Introduction to Contiki

1st Workshop on Appl. of WSN for Env. Monitoring in Developing Countries

Trieste (Italy), 02.03.2011

About the Speakers

- **Carlo Alberto Boano**

  - Ph.D. student at the University of Lübeck, Germany
  - Research focus: wireless and body sensor networks, radio interference mitigation, link quality estimation
  - Contact:

    University of Lübeck – Institute für Technische Informatik
    Ratzeburger Allee 160, Haus 33, Raum 111
    23538, Lübeck, Germany
    cboano@iti.uni-luebeck.de
    http://www.carloalbertoboano.com

About the Speakers

➢ Enrique Muntane Calvo

- R&D Engineer at Zolertia in Barcelona, Spain
- Research Focus: ultra-small, ultra-low-power self-sustainable deploy&forget WSN and distributed detection and estimation algorithms via signal processing or machine learning
- Contact:

Zolertia, Advancare, SL
Av. Parc Tecnologic del Valles, 3, PTV
08290 Cerdanyola del Valles
Barcelona, Spain
ecalvo@zolertia.com

Structure of the Lectures

Weekday	10:30 – 11:30	11:30 – 12:30	14:00 – 16:00	16:30 – 18:30
Wednesday, 02.03.2011	Introduction to Contiki	Contiki: Basic Programming	WSN Lab Contiki: Installation and Overview	WSN Lab Contiki: Basic Examples
Thursday, 03.03.2011	Wireless Sensornet Communications	Contiki: Radio Communication	WSN Lab Contiki: Radio Communication	WSN Lab Contiki: Radio Communication
Friday, 04.03.2011	Crash Course on Electronics (for Wireless Sensor Networks)	Crash Course on Electronics (for Wireless Sensor Networks)	WSN Lab Project II	WSN Lab Project II

Wireless Sensor Networks

Wireless Sensor Networks

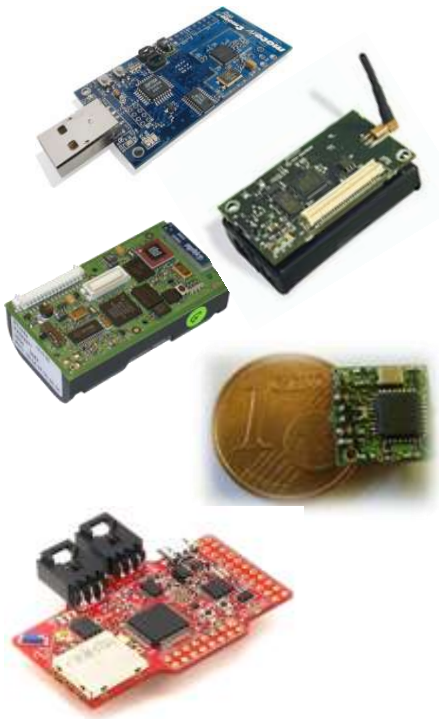
Wireless Sensor Networks

Sensornet Platform	RAM	ROM	External Memory	CPU	Radio (Data Rate)
Sentilla Tmote Sky	10 kB	48 kB	ST M25P80 1 MB	MSP430 (i) 8 MHz	CC2420 250 kbps
CrossBow MicaZ	4 kB	128 kB	512 kB	Atmega 128L 8 MHz @ 8 MIPS	CC2420 250 kbps
Tyndall 10mm	256 B	4 kB	AT25640A 4 kB	nRF9e5 20 MHz	nRF905 100 kbps
Zolertia Z1	8 kB	92 kB	M25P16 16 Mb	MSP430 (ii) 16 MHz	CC2420 250 kbps
ETH BTnode	64 kB	128 kB	60x3 kB SRAM	Atmega 128L 8 MHz @ 8 MIPS	CC1000 76.8 kbps

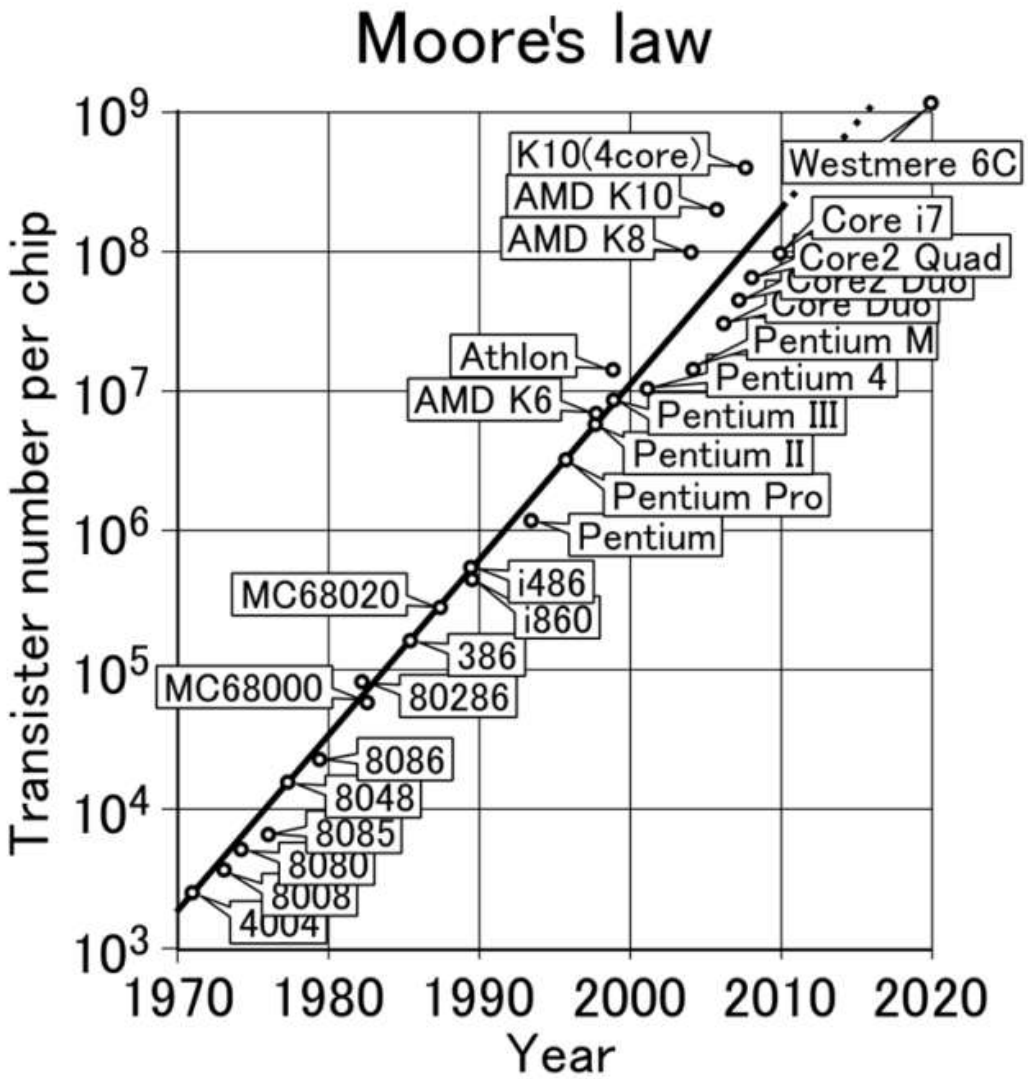
http://en.wikipedia.org/wiki/List_of_wireless_sensor_nodes

Wireless Sensor Networks

Sensornet Platform	RAM	ROM	External Memory	CPU	Radio (Data Rate)
Sentilla Tmote Sky	10 kB	48 kB	ST M25P80 1 MB	MSP430 (i) 8 MHz	CC2420 250 kbps
CrossBow MicaZ	4 kB	128 kB	512 kB	Atmega 128L 8 MHz @ 8 MIPS	CC2420 250 kbps
Tyndall 10mm	256 B	4 kB	AT25640A 4 kB	nRF9e5 20 MHz	nRF905 100 kbps
Zolertia Z1	8 kB	92 kB	M25P16 16 Mb	MSP430 (ii) 16 MHz	CC2420 250 kbps
ETH BTnode	64 kB	128 kB	60x3 kB SRAM	Atmega 128L 8 MHz @ 8 MIPS	CC1000 76.8 kbps



