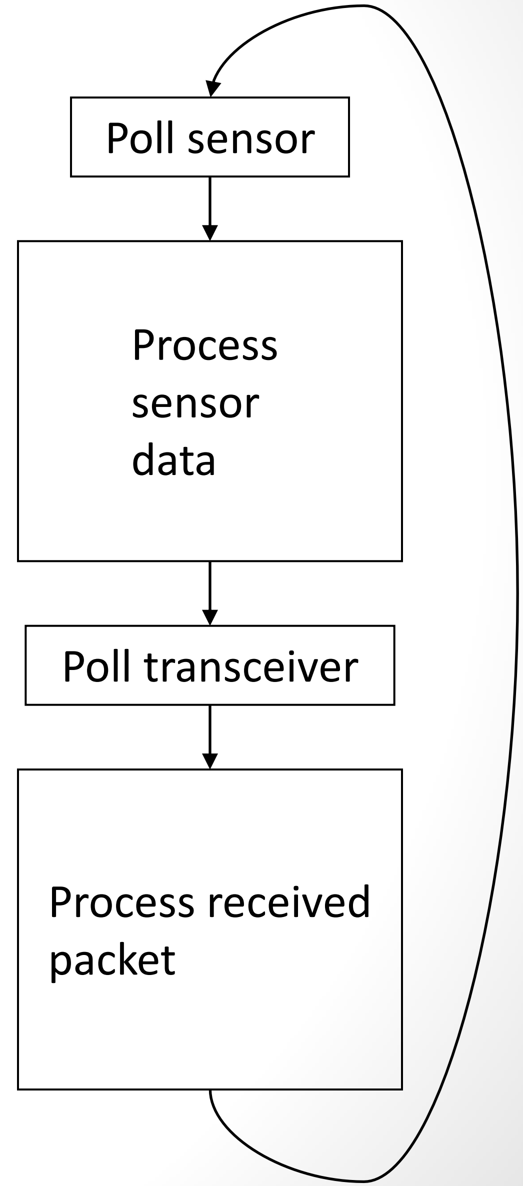
- Possible options
  - Try to implement "as close to an operating system" on WSN nodes
    - In particular, try to provide a known programming interface
    - Namely: support for processes!
    - Sacrifice protection of different processes from each other

    ! Possible, but relatively high overhead
  - Do (more or less) away with operating system
    - After all, there is only a single "application" running on a WSN node
    - No need to protect malicious software parts from each other
    - Direct hardware control by application might improve efficiency
- Currently popular verdict: no OS, just a simple run-time environment
  - Enough to abstract away hardware access details
  - Biggest impact: Unusual programming model

Main issue: How to support concurrency

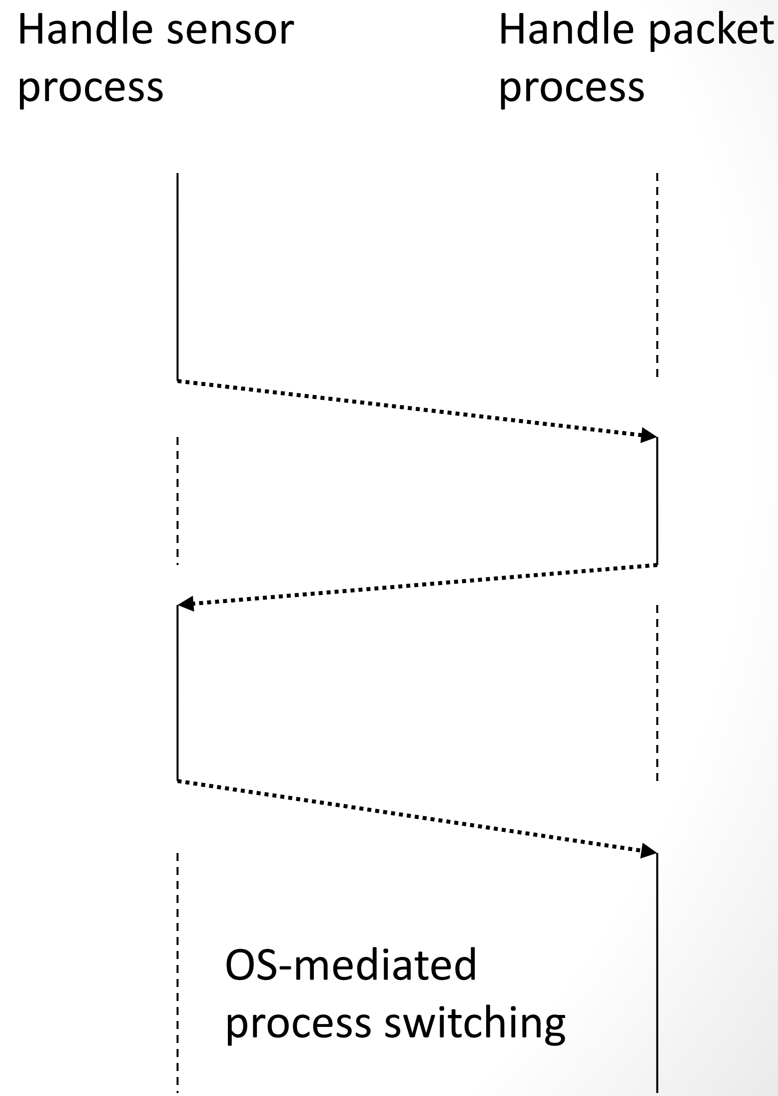
- Simplest option: No concurrency, sequential processing of tasks
  - Not satisfactory: Risk of missing data (e.g., from transceiver) when processing data, etc.
  - ! Interrupts/asynchronous operation has to be supported

