Chapter 2: Single Node Architecture

For use in conjunction with *Protocols and Architectures for Wireless Sensor Networks,* by Holger Karl, Andreas Willig (http://www.wiley.com)

Prof. **Yuh-Shyan Chen**

Department of Computer Science and Information Engineering

National Taipei University

Sep. 2006

Goals of this 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

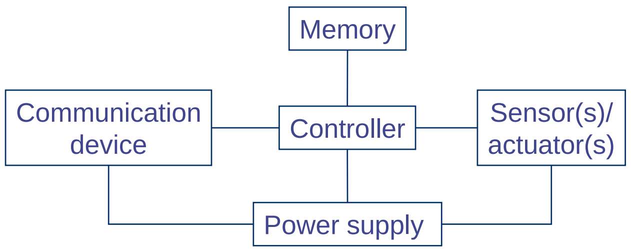
- Note: The details of this chapter are quite specific to WSN; energy consumption principles carry over to MANET as well

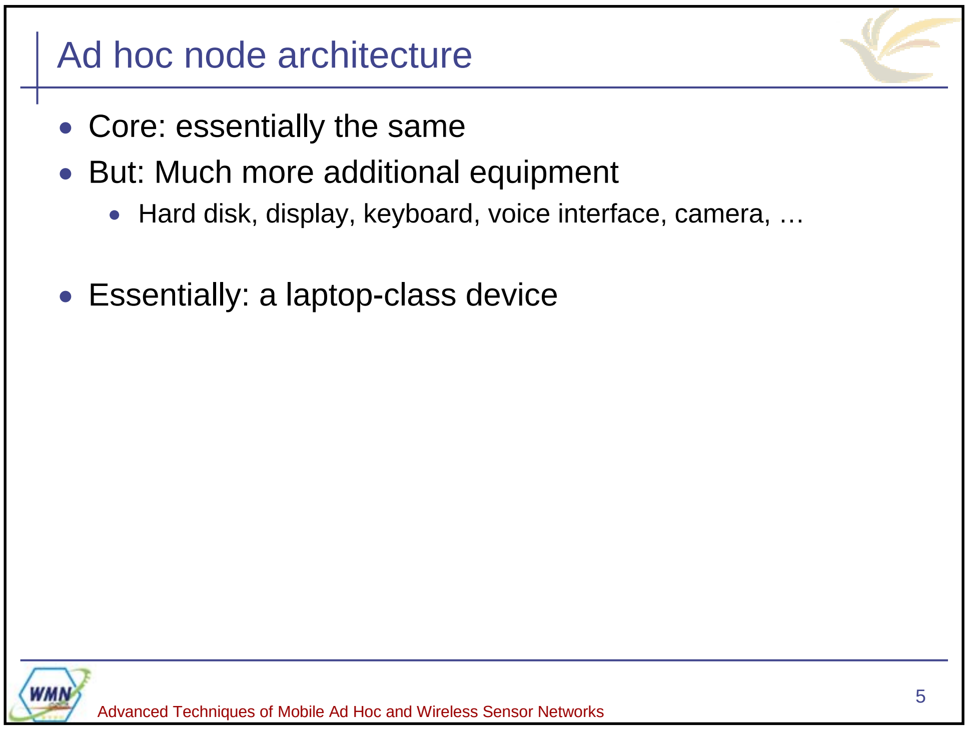
Outline

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

Sensor node architecture

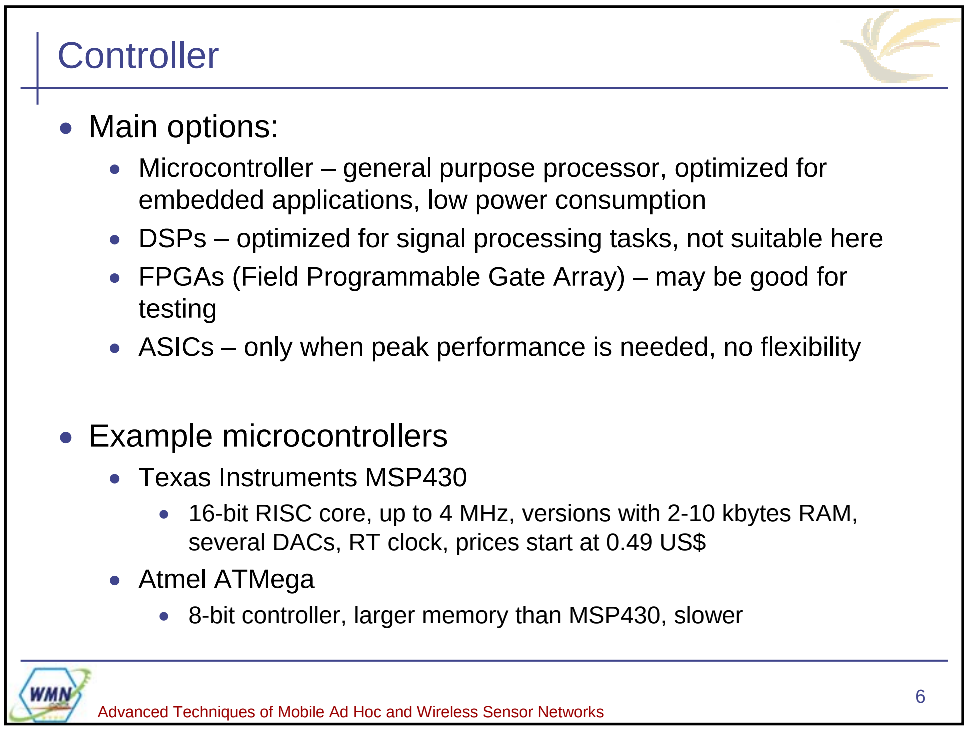
- Main components of a WSN node
  - Controller
  - Communication device(s)
  - Sensors/actuators
  - Memory
  - Power supply