- **Wireless Sensor Network Platforms are <u>severely</u> constrained!**

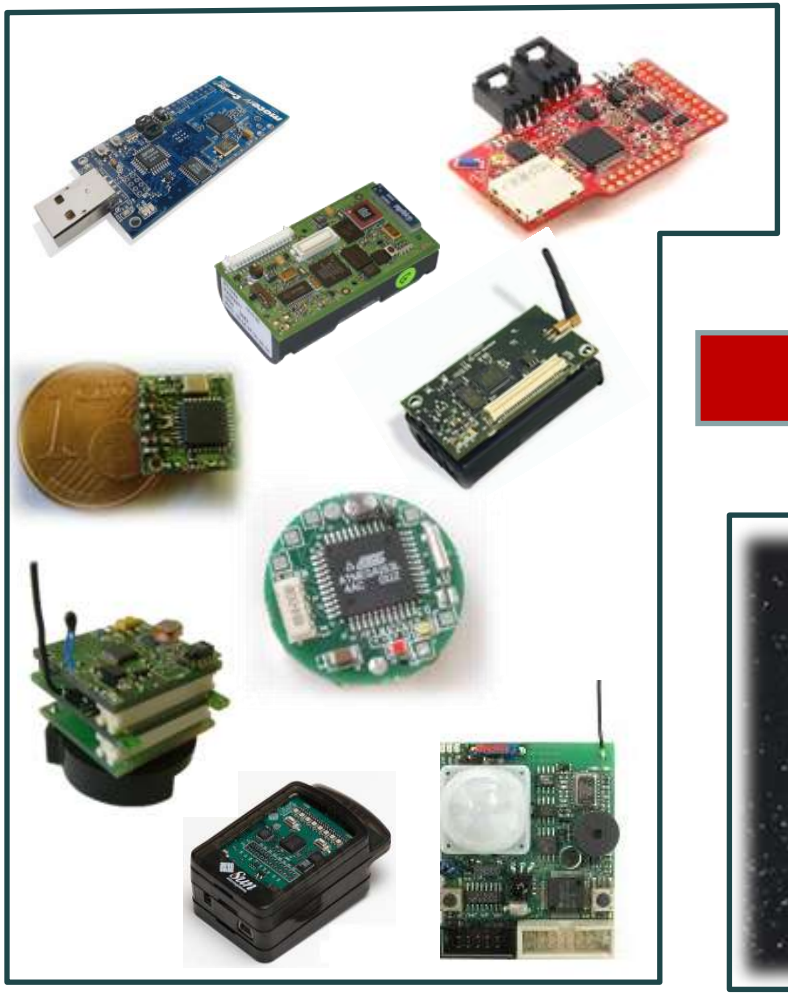

Moore's law
Transister number per chip
Year
10^9
10^8
10^7
10^6
10^5
10^4
10^3
1970
1980
1990
2000
2010
2020
K10(4core)
AMD K10
AMD K8
Westmere 6C
Core i7
Core2 Quad
Core2 Duo
Core Duo
Pentium M
Pentium 4
Athlon
AMD K6
Pentium III
Pentium II
Pentium Pro
Pentium
i486
i860
MC68020
386
MC68000
80286
8086
8048
8085
8080
8008
4004

Wireless Sensor Networks

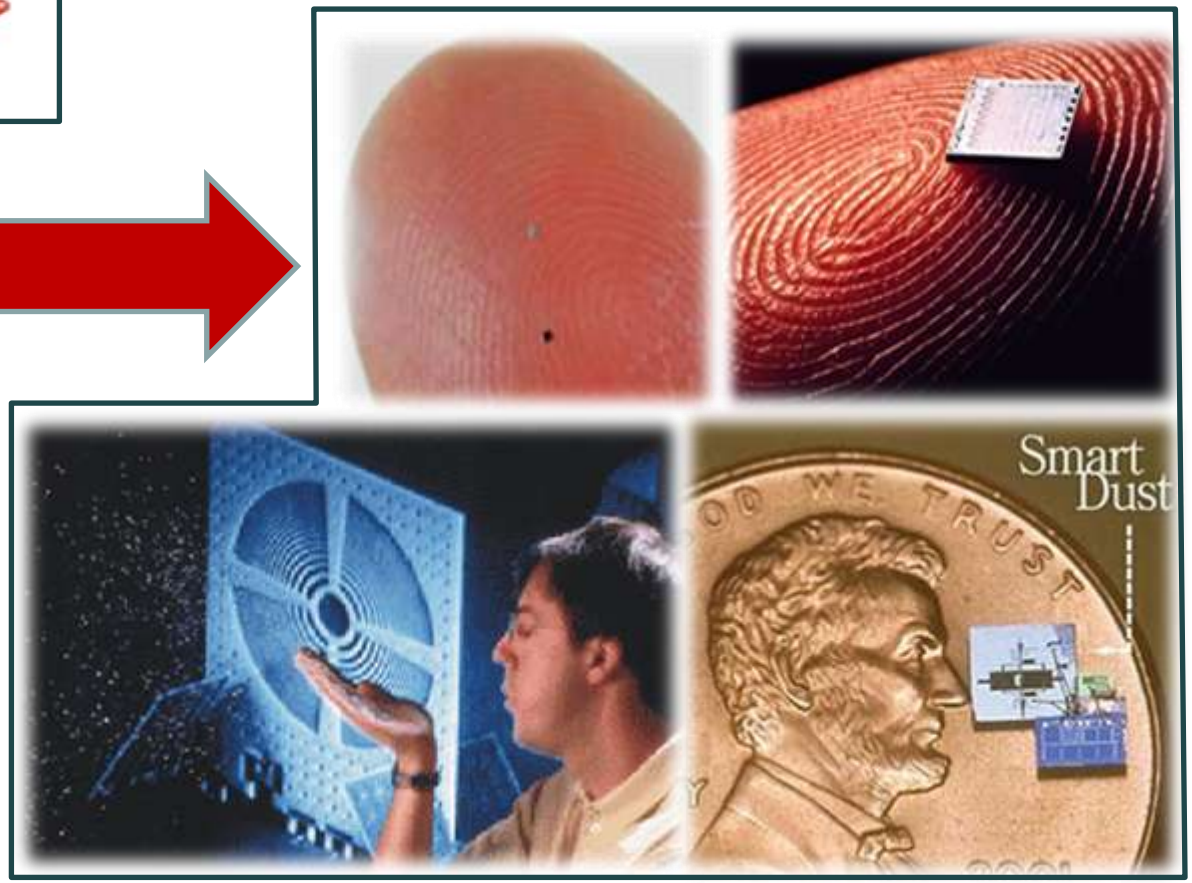
- **Something is wrong with Wireless Sensor Networks…**
  - Interesting keynote at EWSN'11 by Mani Srivastava

  - The size of the sensor nodes did not shrink in the last decade
  - The community tried instead to build more capable nodes

Wireless Sensor Networks

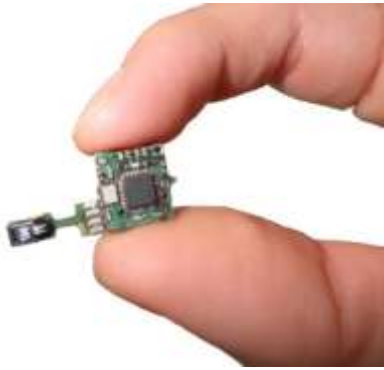


- But this will be the trend…



Wireless Sensor Networks

- **Very Limited Physical Resources**
  - Size is shrinking
  - Capabilities and resources are minimal
  - Low-power operations
    - It takes several years to double the battery energy density
    - Energy harvesting is not enough to power motes



→ <u>The OS should address extremely low-power operation, and provide flexibility in power-management and duty-cycle strategies</u>

P. Levis, et al. TinyOS: An operating system for wireless sensor networks. In *Ambient Intelligence. 2004.*

Wireless Sensor Networks

- **Event-centric Concurrent Applications**
  - Sampling the environment through sensors
  - Manipulation of the environment through actuators
  - Local data processing (signal processing, routing, aggregation, …)

→ The OS should have an approach to concurrency management that respects resource and timing constraints

P. Levis, et al. TinyOS: An operating system for wireless sensor networks. In *Ambient Intelligence. 2004.*

Wireless Sensor Networks

- **Hardware Evolves Continuously…**
  - Moore's law
  - Improvements in nanotechnologies
  - Different applications might have different requirements

→ <u>The OS should permit code reuse and simplify innovation</u>

P. Levis, et al. TinyOS: An operating system for wireless sensor networks. In *Ambient Intelligence. 2004.*

Wireless Sensor Networks

- **Very Limited Physical Resources**
  - The OS should address extremely low-power operation, and provide flexibility in power-management and duty-cycle strategies
- **Event-centric Concurrent Applications**
  - The OS should have an approach to concurrency management that respects resource and timing constraints
- **Hardware Evolves Continuously**
  - The OS should permit code reuse and simplify innovation

→ <u>Need for specific Operat. Systems for low-power embedded devices</u>

Wireless Sensor Networks

- **Need for Specific OS for Low-Power Embedded Devices**
  - Many Operating Systems for wireless sensor networks

- TinyOS	- SenOS	- Bertha
- Contiki	- Mantis OS	- EYES/PEEROS
- Pixie	- Matè	- MagnetOS
- SensorWave	- CORMOS	- NanoRK

  - <u>We will focus on **Contiki**</u>

http://dos.iitm.ac.in/publications/LabPapers/adiWirelessOS.pdf

The Contiki Operating System

- **Background**
  - History and Overview of Contiki
  - Some of Contiki's Main Features
  - Event-driven vs. Multithreaded Systems
  - The Contiki Kernel
  - Contiki's Protothreads
  - Contiki's Processes

The Contiki Operating System

- **Background**
  - History and Overview of Contiki
  - Some of Contiki's Main Features
  - Event-driven vs. Multithreaded Systems
  - The Contiki Kernel
  - Contiki's Protothreads
  - Contiki's Processes

Россия
Russia
Mongolia
中国
China
대한민국
S Korea
日本
Japan
North Pacific Ocean
Canada
United States
North Atlantic Ocean
México
Iceland
Norge
Norway
United Kingdom
Deutschland
Germany
France
España
Spain
Italia
Italy
Algeria
Mauritania
Mali
Niger
Nigeria
Thailand
Indonesia
Papua New Guinea
Venezuela
Colombia
Brasil
Brazil
Bolivia
Chile
Argentina
South Atlantic Ocean
Indian Ocean
Australia
New Zealand
South Pacific Ocean
A
B