- Why concurrency is needed
  - Sensor node's CPU has to service the radio modem, the actual sensors, perform computation for application, execute communication protocol software, etc.



Traditional concurrency: Processes

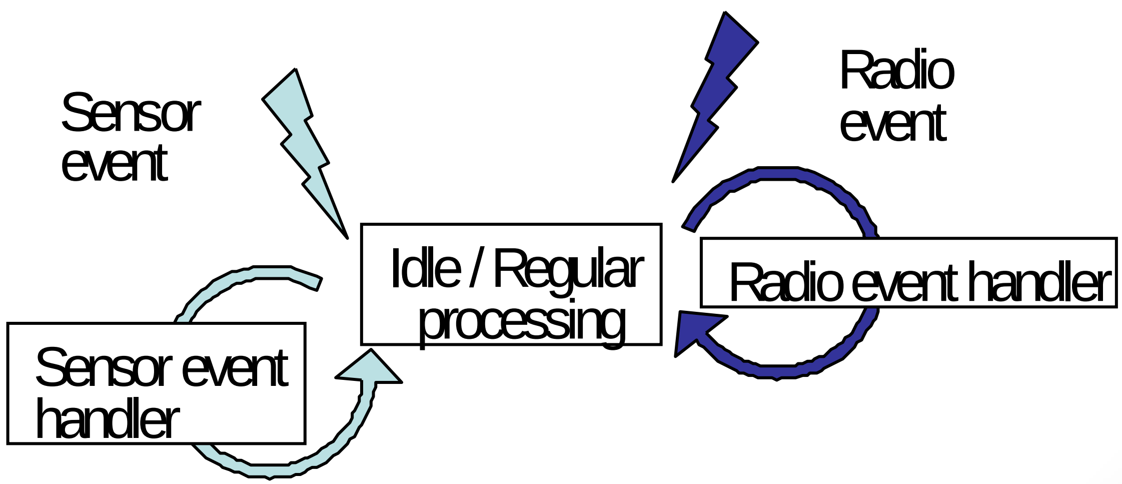
- Traditional OS: processes/threads
  - Based on interrupts, context switching
  - But: not available – memory overhead, execution overhead
- But: concurrency mismatch
  - One process per protocol entails too many context switches
  - Many tasks in WSN small with respect to context switching overhead
- And: protection between processes not needed in WSN
  - Only one application anyway



Event-based concurrency

- Alternative: Switch to ***event-based programming model***
  - Perform regular processing or be idle
  - React to events when they happen immediately
  - Basically: interrupt handler
- Problem: must not remain in interrupt handler too long
  - Danger of loosing events
  - Only save data, post information that event has happened, then return

  ! ***Run-to-completion*** principle
  - Two contexts: one for handlers, one for regular execution



Components instead of processes

- Need an abstraction to group functionality
  - Replacing "processes" for this purpose
  - E.g.: individual functions of a networking protocol

- One option: ***Components***
  - Here: In the sense of TinyOS
  - Typically fulfill only a single, well-defined function
  - Main difference to processes:
    - Component does not have an execution
    - Components access same address space, no protection against each other
  - NOT to be confused with component-based programming!