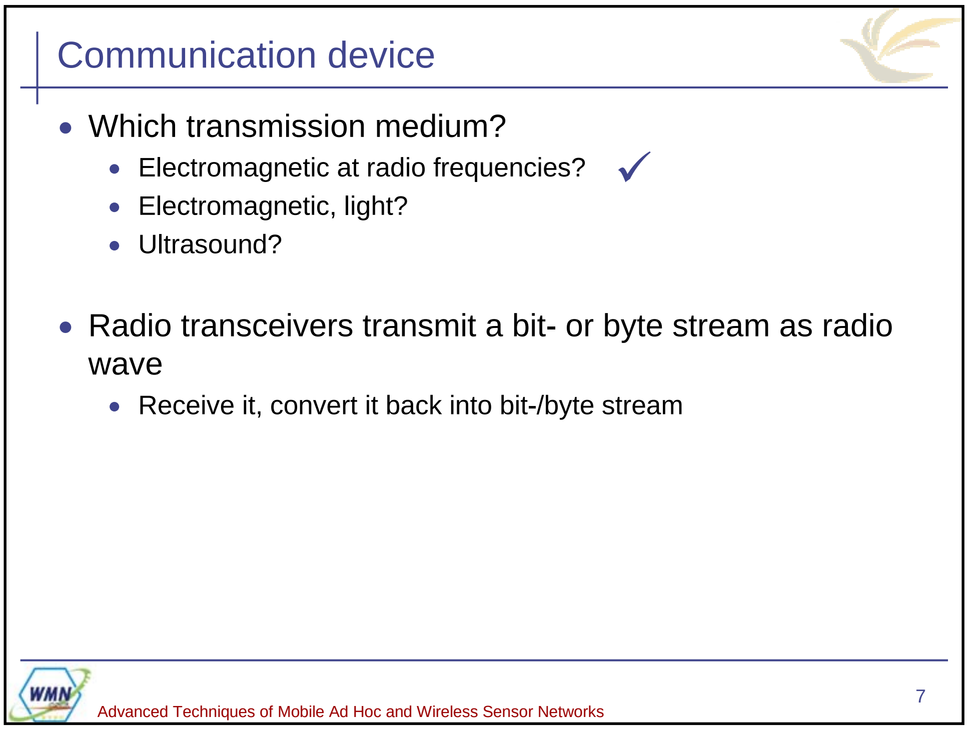
Ad hoc node architecture

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



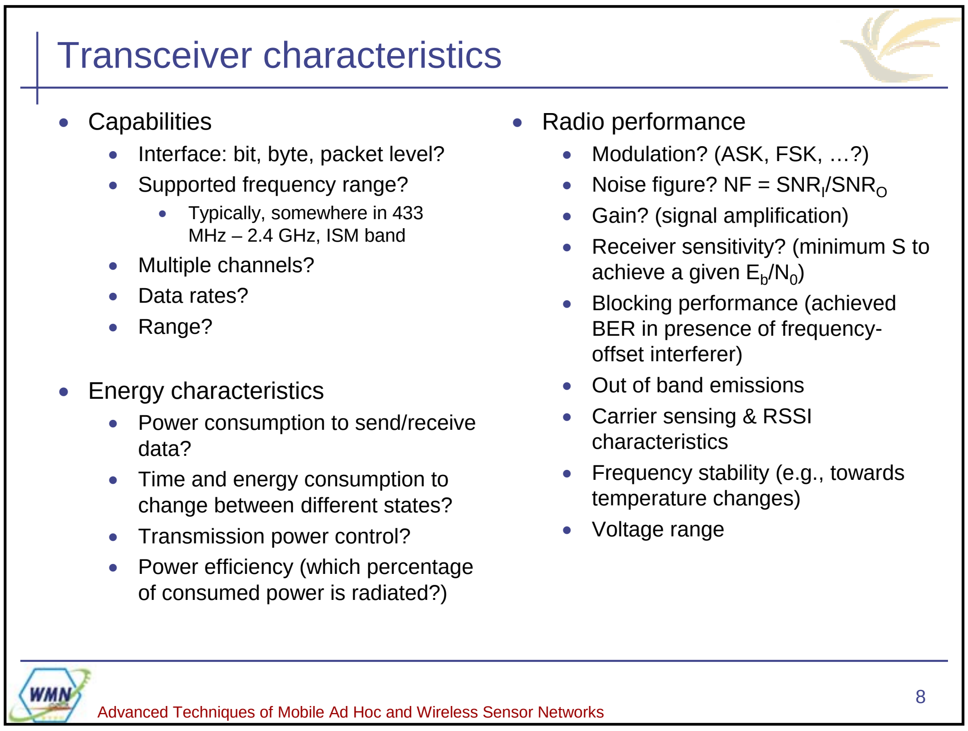
Controller

- Main options:
  - Microcontroller – general purpose processor, optimized for embedded applications, low power consumption
  - DSPs – optimized for signal processing tasks, not suitable here
  - FPGAs (Field Programmable Gate Array) – 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



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!)

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

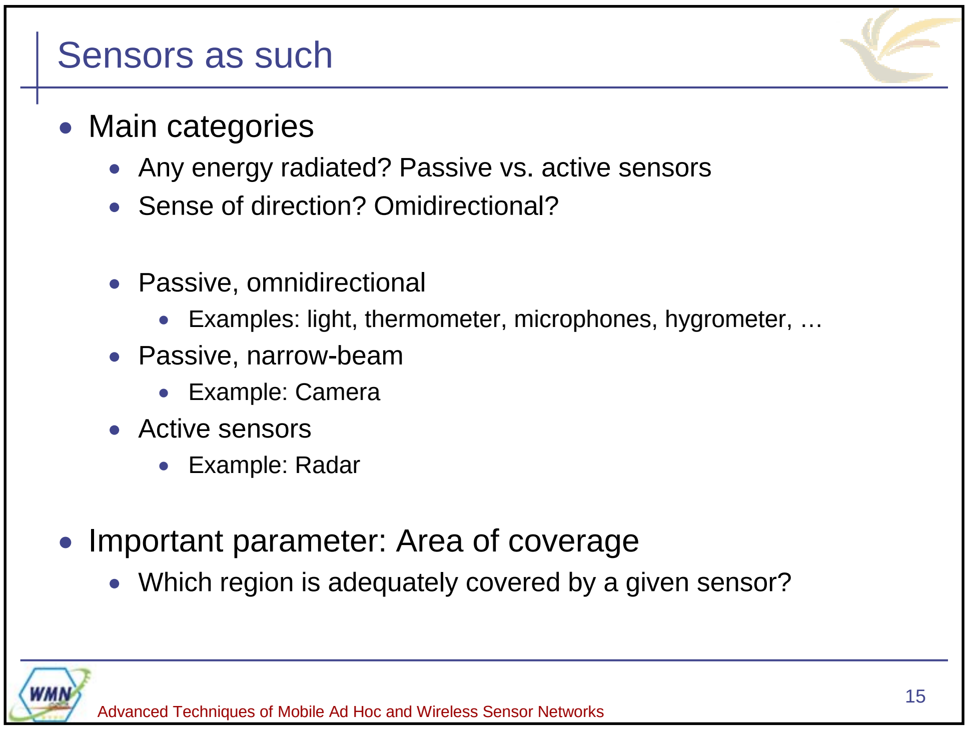
- Ad hoc networks: Usually, higher data rates are required
- Typical: IEEE 802.11 b/g/a is considered
  - Up to 54 MBit/s
  - Relatively long distance (100s of meters possible, typical 10s of meters at higher data rates)
  - Works reasonably well (but certainly not perfect) in mobile environments
  -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