The Contiki Operating System

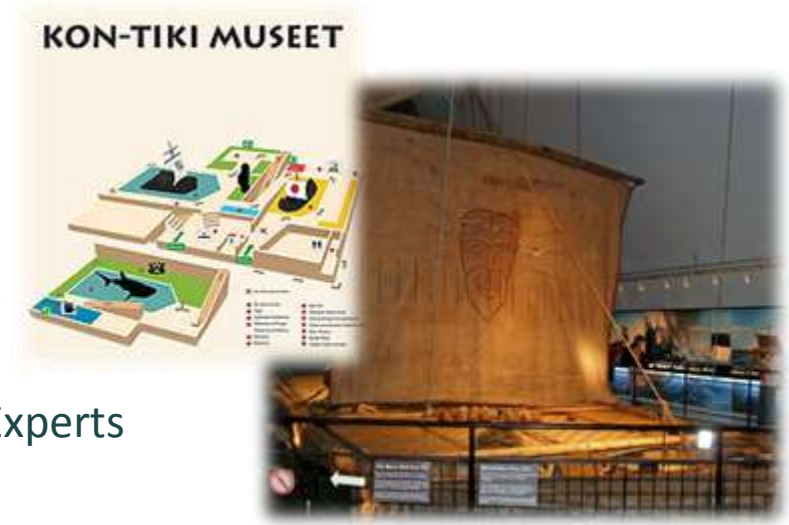
- **The name says it all**
  - Long debate among anthropologists: “could people from South America have settled in Polynesia in pre-Columbian times?”
  - 1947: Thor Heyerdahl organized the Kon-Tiki expedition to demonstrate that it was possible to sail the Ocean despite having only minimal resources

The “Kon-Tiki” name comes from the old name for the Inca sun god

The Contiki Operating System

- **The name says it all**
  - 6 people crew
    - **Thor Heyerdahl** (1914 – 2002)
    - Erik Hesselberg (1914 – 1972)
    - Bengt Danielsson (1921 – 1997)
    - Knut Haugland (1917 – 2009)
    - Torstein Raaby (1920 – 1964)
    - Herman Watzinger (1910 – 1986)

Radio Experts



Kon-Tiki Museum, Oslo, Norway

  - Departure from Callao, Peru on 28.04.1947
  - Arrival to Raroia, Tuamotu on 07.08.1947 (after 101 days)
  - They all survived

The Contiki Operating System

- **The name says it all**
  - Contiki derives his name from the “Kon-Tiki” rudimental raft that sailed the Pacific Ocean in 1947

- **Why such a name for an Operating System?**
  - Navigator, Explorer, Konqueror, …
  - Contiki is a fully-functional operating system with internet connectivity for limited hardware resources
    - Typical configuration: 2kB RAM and 40kB ROM

The Contiki Operating System

- **Original Development Motivated by the Design of the Smallest Web Browser**
  - Year 2000: the Arena Project (LTU, Telia, Ericsson, SICS)
    - Hockey players with BT sensors, camera on helmet
    - Spectators with direct access to pulse/breath readings
    - M16C CPU with 20 kB RAM, 100 kB ROM, and lwIP
    - Luleå Hockey - Brynäs Hockey finished 1-4

  - Year 2001: Adam Dunkels designed uIP, the world's smallest open source, RFC-compliant TCP/IP stack
    - Typical configuration: 5 kB code, 2 kB RAM
    - Minimum configuration: 3 kB code, 128 B RAM

The Contiki Operating System

➢ Brief History of Contiki

- March 2003: Contiki version 1.0 was released
  - Internet-enabled Commodore 64 (1982, 64kB RAM, 16 colors)
- End of 2003: First port to a wireless sensor network platform
  - Scatterweb ESB (Embedded Sensor Board)
- November 2010: Contiki 2.5 Release Candidate
  - Many sensor networks platforms supported
  - Scatterweb ESB, MSB430, Sentilla Tmote Sky
  - TelosB, Atmel Raven, Sensinode, MicaZ
  - Sentilla Tmote Sky and JCreate, Zolertia Z1, …

The Contiki Operating System

➢ The Contiki Team

- Main author and project leader: Adam Dunkels
  - Core group: Durvy, Eriksson, Finne, Schmidt, Tsiftes, Österlind
  - Several developers from SICS, SAP, Cisco, NewAE, TU Munich, Atmel

➢ Official Website

- http://www.sics.se/contiki/
  - Examples
  - Tutorials
  - FAQs

The Contiki Operating System

- **Background**
  - History and Overview of Contiki
  - Some of Contiki's Main Features
  - Event-driven vs. Multithreaded Systems
  - The Contiki Kernel
  - Contiki's Protothreads
  - Contiki's Processes

Some of Contiki's Main Features

Open-source C code

Multitasking with an event-driven kernel

Simulators (COOJA, MSPsim)

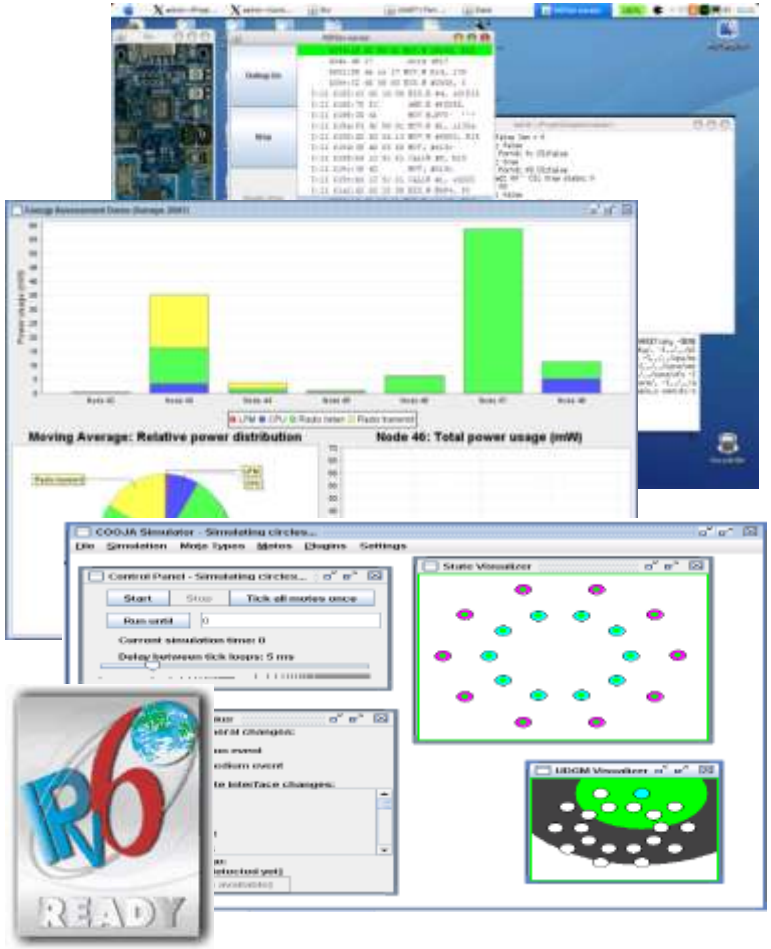
Software-based energy consumption

IPv4 and IPv6 connectivity

Support for several platforms/MCUs (MSP430, Atmel, …)

Dynamic loading

File System (Coffee)



Some of Contiki's Main Features

- **Internet connectivity**
  - uIP: World's smallest, fully compliant TCP/IP stack
    - Compliant to host-to-host communication requirements (RFC1122)
    - Supports unicast, broadcast, multicast packets, TCP, UDP, ICMP
    - No support for Path MTU discovery or ICMP redirect messages
    - No soft error reporting mechanism
    - No dynamically configurable type-of-service bits for TCP conn.
    - No explicit dynamic memory allocation
    - Single buffer for holding packets/fixed table for holding connections
  - IPv6 Ready Phase 1 certified
  - IPSO Alliance for the use of IP for Smart Objects

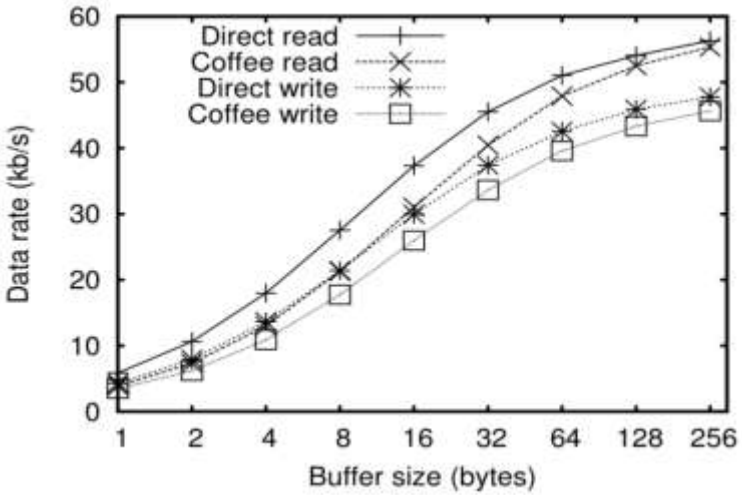
Some of Contiki's Main Features

- **Software-based energy estimation**
  - Energy consumption critical in sensor networks
    - Main design constraint
    - To provide a long sensor network lifetime, it is crucial to control and reduce the power consumption of each sensor node
  - Contiki provides a software-based power profiling tool
    - Keeps track of the energy expenditure of components
    - No need of additional hardware
    - Good accuracy, low overhead

