Network Kernel Architectures and Implementation (01204423)

Single-Node Architectures

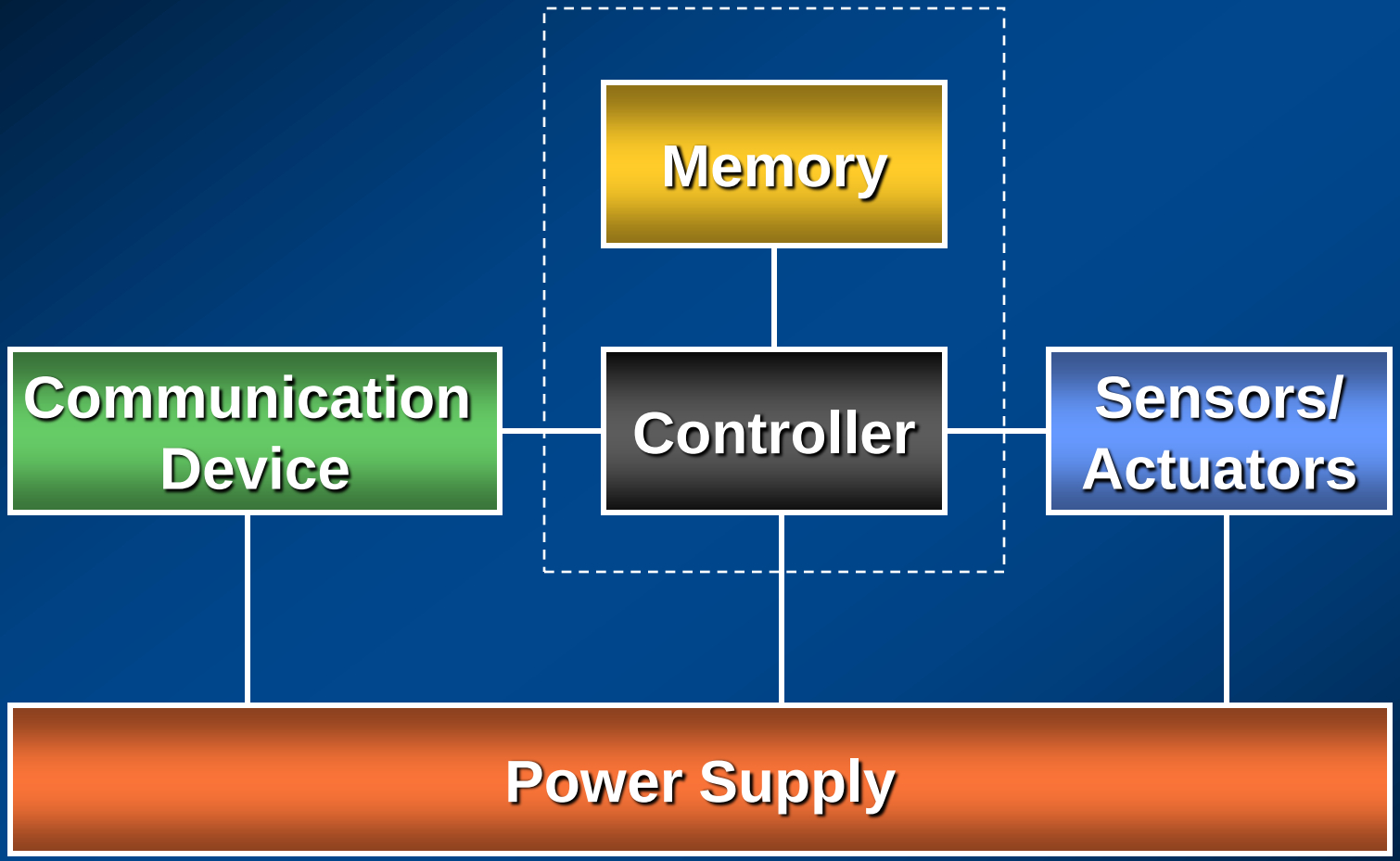
Chaiporn Jaikaeo
chaiporn.j@ku.ac.th
Department of Computer Engineering
Kasetsart University

Materials taken from lecture slides by Karl and Willig
Contiki materials taken from slides by Adam Dunkels

Outline

- Main components of a wireless sensor node
  - Processor, radio, sensors, batteries
- Energy supply and consumption
- Operating systems and execution environments
  - IWING's MoteLib
  - TinyOS
  - Contiki
- Sample implementations

Main Components



Controller

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

Microcontroller Examples

- Texas Instruments MSP430
  - 16-bit RISC core, 4 MHz
  - Up to 120 KB flash
  - 2-10 KB RAM
  - 12 ADCs, RT clock
- Atmel ATMega
  - 8-bit controller, 8 MHz
  - Up to 128KB Flash
  - 4 KB RAM

Communication Device

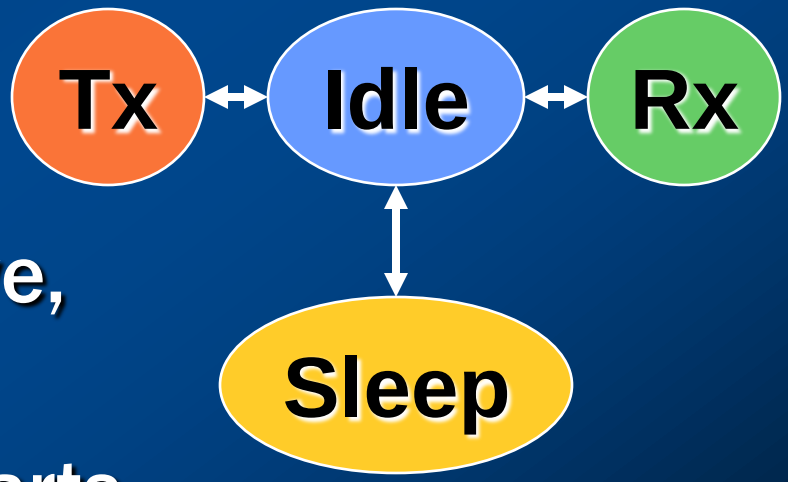
- Medium options
  - Electromagnetic, RF
  - Electromagnetic, optical
  - Ultrasound

Transceiver Characteristics

- Service to upper layer: packet, byte, bit
- Power consumption
- Supported frequency, multiple channels
- Data rate
- Modulation
- Power control
- Communication range
- etc.

Transceiver States

- Transceivers can be put into different operational ***states***, typically:
  - ***Transmit***
  - ***Receive***
  - ***Idle*** – ready to receive, but not doing so
  - ***Sleep*** – significant parts of the transceiver are switched off

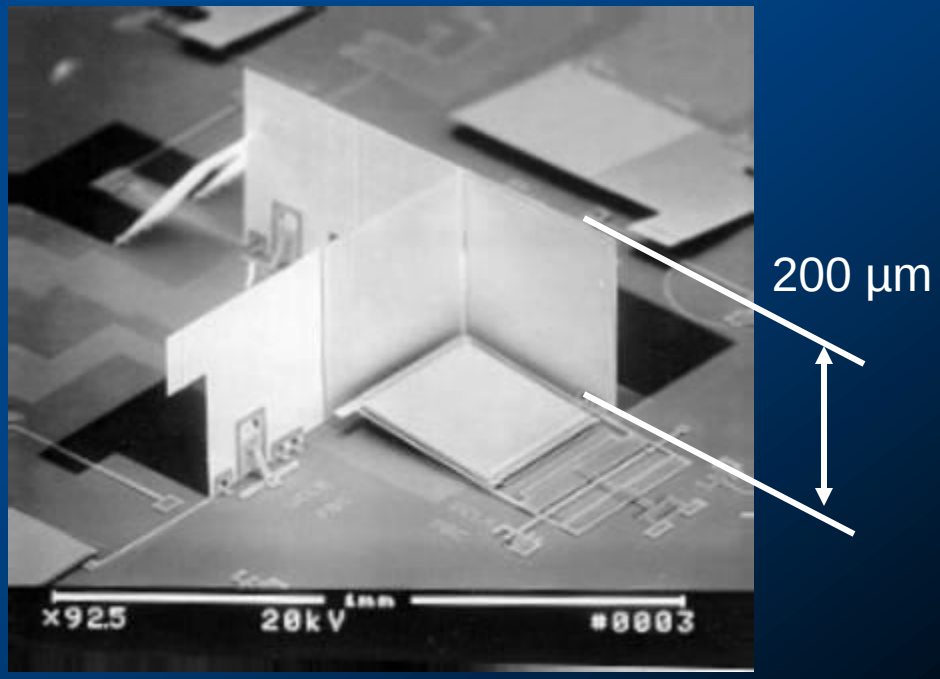


Wakeup Receivers

- 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

Optical Communication

- Optical communication can consume less energy
- Example: passive readout via corner cube reflector
  - Laser is reflected back directly to source if mirrors are at right angles
  - Mirrors can be “tilted” to stop reflecting
  - Allows data to be sent back to laser source



Sensors

- Main categories
  - 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

- Main components of a wireless sensor node
  - Processor, radio, sensors, batteries
- *Energy supply and consumption*
- Operating systems and execution environments
  - IWING's MoteLib
  - TinyOS
  - Contiki
- Example implementations

Energy Supply

- 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

Energy Supply - Requirements

- Low self-discharge
- Long shelf life
- Capacity under load
- Efficient recharging at low current
- Good relaxation properties (seeming self-recharging)
- Voltage stability (to avoid DC-DC conversion)

Battery Examples

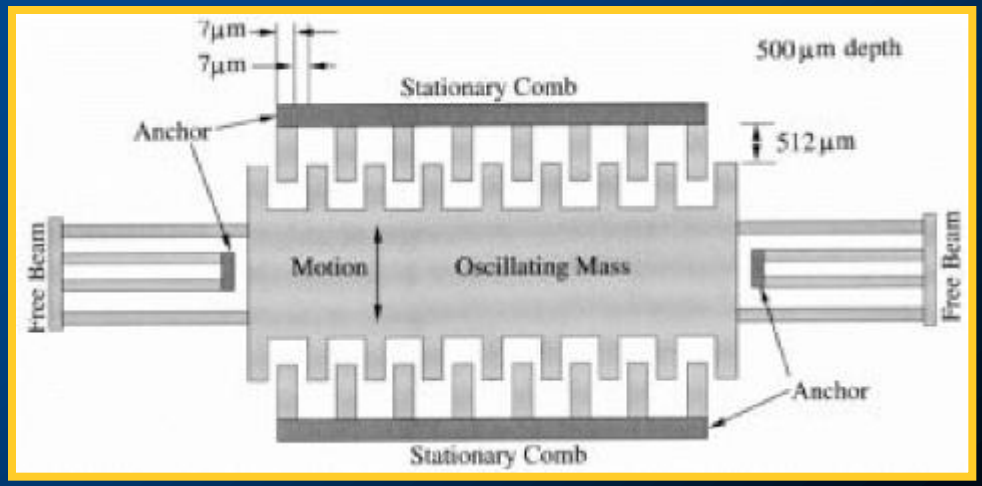
- Energy per volume (Joule/cc):