Ultra-wideband communication

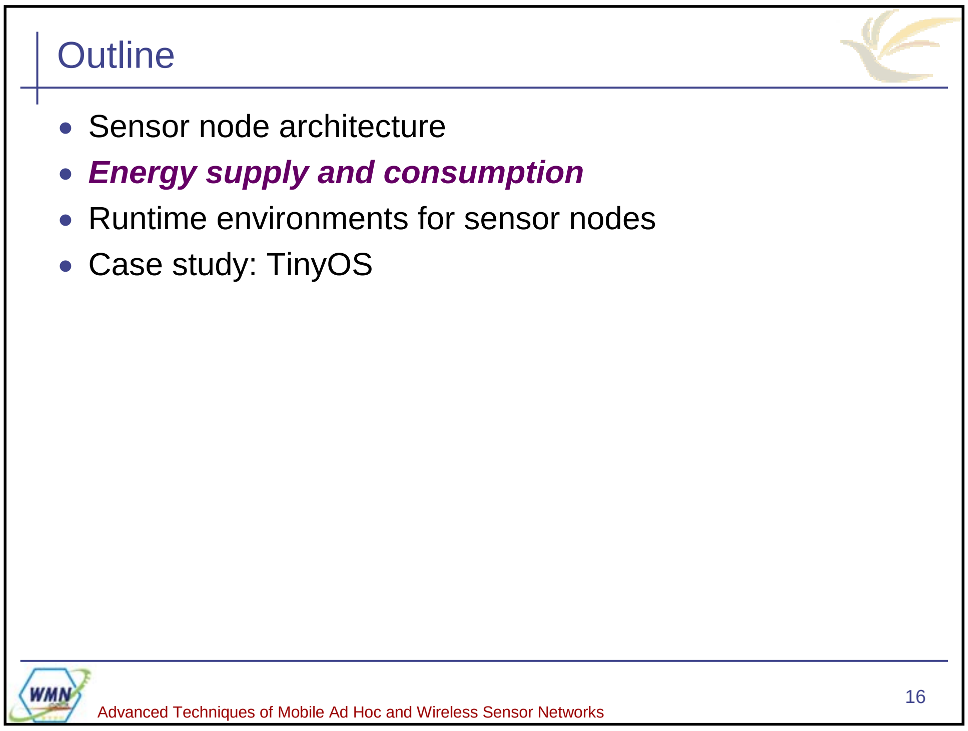
- Standard radio transceivers: Modulate a signal onto a carrier wave
  - Requires relatively small amount of bandwidth
- Alternative approach: Use a large bandwidth, do not modulate, simply emit a “burst” of power
  - Forms almost rectangular pulses
  - Pulses are very short
  - Information is encoded in the presence/absence of pulses
  - Requires tight time synchronization of receiver
  - Relatively short range (typically)
- Advantages
  - Pretty resilient to multi-path propagation
  - Very good ranging capabilities
  - Good wall penetration



Sensors as such

- 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?



Outline

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

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)

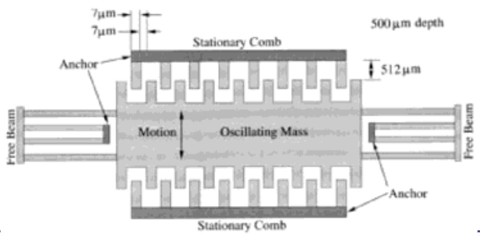
Battery examples

- 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 $\rightarrow$ 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 mWh/cm^3
Batteries (rechargable lithium)	300 mWh/cm^3 (at 3 – 4 V)
Energy source	**Power density**
Solar (outdoors)	15 mW/cm^2 (direct sun)
	0.15 mW/cm^2 (cloudy day)
Solar (indoors)	0.006 mW/cm^2 (standard office desk)
	0.57 mW/cm^2 (< 60 W desk lamp)
Vibrations	0.01 – 0.1 mW/cm^3
Acoustic noise	$3 \cdot 10^{-6}$mW/cm^2 at 75 Db
	$9,6 \cdot 10^{-4}$mW/cm^2 at 100 Db
Passive human-powered systems	1.8 mW (shoe inserts)
Nuclear reaction	80 mW/cm^3, 10^6 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 $\approx$ ***11.5 μW*** as max. sustained power consumption!
- Not feasible!

Multiple power consumption modes

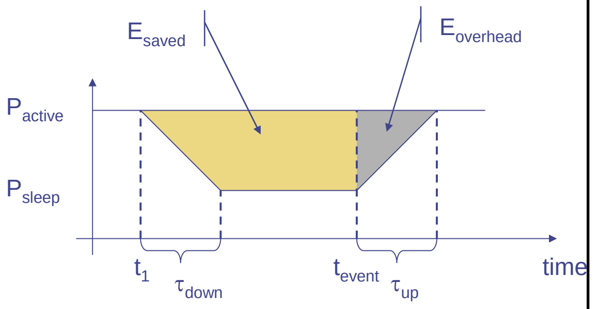
- Way out: Do not run sensor node at full operation all the time
  - If nothing to do, switch to ***power safe mode***
  - Question: When to throttle down? How to wake up again?
- Typical modes
  - Controller: Active, idle, sleep
  - Radio mode: Turn on/off transmitter/receiver, both
- Multiple modes 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
    - $P \propto f V^2$
  - 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: $\approx$ 1.1 nAh per byte
  - Writing: $\approx$ 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: $n / (R \cdot R_{code})$
  - R nomial data rate, R_{code} coding rate
- To leave sleep mode
  - Time T_{start}, average power P_{start}