Some of Contiki's Main Features

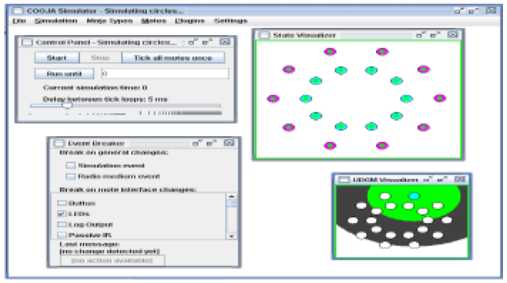
- **Coffee: a Flash-based file system**
  - Programming interface for building efficient and portable storage abstractions for flash-based sensor devices

  - High-speed
    - 92% of raw flash throughput
  - Lightweight
    - 5 kb ROM
    - Constant RAM footprint < 0.5 kb
    - Only stores EoF pointers of open files and a small directory cache

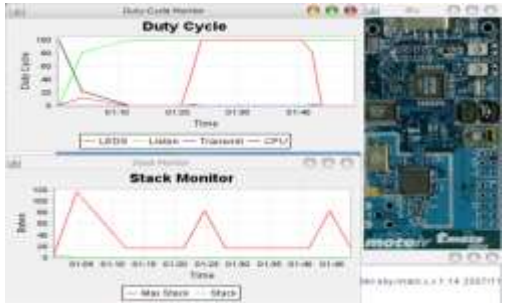


Some of Contiki's Main Features

- **Simulators**
  - To ease software development and debugging
  - COOJA
    - Extensible cross-level Java-based simulator
    - Contiki/Java/Emulated MSP430 nodes



  - MSPsim
    - Java-based instruction level simulator for the MSP430
    - Cycle counting, power profiling, debugging
    - Integrated into COOJA or standalone



The Contiki Operating System

- **Background**
  - History and Overview of Contiki
  - Some of Contiki's Main Features
  - Event-driven vs. Multithreaded Systems
  - The Contiki Kernel
  - Contiki's Protothreads
  - Contiki's Processes

Event-driven vs. Multithreaded Systems

- **Desired Properties for a Sensornet Operating System**
  - It should address extremely low-power operation, and provide flexibility in power-management and duty-cycle strategies
  - It should have an approach to concurrency management that respects resource and timing constraints
  - Like the applications, the OS should be event-centric
  - It should permit code reuse and simplify innovation

Event-driven vs. Multithreaded Systems

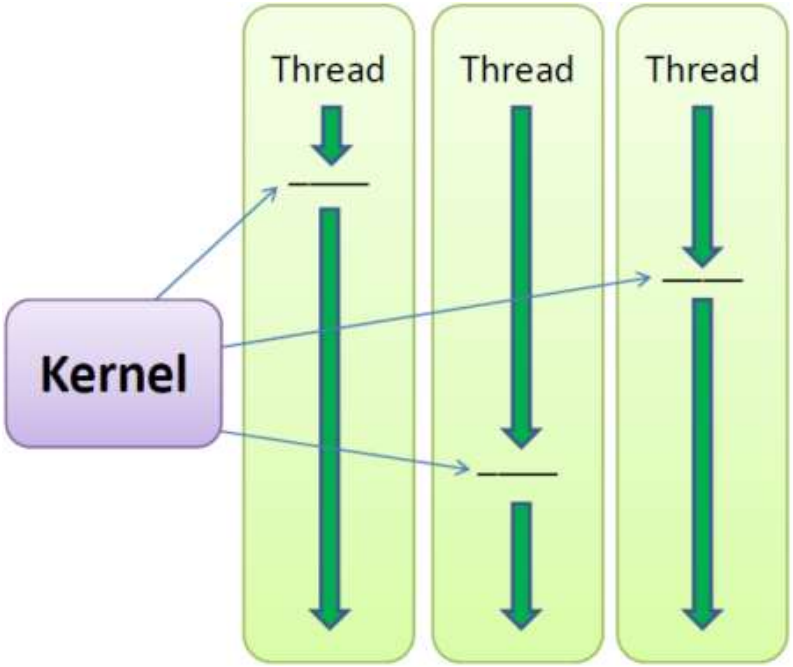
- **What are Events?**

Event-driven vs. Multithreaded Systems

- **How to address concurrency management?**
  - Multithreaded approach: processes are implemented as concurrent threads and the processor switches between them

Pro's

- Sequential code flow
- Preemption is possible
- Suitable for long computations
- wait() statement allowed

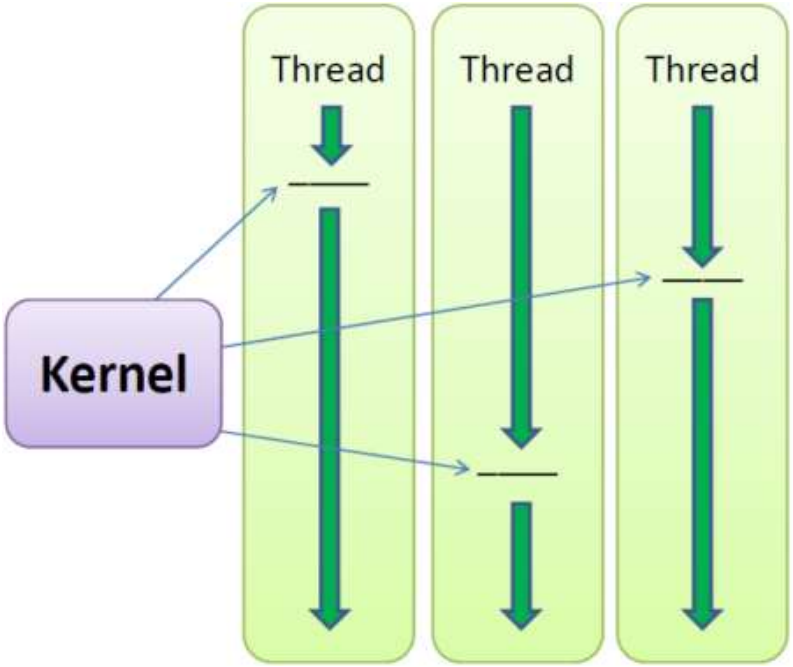


Event-driven vs. Multithreaded Systems

- **How to address concurrency management?**
  - Multithreaded approach: processes are implemented as concurrent threads and the processor switches between them

Con's

- Large code overhead
- Locking mechanisms required (reentrant code needed)
- Memory not shared among concurrent threads
  - -> Each thread has its own stack (typically over-provisioned)
  - -> High memory consumption

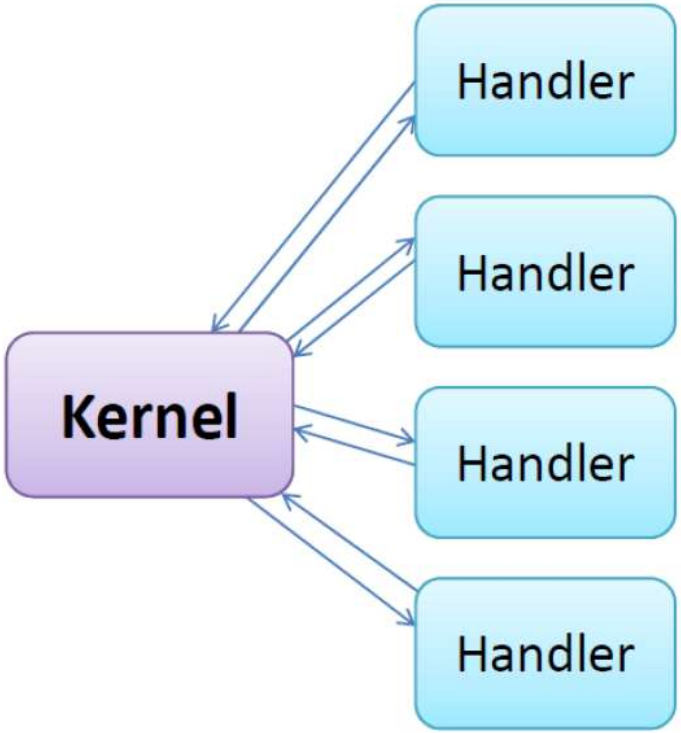


Event-driven vs. Multithreaded Systems

- **How to address concurrency management?**
  - Event-driven approach: processes are implemented as event handlers that run to completion