API to an event-based protocol stack

- Usual networking API: sockets
  - Issue: blocking calls to receive data
  - Ill-matched to event-based OS
  - Also: networking semantics in WSNs not necessarily well matched to/by socket semantics
- API is therefore also event-based
  - E.g.: Tell some component that some other component wants to be informed if and when data has arrived
  - Component will be posted an event once this condition is met
  - Details: see TinyOS example discussion below

Dynamic power management

- Exploiting multiple operation modes is promising
- Question: When to switch in power-safe mode?
  - Problem: Time & energy overhead associated with wakeup; greedy sleeping is not beneficial (see exercise)
  - Scheduling approach
- Question: How to control dynamic voltage scaling?
  - More aggressive; stepping up voltage/frequency is easier
  - Deadlines usually bound the required speed form below
- Or: Trading off fidelity vs. energy consumption!
  - If more energy is available, compute more accurate results
  - Example: Polynomial approximation
    - Start from high or low exponents depending where the polynomial is to be evaluated

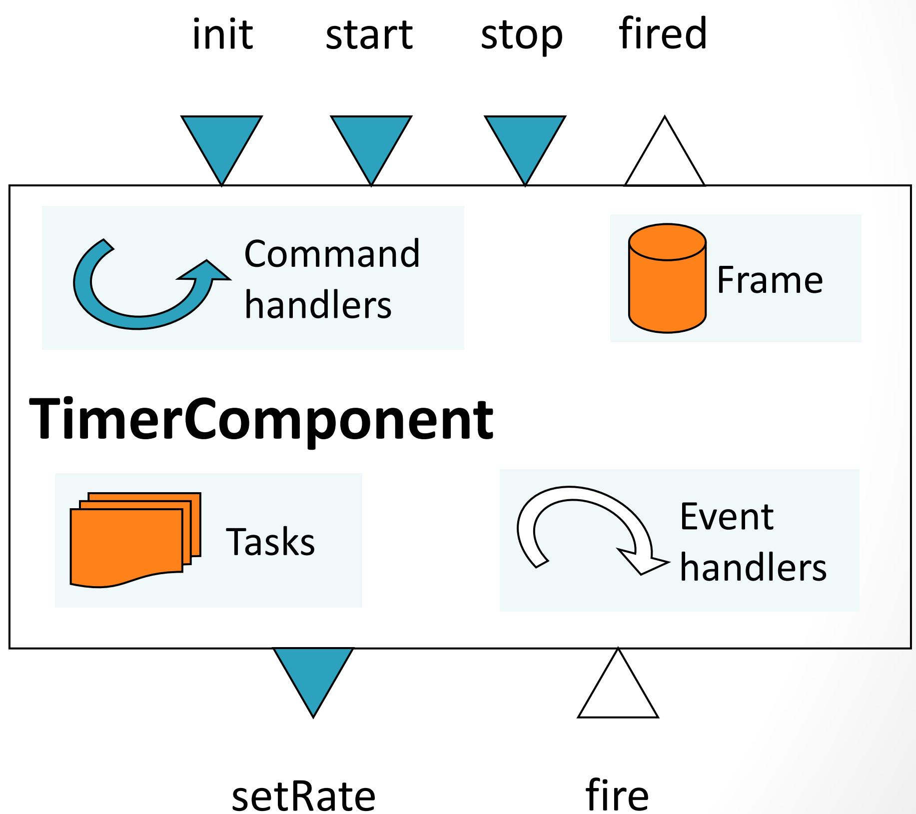
CASE STUDY: TINYOS

Case study embedded OS: TinyOS & nesC

- TinyOS developed by UC Berkely as runtime environment for their "motes"
- nesC as adjunct "programming language"
- Goal: Small memory footprint
  - Sacrifices made e.g. in ease of use, portability
  - Portability somewhat improved in newer version
- Most important design aspects
  - Component-based system
  - Components interact by exchanging asynchronous events
  - Components form a program by ***wiring*** them together (akin to VHDL – hardware description language)