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

- 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 $n / (R \cdot R_{code})$
- Receiver electronics needs P_{rxElec}
- Plus: energy to decode n bits $E_{decBits}$

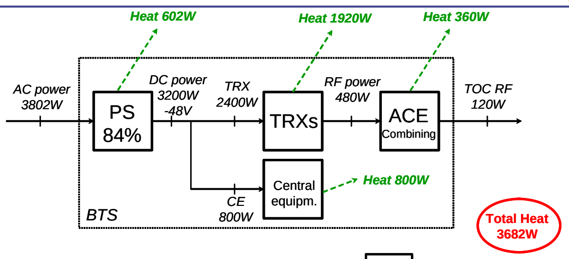
$$\rightarrow \mathbf{E_{rx} = T_{start}\ P_{start} + n / (R \cdot R_{code})\ P_{rxElec} + E_{decBits}\ (R)}$$

Some transceiver numbers

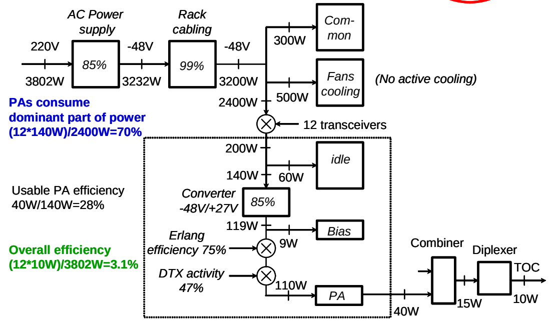
Symbol	Description	Example transceiver μAMPS-1 [559]	WINS [670]	MEDUSA-II [670]
α_{amp}	Eq. (2.4)	174 mW	N/A	N/A
β_{amp}	Eq. (2.4)	5.0	8.9	7.43
P_{amp}	Amplifier pwr.	179 – 674 mW	N/A	N/A
P_{rxElec}	Reception pwr.	279 mW	368.3 mW	12.48 mW
P_{rxIdle}	Receive idle	N/A	344.2 mW	12.34 mW
P_{start}	Startup 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

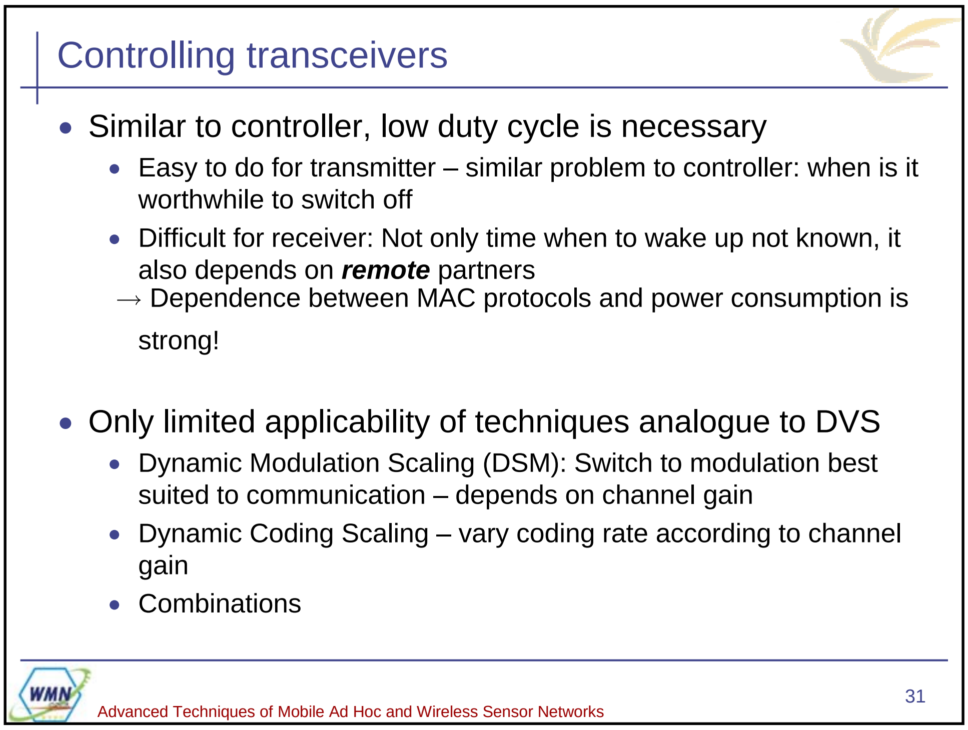
- Overview



- Details



- (just to put things into perspective)