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

http://en.wikipedia.org/wiki/Energy_density

Energy Harvesting

- 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 $\mu W/cm^2$ and 15 mW/cm^2
  - Temperature gradients – 80 $\mu W/cm^2$ @ 1 V from 5K difference
  - Vibrations – between 0.1 and 10000 $\mu W/cm^3$
  - Pressure variation (piezo-electric) – 330 $\mu W/cm^2$ from the heel of a shoe
  - Air/liquid flow (MEMS gas turbines)



Portable Solar Chargers

- Foldable Solar Chargers
  - http://www.energyenv.co.uk/FoldableChargers.asp
- Solargorilla
  - http://powertraveller.com/iwantsome/primatepower/solargorilla/

Multiple Power Consumption Modes

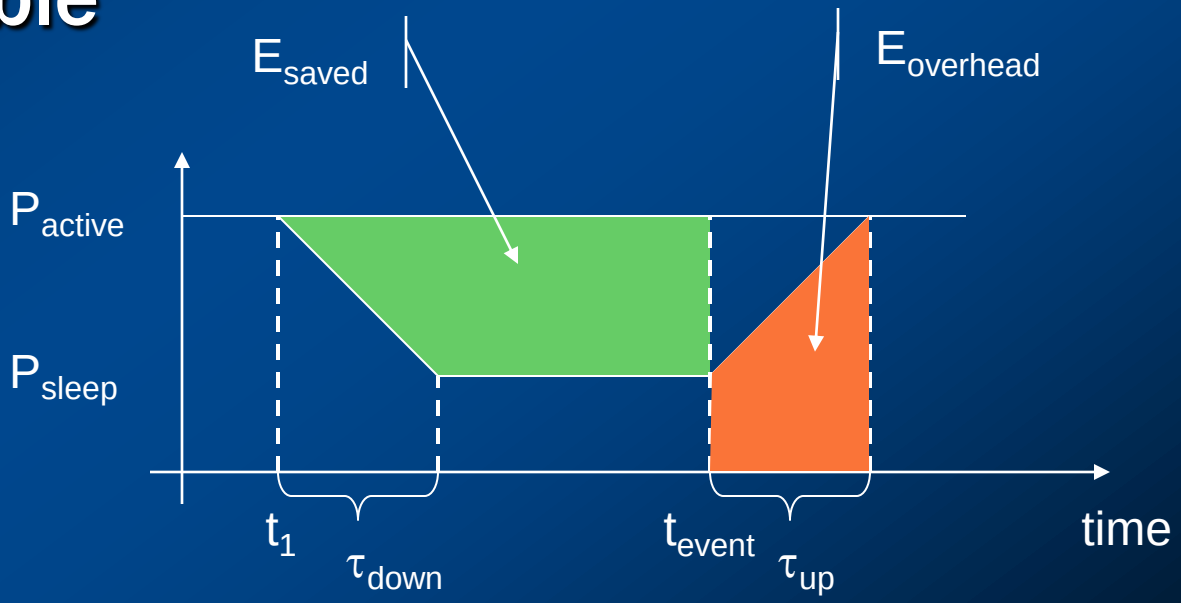
- Do not run sensor node at full operation all the time
  - If nothing to do, switch to power safe mode
- Typical modes
  - Controller: Active, idle, sleep
  - Radio mode: Turn on/off transmitter/receiver, both
  - Strongly depends on hardware
- Questions:
  - When to throttle down?
  - How to wake up again?

Energy Consumption Figures

- TI MSP 430 (@ 1 MHz, 3V):
  - Fully operation 1.2 mW
  - One fully operational mode + four sleep modes
  - 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
  - Six modes of operations
  - 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



Should We Switch?

- Switching modes is beneficial if

$$E_{overhead} < E_{saved}$$

which is equivalent to

$$(t_{event} - t_1) > \frac{1}{2}\left(\tau_{down} + \frac{P_{active} + P_{sleep}}{P_{active} - P_{sleep}}\tau_{up}\right)$$

Computation vs. Communication Energy Cost

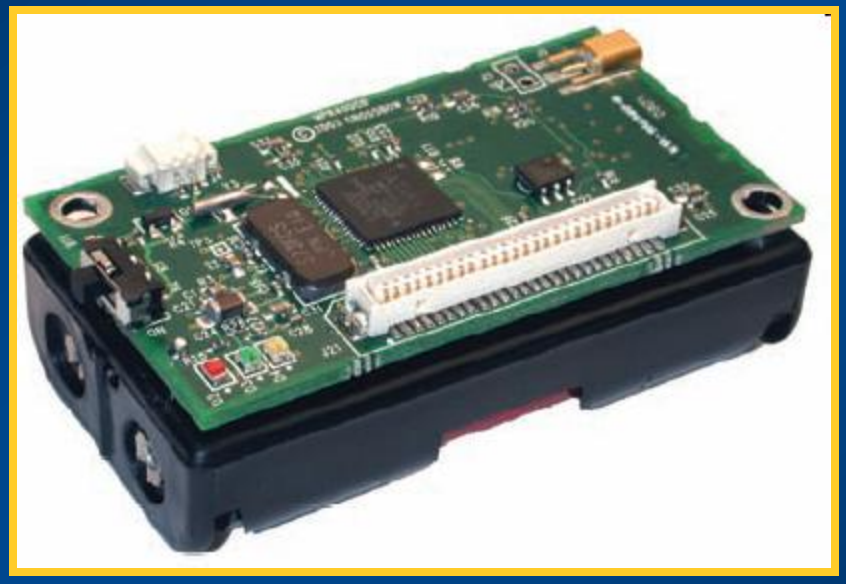
- Sending one bit vs. running one instruction
  - Energy ratio up to 2900:1
  - I.e., send & receive one KB = running three million instruction
- So, *try to compute instead of communicate whenever possible*
- Key technique – ***in-network processing***
  - Exploit compression schemes, intelligent coding schemes, aggregate data, …

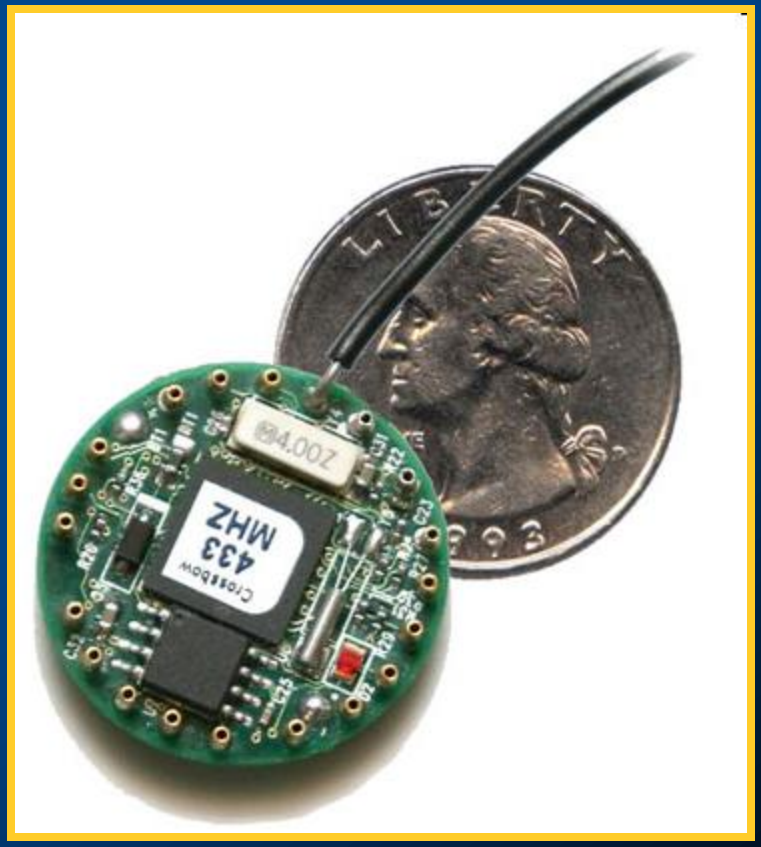
Outline

- Main components of a wireless sensor node
  - Processor, radio, sensors, batteries
- Energy supply and consumption
- Operating systems and execution environments
  - IWING's MoteLib
  - TinyOS
  - Contiki
- *Example implementations*

Mica Motes

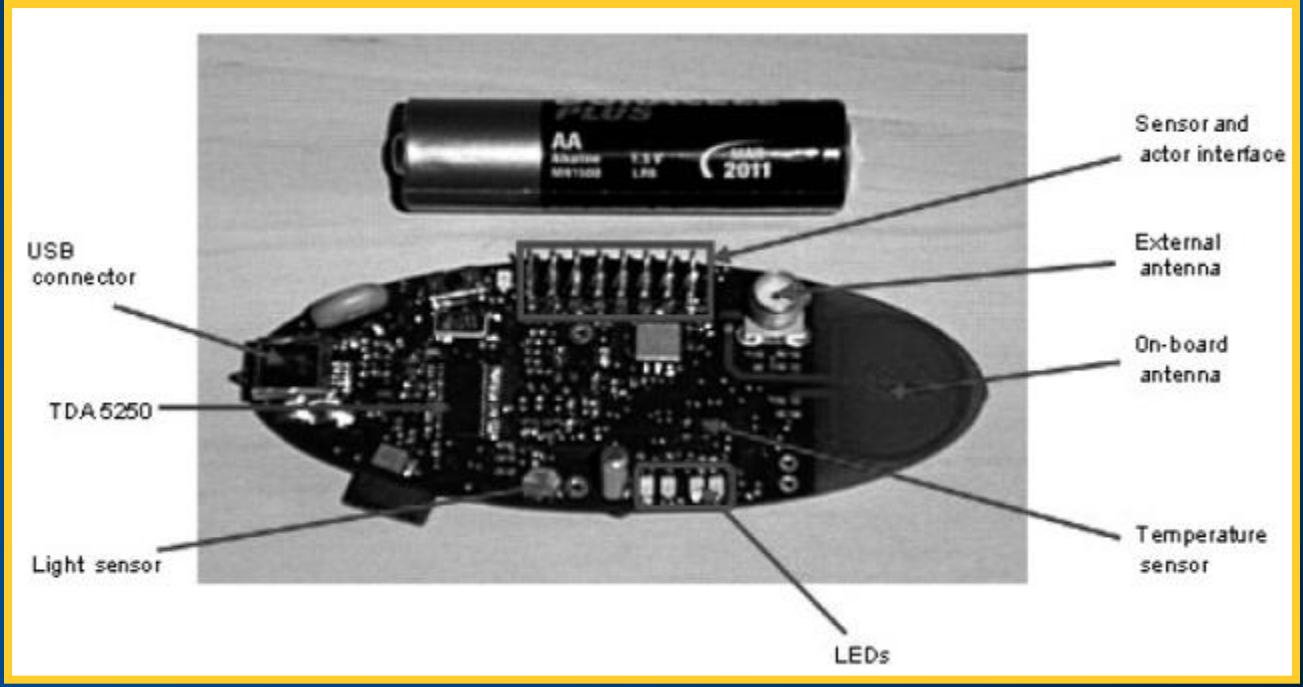
- By Crossbow, USA
- MCU:
  - Atmel ATMega128L
- Comm: RFM TR1000