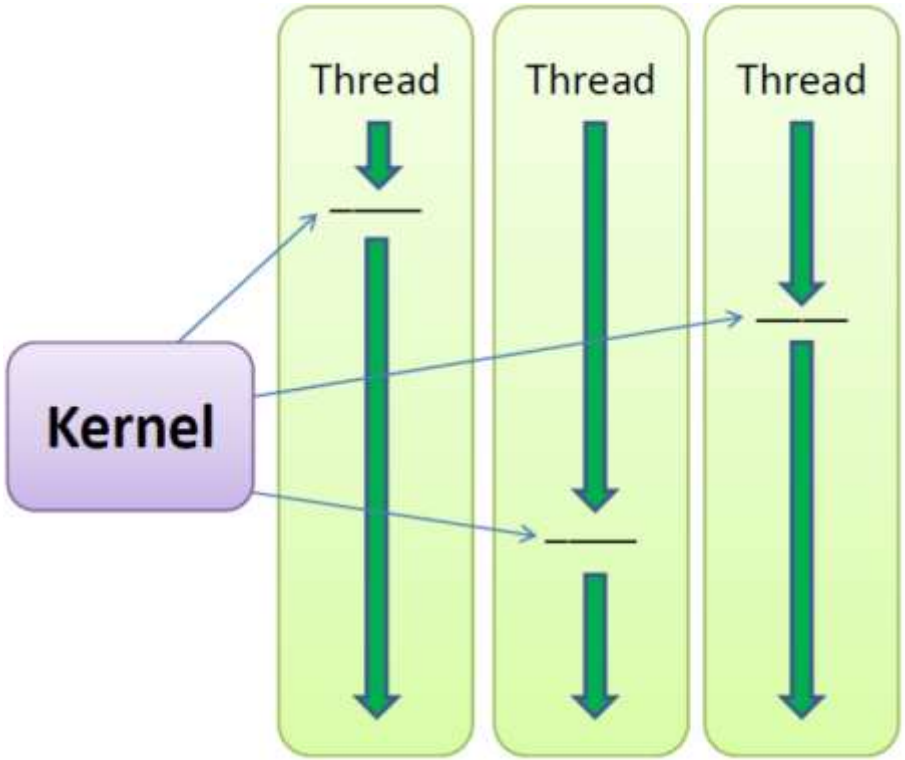
Pro's

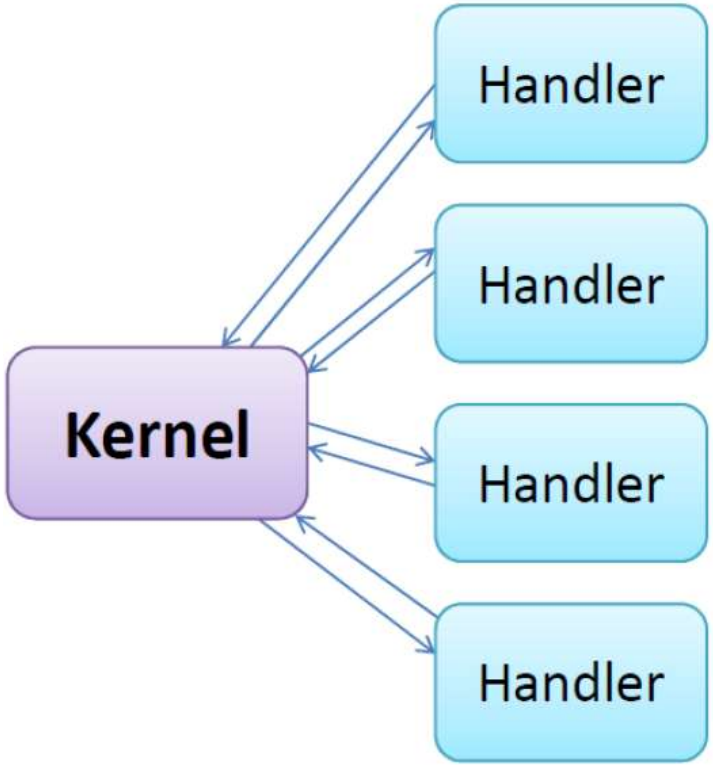
- Memory efficient (same stack for all processes)
- Low context switching overhead
- No need for locking mechanisms
- Compact code



Event-driven vs. Multithreaded Systems

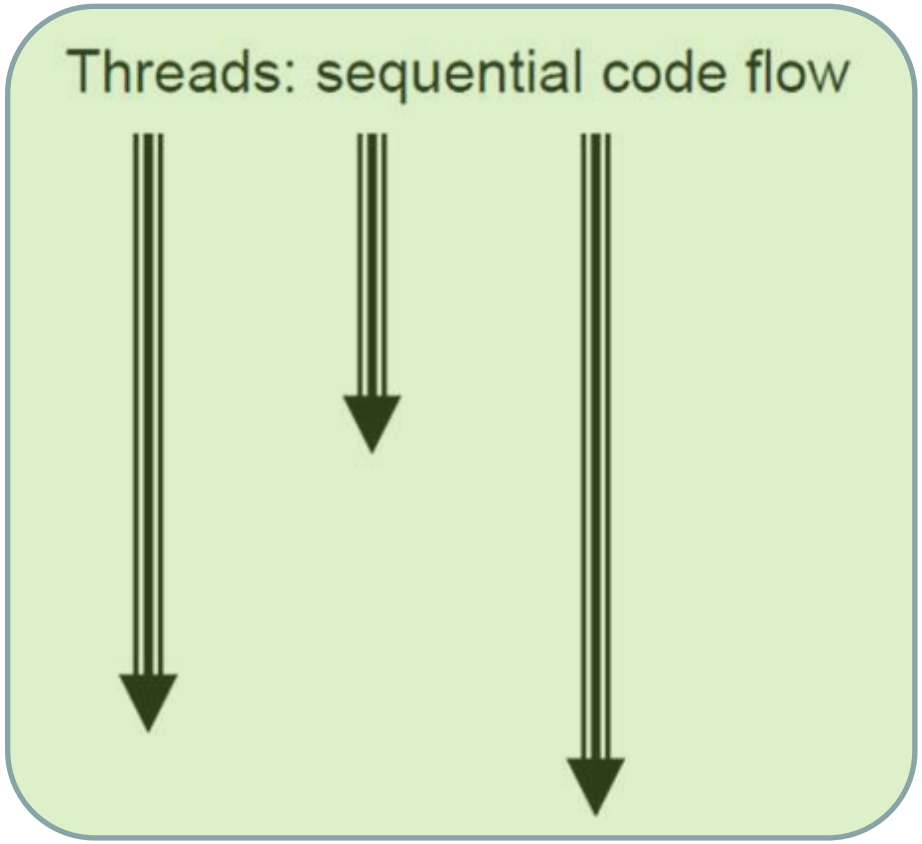
- **What is the Main Difference when Coding?**

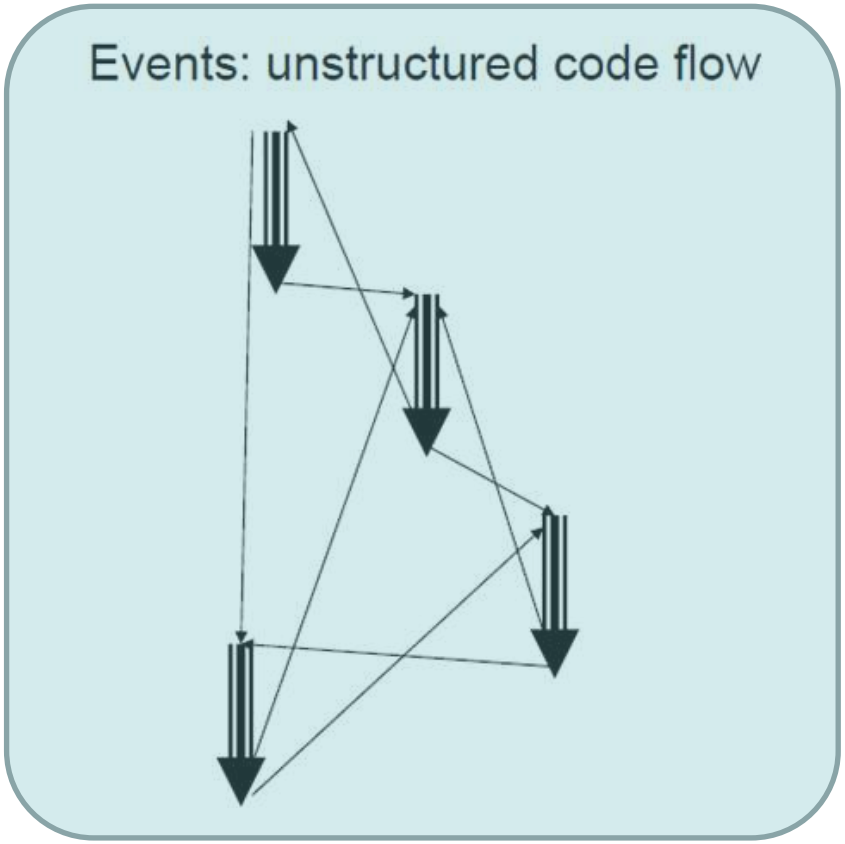




Event-driven vs. Multithreaded Systems

- **What is the Main Difference when Coding?**



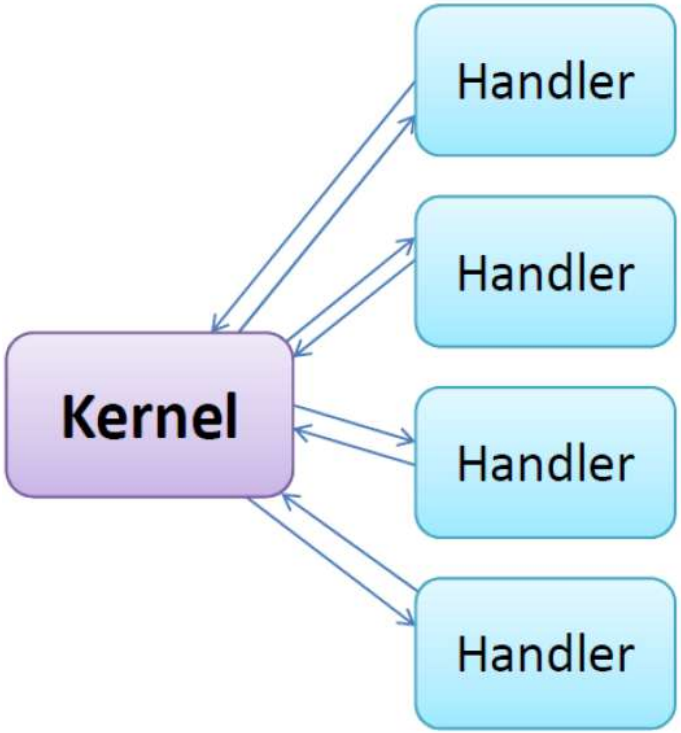


Event-driven vs. Multithreaded Systems

- **How to address concurrency management?**
  - Event-driven approach: processes are implemented as event handlers that run to completion

