

# Small Stacks

---





# Sensors to WSN (Smart Buildings)

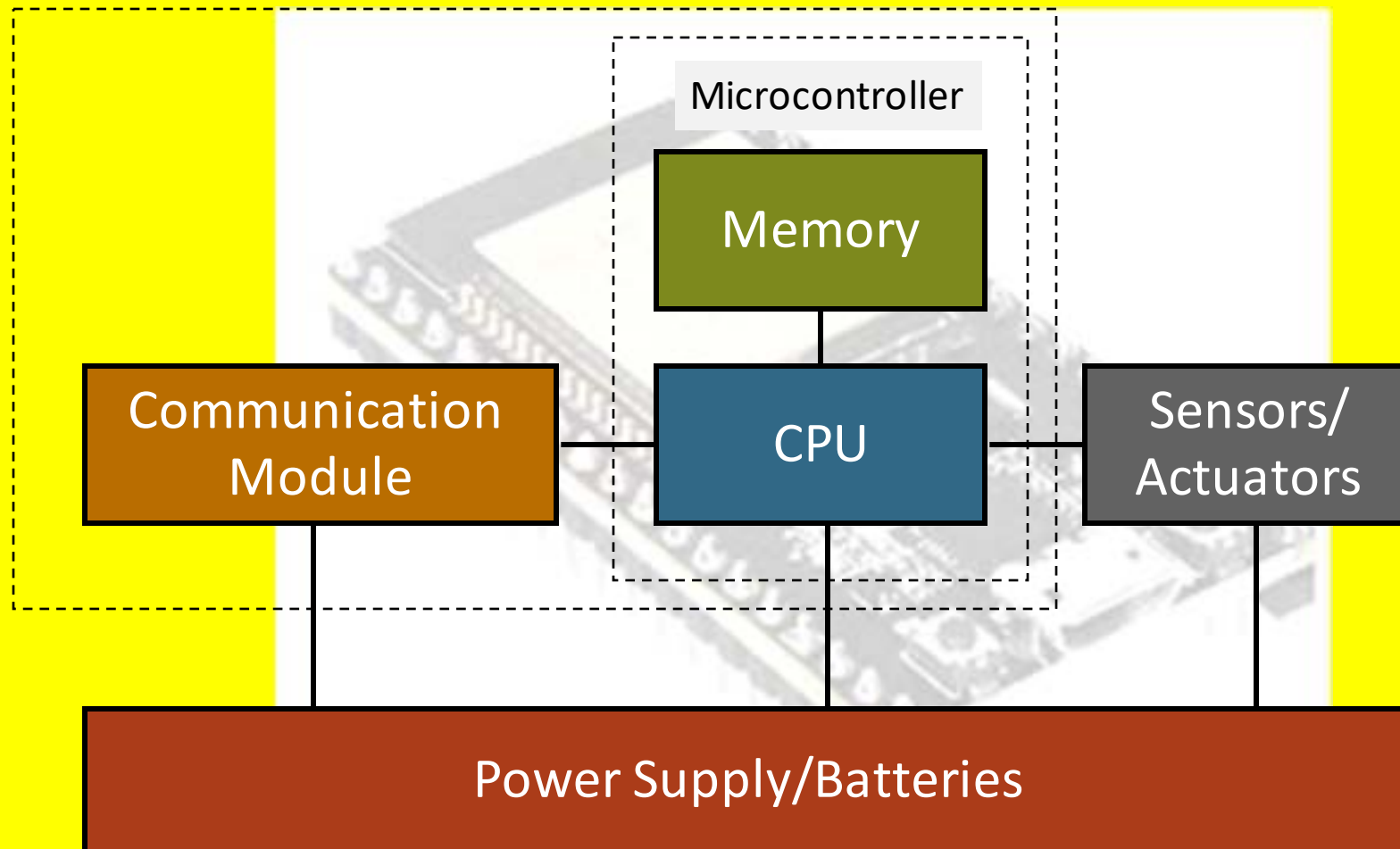
More than 100 sensors were installed to monitor the health of the bridge.

- **Weather stations** (2) (measures: wind speed and direction, ambient temperature and relative humidity)
- **Road temperature sensors** (4)
- **Concrete deck temperature sensors** (5)
- **Accelerometers** (42)
  - On shore (2)
  - On pylons (12)
  - On deck (15)
  - On stays (13)
- **Load cells on stays** (16)
- **Load cell on fuse** (4 digital+4 analog)
- **Joint displacement sensor** (on both expansion joints)
- **Water detection sensors** (4)
- **Strain gauges on gussets** (16)



<https://placetechnet/analysis/worlds-smartest-buildings-the-edge-amsterdam/>

# Small Systems: WSN – IoT



# 0. Microcontroller Examples

- Microchip ATMEga (PIC ...)

- 8-bit controller, 8 MHz
- Up to 128KB Flash
- 4 KB RAM
- 8-channel ADC
- 6 sleep modes



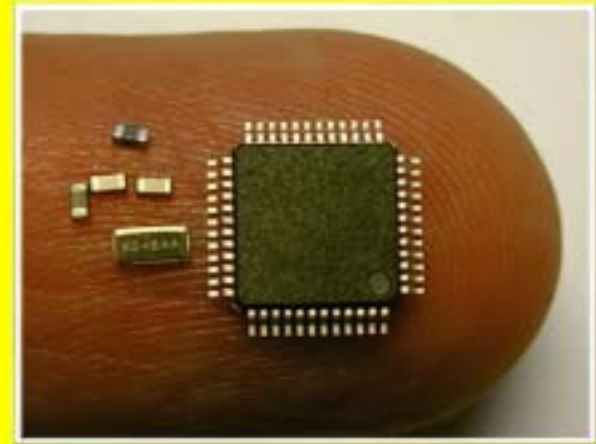
- Texas Instruments MSP430

- 16-bit RISC core, 4 MHz
  - Up to 120 KB flash
  - 2-10 KB RAM
  - 12-channel ADC
- ⇒ Picosatellites



# Microcontroller Examples

- ATSAM21
  - 32-bit ARM Cortex-M0+ @48MHz
  - 256KB flash, 32KB RAM
  - 20-channel ADC
  - Full-speed USB 2.0 interface
- ESP32
  - 32-bit Xtensa dual-core @240MHz
  - 448KB flash, 520KB RAM
  - 18-channel ADCs
  - Integrated WiFi and Bluetooth