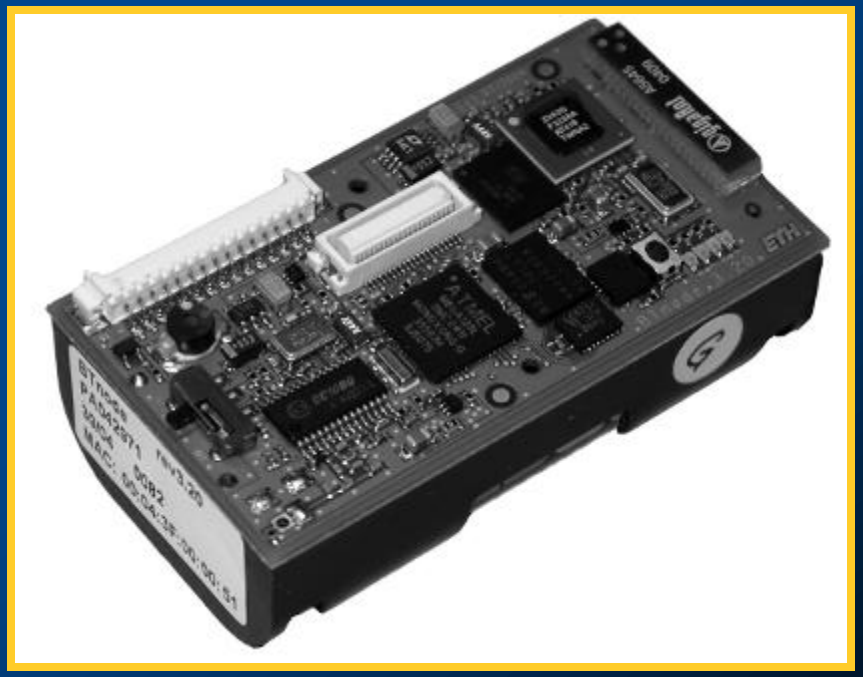
EYES Nodes

- By Infineon, EU
- MCU: TI MSP430
- Comm: Infineon radio modem TDA5250



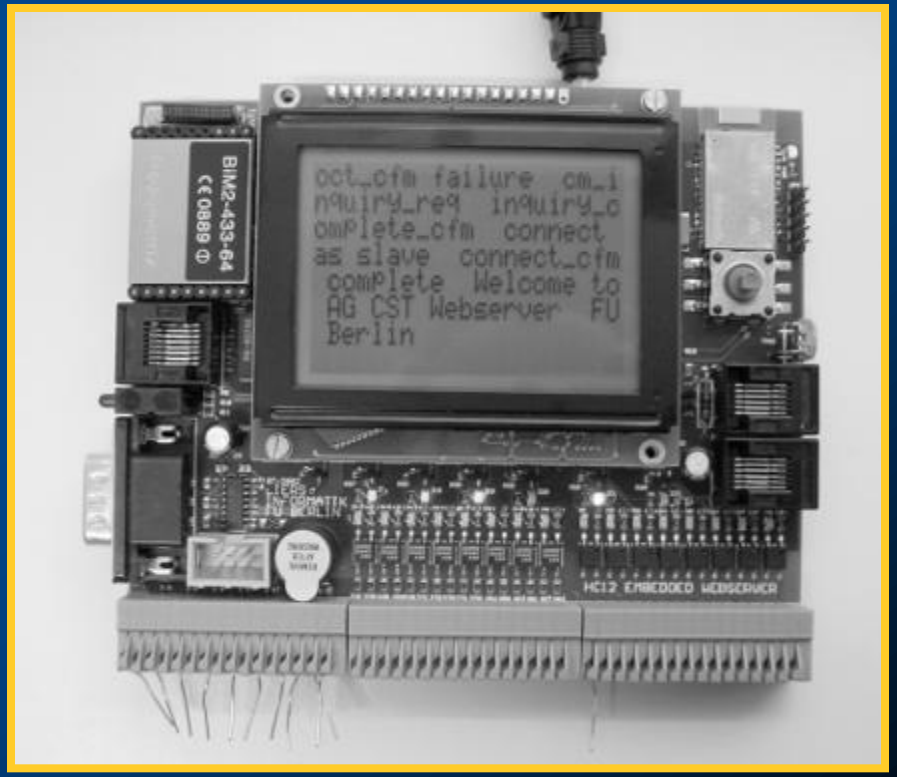
Btnote

- By ETH Zurich
- MCU:
  - Atmel ATMega128L
- Comm:
  - Bluetooth
  - Chipcon CC1000



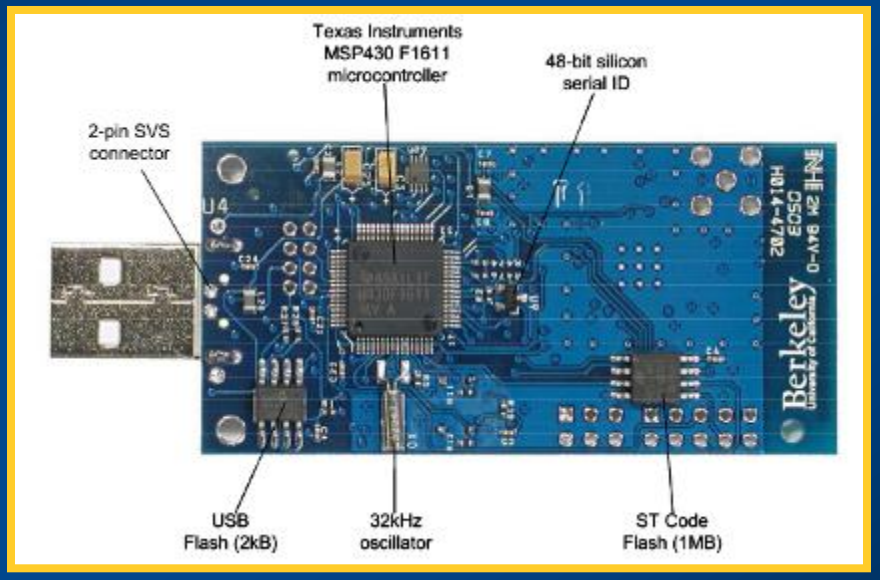
ScatterWeb

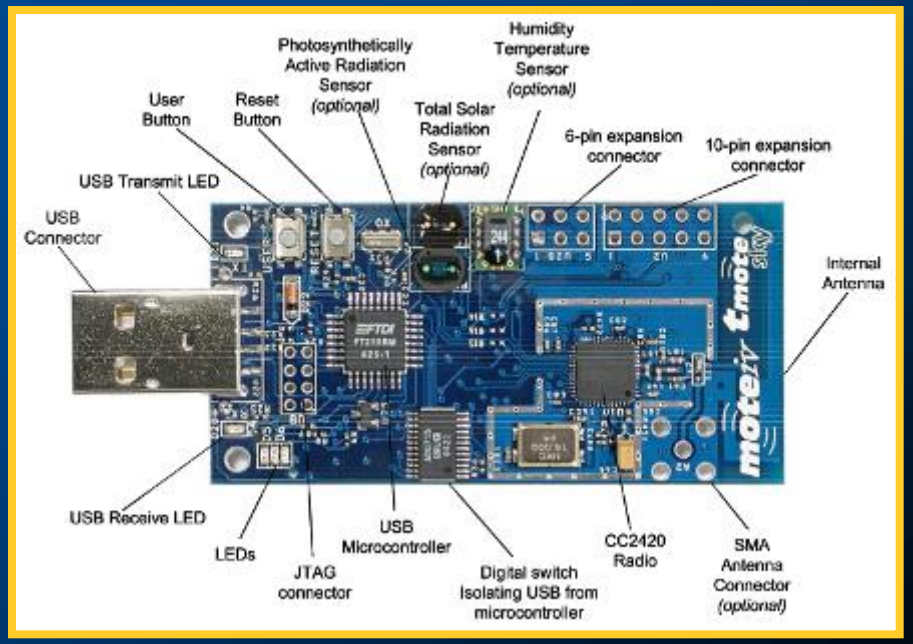
- By Computer Systems & Telematics group, Freie Universitat Berlin
- MCU:
  - TI MSP 430
- Comm:
  - Bluetooth, I^2C, CAN



Tmote Sky

- By Sentilla (formerly Moteiv), USA
- MCU:
  - TI MSP430
- Comm:
  - Chipcon CC2420 (IEEE 802.15.4)

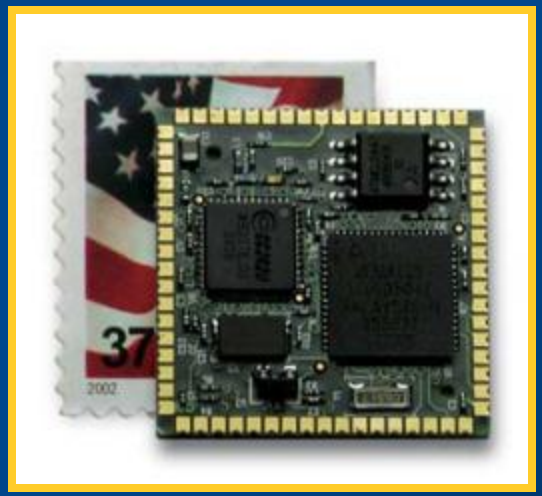




IRIS Motes

- By Crossbow, USA
- MCU: ATMega128L
- Comm: Atmel's RF230 (IEEE 802.15.4)
- 3x radio range compared to Tmote

- "Postage-stamp" form factor costs as low as $29 per unit (when purchased in large volumes)

