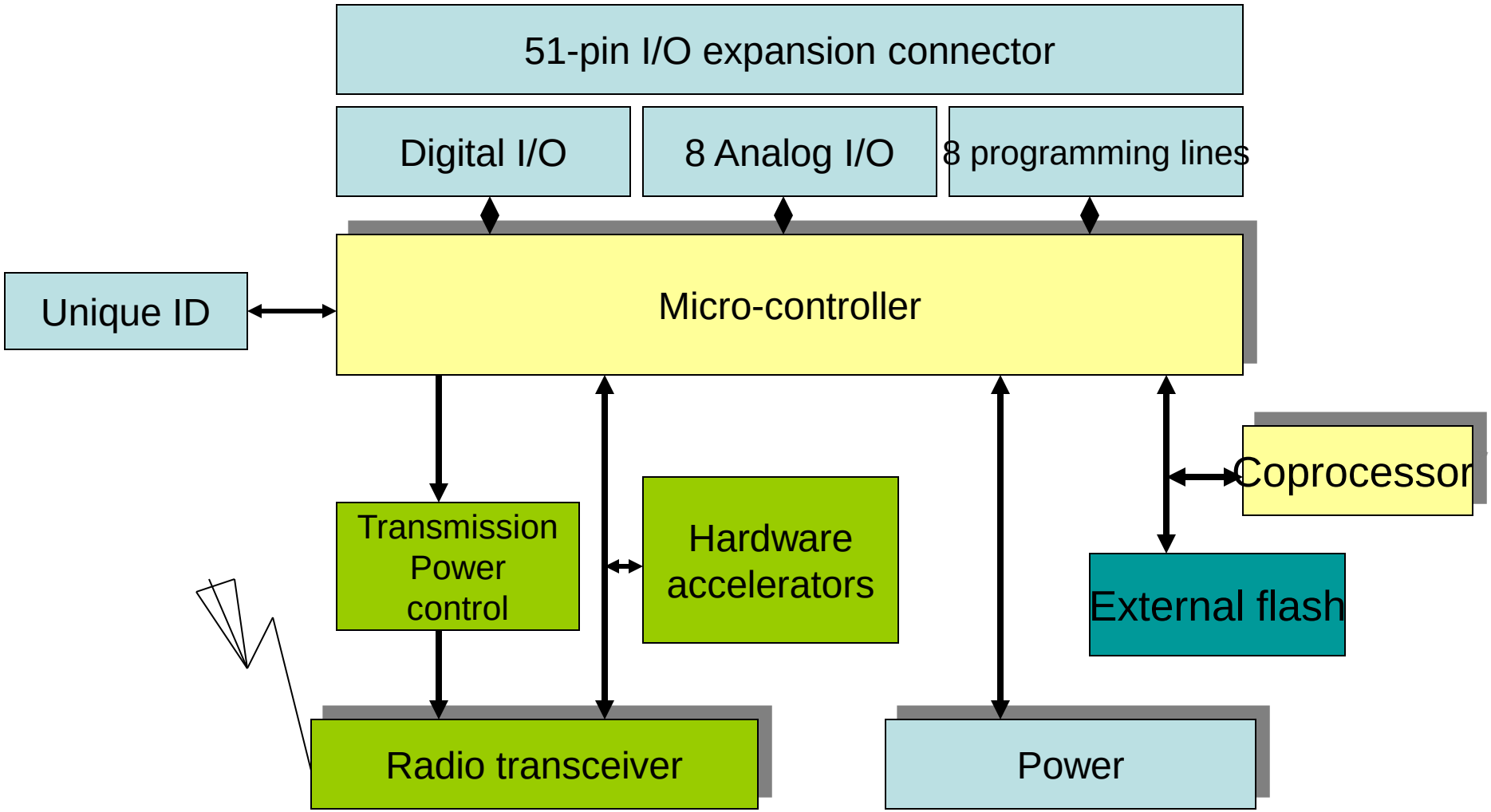
# Lecture 5

## Process management in a sensor OS

# MICA mote architecture



# TinyOS

- This is an OS for tiny sensors that provides a set of system software components.
- Only the necessary parts 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 in a scheduler and a graph of components*.
- A component has four interrelated parts:
  - a set of command handlers,
  - a set of event handlers,
  - a fixed-size frame and
  - a bundle of tasks.
- Tasks, events and commands execute in the context of the frame and operate on its state.

# A. The program model

- In TinyOS, tasks 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.

- Tasks don’t preempt each other.
- The scheduler invokes a new task from the queue only when the current task has completed.

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

# Events

- Events are generated by HW (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, an event has as result a task being scheduled…

- Event handlers should be small!

# B. Schedulers and tasks

- TinyOS 1.x provides a single kind of task, a parameter-free function, and a single scheduling policy, FIFO.
- The task queue in TinyOS 1.x is 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 has several issues:
  - some components do not have 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.