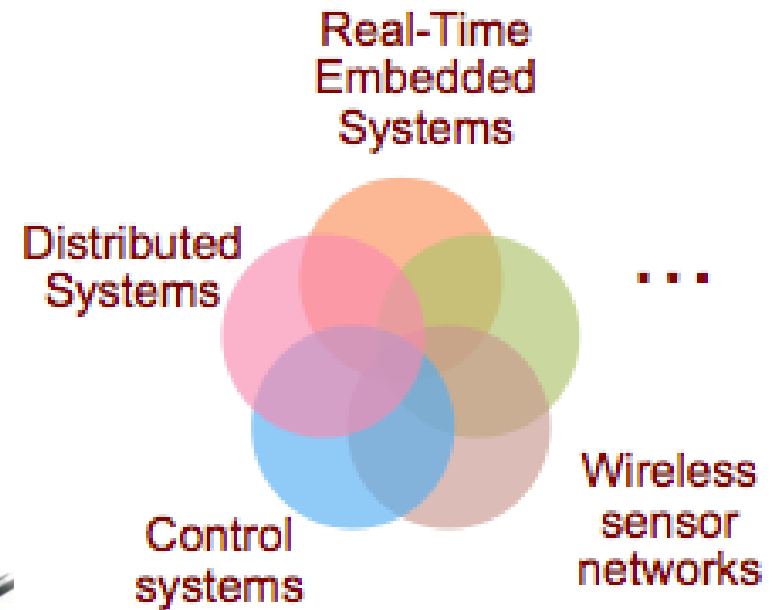
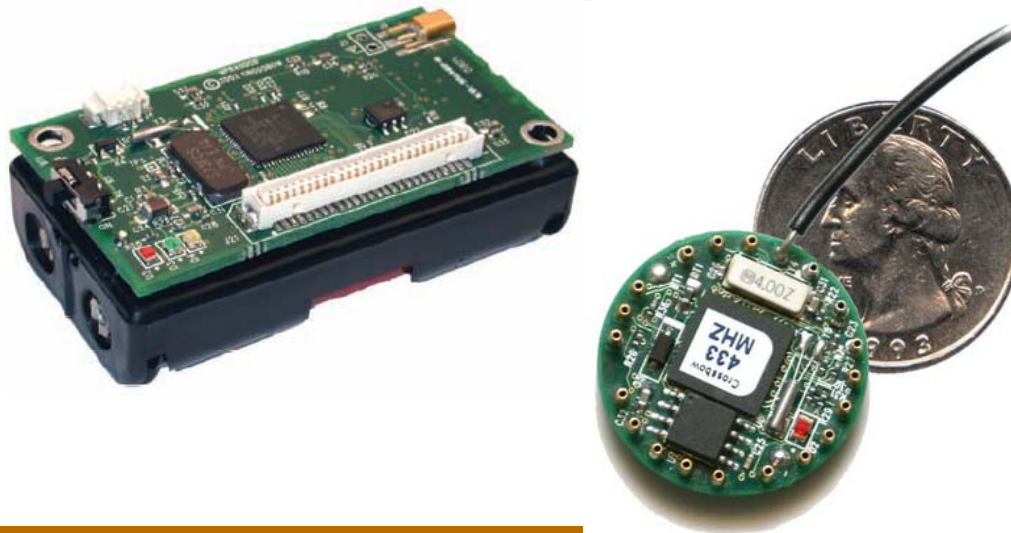


Sources:

- <http://www.avdweb.nl/arduino/samd21/sam-d21.html>
- <https://www.espressif.com>

# Mica Motes

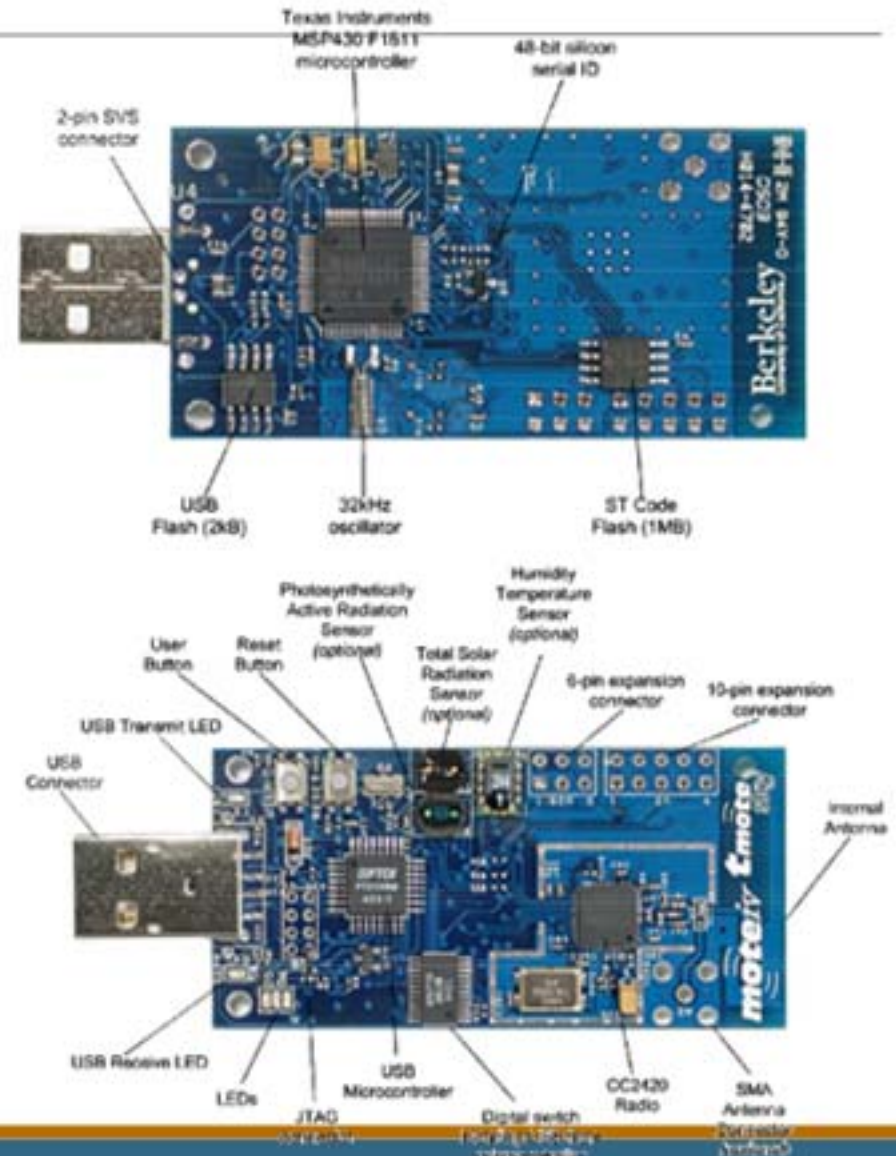
- By Crossbow, USA
- Controller
  - 8-bit Atmel ATMega128L
- Communication
  - RFM TR1000