Con's

- No preemption possible
- State-driven programming hard to manage (recursive callbacks)
- Not all programs are easily expressible as state machines
- Unstructured code flow



The Contiki Operating System

- **Background**
  - History and Overview of Contiki
  - Some of Contiki's Main Features
  - Event-driven vs. Multithreaded Systems
  - The Contiki Kernel
  - Contiki's Protothreads
  - Contiki's Processes

The Contiki Kernel

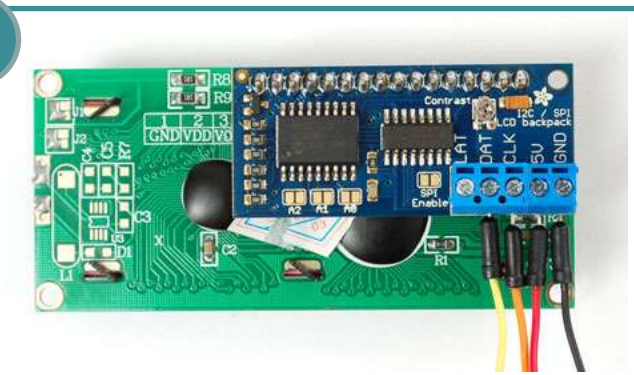
- **Event-driven Kernel**
  - Several processes executing concurrently and waiting for events
  - Events are posted, so when an event occurs:
    - The kernel activates the process
    - An ISR will post the event to the process

An event handler can NOT block!

The Contiki Kernel

- **Event-driven Kernel**
  - Three types of events
    1. Timer events
    2. External events
    3. Internal events
  - Event information
    - Process
    - Event type
    - Data

2



Peripheral devices with interrupt capabilities connected to I/O pins

1

Timer events

3

Interprocess communication

The Contiki Kernel

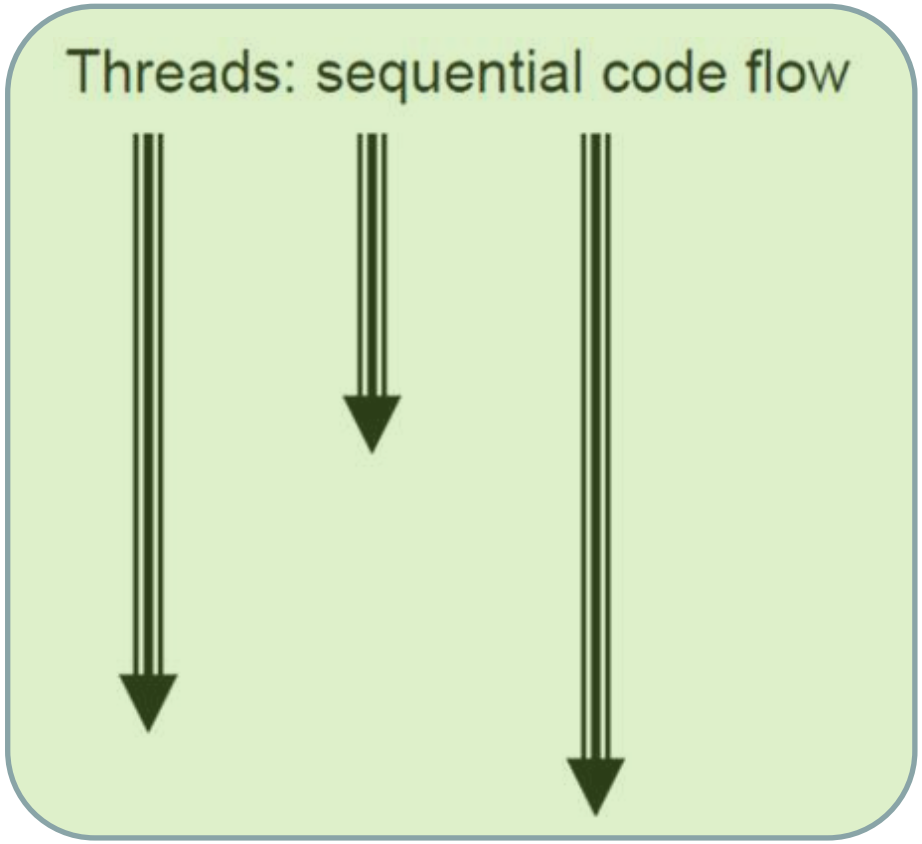
- **Contiki's Main Principle**
  - Processes do not run without events
  - When an event is posted, the kernel invokes the corresponding event handlers, and the processes will execute until they block waiting for another event
- **Contiki's Processes are Protothreads**
  - What are Protothreads?

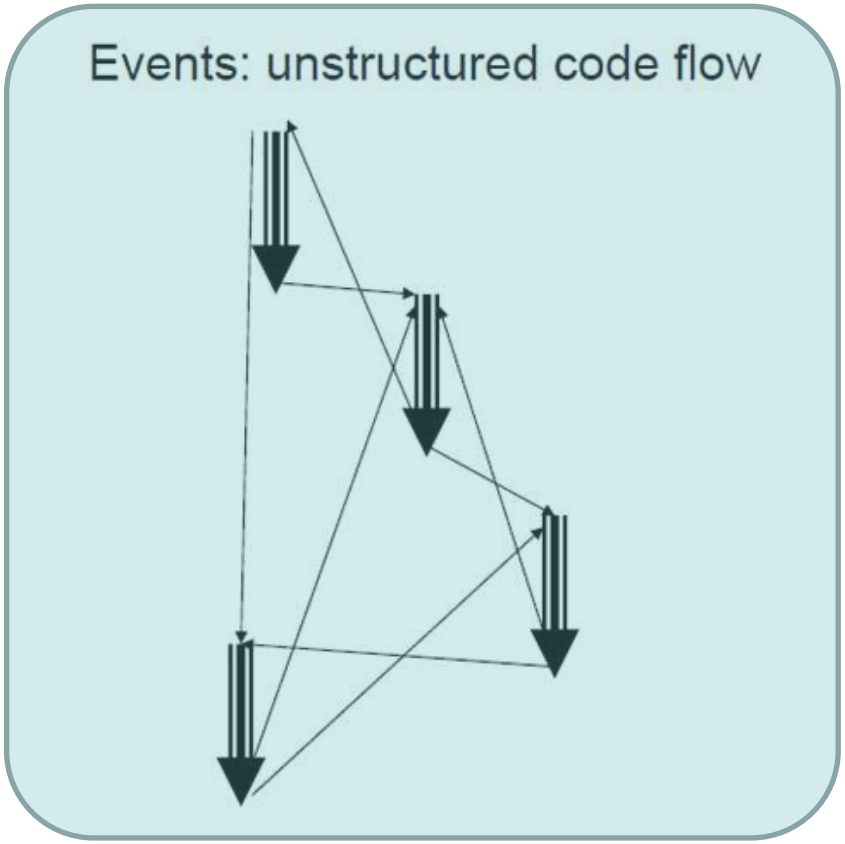
The Contiki Operating System

- **Background**
  - History and Overview of Contiki
  - Some of Contiki's Main Features
  - Event-driven vs. Multithreaded Systems
  - The Contiki Kernel
  - Contiki's Protothreads
  - Contiki's Processes

Event-driven vs. Multithreaded Systems

- **Sequential Code Flow is Nice**





Contiki's Protothreads

- **A New Programming Abstraction**
  - Design point combining the advantages of event-driven systems and multithreaded systems

Stackless
Protothreads run on the same stack

→ Low memory usage

→ Context switching by stack rewinding

Thread-like Construct
Simple sequential flow of control

→ No explicit state machine

→ Blocking-wait semantic

→ Sequential code structure

Implemented using local continuation
(The state of execution is stored and resumed)

Contiki's Protothreads

- **Example**

```
int a_protothread(struct pt *pt) {
  PT_BEGIN(pt);
  /* ... */
  PT_WAIT_UNTIL(pt, condition1);
  /* ... */
  if(something) {
    /* ... */
    PT_WAIT_UNTIL(pt, condition2);
    /* ... */
  }
  PT_END(pt);
}
```