TinyOS components

- Components
  - Frame – state information
  - Tasks – normal execution program
  - Command handlers
  - Event handlers
- Handlers
  - Must run to completion
  - Form a component's interface
  - Understand and emits commands & events
- Hierarchically arranged
  - Events pass upward from hardware to higher-level components
  - Commands are passed downward



Handlers versus tasks

- Command handlers and events must run to completion
  - Must not wait an indeterminate amount of time
  - Only a ***request*** to perform some action
- Tasks, on the other hand, can perform arbitrary, long computation
  - Also have to be run to completion since no non-cooperative multi-tasking is implemented
  - But can be interrupted by handlers

! No need for stack management, tasks are atomic with respect to each other

Split-phase programming

- Handler/task characteristics and separation has consequences on programming model
  - How to implement a blocking call to another component?
  - Example: Order another component to send a packet
  - Blocking function calls are not an option

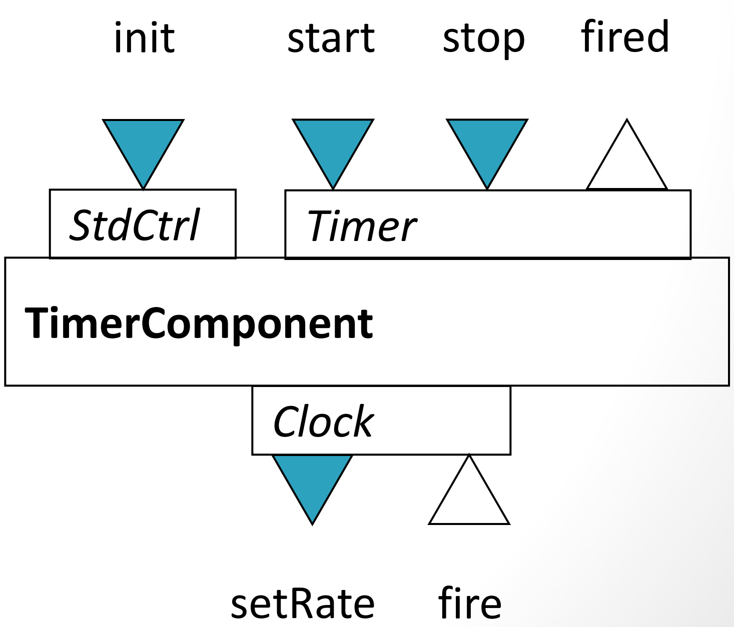
! Split-phase programming

  - First phase: Issue the command to another component
    - Receiving command handler will only receive the command, post it to a task for actual execution and returns immediately
    - Returning from a command invocation does not mean that the command has been executed!
  - Second phase: Invoked component notifies invoker by event that command has been executed
  - Consequences e.g. for buffer handling
    - Buffers can only be freed when completion event is received

