Outline

• Operating Systems for Sensor Nodes: Principles

• Case studies:
  – TinyOS
  – Contiki
  – Reflex

Operating system challenges in WSN

• Usual operating system goals
  – Make access to device resources abstract (virtualization)
  – Protect resources from concurrent access

• Usual means
  – Protected operation modes of the CPU – hardware access only in these modes
  – Process with separate address spaces
  – Support by a memory management unit

• Problem: These are not available in microcontrollers (only partially true)
  – No separate protection modes, no memory management unit
  – Would make devices more expensive, more power-hungry

• Possible options
  – Try to implement “as close to an operating system” on WSN nodes
    • In particular, try to provide a known programming interface
    • Namely: support for processes!
    • Sacrifice protection of different processes from each other
    – Possible, but relatively high overhead
  – Do (more or less) away with operating system
    • After all, 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

• Currently popular verdict: no OS, just a simple run-time environment
  – Enough to abstract away hardware access details
  – Biggest impact: Unusual programming model
Main issue: How to support concurrency

- Simplest option: No concurrency, sequential processing of tasks
  - Not satisfactory: Risk of missing data (e.g., from transceiver) when processing data, etc.
  - Interrupt asynchronous operation has to be supported

- Why concurrency is needed
  - Sensor node’s CPU has to service the radio modem, the actual sensors, perform computation for application, execute communication protocol software, etc.

Traditional concurrency: Processes

- Traditional OS: processes/threads
  - Based on interrupts, context switching
  - But: not available – memory overhead, execution overhead

- But: concurrency mismatch
  - One process per protocol entails too many context switches
  - Many tasks in WSN small with respect to context switching overhead

- And: protection between processes not needed in WSN
  - Only one application anyway

Event-based concurrency

- Alternative: Switch to event-based programming model
  - Perform regular processing or be idle
  - React to events when they happen immediately
  - Basically: interrupt handler

- Problem: must not remain in interrupt handler too long
  - Danger of losing events
  - Only save data, post information that event has happened, then return!
    - Run-to-completion principle
  - Two contexts: one for handlers, one for regular execution

Components instead of processes

- Need an abstraction to group functionality
  - Replacing “processes” for this purpose
  - E.g.: individual functions of a networking protocol

- One option: Components
  - Here: In the sense of TinyOS
  - Typically fulfill only a single, well-defined function
  - Main difference to processes:
    - Component does not have an execution
    - Components access same address space, no protection against each other
  - NOT to be confused with component-based programming!
API to an event-based protocol stack

- Usual networking API: sockets
  - Issue: blocking calls to receive data
  - Ill-matched to event-based OS
  - Also: networking semantics in WSNs not necessarily well matched to/socket semantics

- 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 example discussion below

Components and Publish/Subscribe

- Protocol functions can be considered as components in a flexible computing environment
  - Components fit very well for push based applications
  - E.g. a value can be delivered to the receiver as soon as it is acquired

- For cross-layer aspects often some additional information from other layers is required
  - RSSI value can be used for: routing, FEC/ARQ, location determination (guessing) etc.
  - There might be several subscribers to this kind of information
  - Event based processing is not the best model to process these structures

- Use a Publish/Subscribe paradigm for pull based information
  - Data is put onto a blackboard
  - Each subscriber can access the information at any time independent of the data acquisition

Dynamic power management

- Exploiting multiple operation modes is promising
- Question: When to switch in power-safe mode?
  - Problem: Time & energy overhead associated with wakeup; greedy sleeping is not beneficial
  - Scheduling approach

- Question: How to control dynamic voltage scaling?
  - More aggressive; stepping up voltage/frequency is easier
  - Deadlines usually bind the required speed

- Or: Trading off fidelity vs. energy consumption!
  - If more energy is available, compute more accurate results
  - Example: Polynomial approximation
    - Start from high or low exponents depending where the polynomial is to be evaluated
Outline

• Case study: TinyOS

Case study embedded OS: TinyOS & nesC

• TinyOS developed by UC Berkeley as runtime environment for their "motes"
• nesC as adjunct "programming language"
• Goal: Small memory footprint
  – Sacrifices made e.g. in ease of use, portability
  – Portability somewhat improved in newer version
• Most important design aspects
  – Component-based system
  – Components interact by exchanging asynchronous events
  – Components form a program by stringing them together (akin to VHDL – hardware description language)