# Tmote Sky

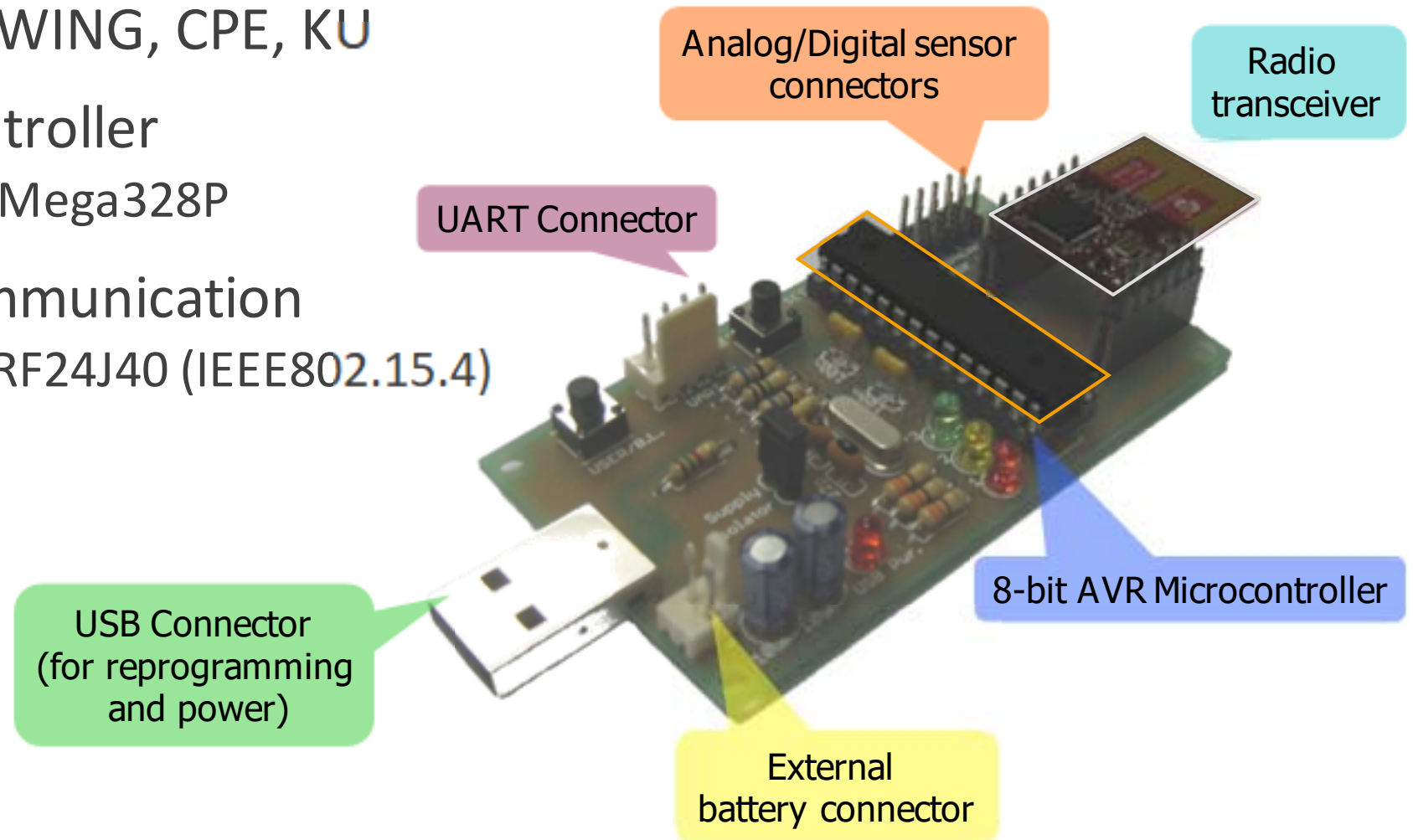
- By Sentilla (formerly Moteiv), USA
- Controller
  - 16-bit TI MSP430
- Communication
  - Chipcon CC2420 (IEEE 802.15.4)





# IWING-MRF Motes

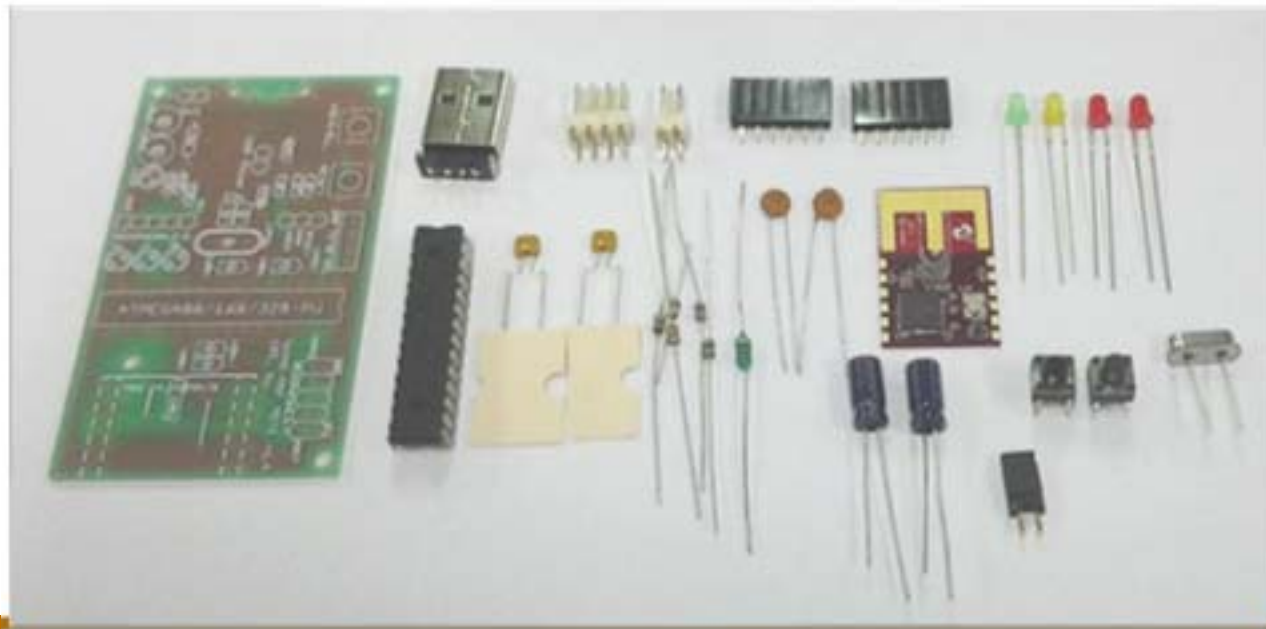
- By IWING, CPE, KU
- Controller
  - ATmega328P
- Communication
  - MRF24J40 (IEEE802.15.4)



# IWING-MRF Motes

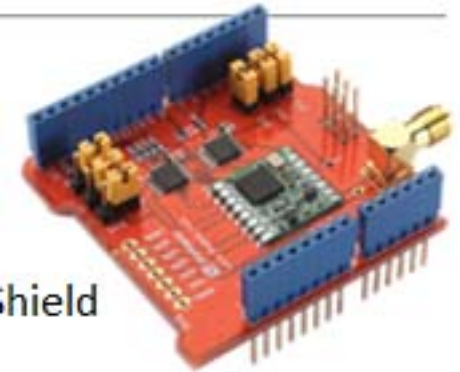
---

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



# Arduino Boards

- Various communication "shields"



LoRa Shield



WiFi Shield



NBIoT Shield

- <https://www.arduino.cc/en/Main/Boards>
- <http://www.dragino.com/products/nb-iot/item/130-nb-iot-shield.html>
- <https://store.arduino.cc/usa/arduino-wifi-shield>
- <http://www.dragino.com/products/lora/item/102-lora-shield.html>