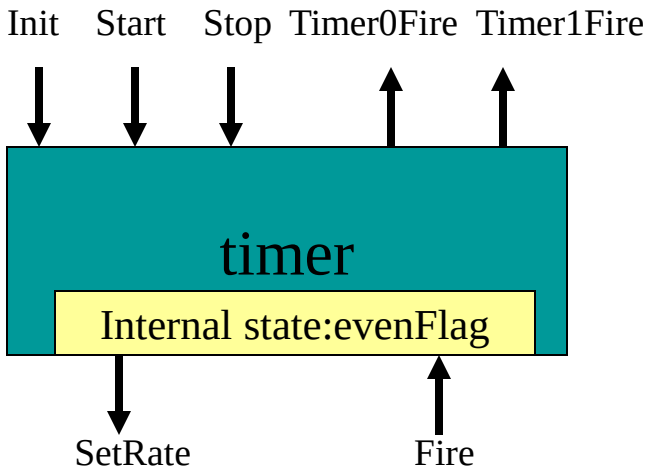
- In TinyOS 2.x, a basic post will only fail if and only if the task has already been posted and has not started execution. A task can always run, but can only have one outstanding post at any time.

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

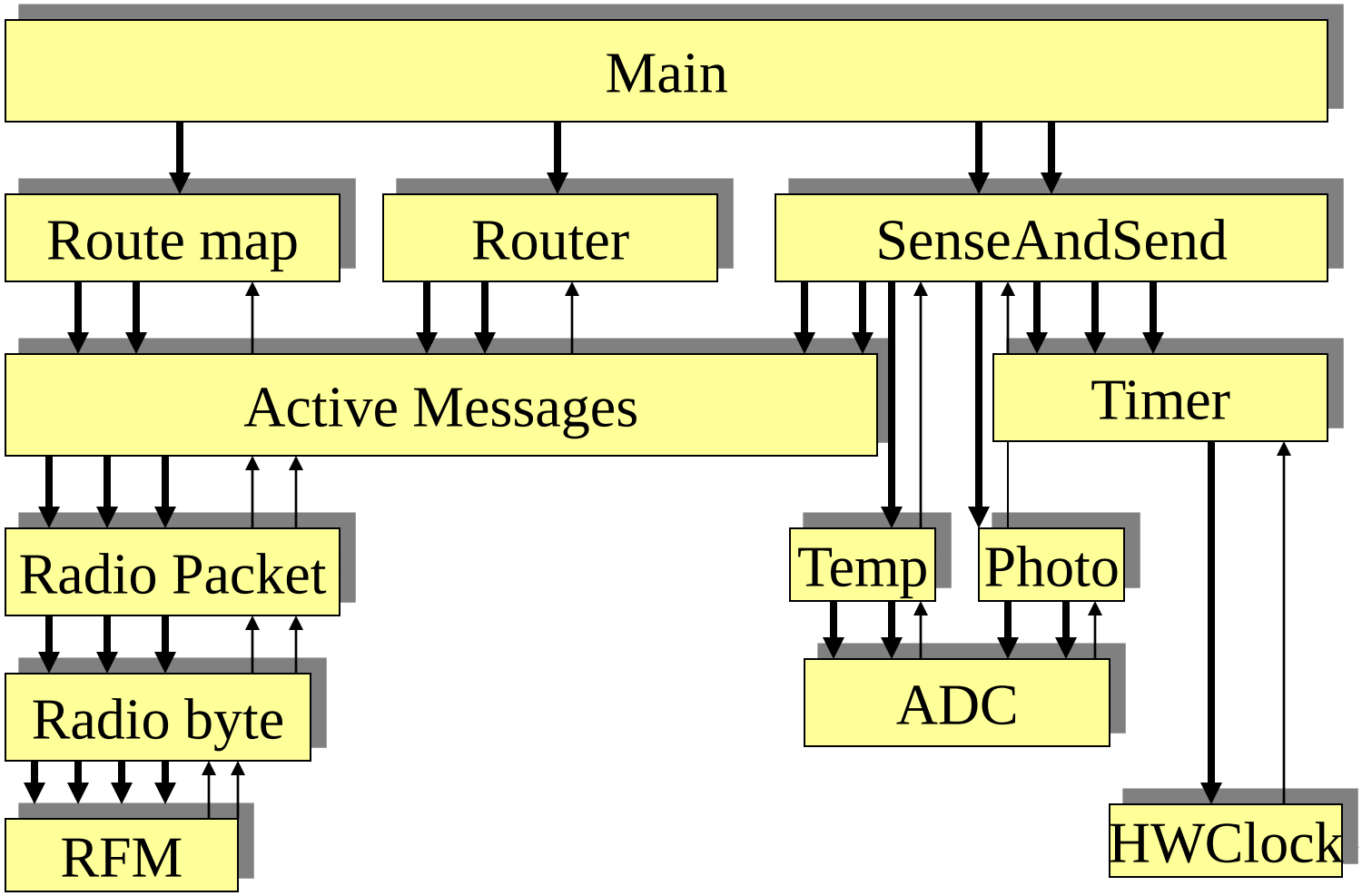
- Using this task interface, a component could post a task with a uint16_t parameter. When the scheduler runs the task, it will signal the runTask event with the passed parameter, which contains the task's logic.

# The Timer TinyOS component

- It works with a lower layer HWClock component = SW wrapper around a HW 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

- 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 performing the body of the operation; 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 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 pending 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;
     }
……..
}
```

# Links

- http://csl.stanford.edu/~pal/pubs/tinyos-programming-1-0.pdf
- www.tinyos.net
- http://sourceforge.net/projects/tinyos

- Lots of sensor sites include sections on OS.