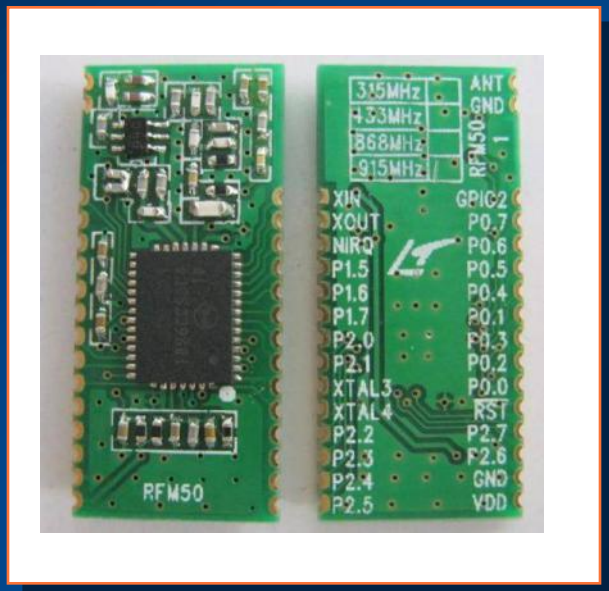


IMote2

- By Intel Research
- MCU: PXA271 XScale
- Comm: Chipcon CC2420 (IEEE802.15.4)

Other WSN-Capable Modules

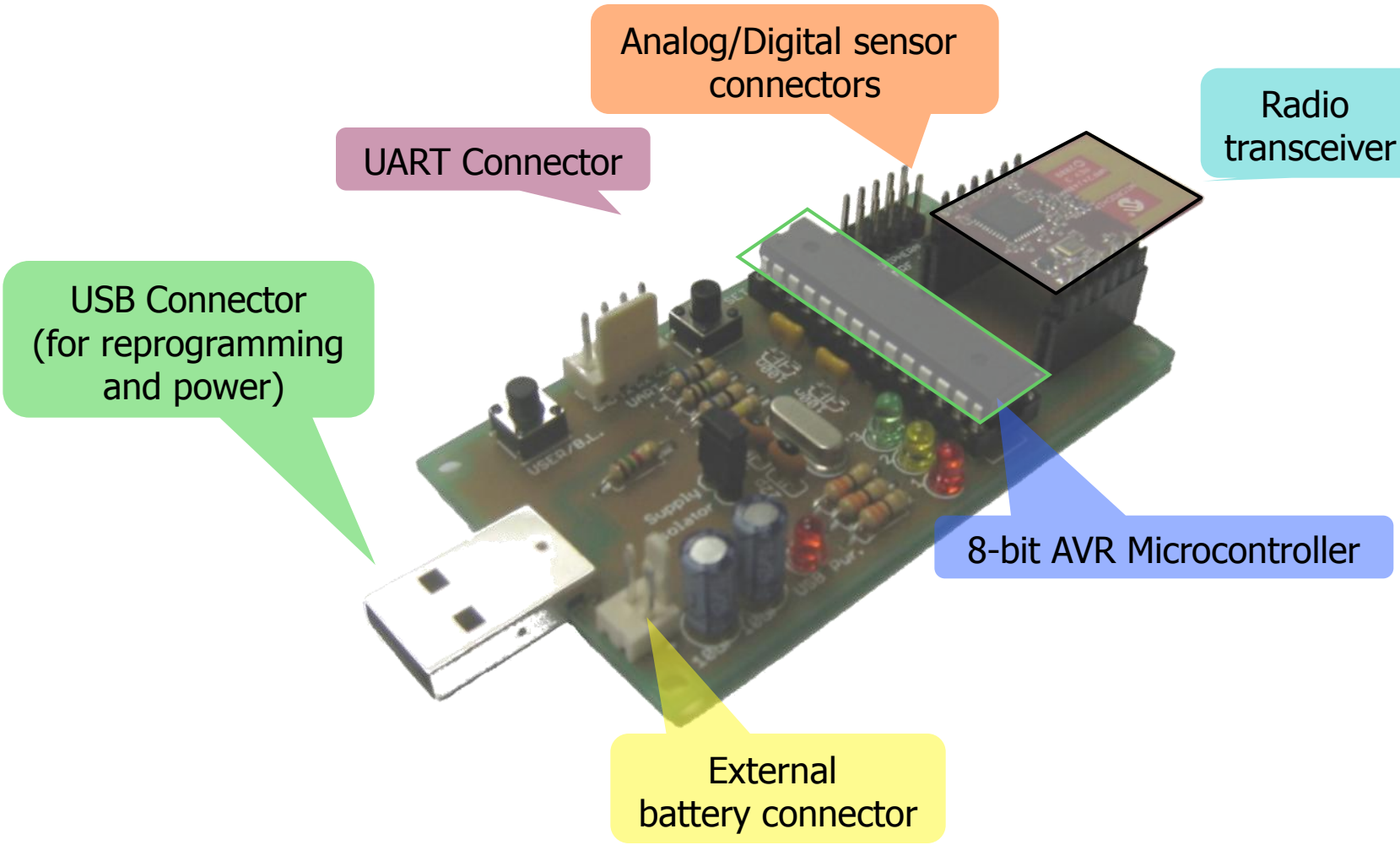
- Many low-cost, wireless SoC modules already available



HopeRF 433 MHz module
based on Silicon Labs's SoC
(~6 USD/module)

Synapse Wireless 2.4 GHz module
based on Atmel's SoC
SNAP OS / embedded Python
(~25 USD/module)

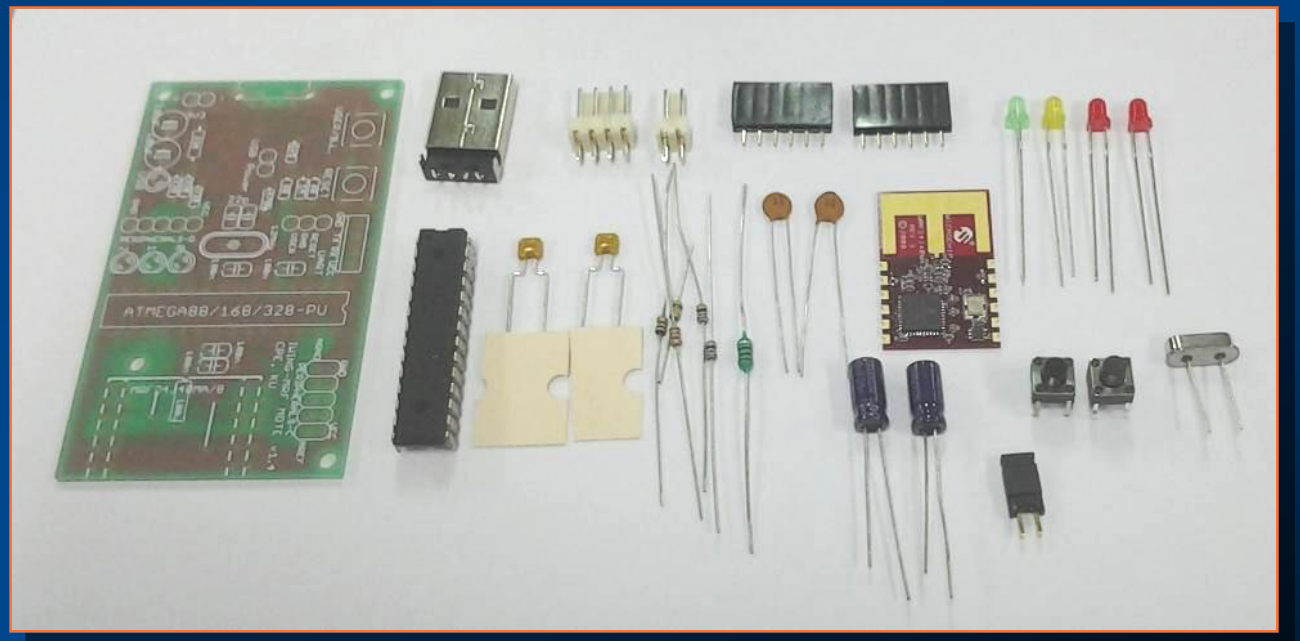
IWING-MRF Motes



Morakot Saravanee, Chaiporn Jaikaeo, 2010. Intelligent Wireless Network Group (IWING), KU

IWING-MRF Motes

- Built from off-the-shelf components
- Built-in USB boot loader
  - Reprogrammed via USB
- Easy to modify and extend hardware



IWING-MRF Mote

- Processor
  - 8-bit AVR microcontroller ATMega88/168/328, 12 MHz
  - 16KB flash, 2KB RAM
- RF transceiver
  - Microchip's MRF24J40A/B/C, 2.4GHz IEEE 802.15.4
  - SPI interface
- External connectors
  - 6 ADC connectors (can also be used as TWI)
  - 1 UART
- Power options
  - 3 – 3.6 VDC
  - USB or 2 AA batteries

IWING-JN Motes

- Built on JN5168 wireless microcontroller
- 32-bit RISC architecture
  - Operating at 32 MHz
  - 256 KB flash, 32 KB RAM
- IEEE 802.15.4 RF transceiver
- 4 ADC channels (10-bit)
- ~20 general-purpose digital I/O
- 2 UART interfaces
- Hardware access via C-language API

Outline

- Main components of a wireless sensor node
  - Processor, radio, sensors, batteries
- Energy supply and consumption
- *Operating systems and execution environments*
  - IWING's MoteLib
  - TinyOS
  - Contiki
- Example implementations

Operating System Challenges

- Usual operating system goals
  - Make access to device resources abstract (virtualization)
  - Protect resources from concurrent access
- Usual means
  - Protected operation modes of the CPU
  - Process with separate address spaces
- These are not available in microcontrollers
  - No separate protection modes, no MMU
  - Would make devices more expensive, more power-hungry

Possible OS Options

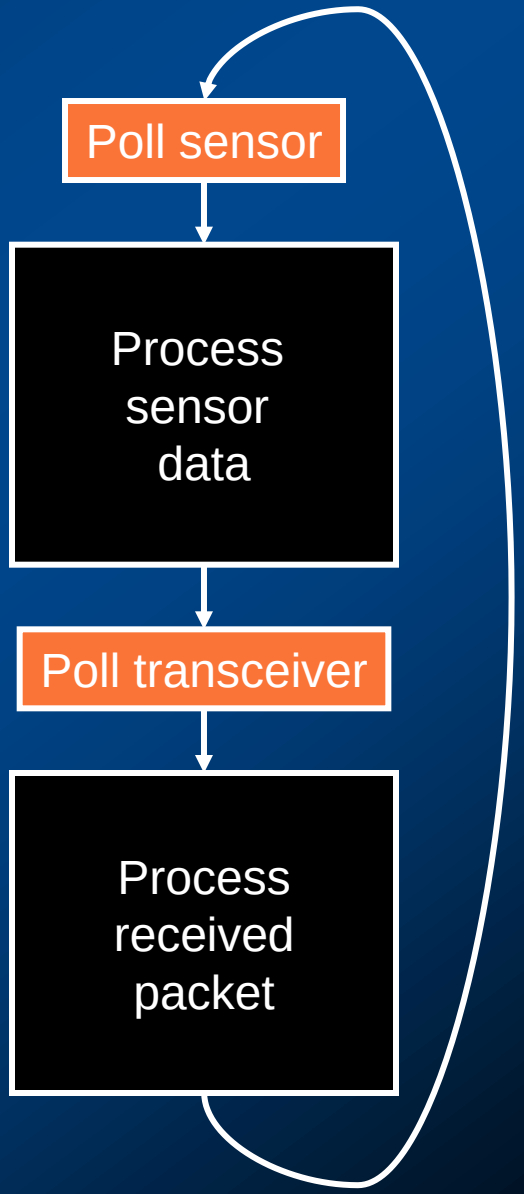
- Try to implement “as close to an operating system” on WSN nodes
  - Support for processes!
  - Possible, but relatively high overhead
- Stay away with operating system
  - There is only a single “application” running on a WSN node
  - No need to protect malicious software parts from each other
  - Direct hardware control by application might improve efficiency