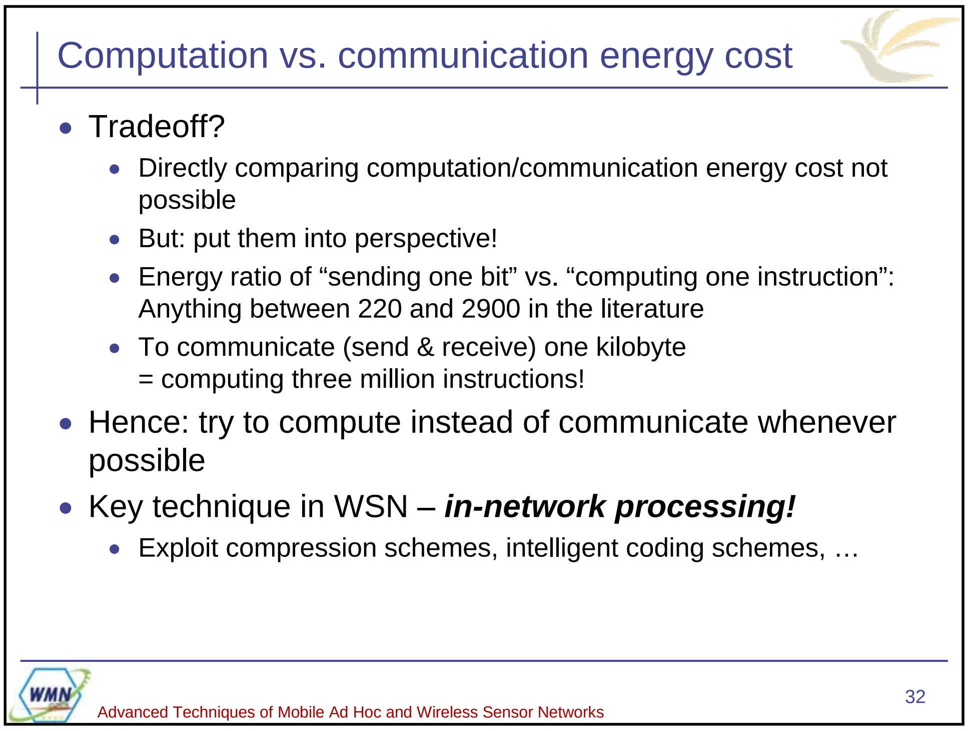


Controlling transceivers

- 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, …

Outline

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

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

$\rightarrow$???

Operating system challenges in WSN

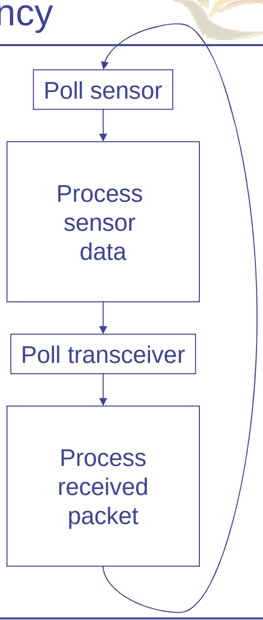
- Possible options
  - Try to implement "as close to an operating system" on WSN nodes
    - In particular, try to provide a known programming interface
    - Namely: support for processes!
    - Sacrifice protection of different processes from each other

    → 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

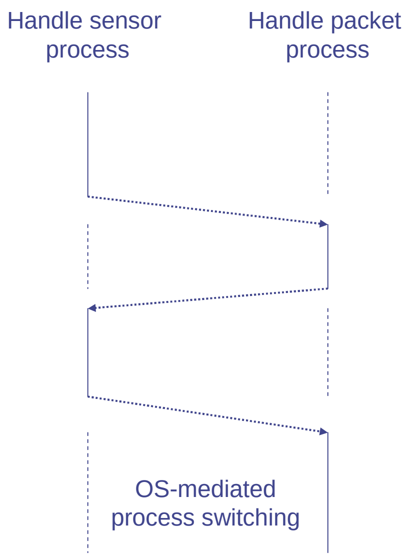
- Simplest option: No concurrency, sequential processing of tasks
  - Not satisfactory: Risk of missing data (e.g., from transceiver) when processing data, etc.

  → 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.



