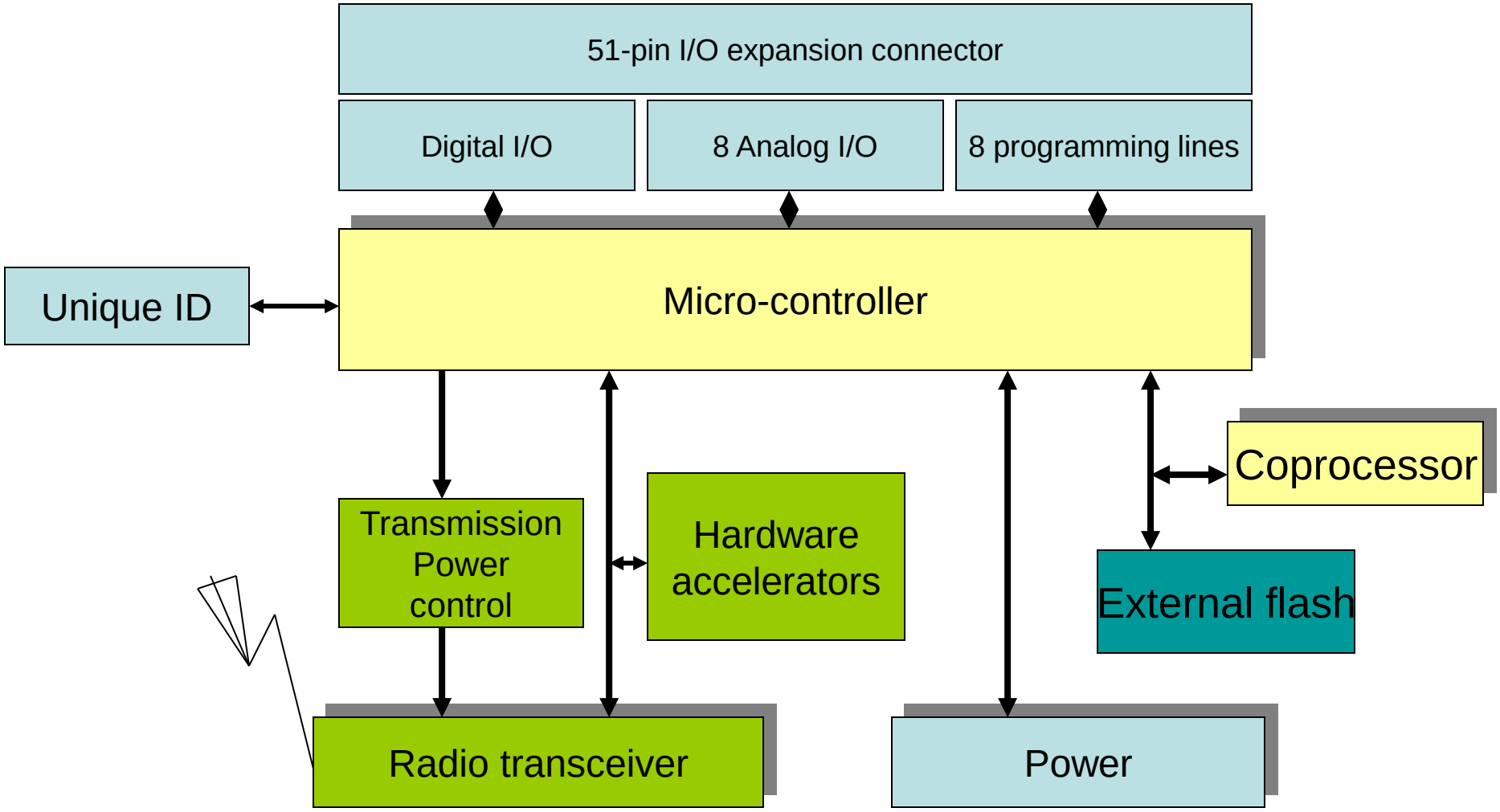
# Lecture 6

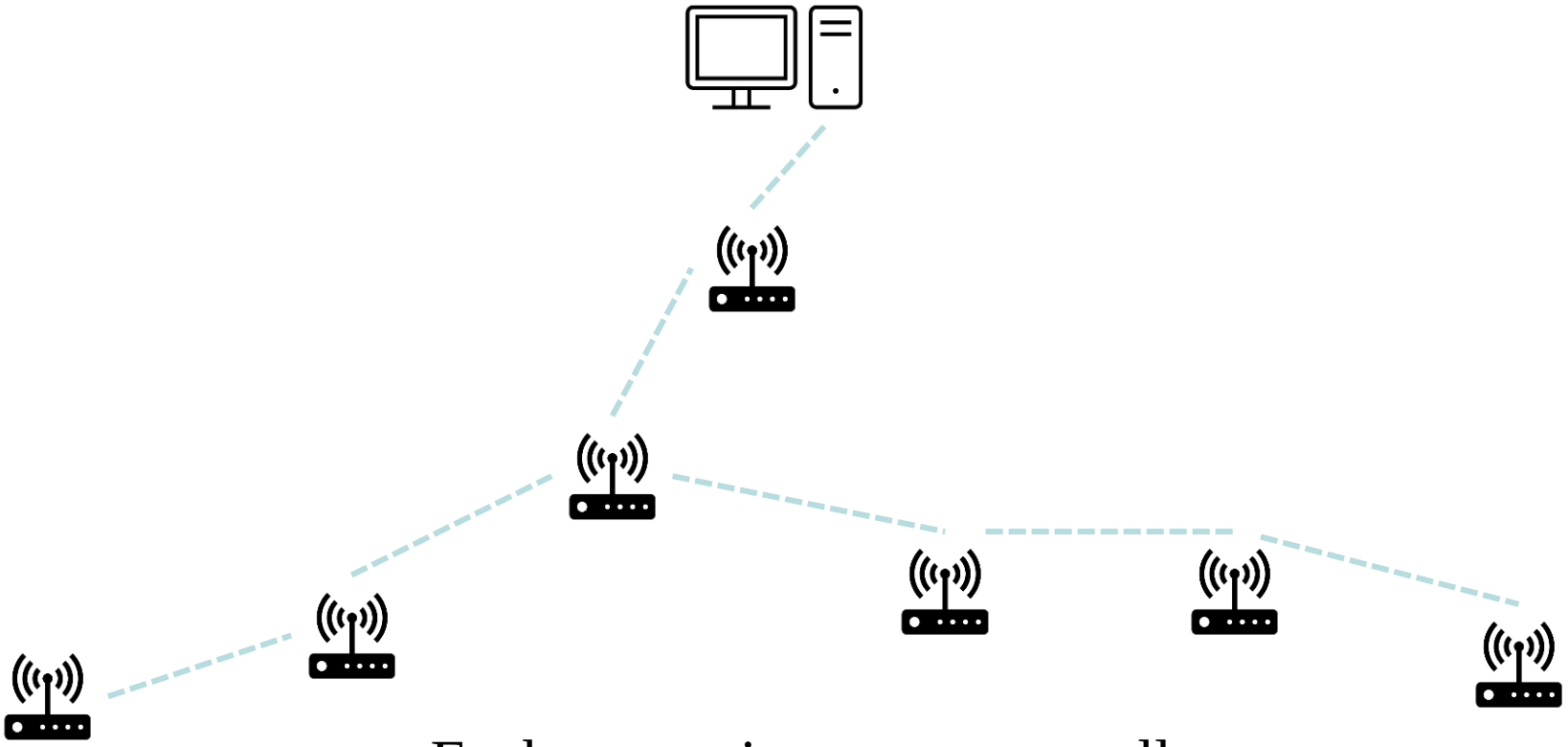
## Processes in a sensor OS

# Sensor device architecture



# Sensor networks

Server receives and stores sensors data.



Each sensor is a router as well.

# TinyOS

- This is an OS that provides a set of system software components for tiny sensors.
- Only the necessary components of the OS are compiled with the application → each application is built into the OS.
- An application wires OS components together with application-specific components – *a complete system consists of a scheduler and a graph of components*.
- A component has four interrelated parts:
  1. a set of command handlers;
  2. a set of event handlers;
  3. a fixed-size frame;
  4. a bundle of tasks.
- Tasks, events and commands execute in the context of the frame and operate on its state.

# A. The programing model

- In TinyOS, tasks (equivalent to processes) and events provide two sources of concurrency.

- A hardware event triggers a processing chain that can go upward and can bend downward by commands.
- To avoid cycles, commands cannot signal events.