Possible OS Options

- Currently popular approach
  - ⇨ No OS, just a simple run-time environment
  - ➢ Enough to abstract away hardware access details
  - ➢ Biggest impact: Unusual programming model

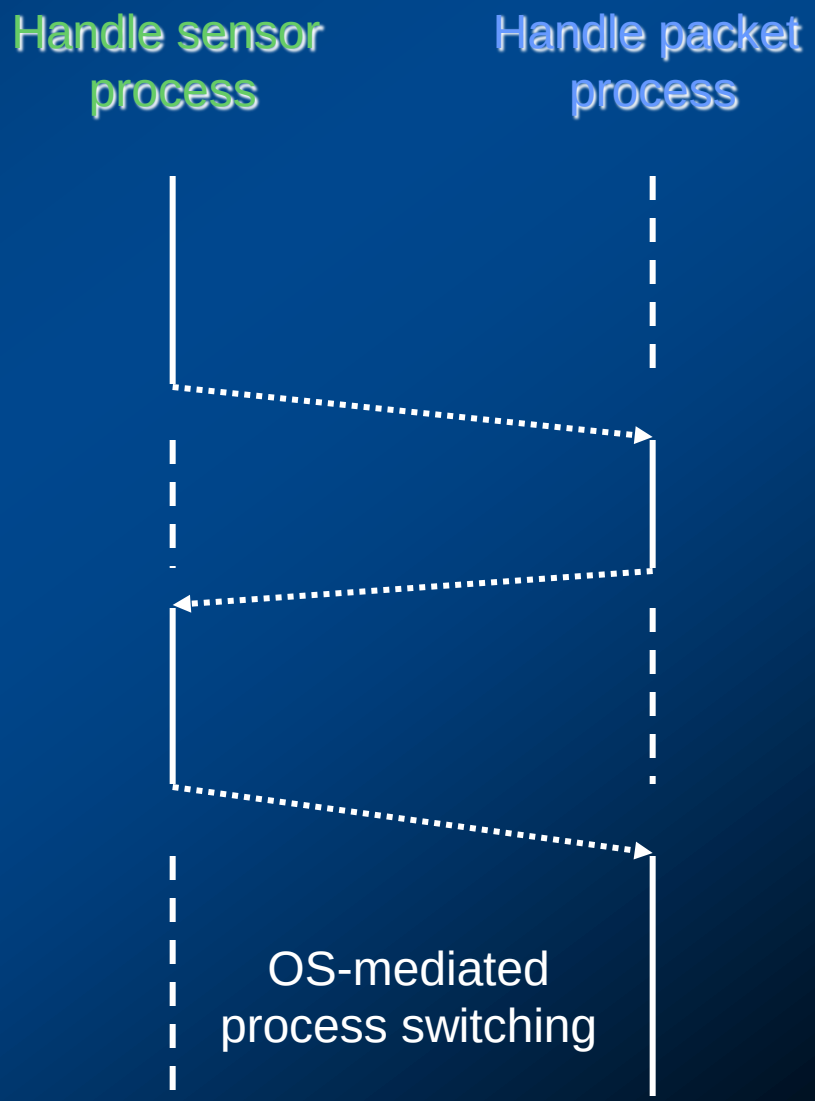
Concurrency Support

- Simplest option: No concurrency, sequential processing of tasks
  - Risk of missing data
  - Should support interrupts/asynchronous operations



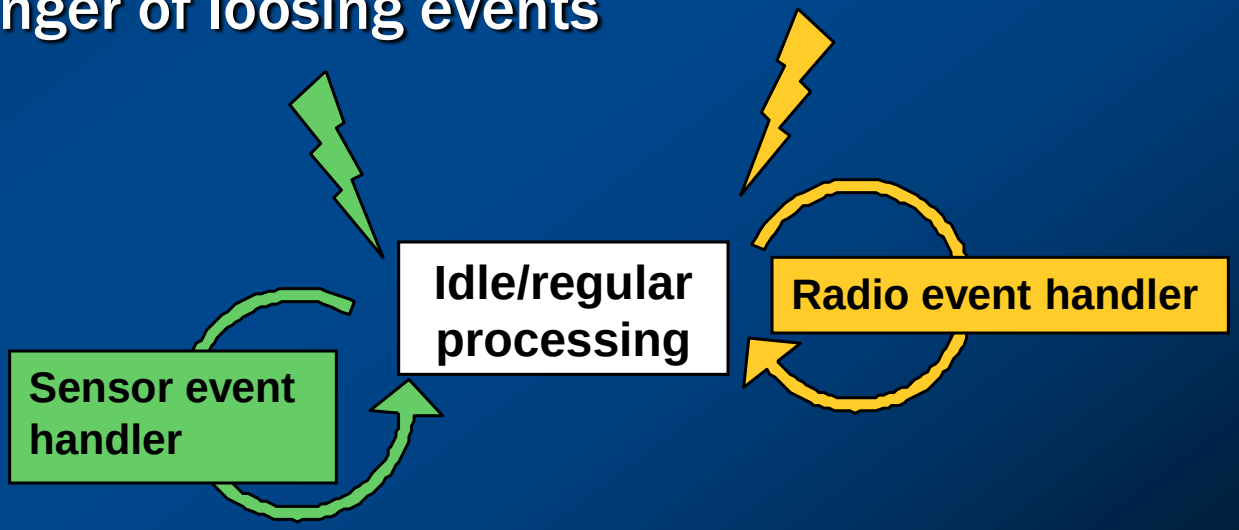
Processes/Threads

- Based on interrupts, context switching
- Difficulties
  - Too many context switches
    - Most tasks are short anyway
  - Each process required its own stack



Event-Based Concurrency

- ***Event-based programming model***
  - Perform regular processing or be idle
  - React to events when they happen immediately
  - Basically: interrupt handler
- Must not remain in interrupt handler too long
  - Danger of loosing events



Components Instead of Processes

- An abstraction to group functionality
- Typically fulfill only a single, well-defined function
  - E.g., individual functions of a networking protocol
- Main difference to processes:
  - Component does not have an execution
  - Components access same address space, no protection against each other

Event-based Protocol Stack

- Usual networking API: sockets
  - Issue: blocking calls to receive data
  - Not match to event-based OS
- 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 IWING's MoteLib and TinyOS

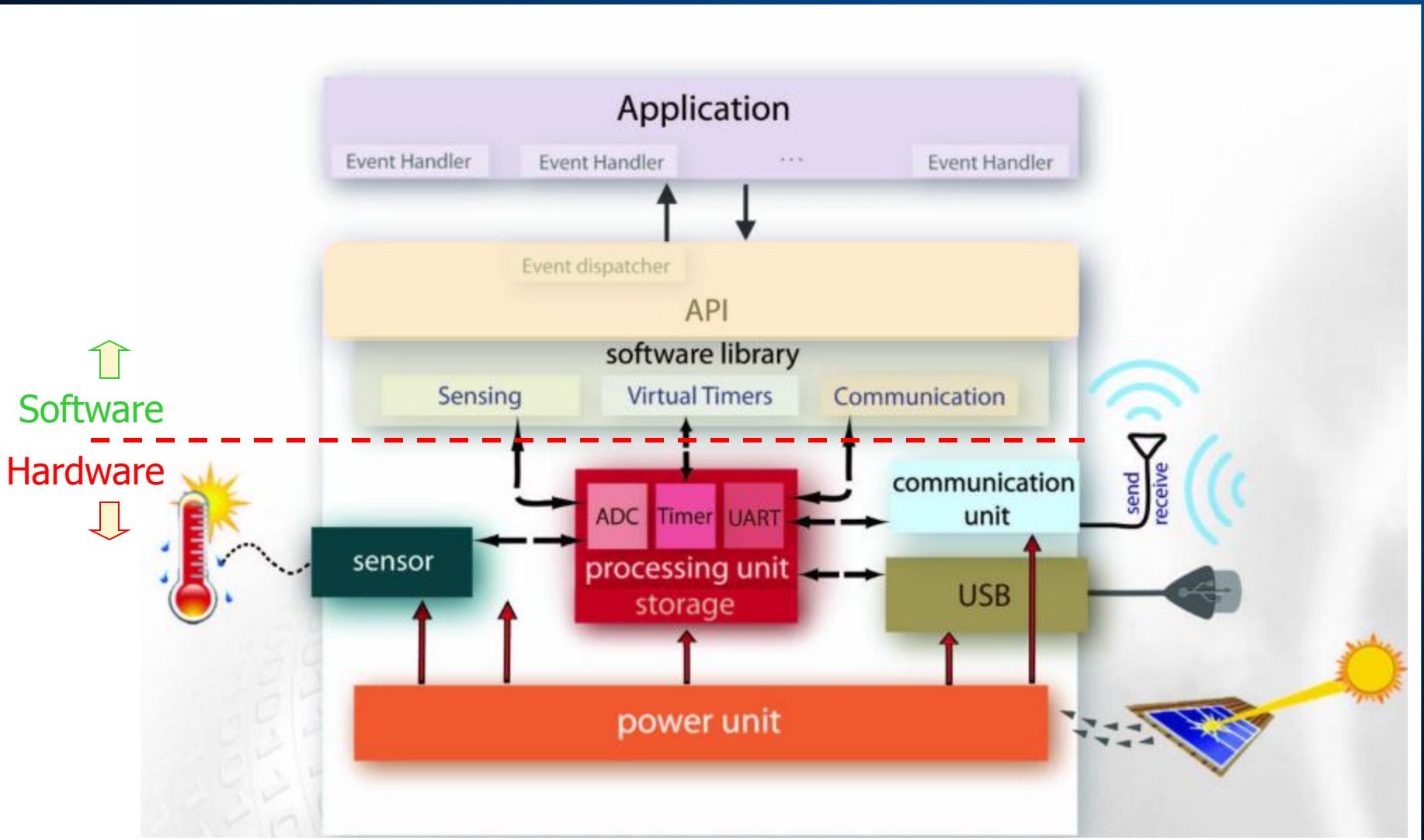
Outline

- Main components of a wireless sensor node
  - Processor, radio, sensors, batteries
- Energy supply and consumption
- Operating systems and execution environments
  - *IWING's MoteLib*
  - TinyOS
  - Contiki
- Example implementations