Structuring commands/events into interfaces

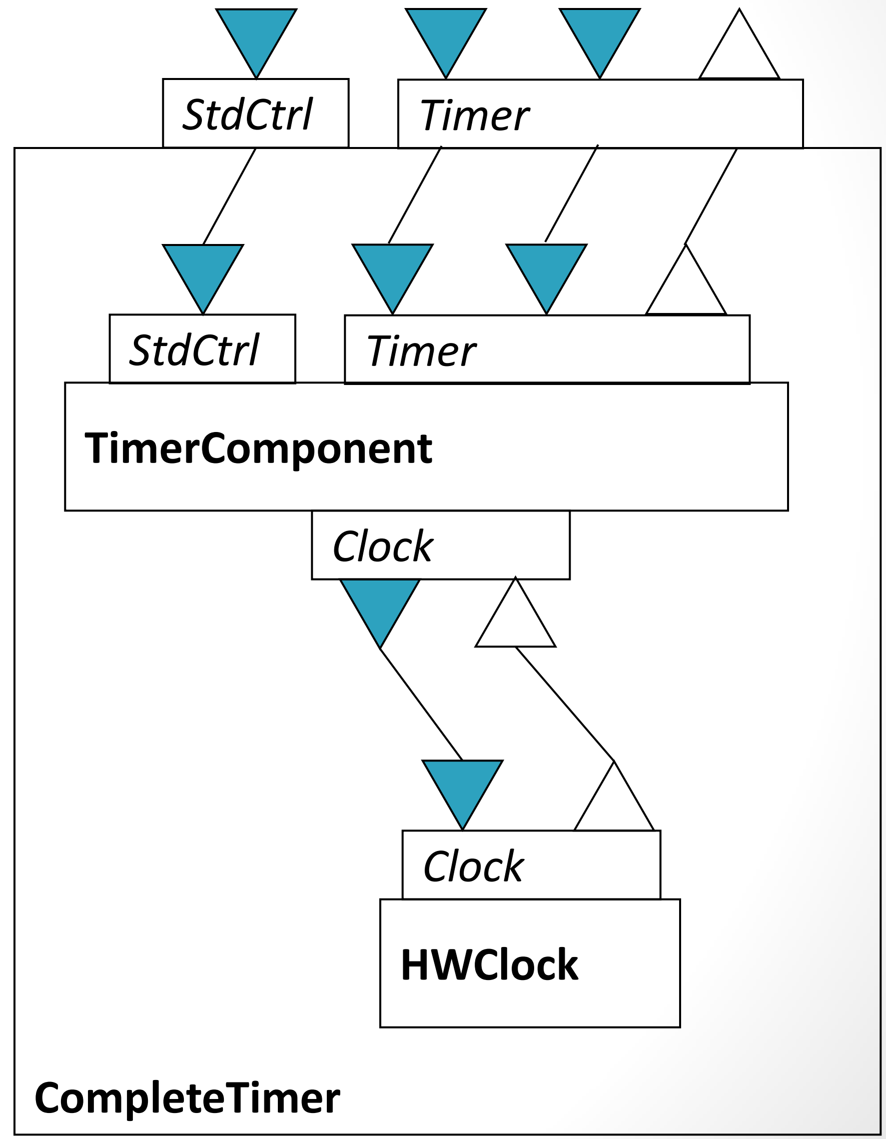
- Many commands/events can add up
- nesC solution: Structure corresponding commands/events into ***interface types***
- Example: Structure timer into three interfaces
  - StdCtrl
  - Timer
  - Clock

- Build configurations by wiring together corresponding interfaces



Building components out of simpler ones

- Wire together components to form more complex components out of simpler ones
- New interfaces for the complex component



Defining modules and components in nesC

```
interface StdCtrl {
  command result_t init();
}

interface Timer {
  command result_t start (char type, uint32_t interval);
  command result_t stop ();
  event result_t fired();
}

interface Clock {
  command result_t setRate (char interval, char scale);
  event result_t  fire ();
}

module TimerComponent {
  provides {
    interface StdCtrl;
    interface Timer;
  }
  uses interface Clock as Clk;
}
```

Wiring components to form a configuration

```
configuration CompleteTimer {
  provides {
    interface StdCtrl;
    interface Timer;
  }
  implementation {
    components TimerComponent, HWClock;
    StdCtrl = TimerComponent.HWClock;
    Timer = TimerComponent.Timer;
    TimerComponent.Clk = HWClock.Clock;
  }
}
```

Summary

- For WSN, the need to build cheap, low-energy, (small) devices has various consequences for system design
  - Radio frontends and controllers are much simpler than in conventional mobile networks
  - Energy supply and scavenging are still (and for the foreseeable future) a premium resource
  - Power management (switching off or throttling down devices) crucial
- Unique programming challenges of embedded systems
  - Concurrency without support, protection
  - De facto standard: TinyOS

- For more details, refer to:
  - Chapter 2, H. Karl and A. Willig, **Protocols and Architectures for Wireless Sensor Networks**, Wiley 2005.
- The lecture is available online at:
  - http://bu.edu.eg/staff/ahmad.elbanna-courses/12189
- For inquires, send to:
  - ahmad.elbanna@feng.bu.edu.eg