TinyOS components

• Components
  – Frame – state information
  – Tasks – normal execution program
  – Command handlers
  – Event handlers
• Handlers
  – Must run to completion
  – From a component’s interface
  – Understand and emit commands & events
• Hierarchically arranged
  – Events pass upward from hardware to higher-level components
  – Commands are passed downward

Handlers versus tasks

• Command handlers and events must run to completion
  – Must not wait an indeterminate amount of time
  – Only a request to perform some action
• Tasks, on the other hand, can perform arbitrary, long computation
  – Also have to be run to completion since no non-cooperative multi-tasking is implemented
  – But can be interrupted by handlers
  – No need for stack management, tasks are atomic with respect to each other
**Split-phase programming**

- Handler/task characteristics and separation has consequences on programming model
  - How to implement a blocking call to another component?
  - Example: Order another component to send a packet
  - Blocking function calls are not an option

! Split-phase programming
- First phase: Issue the command to another component
  - Receiving command handler will only receive the command, post it to a task for actual execution and returns immediately
  - Returning from a command invocation does not mean that the command has been executed!
- Second phase: Invoked component notifies invoker by event that command has been executed
- Consequences e.g. for buffer handling
  - Buffers can only be freed when completion event is received

**Structured commands/events into interfaces**

- Many commands/events can add up
- nesC solution: Structure corresponding commands/events into interface types
- Example: Structure timer into three interfaces
  - StdCtrl
  - Timer
  - Clock
- Build configurations by wiring together corresponding interfaces

**Building components out of simpler ones**

- Wire together components to form more complex components out of simpler ones
- New interfaces for the complex component

**Defining modules and components in nesC**

```nesC
interface StdCtrl {
  command result_t init();
}

interface Timer {
  command result_t start (char type, uint32_t interval);
  command result_t stop ();
  event result_t fired();
}

interface Clock {
  command result_t setRate (char interval, char scale);
  event result_t fired();
}

module TimerComponent {
  provides:
    interface StdCtrl;
    interface Timer;
}

uses interface Clock as Clk;
```
Wiring components to form a configuration

```c
configuration CompleteTimer {
    provides |
    interface StdCtlr;
    interface Timer;
}
implementation (
    components TimerComponent, RMClock;
    StdCtlr = TimerComponent.RMClock;
    Timer = TimerComponent.Timer;
    TimerComponent.Clock = RMClock.Clock;
)
```

Outline

- Case study: Contiki

Contiki

- Small operating system for **tiny networked devices**
  - Developed by Adam Dunkels (SICS)
- Typical tiny device
  - 8-bit microcontroller
  - 1 kilobyte RAM, 10 kilobytes ROM (code)
  - Network: Radio, Ethernet, ...

Contiki

- Ported to ~20 platforms
  - MSP430, 6802, x86, AVR, Z80, PowerPC, ...
  - < 10 by Adam Dunkels, rest by others
- Full TCP/IP support, provided by µIP
  - World's smallest full TCP/IP stack
- Quite well-known
  - 10+ newspaper articles, ...

Contiki – goals

- Portability
- Flexibility
  - Loadable application programs, device drivers
- Multitasking
- Networking (TCP/IP)
- Using little memory!!

Runtime partitioning

- A running Contiki system is partitioned into two parts
  - Core
  - Programs
- Core always present in memory (ROM)
- Programs are loaded at runtime

Partitioning II

Core

- Core, always present in memory:
  - kernel
  - communication drivers
  - parts of language runtime (C library)
  - libraries
- In general not changed after deployment
  - But not impossible to do so
Contiki: event-driven kernel

- Small
  - Compact code (1k)
  - Very low memory requirements (a few bytes)
- Multitasking
  - Each process must have an event handler
  - Less memory than threaded system (threads require stack)
- Inter-process communication
  - Processes post events to each other

Event-driven: control flow

One event handled, then pollhandlers executed

Event-driven: drawbacks

- Hard to program
  - Programs must be implemented as state-machines
  - Modifications cumbersome
- Not suitable for all applications
  - Long running computations may be hard to discretize
- But Contiki also supports threads...