Case Study: IWING's MoteLib

- Developed by IWING (CPE, KU) along with IWING motes
- Provides hardware abstraction and virtualization in standard C interfaces
- Follows event-based programming model

MoteLib Architecture and API



Example: *Count and Send*

- Node#0 runs a counter and broadcasts its value
- Other nodes display received values on LEDs

Example: *Count and Send*

```
#include <motelib/system.h>
#include <motelib/timer.h>
#include <motelib/radio.h>
#include <motelib/led.h>

Timer   timer;
uint8_t counter;

void timerFired(Timer *t)
{
    counter++;
    radioRequestTx(BROADCAST_ADDR, 0, (char*)&counter, sizeof(counter), NULL);
}

void receive(Address source, MessageType type, void *message, uint8_t len)
{
    ledSetValue(((char*)message)[0]);
}

void boot()
{
    counter = 0;
    if (getAddress() == 0)
    {
        timerCreate(&timer);
        timerStart(&timer, TIMER_PERIODIC, 500, timerFired);
    }
    else
        radioSetRxHandler(receive);
}
```

called when timer expires

called when node receives a radio packet

called when node booted

Outline

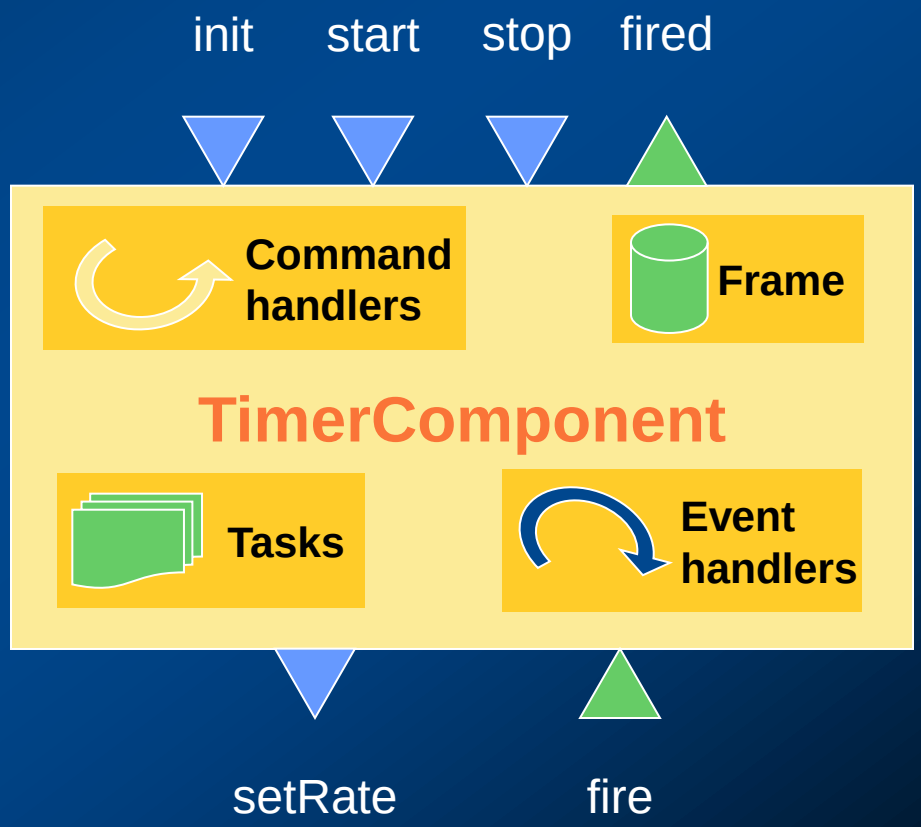
- Main components of a wireless sensor node
  - Processor, radio, sensors, batteries
- Energy supply and consumption
- Operating systems and execution environments
  - IWING's MoteLib
  - *TinyOS*
  - Contiki
- Example implementations

Case Study: TinyOS

- Developed by UC Berkeley as runtime environment for their motes
- nesC (network embedded system C) as adjunct programming language
- 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)
- Website http://www.tinyos.net

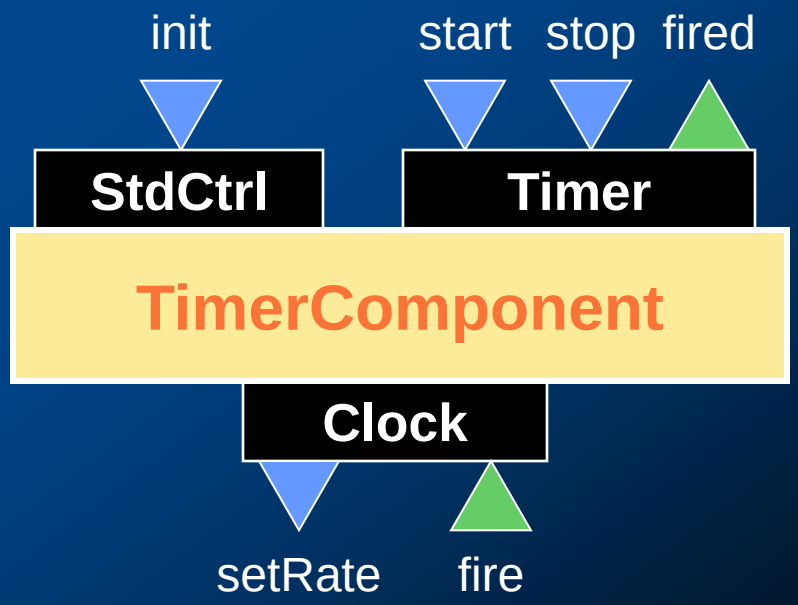
TinyOS Components

- Components
  - Frame – state information
  - Tasks – normal execution program
  - Command handlers
  - Event handlers
- Hierarchically arranged
  - Events are passed upward from hardware
  - Commands are passed downward

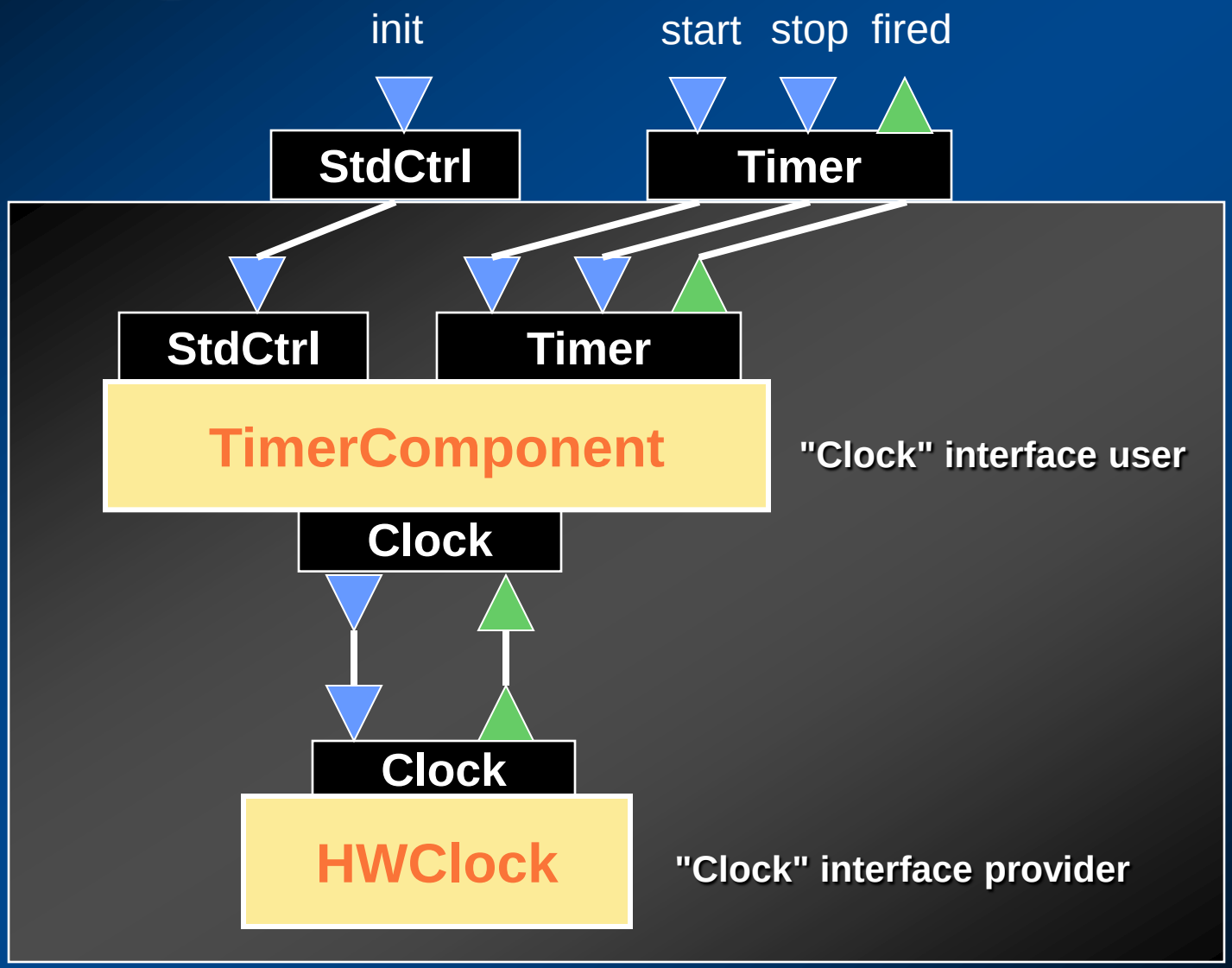


Interfaces

- Many commands/events can be grouped
- nesC structures corresponding commands/events into ***interface types***
- Example: Structure timer into three interfaces
  - StdCtrl
  - Timer
  - Clock
- The TimerComponent
  - **Provides:** StdCtrl, Timer
  - **Uses:** Clock