PT_WAIT_UNTIL(pt, c1)
Conditional blocking wait

The protothread blocks until condition1 is true

Contiki's Protothreads

➢ Initialization, Declaration, and Definition

Initializing a Protothread

- PT_INIT (*pt, condition*)

Declaring a Protothread

- PT_THREAD (*name_args*)
- PT_BEGIN(pt)
- PT_END(pt)

```
int a_protothread(struct pt *pt) {
  PT_BEGIN(pt);
  /* … */
  PT_WAIT_UNTIL(pt, condition1);
  /* … */
  if(something) {
    /* … */
    PT_WAIT_UNTIL(pt, condition2);
    /* … */
  }
  PT_END(pt);
}
```

http://www.sics.se/~adam/pt/pt-1.4-refman/a00014.html

Contiki's Protothreads

- **Conditional Blocking Wait**
  - PT_WAIT_UNTIL (*pt, condition**)
    - The protothread is blocked until the statement evaluates to true
  - PT_WAIT_WHILE (*pt, condition**)
    - The protothread is blocked while the statement evaluates to true
  - PT_WAIT_WHILE (*pt, thread*)
    - The protothread is blocked until a child protothread completes
  - PT_SPAWN (*pt, child, thread*)
    - Spawns a child protothread and waits until it exits

**condition* can be any conditional statement or boolean expression

http://www.sics.se/~adam/pt/pt-1.4-refman/a00014.html

Contiki's Protothreads

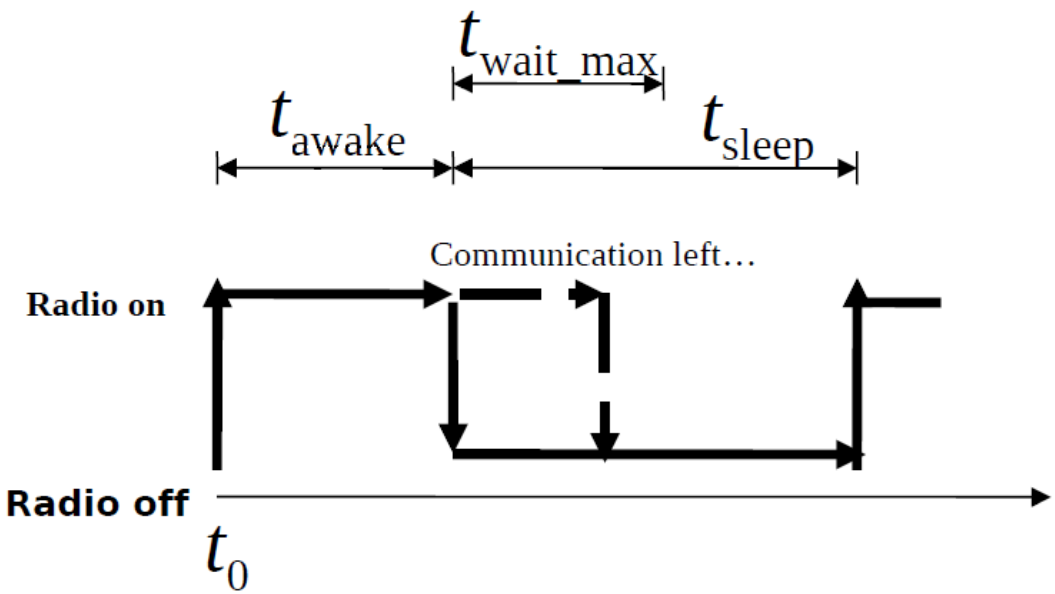
- **Calling, Exiting, Yielding, and Restarting Protothreads**
  - PT_YIELD (*pt*)
  - PT_YIELD_UNTIL (*pt, condition**)
    - Temporarily blocks the protothread, allowing other processing
  - PT_RESTART (*pt*)
  - PT_EXIT (*pt*)
  - PT_SCHEDULE (*function*)
    - *function* is the call to the C function implementing the protothread to be scheduled

http://www.sics.se/~adam/pt/pt-1.4-refman/a00014.html

Contiki's Protothreads

➢ Example: Controlling Radio Transmissions

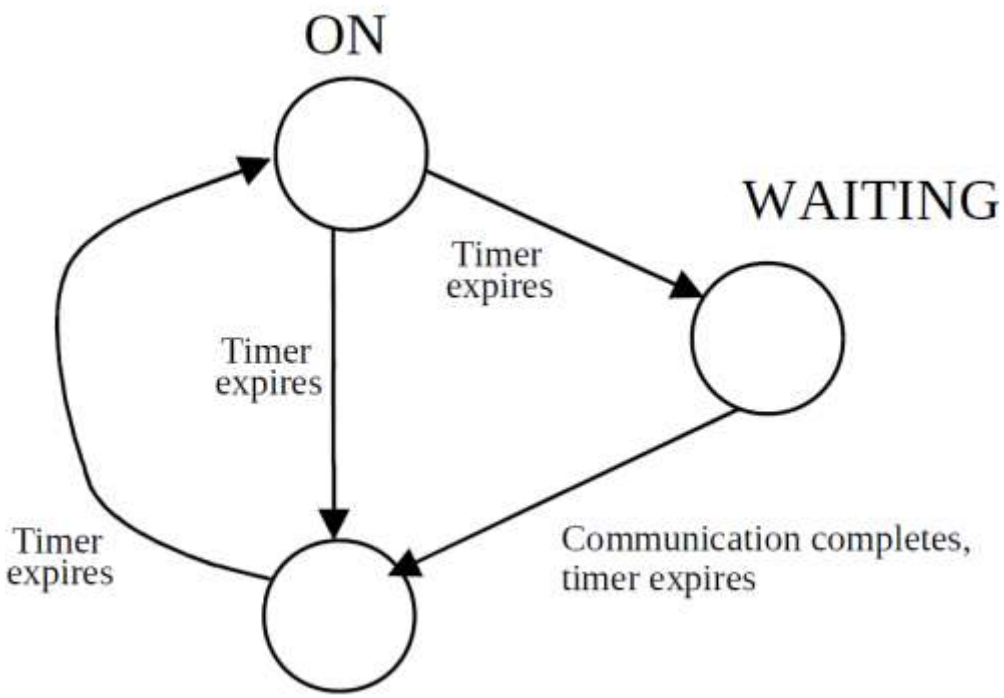
- Controlling radio transmissions using protothreads



1. Turn radio on
2. Wait until $t = t_0 + t_{awake}$
3. If communication has not been completed yet, wait until it has completed or until $t = t_0 + t_{awake} + t_{wait_max}$
4. Turn the radio off. Wait until $t = t_0 + t_{awake} + t_{sleep}$
5. Repeat from step 1

Contiki's Protothreads

- **Example: Controlling Radio Trai**
  - Event-driven state machine and im

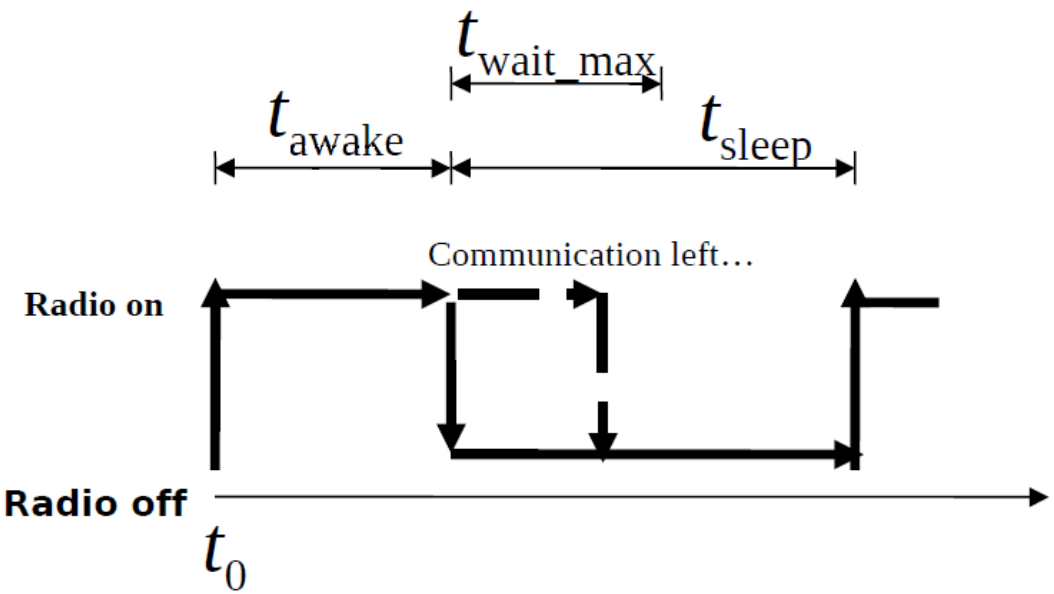



```
enum {ON, WAITING, OFF} state;

void eventhandler() {
  if(state == ON) {
    if(expired(timer)) {
      timer = t_sleep;
      if(!comm_complete()) {
        state = WAITING;
        wait_timer = t_wait_max;
      } else {
        radio_off();
        state = OFF;
      }
    }
  } else if(state == WAITING) {
    if(comm_complete() ||
       expired(wait_timer)) {
      state = OFF;
      radio_off();
    }
  } else if(state == OFF) {
    if(expired(timer)) {
      radio_on();
      state = ON;
      timer = t_awake;
    }
  }
```