- There is only one queue for tasks; the scheduler invokes a new task from the queue when the current task has completed.
- Tasks don’t preempt each other.

- When there is no task in the queue, the scheduler puts the core into the *sleep* mode but not the peripherals.

# Events

- Events are generated by hardware (interrupts).
- The execution of an interrupt handler is called an *event context*.

- The processing of events also run to completion, but it preempts tasks and can be preempted by other events.
- If the task queue is empty, the result of an event is a task being posted to the queue…

- Event handlers should execute very quickly!

# B. Schedulers and tasks

- TinyOS 1.x provided a single kind of task (function) and a single scheduling policy, FIFO.
- The task queue in TinyOS 1.x was implemented as a fixed size circular buffer of function pointers. Posting a task puts the task's function pointer in the next free element of the buffer; if there are no free elements, the post returns *fail*.

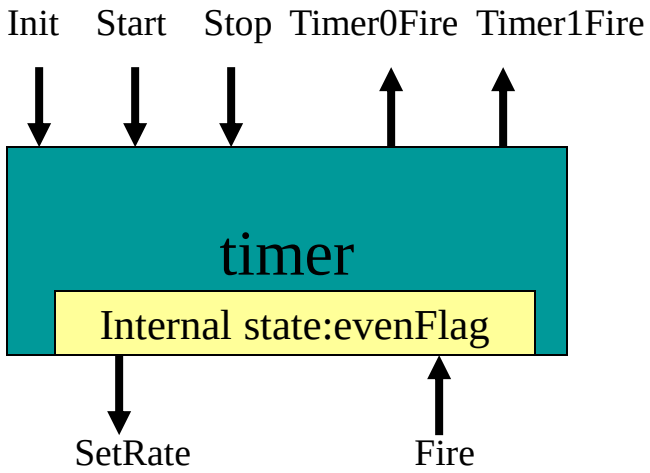
- This model had several problematic issues:
  - some components may not had a reasonable response to a failed post;
  - as a given task can be posted multiple times, it can consume more than one element in the buffer;
  - all tasks from all components share a single resource: one misbehaving component can cause other's posts to fail.

- In TinyOS 2.x, a basic post will only fail if and only if the task has already been posted and has not started execution. A task can always run but can only have one outstanding post at any time.

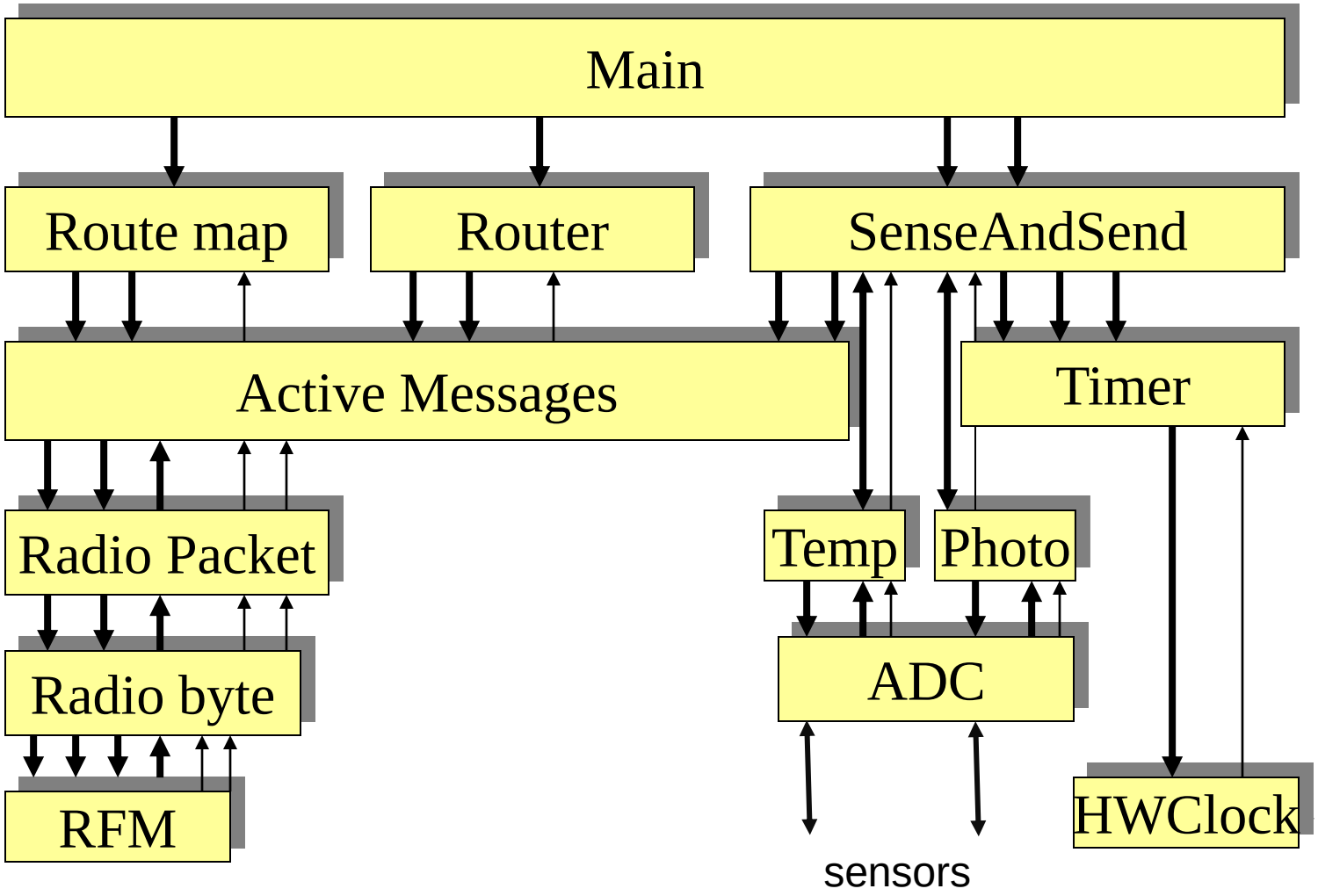
- 2.x introduces task interfaces for additional task models. Task interfaces allow users to extend the syntax and semantics of tasks – use priority, earliest deadline first, etc.