Forming New Components

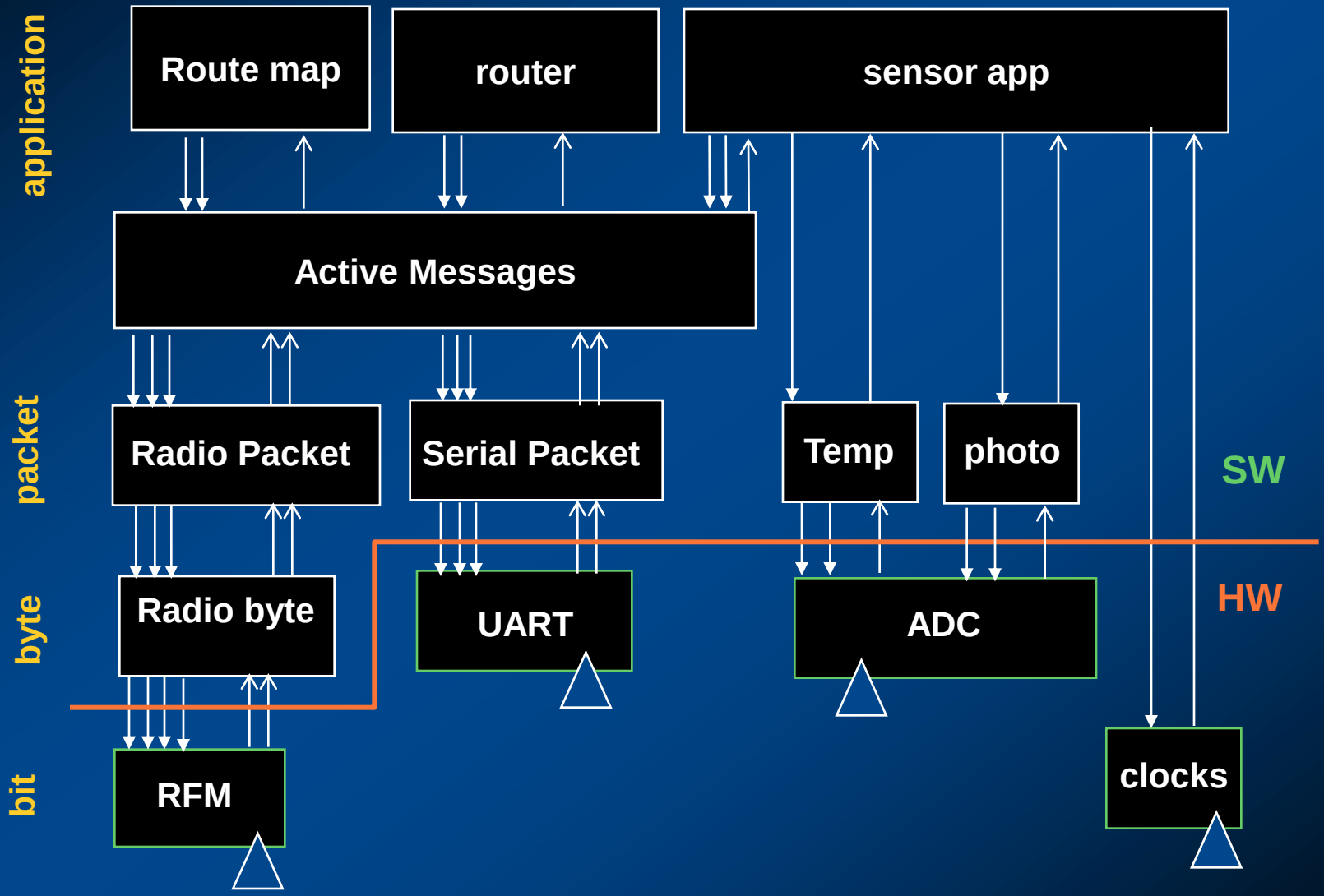


Sample nesC Code

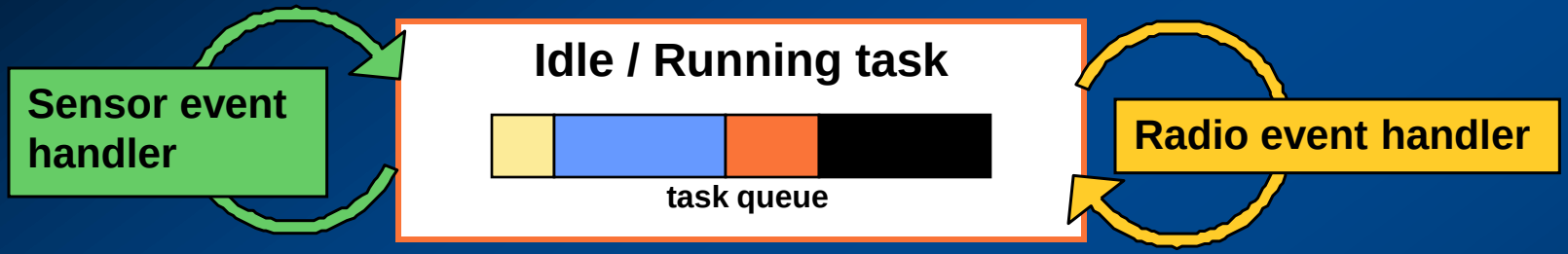
```
configuration CompleteTimer
{
  provides
  {
    interface StdCtrl;
    interface Timer;
  }

  implementation
  {
    components TimerComponent, HWClock;
    StdCtrl = TimerComponent.StdCtrl;
    Timer = TimerComponent.Timer;
    TimerComponent.Clock -> HWClock.Clock;
  }
}
```

Sample App Configuration

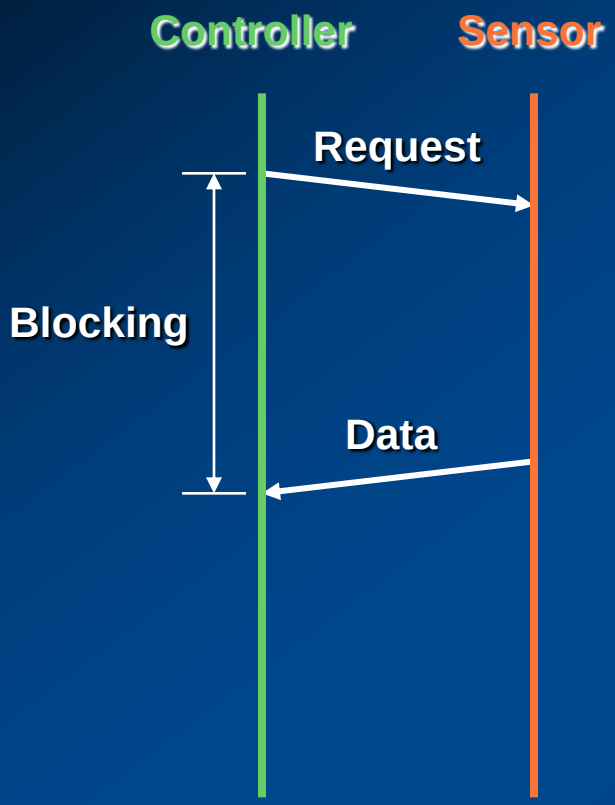


Handlers versus Tasks

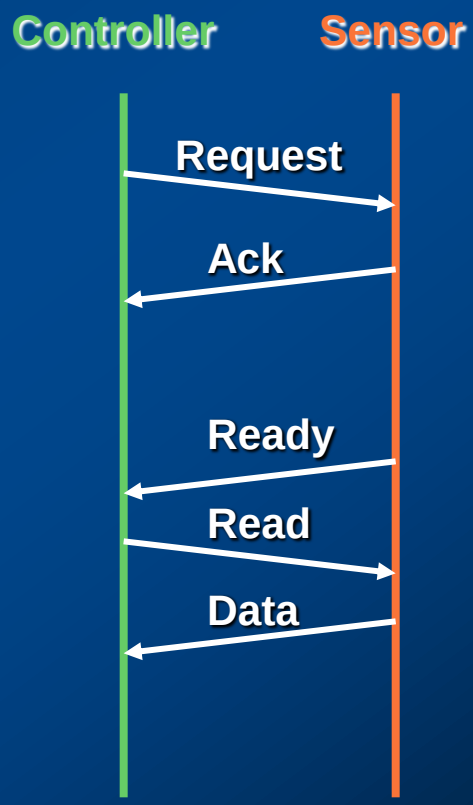


- Command/event handlers must run to completion
  - Must not wait an indeterminate amount of time
  - Only a ***request*** to perform some action
- Tasks can perform arbitrary, long computation
  - Can be interrupted by handlers
  - Also have to be run to completion
    - Preemptive multitasking not implemented

Split-Phase Operations



Synchronous Operation



Asynchronous Operation

Outline

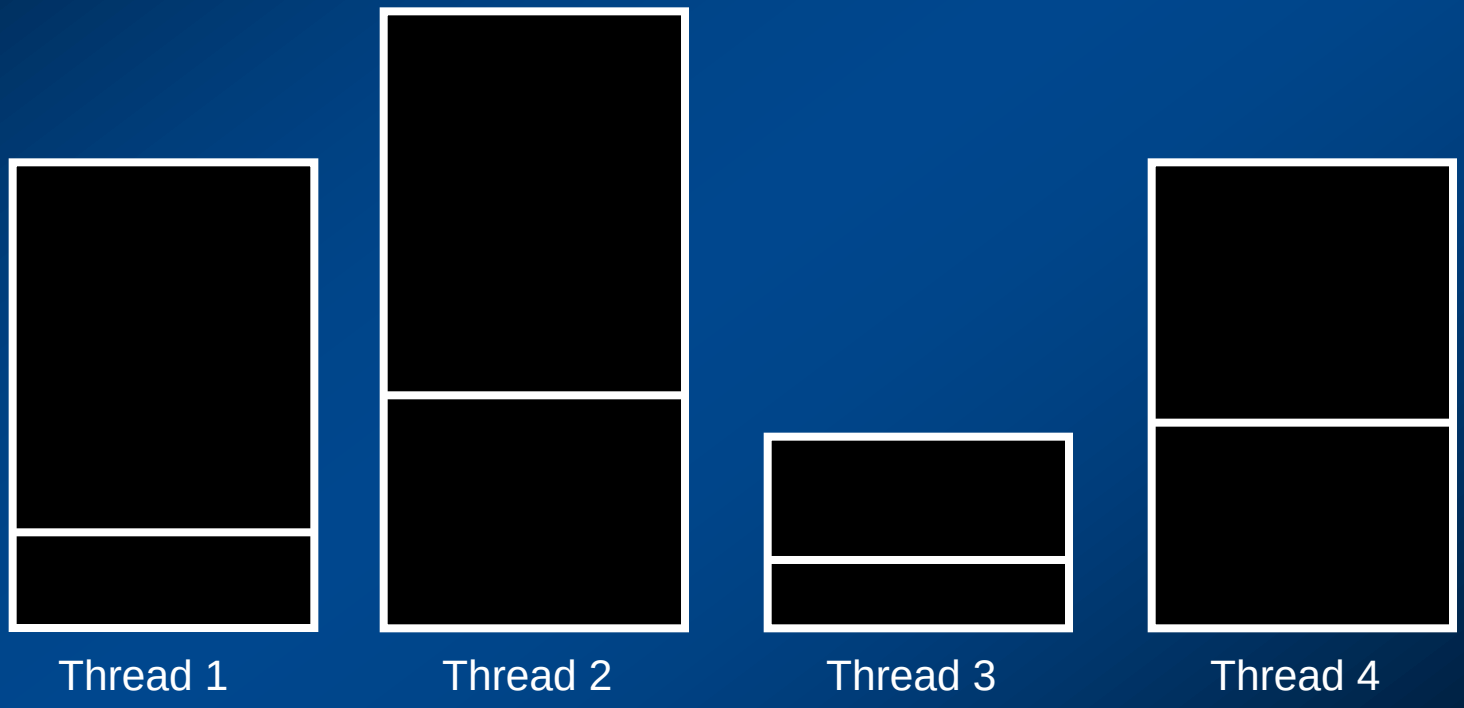
- Main components of a wireless sensor node
  - Processor, radio, sensors, batteries
- Energy supply and consumption
- Operating systems and execution environments
  - IWING's MoteLib
  - TinyOS
  - *Contiki*
- Example implementations