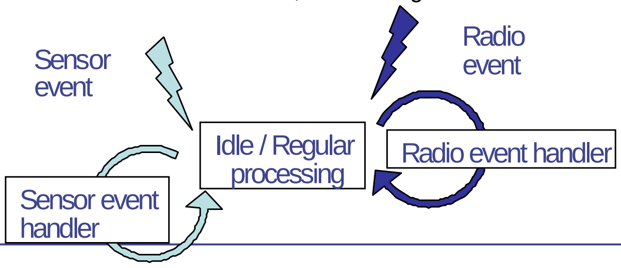
Traditional concurrency: Processes

- 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

Outline

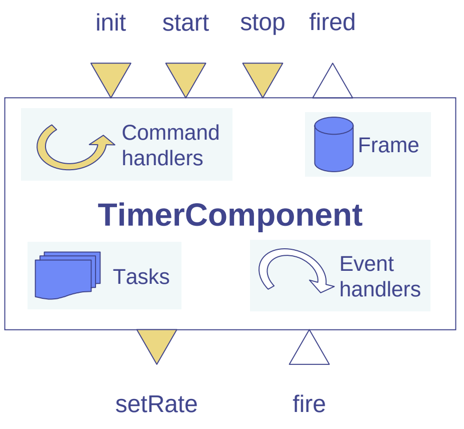
- Sensor node architecture
- Energy supply and consumption
- Runtime environments for sensor nodes
- ***Case study: TinyOS***