# The Timer TinyOS component

- It works with a lower layer HWClock component. Therefore, it is a software wrapper around a hardware clock that generates periodic interrupts.
- An arrow pointing into the component denotes a call from other components.
- An arrow pointing outside is a call from this component.



# C. The FieldMonitor application



# Split-phase operation

- The split-phase operation separates the initiation of a method call from the return of the call.
- A call to a split-phase operation returns immediately without actually waiting for the operation to be completed; the execution of the operation is scheduled later.
- When the execution finishes, the operation notifies the original caller through a separate method call.

- Example: *packet send method* in Active Messages (AM) component. This is a long operation, involving converting the packets to bytes, then to bits and ultimately driving the RF circuits to send the bits, one by one.

# Resource contention

- Resource contention is handled by explicit rejection of concurrent requests.
- All split-phase operations return Boolean values indicating whether a request to perform the operation is accepted.
- In the previous example, a send() call, when the AM component is still busy, will result in an error signaled by the AM component.
- The caller needs to implement a lock, to remember not to call send until the sendDone() is called. The caller may need a queue as well.

# SenseAndSend implementation

```
Module SenseAndSend {
    provides interface StdControl;
        uses interface ADC;
        uses interface Timer;
        uses interface Send;
}

Implementation {
    bool busy
    norace uint16_t sensorReading;

    command result_t StdControl.init() {
        busy = FALSE;
    }
    event result_t Timer.timer0Fire() {
         bool localBusy;
         atomic {
             localBusy = busy;
             busy = TRUE;
         }
```

```
        if (!localBusy) {
            call ADC.getData();
            return SUCCESS;
        }   else {
            return FAILED;
        }
    }
    task void sendData() {  // send sensorReading
        adcPacket.data = sensorReading;
        call Send.send(&adcPacket, sizeof adcPacket.data);
        return SUCCESS;
    }
    event result_t ADC.dataReady(uint16_t data) {
        sensorReading = data;
        post sendData();
        atomic {
            busy = FALSE;
        }
        return SUCCESS;
     }
……..
}
```

# Conclusions

- The footprint of a sensor OS should be very small. Only the key OS functions are implemented: scheduling, networking protocol(s), timer.
- A sensor OS is event-driven. Events trigger the scheduling and execution of tasks.
- Tasks can’t pre-empt tasks.
- Split-phase allows for asynchronous execution.
- Communication can take place at the same time with sensing.
- Some sensor OS have auto-diagnosing and remote upgrading.