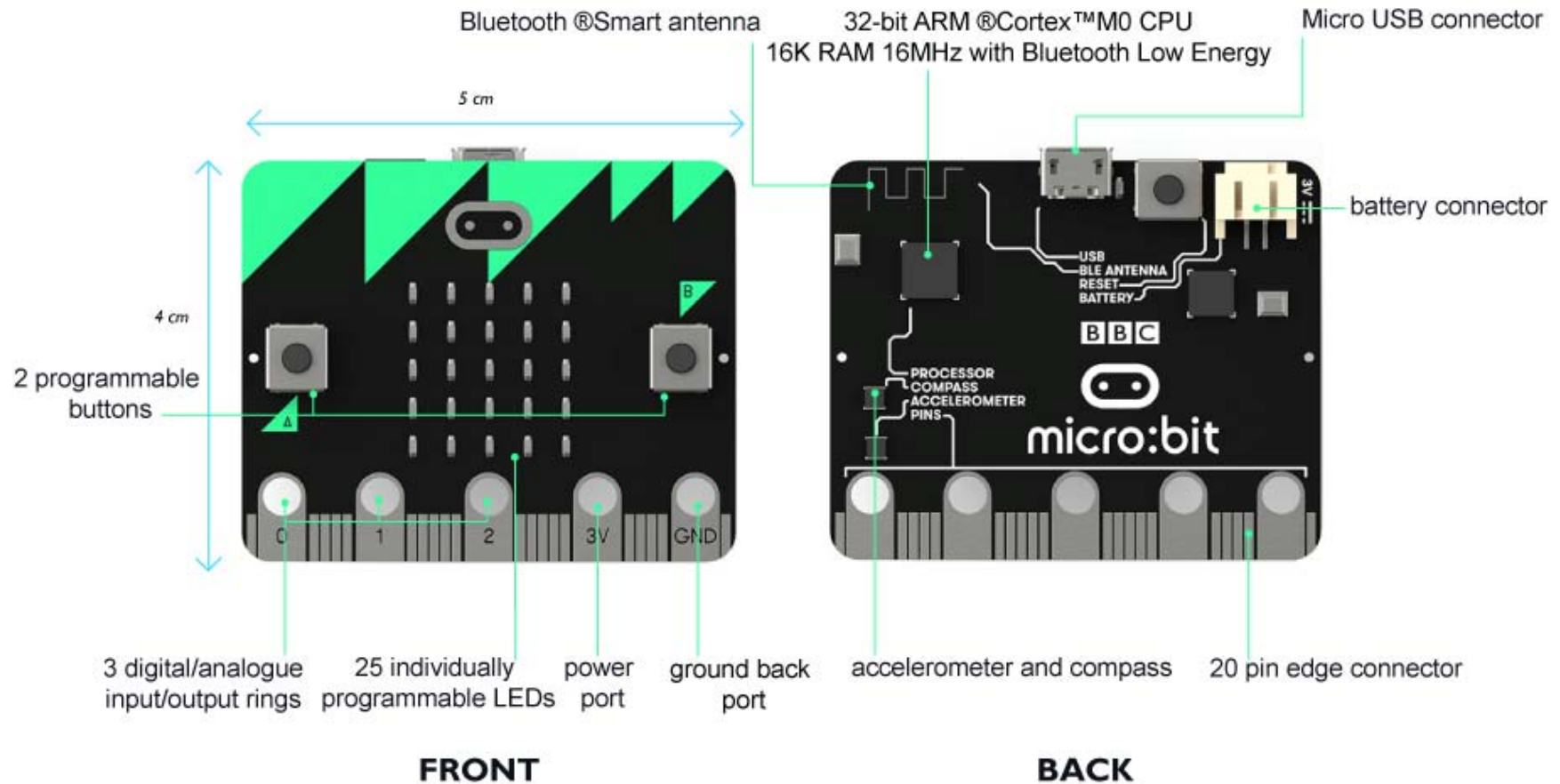
# ESP32 Modules

- By Espressif Systems
- Controller
  - 32-bit Xtensa dual-core
- Communication
  - WiFi and Bluetooth



<https://www.espressif.com/en/products/hardware/modules>

# BBC's Micro:bit



<https://microbit.org/guide/features/>

# Raspberry Pi

---

- By Raspberry Pi Foundation
- Controller
  - 1.4 GHz 64/32-bit quad-core ARM Cortex-A53
- Communication
  - WiFi and Bluetooth



Raspberry Pi 3 Model B+



Raspberry Pi Zero W

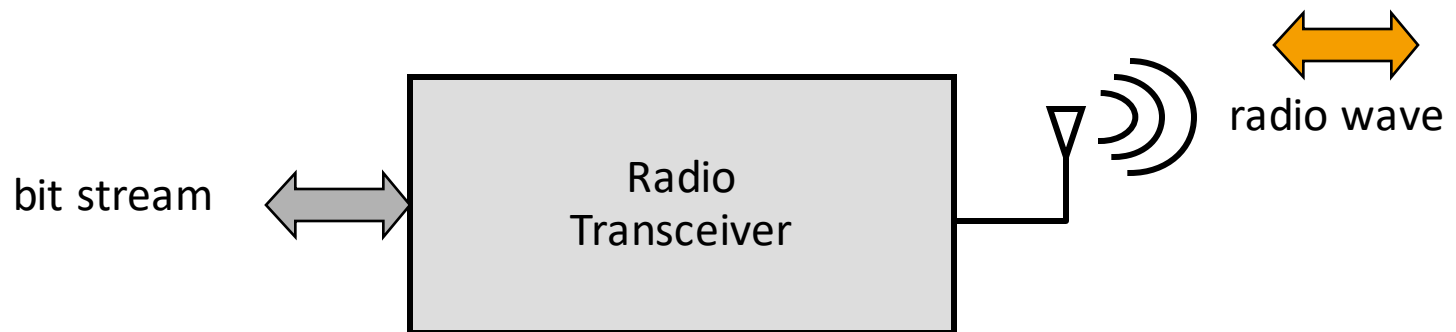
<https://www.raspberrypi.org/products/>



# 1. Communication Device

---

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



# Transceiver Characteristics

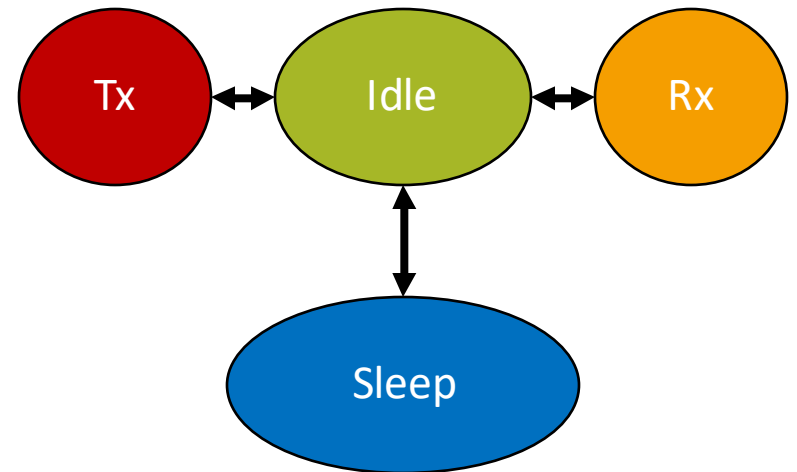
---

- Service to upper layers: packet, byte, bit
- Data rate
- 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