Concurrency in Contiki

- Contiki supports two additional concurrency models
  - Threads: preemptive / non-preemptive
  - Protothreads: stackless threads
- Threads make programming easier (in many cases)
  - Support blocked waiting (calling sleep() in event-based system blocks all execution)
Threads

- Implemented as application library
  - Applied on a per-application basis
- Scheduling on application level
  - Event handler schedules thread
- May be preempted
- Drawbacks
  - Requires stack to be allocated
  - Locking

Protothreads – stackless threads

- Extremely lightweight threads
  - Can only run within a single C function
  - Do not require any stack!
- Support blocked waiting
- Scheduled within event handler, poll handler, TCP/IP handler, etc

Example – Radio Sleep Cycle

1. Turn radio on.
2. Wait until $t = t_0 + t_{awake}$.
3. Turn radio off, but only if all communication has completed.
4. If communication has not completed, wait until it has completed or $t = t_0 + t_{awake} + t_{wait\_max}$.
5. Turn the radio off. Wait until $t = t_0 + t_{awake} + t_{sleep}$.
6. Repeat from step 1

Radio Sleep Cycle Algorithm
Radio Cycle – Event-driven Code

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

void radio_wake_eventhandler() {
    switch(state) {
    case ON:
        if(timer_expired(&timer)) {
            timer_set(&timer, T_SLEEP);
            if(!communication_complete()) {
                state = WAITING;
                timer_set(&wait_timer, T_MAX_WAIT);
            } else {
                radio_off();
                state = OFF;
            }
        }
        break;
    case WAITING:
        if(communication_complete() ||
                timer_expired(&wait_timer)) {
            state = OFF;
            radio_off();
        }
        break;
    case OFF:
        if(timer_expired(&timer)) {
            radio_on();
            state = ON;
            timer_set(&timer, T_AWAKE);
        }
        break;
    }
}
```

Radio Cycle as Protothread