Case Study: Contiki

- Multitasking OS developed by Swedish Institute of Computer Science (SICS)
- The kernel is event driven
- Processes are protothreads
  - Very light weight threads
  - Provide a linear, thread-like programming model
- Comes with various communication stacks: uIP, uIPv6, Rime
- Website http://www.contiki-os.org/

Problem with Multithreading

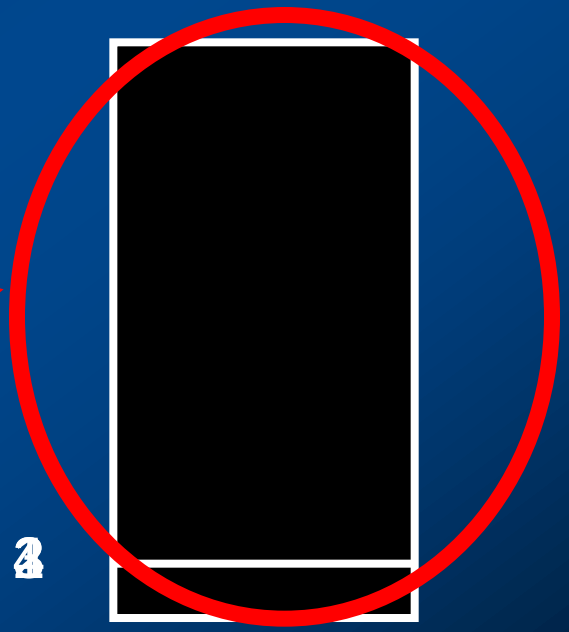
- Four threads, each with its own stack



Events Require One Stack

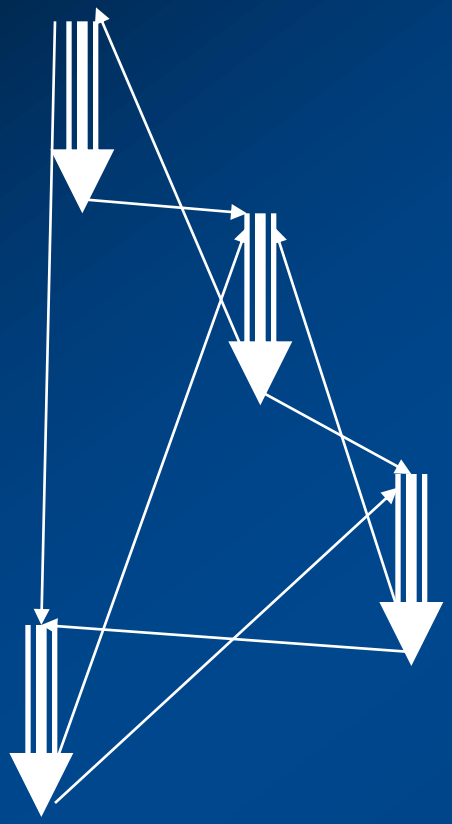
- Four event handlers, one stack

Stack is reused for every event handler

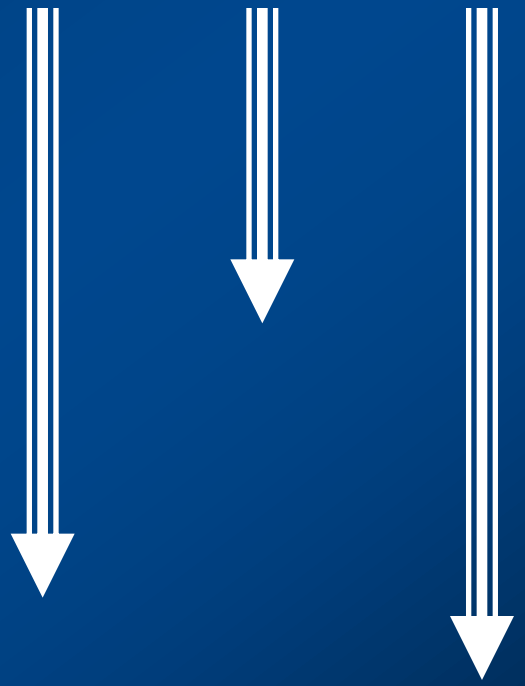


Problem with Event-based Model

Events: unstructured code flow



Threads: sequential code flow



Very much like programming with GOTOs

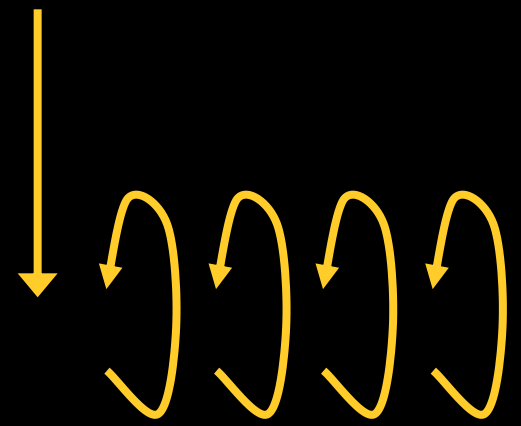
Protothreads

- Protothreads require only one stack
- E.g, four protothreads, each with its own stack

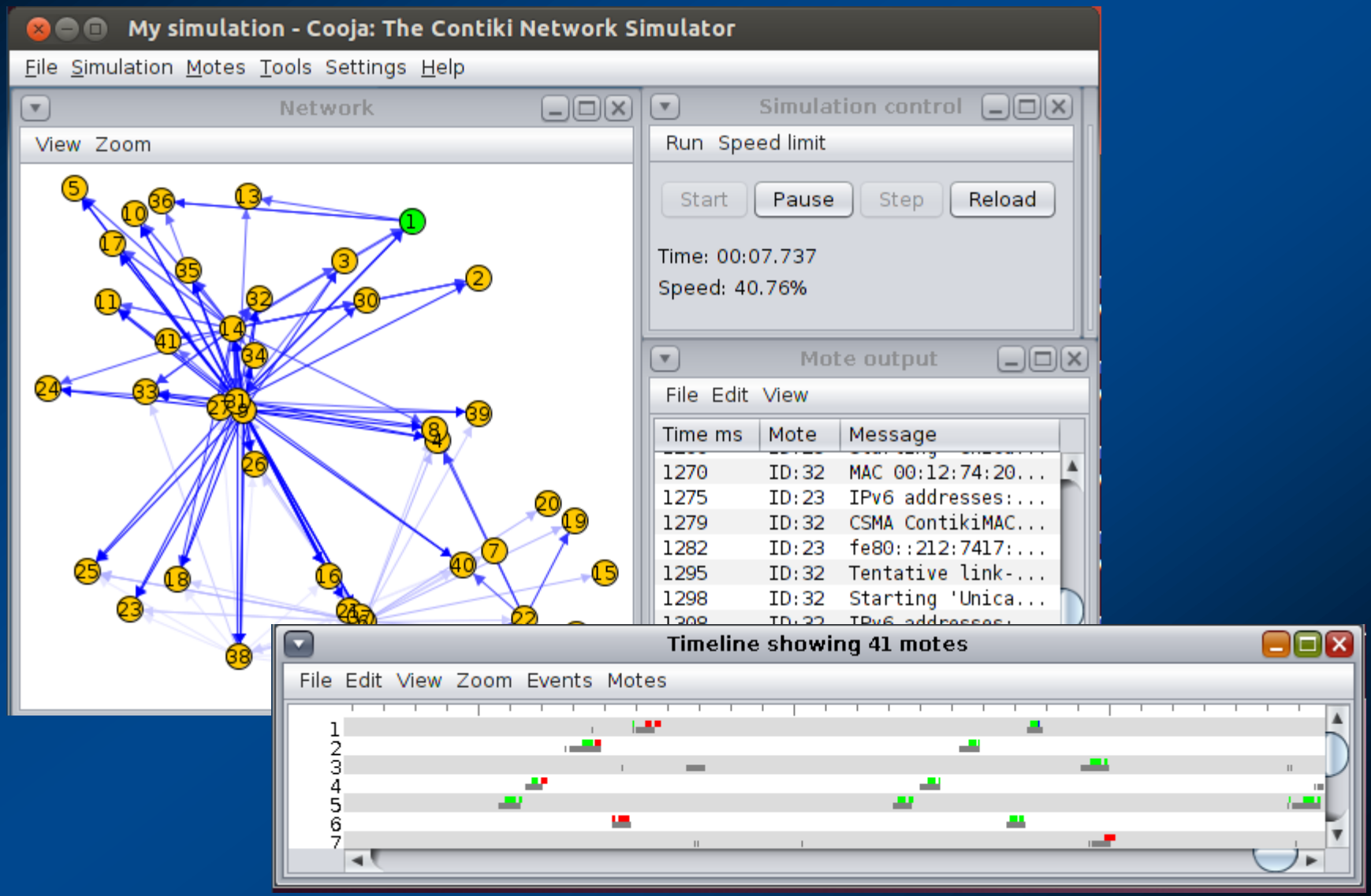
Contiki Processes

- Contiki processes are protothreads

```
PROCESS_THREAD(hello_world_process, ev, data)
{
  PROCESS_BEGIN();
  printf("Hello, world!\n");
  while(1) {
    PROCESS_WAIT_EVENT();
  }
  PROCESS_END();
}
```



Contiki's Cooja Simulator



Summary

- The need to build cheap, low-energy, (small) devices has various consequences
  - Much simpler radio frontends and controllers
  - Energy supply and scavenging are a premium resource
  - Power management is crucial
- Unique programming challenges of embedded systems
  - Concurrency without support, protection
  - De facto standard:
    - Event-based programming model: TinyOS
    - Multithreaded programming model: Contiki