# 2. 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?

# 3. 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
- Capacity under load
- Efficient recharging at low current
- Voltage stability (to avoid DC-DC conversion)

# Battery Examples

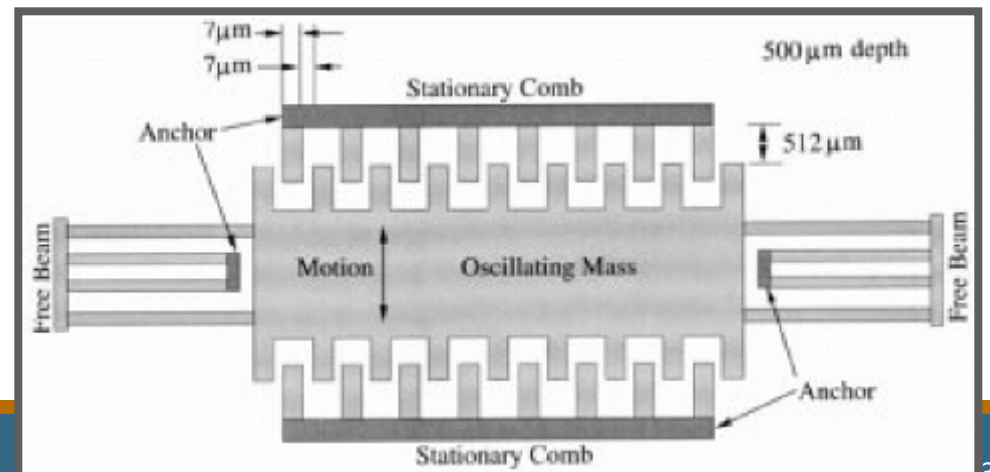
- Energy per volume (Joule/cc):

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

[http://en.wikipedia.org/wiki/Energy\\_density](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  $10 \mu\text{W}/\text{cm}^2$  and  $15 \text{ mW}/\text{cm}^2$
  - Temperature gradients –  $80 \mu\text{W}/\text{cm}^2$  @ 1 V from 5K difference
  - Vibrations – between 0.1 and  $10000 \mu\text{W}/\text{cm}^3$
  - Pressure variation (piezo-electric) –  $330 \mu\text{W}/\text{cm}^2$  from the heel of a shoe
  - Air/liquid flow (MEMS gas turbines)



# 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?

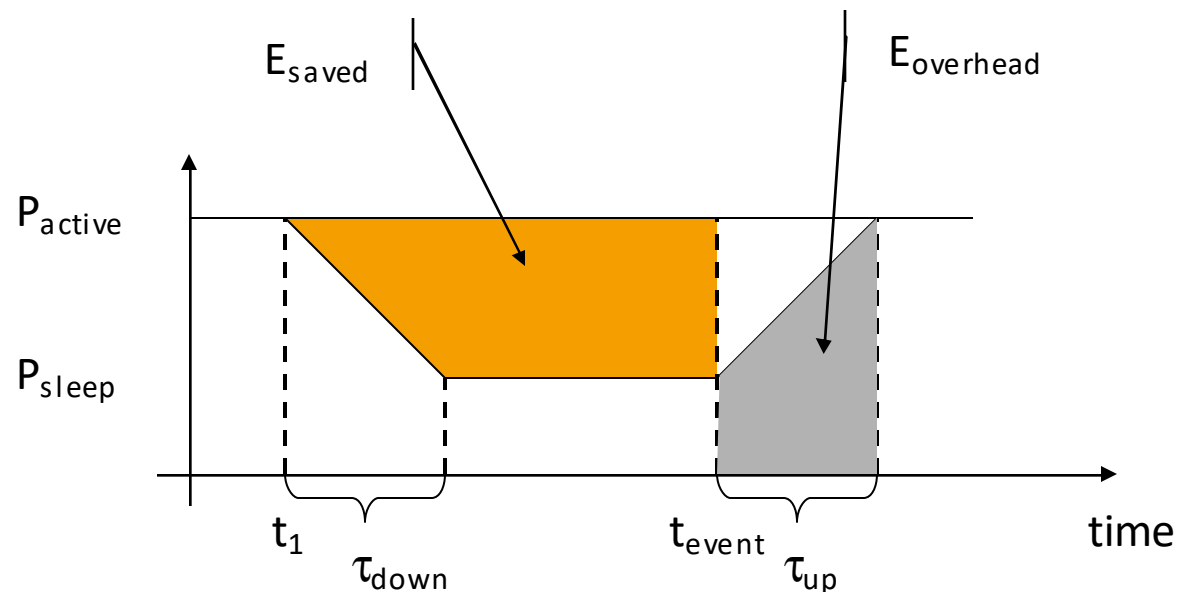
# Energy Consumption Figures

---

- TI MSP 430 (@ 1 MHz, 3V):
  - Fully operational mode 1.2 mW
  - Four sleep modes
  - Deepest sleep mode 0.3  $\mu$ W – 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  $\mu$ W

# 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

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

# Computation vs. Communication

---

- 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, ...