```c
PT_THREAD(radio_wake_thread(struct pt *pt)) {
    PT_BEGIN(pt);
    while(1) {
        radio_on();
        timer_set(&timer, T_AWAKE);
        PT_WAIT_UNTIL(pt, timer_expired(&timer));
        timer_set(&timer, T_SLEEP);
        if(!communication_complete()) {
            timer_set(&wait_timer, T_MAX_WAIT);
            PT_WAIT_UNTIL(pt, communication_complete() ||
                        timer_expired(&wait_timer));
        }
        radio_off();
        PT_WAIT_UNTIL(pt, timer_expired(&timer));
    }
    PT_END(pt);
}
```

Service

- **Service**:  
  - Process implementing functionality that can be used by other processes  
  - Similar to shared library
- **Service consists of**:  
  - Interface (version #; table with pointers to functions)  
  - Process which implements the service
- **Applications use a stub lib for communication**
- **Service layer**:  
  - Finding installed services  
  - Book keeping of running processes  
  - Conceptually sitting directly next to the kernel
Libraries

- Kernel provides only the very minimum of essential features i.e. multiplexing and event handling stuff
- All other functionality is provided via libraries
- Programs and libraries can be linked:
  - Statically with libs that are part of the core
  - Statically with libs that are part of the loadable program
  - Dynamically (no real linking) calling services that implement a certain lib.
- Libraries are likely to be part of:
  - the core: e.g. often used parts of language runtime libs.
  - The loadable program, e.g. application specific libs, or rarely used programming language stuff

Loosely coupled communication stack

- Service calls for stack internal communication
- Events for communication with applications
- Communication modules are realized as services
  - Run-time replacement of modules possible
  - Several communication stacks can be used simultaneously

Programs

- Programs, loaded at runtime
  - Applications
  - May replace drivers, services in core
- Examples
  - Web server
  - Telnet server
  - Event logging
  - Log replicator

Outline

- Case study: Reflex
Introduction

- operating system platform for deeply embedded systems
  - Ram down to 1KB
  - Rom down to 8KB
  - clock a few MHz
  - no memory management unit
- objectives:
  - usability
  - portability
  - real-time support
  - memory efficiency
  - universal applicability
The Event Flow Model

Port Based Object

- Chimera and Chimera II OS [6]
- Boldstroke model used by Boeing [5]
- TinyOS [3]
  - developed for deeply embedded systems
  - C language extension NesC [2]

Call-Based vs. Port-Based Objects

- easy to synchronise
- easy to analyse
- dependability through analysability
- message based communication between components
- scheduling scheme independence
- proposal of port-based objects
- change of view: system now invokes tasks not otherwise

Call-Based in polled and event driven systems

Hon.-Prof. Dr. Peter Langendörfer
2008
- the combination of traits
  - eventdriven
  - synchronized channels with buffer semantic
  - single-threaded execution
  - native object-oriented implementation
- integrated power management scheme

**The Event Flow Model**

**Single-Threaded Execution 1/2**

Figure: stack allocation in multi- and single-threaded systems

**The Event Flow Model**

**Single-Threaded Execution 2/2**

Figure: stack sharing in preemptively scheduled systems

**Implicit Synchronization**

Figure: Synchronization Scenarios

- non-preemptive > only buffers written from interrupt handler
- preemptive > priority of activities determines sync-points
- state-constraint problem [4] easy to handle
The Event Flow Model

- objects are not embedded in complex call graphs
- load is always caused by interrupts

Figure: event flow graph annotated for timing analysis

Implementation

- sync with system and central control
- tagging (what type of interrupt source)

Figure: common interrupt handling scheme

- structure is implemented statically with a typedef
- activities store scheduling related information
- scheduler structure is similar
- per activity scheduling count

Figure: logical structure for activities and schedulers
• eventchannels with data-type-driven interface
  – void notify() for pure events
  – void assign(T data) for events with data

• pre-fabricated event flow buffers
  – SingleValue
  – Fifo
  – Queue

• Distributor for fan out

2 level scheme
  – driver shuts off components if nothing to do
  – scheduler signals idle state to power manager

4 power saving modes
  - wait idle/no power saving requested
  - dream: wake-on-networking
  - sleep: wake-on-RTC
  - stop: wake-on-reset or external interrupt

user activity raises power saving request (dream, sleep, stop)

InterruptGuardian disables all interrupts, except
Sensornetze WS2007/2008

• user activity raises power saving request (dream, sleep, stop)
  • InterruptGuardian disables all interrupts, except
    – wakeup
  – secondary interrupts

• if idle & no pending activities > power down

Implementation
Power Management 2/2

Sensornetze WS2007/2008
user activity raises power saving request (dream, sleep, stop)

- InterruptGuardian disables all interrupts, except
  - wakeup
  - secondary context
- if idle & no pending activities > power down
- a
- wakeup to active state

Implementation

Power Management 2/2

Sensornetze WS2007/2008

Using Reflex

Workflow

- install toolchain
- set up application directory
- implement activities/components if needed
- implement construction plan (NodeConfiguration)
- build
- download

Using Reflex

Construction