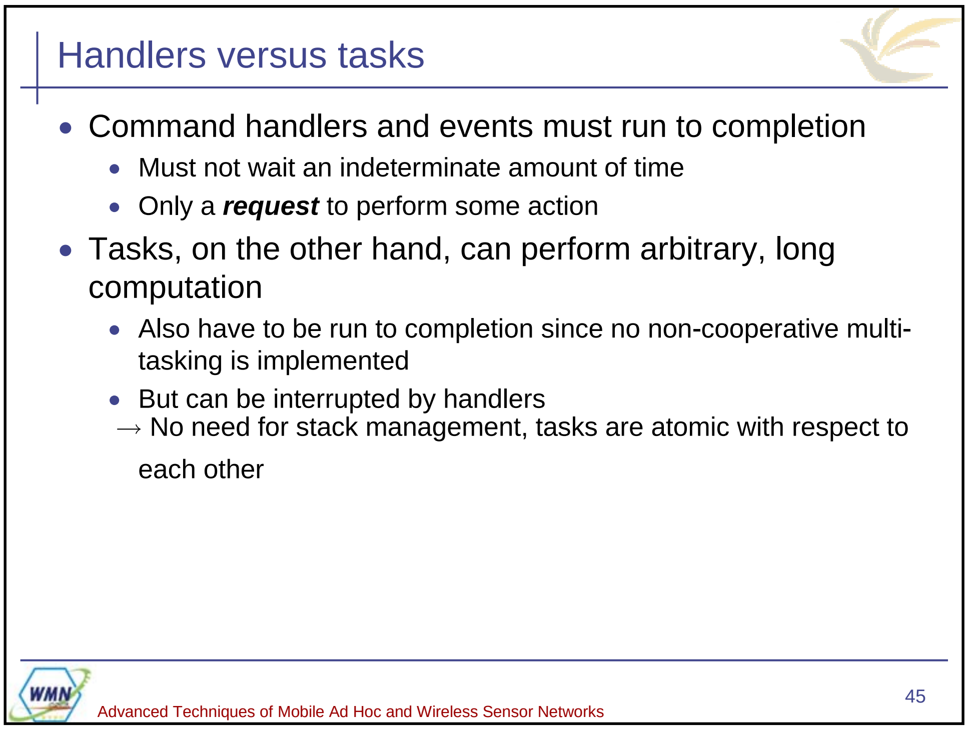
Case study embedded OS: TinyOS & nesC

- 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

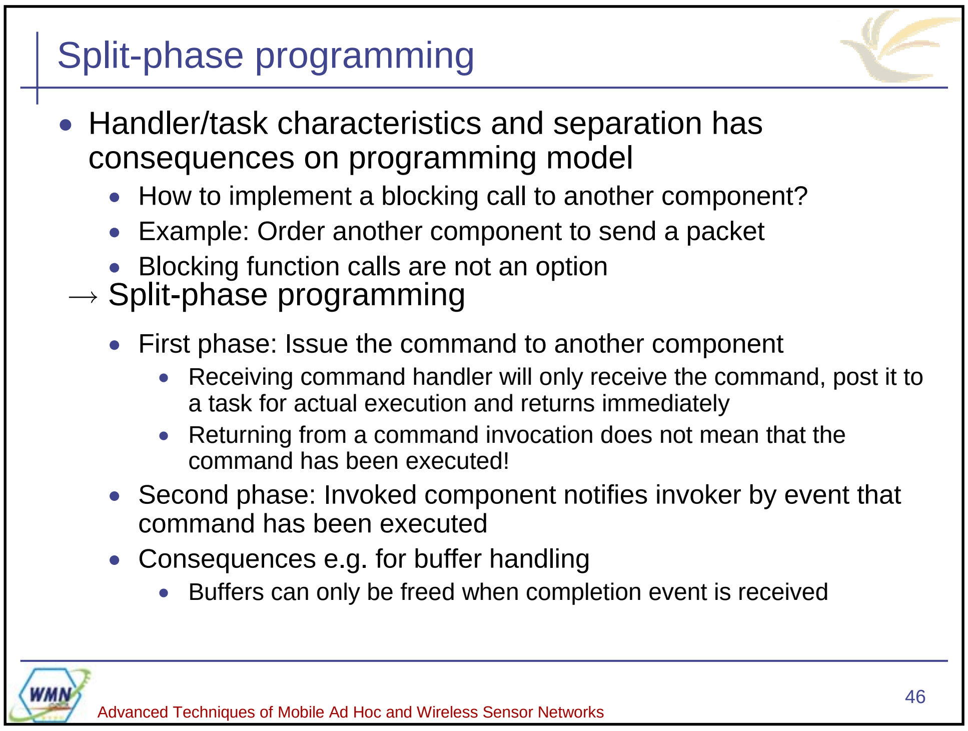




Handlers versus tasks

- 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

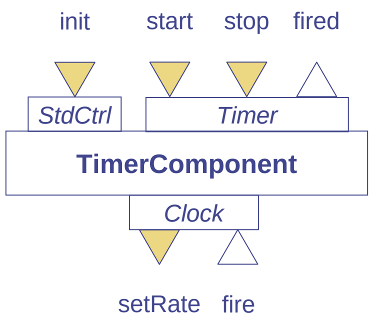
- Handler/task characteristics and separation has consequences on programming model
  - How to implement a blocking call to another component?