# 4. 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

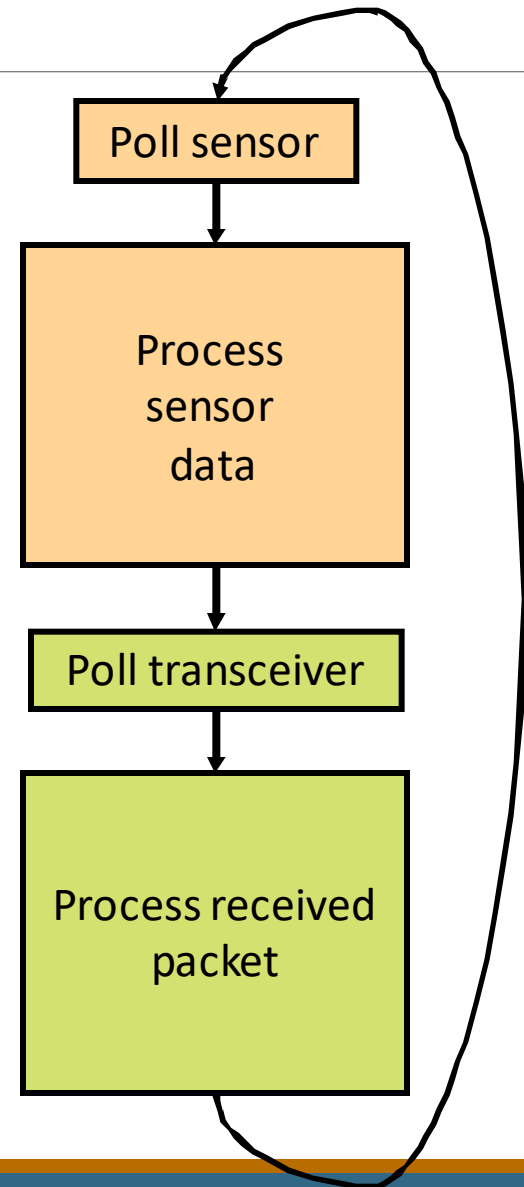
# Possible OS Options

---

- 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

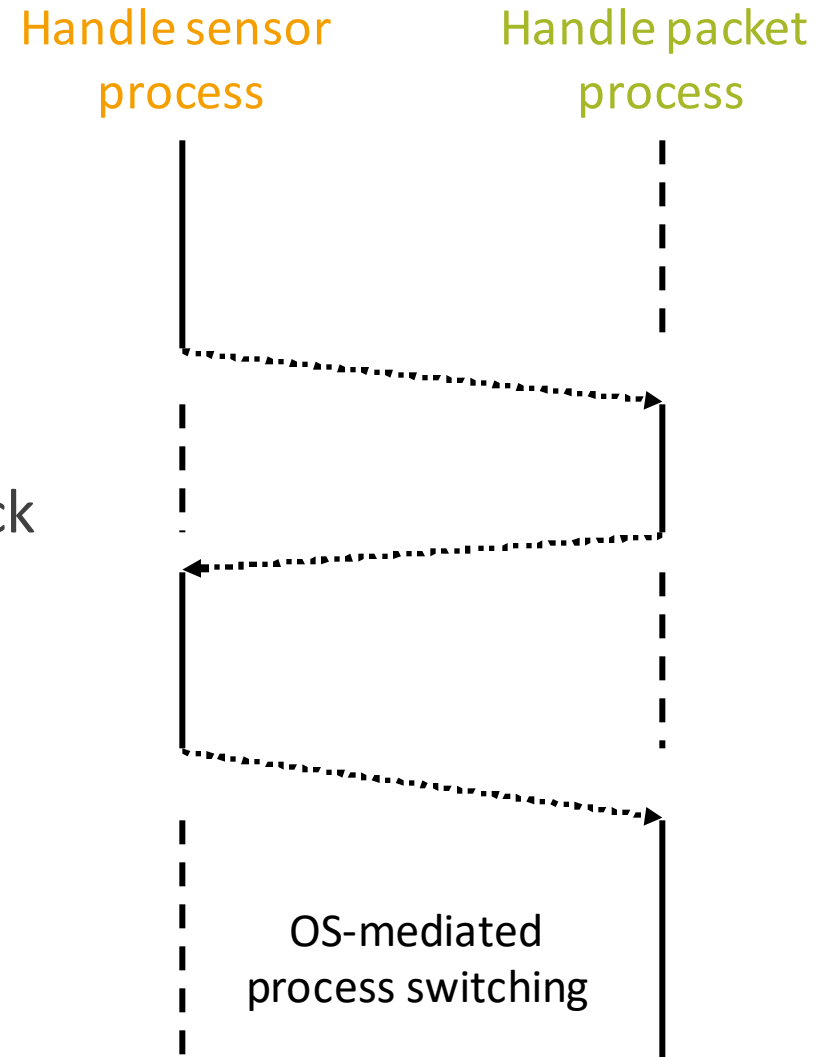
# Concurrency Support

- 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
- Not much of a problem on modern microcontrollers