Contiki's Protothreads

➢ Example: Controlling Radio Transmissions

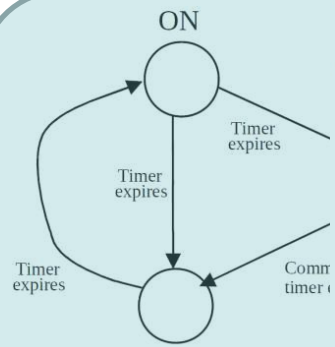
- Protothread-based implementation




```
int protothread(struct pt *pt) {
  PT_BEGIN(pt);
  while(1) {
    radio_on();
    timer = t_awake;
    PT_WAIT_UNTIL(pt, expired(timer));
    timer = t_sleep;
    if(!comm_complete()) {
      wait_timer = t_wait_max;
      PT_WAIT_UNTIL(pt, comm_complete()
                || expired(wait_timer));
    }
    radio off();
    PT_WAIT_UNTIL(pt, expired(timer));
  }
  PT_END(pt);
}
```

Contiki's Protothreads

```
int protothread(struct pt *pt) {
  PT_BEGIN(pt);
  while(1) {
    radio_on();
    timer = t_awake;
    PT_WAIT_UNTIL(pt, expired(timer));
    timer = t_sleep;
    if(!comm_complete()) {
      wait_timer = t_wait_max;
      PT_WAIT_UNTIL(pt, comm_complete()
                || expired(wait_timer));
    }
    radio off();
    PT_WAIT_UNTIL(pt, expired(timer));
  }
  PT_END(pt);
}
```




```
enum {ON, WAITING, OFF} state;

void eventhandler() {
  if(state == ON) {
    if(expired(timer)) {
      timer = t_sleep;
      if(!comm_complete()) {
        state = WAITING;
        wait_timer = t_wait_max;
      } else {
        radio_off();
        state = OFF;
      }
    }
  } else if(state == WAITING) {
    if(comm_complete() ||
       expired(wait_timer)) {
      state = OFF;
      radio_off();
    }
  } else if(state == OFF) {
    if(expired(timer)) {
      radio_on();
      state = ON;
      timer = t_awake;
    }
  }
```

- **That's why protothreads are so powerful!**

Contiki's Protothreads

- **Protothread also have Limitations!**
  - Protothreads do not save the stack context across a blocking wait, so local variables are <u>not</u> preserved
    - Workaround: <u>use global variables instead</u>
  - Constraints on the use of switch() constructs in programs
    - Workaround: <u>do not use switches</u>

The Contiki Operating System

- **Background**
  - History and Overview of Contiki
  - Some of Contiki's Main Features
  - Event-driven vs. Multithreaded Systems
  - The Contiki Kernel
  - Contiki's Protothreads
  - Contiki's Processes

Contiki's Processes

- **Contiki's Processes are Protothreads**
  - Defined by event handler and optional poll handler functions
  - Every process is kept in its private memory
    - The kernel only keeps a pointer to the process state
  - All processes share the same address space
    - No memory protection
  - Contiki does not disable interrupts
    - Events cannot be posted by interrupt handlers because that would lead to race conditions in the event handler
    - The kernel provides a polling flag to request a poll event

Check: contiki\core\sys\process.h

Contiki's Processes

- **Contiki's Processes are Protothreads**
  - PROCESS_THREAD defines a new process (name, events, data)
  - PROCESS_BEGIN() and PROCESS_END() enclose the process
  - PROCESS_WAIT_EVENT() or PROCESS_YIELD() waits for a new event to be posted to the process
  - PROCESS_WAIT_EVENT_UNTIL(condition c) waits for an event to be posted with an extra condition
    - May be an interrupt coming from a button that has been pressed
    - May be an expired timer

Check: contiki\core\sys\process.h

Contiki's Processes

- **Contiki's Processes are Protothreads**
  - PROCESS_PAUSE() yields the process for a short while
  - Post (a)synchronous events
    - PROCESS_POST(): the process will be invoked later
    - PROCESS_POST_SYNCH(): the process will be invoked now
    - These functions must not be called from an interrupt
  - PROCESS_POLL() polls the process by sending a PROCESS_EVENT_POLL event to the process
  - Events can be defined as *#define EVENT_EX 1*

Check: contiki\core\sys\process.h

Contiki's Processes

From Theory to Practice

- Hello World!

1 **Define a new process**

PROCESS_THREAD defines a process (name, events, data)

2 **Enclose the process**

PROCESS_BEGIN()
PROCESS_END()

```
PROCESS_THREAD(hello_world_p, ev, data) {
  PROCESS_BEGIN();
      printf("Hello world!\n");
  PROCESS_END();
}
```

Contiki's Processes

From Theory to Practice

- Hello World!

3 **Wait for an event**

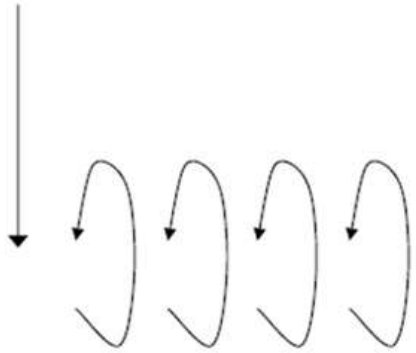
PROCESS_WAIT_EVENT() waits for a new event to be posted to the process

```
PROCESS_THREAD(hello_world_p, ev, data) {
    PROCESS_BEGIN();
        printf("Hello world!\n");

        while(1){
            PROCESS_WAIT_EVENT();
        }

    PROCESS_END();
}
```

Contiki's Processes

- **From Theory to Practice**
  - Hello World!

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



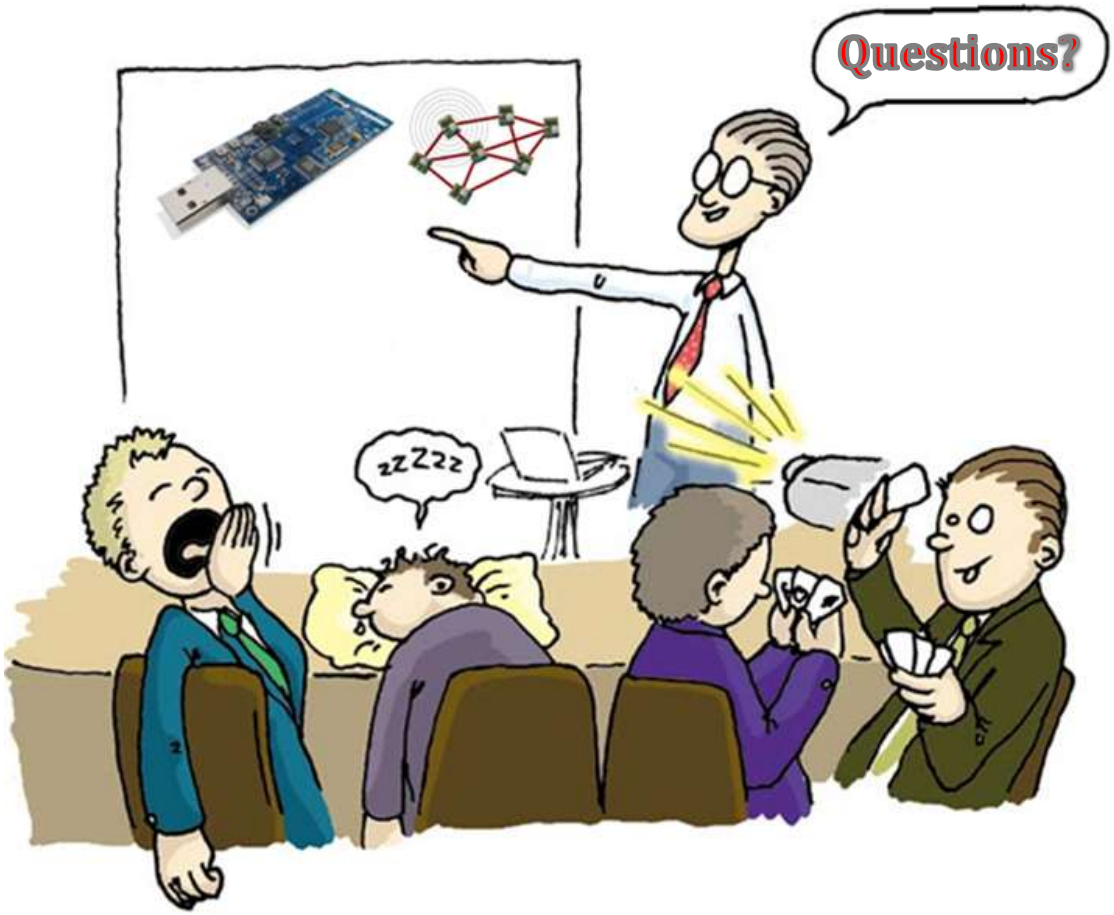
Waits forever!

References

Contiki

- A lightweight and flexible operating system for tiny networked sensors
  http://www.sics.se/~adam/dunkels04contiki.pdf
- Official website
  http://www.sics.se/contiki

Protothreads

- Using protothreads for sensor node programming
  http://www.sics.se/~adam/dunkels05using.pdf
- Simplifying event-driven programming of memory-constrained emb. systems
  http://www.sics.se/~adam/dunkels06protothreads.pdf

Questions & Answers