  - Example: Order another component to send a packet
  - Blocking function calls are not an option

→ 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

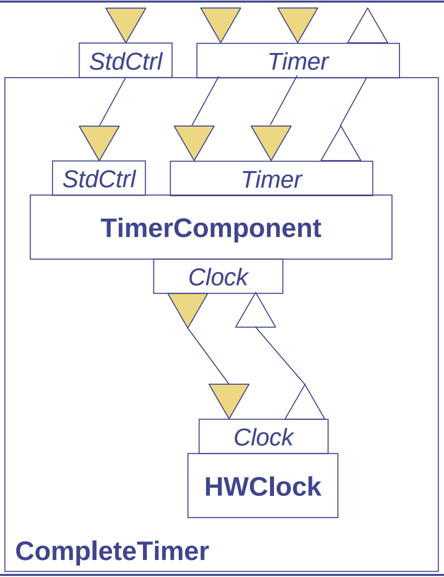
Structuring commands/events into interfaces

- 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

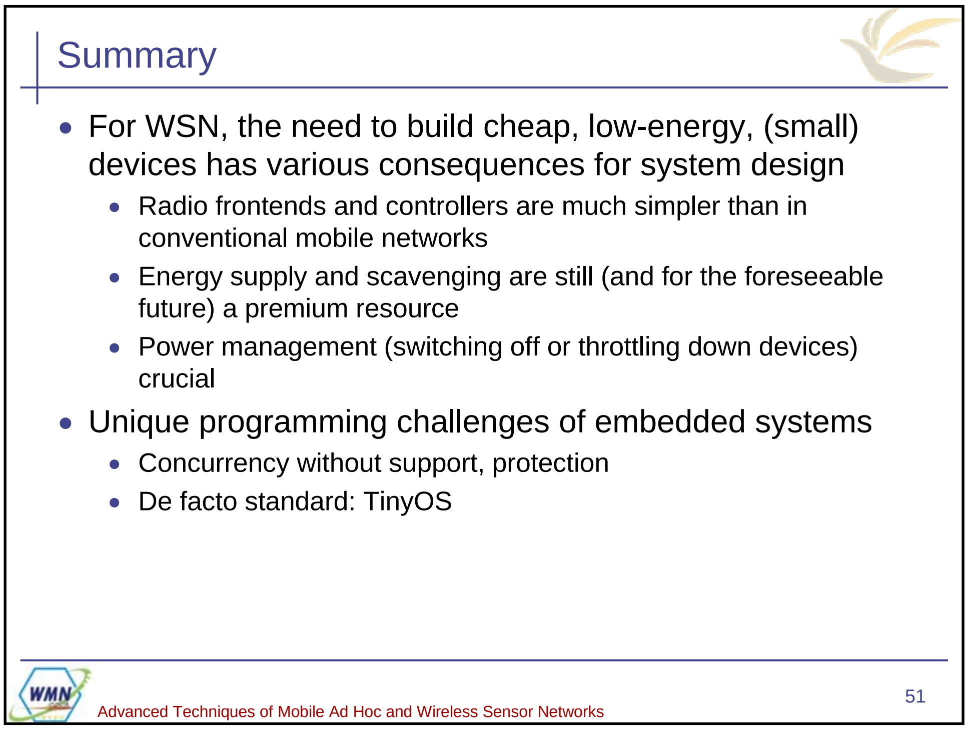


Defining modules and components in nesC

```
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