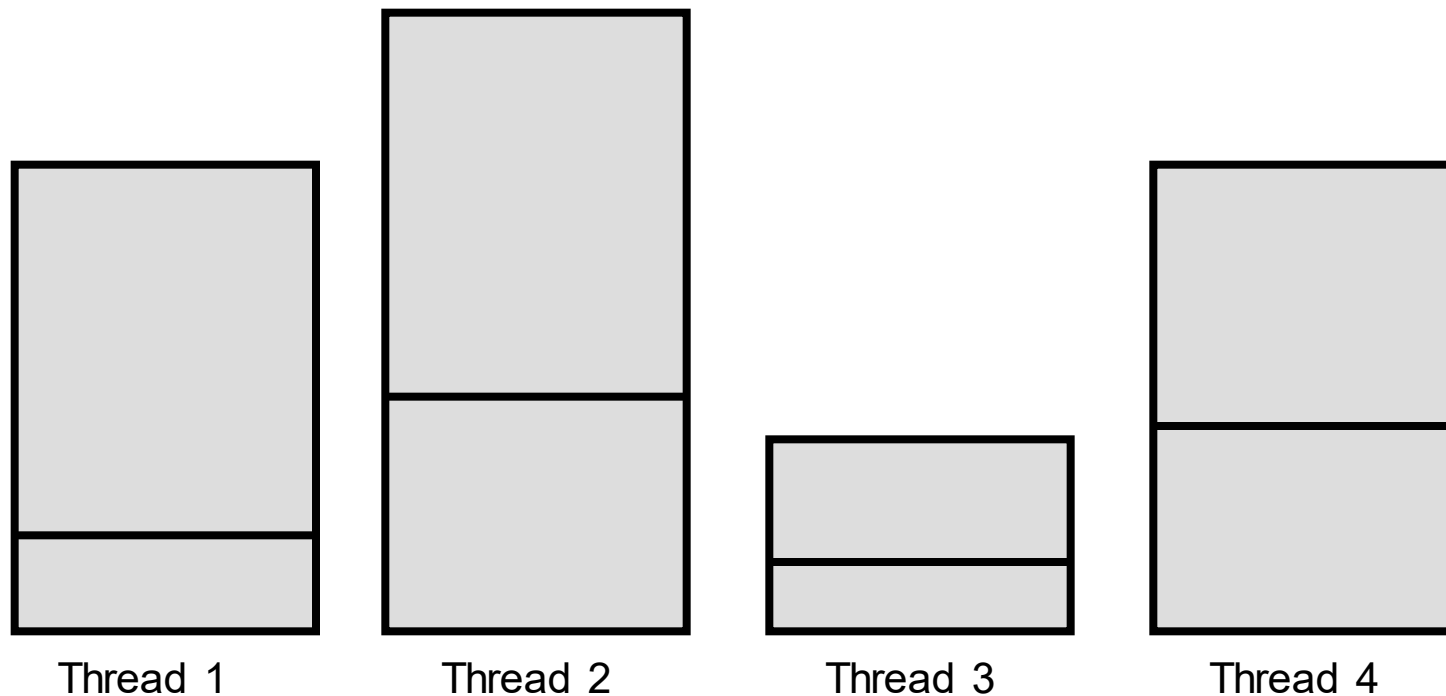




# Problems with Multithreads

---

- Four threads, each with its own stack



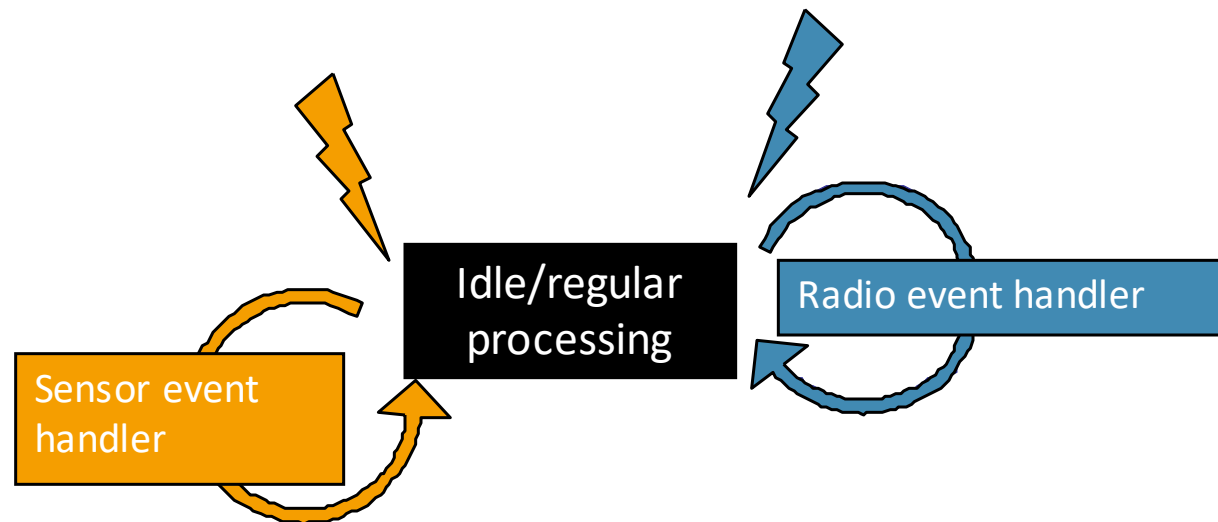
# Problems with Multithreads

---

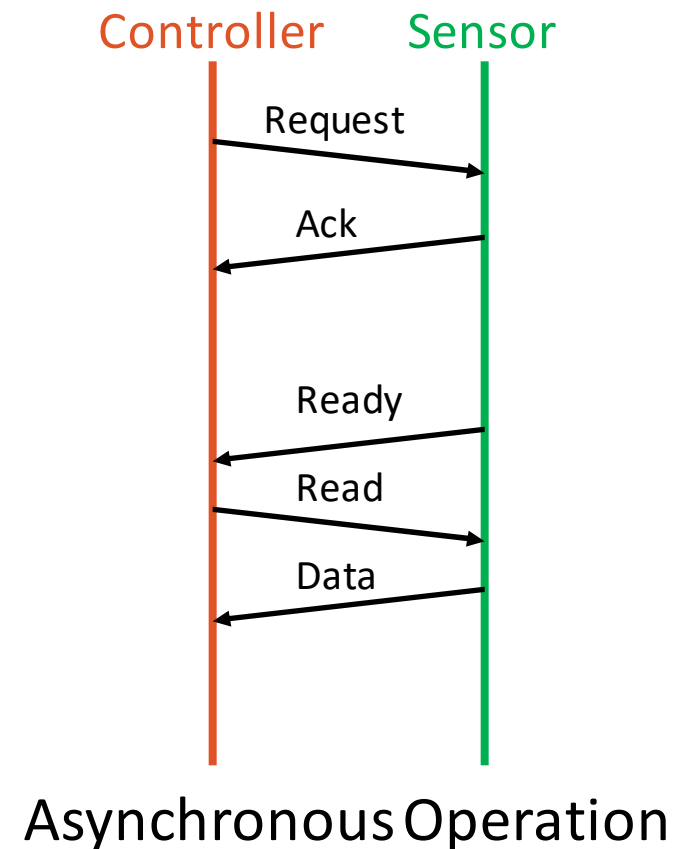
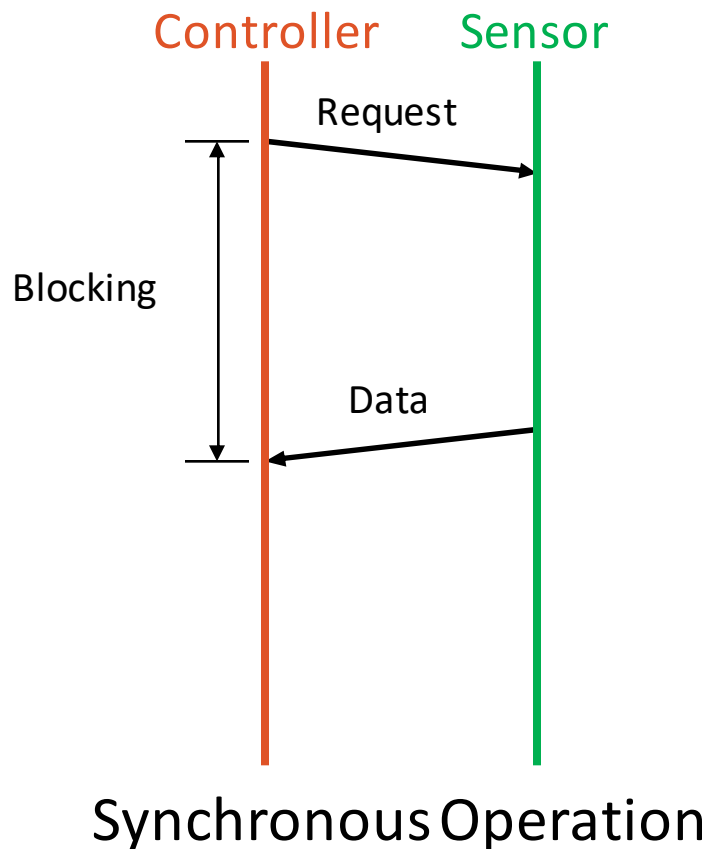
- Code employing preemptive threading library must ensure *thread-safe* operations

# Event-Based Concurrency

- Perform regular processing or be idle
- React to events when they happen immediately
- Examples
  - Bare-metal programming with interrupt handlers
  - Contiki, Zephyr, RIOT, TinyOS, ROS, SHARK, ERICA...



# Split-Phase Operations

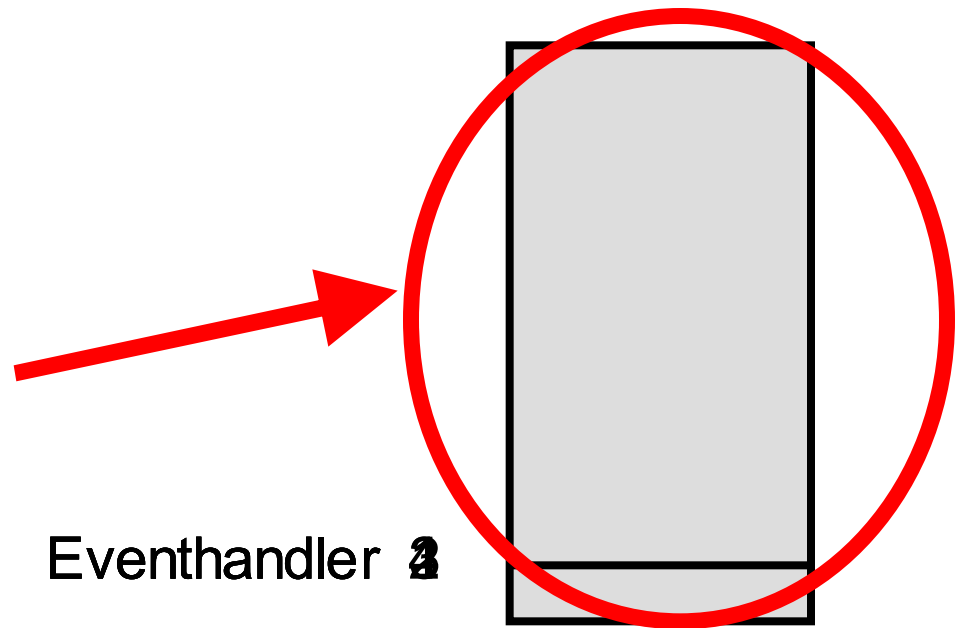


# Events Require One Stack

---

- Four event handlers, one stack

Stack is reused for  
every event handler



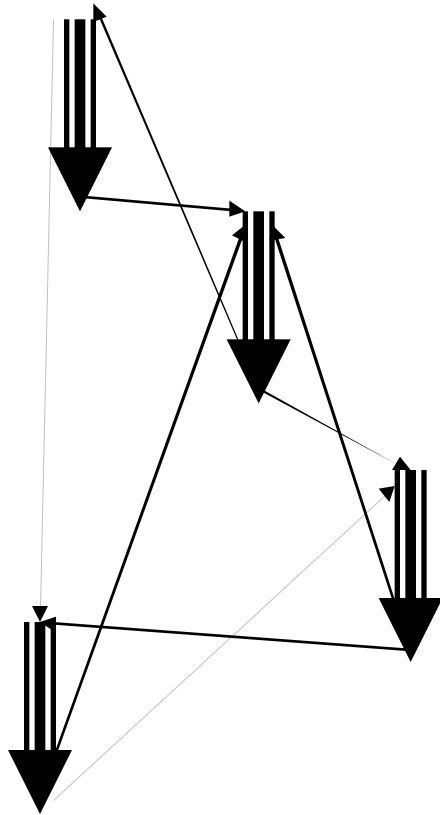
# Event-based Protocol Stack

---

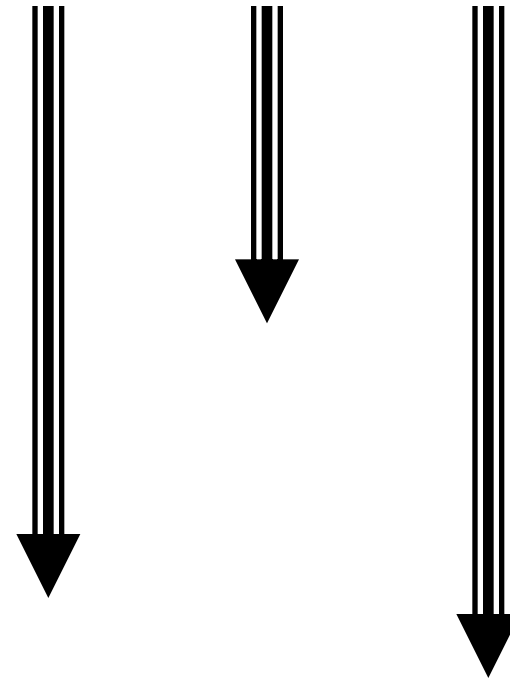
- Usual networking API: **sockets**
  - Issue: blocking calls to receive data
  - Not match to event-based OS
- API is therefore also event-based
  - E.g., Tell some component that some other component wants to be informed if and when data has arrived
  - Component will be posted an event once this condition is met
  - Details: see TinyOS

# Problem with Event-based Model

Events: unstructured code flow



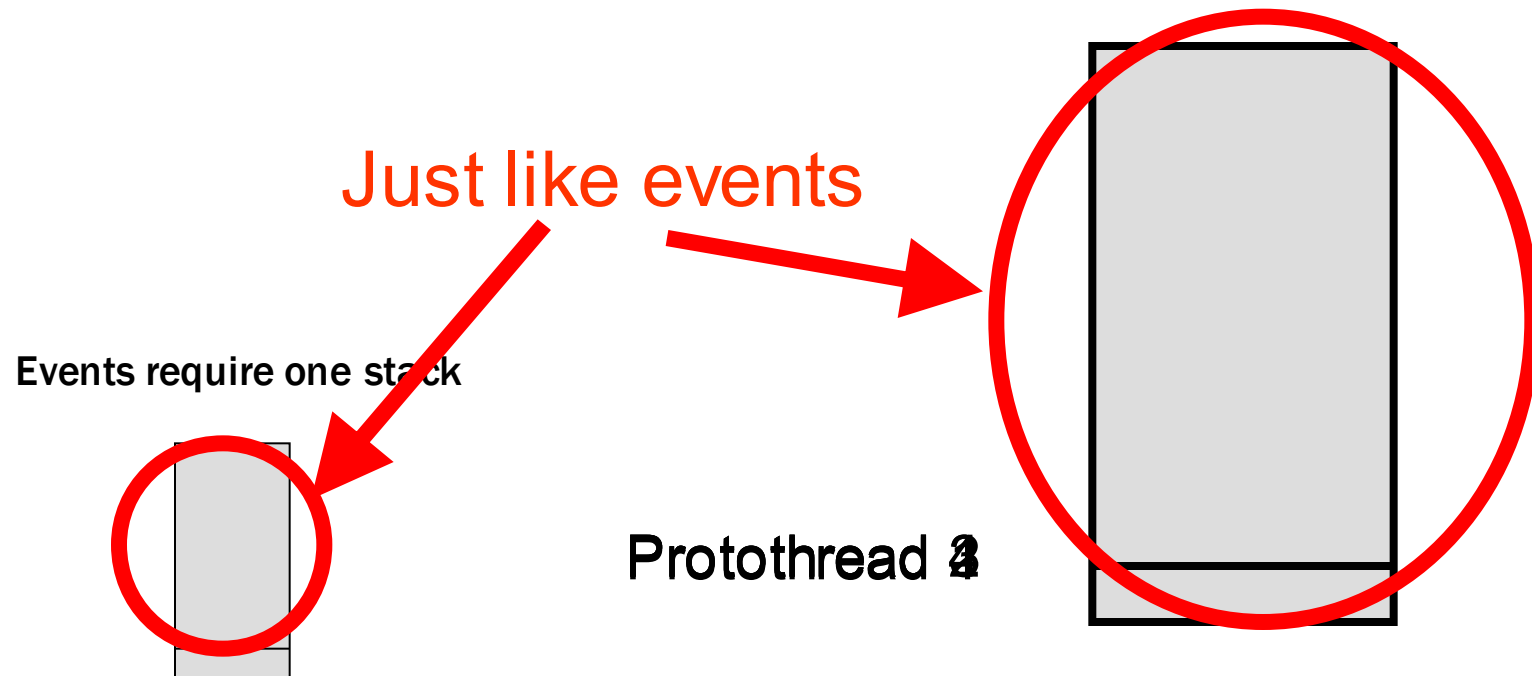
Threads: sequential code flow



Very much like programming with GOTOs

# Protothreads

- Protothreads provides thread-like operations but requires only one stack
- E.g, four protothreads, each with its own stack






# Protothreads in Contiki

- Contiki processes are protothreads

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



# 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
    - Multithreaded programming model