Building an application with fixed priority activities

Sensornetze WS2007/2008

Using Reflex

Activity Example

Figure: Scheduling-scheme independent activity implementation

Sensornetze WS2007/2008

\begin{verbatim}
class NodeConfiguration : public system {
    public:
        // methods for configuration
        void setConfiguration();
        int getConfiguration();
    private:
        // variables for configuration
    }

    // this method is called by the scheduler
    void execute();
}
\end{verbatim}
Using Reflex
Sensornetze WS2007/2008

- sophisticated heating control
  - state based Bajorath algorithm used for heating
  - moveable solar panel
- alarm system with comfort functions
  - DCF77 receiver 1.5KB RAM and 15KB ROM
  - next version features
    - Ethernet connection
    - digital temperature sensors

Evaluation

- component based design
  - re-use-able
  - hardware independency
- implicit synchronization
- scheduling framework
- implicit power management
Evaluation

Portability

- less than 100 lines of assembler code per port
  - enabling/disabling interrupts
  - interrupt wrapper
  - sleep states
- driver components concept solves hardware dependencies
- 5 supported microcontrollers
- 7 supported boards
- ports for simulators and guest environments

Evaluation

Real-Time Support

- execution model supports real-time analysis
- implementation cares about interrupt locking times
- special problems which arise in deeply embedded systems are solved
  - bounded timeline (usually 8bit counter)

Evaluation

Memory Consumption 1/2

<table>
<thead>
<tr>
<th></th>
<th>Texas Instruments MSP430</th>
</tr>
</thead>
<tbody>
<tr>
<td>REFLEX (EDF)</td>
<td>84 2274</td>
</tr>
<tr>
<td>REFLEX (FP)</td>
<td>84 2162</td>
</tr>
<tr>
<td>REFLEX (FIFO)</td>
<td>84 1936</td>
</tr>
<tr>
<td>TinyOS</td>
<td>230 3876</td>
</tr>
<tr>
<td>RamRom</td>
<td>230 3876</td>
</tr>
</tbody>
</table>

Table: memory consumption of base system

<table>
<thead>
<tr>
<th></th>
<th>on</th>
<th>off</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>1976</td>
<td>2196</td>
</tr>
<tr>
<td>Texas Instruments</td>
<td>1220 35</td>
<td></td>
</tr>
<tr>
<td>HEFLEX (EDF)</td>
<td>1936 84</td>
<td></td>
</tr>
<tr>
<td>HEFLEX (FP)</td>
<td>2142 84</td>
<td></td>
</tr>
<tr>
<td>HEFLEX (FIFO)</td>
<td>1276 84</td>
<td></td>
</tr>
</tbody>
</table>

Evaluation

Memory Consumption 2/2

<table>
<thead>
<tr>
<th></th>
<th>Texas Instruments MSP430</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-Mac</td>
<td>840 8592</td>
</tr>
<tr>
<td>DumpSensors</td>
<td>264 6082</td>
</tr>
<tr>
<td>TinyChat</td>
<td>236 4796</td>
</tr>
<tr>
<td>HelloWorld</td>
<td>160 3466</td>
</tr>
<tr>
<td>n-Mac</td>
<td>2889 18076</td>
</tr>
</tbody>
</table>

Table: memory consumption for different applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Texas Instruments MSP430</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reflex</td>
<td>76 364</td>
</tr>
<tr>
<td>TinyChat</td>
<td>230 3876</td>
</tr>
<tr>
<td>S-Mac</td>
<td>840 8592</td>
</tr>
<tr>
<td>DumpSensors</td>
<td>264 6082</td>
</tr>
<tr>
<td>TinyChat</td>
<td>236 4796</td>
</tr>
<tr>
<td>HelloWorld</td>
<td>160 3466</td>
</tr>
<tr>
<td>n-Mac</td>
<td>2889 18076</td>
</tr>
</tbody>
</table>

* size in bytes for Texas Instruments MSP430
**Evaluation**

- complex applications already implemented
- used for the Cocos framework
- SDL-to-Reflex mapping successfully done at IHP
- integration in Siemens-SPS-environment in progress

**Conclusion & Outlook**

- Reflex meets its targets
- an IDE would be nice
- validation of power management scheme
- putting remote update and power management into a release
- online monitoring/debug interface

**Outline**

- **OS Comparison**

**TinyOS versus Reflex**

- Implementation Language
  - TinyOS:
    - NesC; Components do not have state
      - Special compiler and preprocessor needed
  - Reflex:
    - C++; full multi-threaded stateful components
      - Standard C++ tool-chain applicable
- System Configuration
  - TinyOS:
    - Components; Handlers; Frame
    - Command Handlers; Event Handlers; Tasks
    - Commands run to completion; Handlers cannot be interrupted
    - Connections are done during compile time
  - Reflex:
    - Objects
      - Objects are connected to objects via event channels (an abstract object class)
      - Connections are done during runtime
TinyOS versus Reflex 2

- **Scheduling**
  - **TinyOS**
    - Simple FIFO scheme;
      - no preemption possible but via interrupts;
    - No nested interrupts possible to avoid stack management.
    - No Hard Realtime possible
  - **Reflex**
    - Flexible and dynamic Scheduling,
      - Three schedulers are available already and can be extended and amended.
    - Preemption possible but no nested or recursive call (will lock in run method)
  - **Event Flow Support**
    - **TinyOS**
      - Simple parameterless event call chain
      - No implicit synchronization
  - **Reflex**
    - Event channels via synchronized buffers
      - Buffers are typed and transparent to allow high flexibility

TinyOS versus Reflex 3

- **Codesize comparison:**

<table>
<thead>
<tr>
<th>Application</th>
<th>TinyOS</th>
<th>Reflex</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorld</td>
<td>2762</td>
<td>2620</td>
</tr>
<tr>
<td>TinyChat</td>
<td>11988</td>
<td>5504</td>
</tr>
<tr>
<td>DumbSensors</td>
<td>12182</td>
<td>6002</td>
</tr>
<tr>
<td>S.Mac</td>
<td>25963</td>
<td>6592</td>
</tr>
</tbody>
</table>

Summary

- For WSN, the need to build cheap, low-energy, (small) devices has various consequences for system design
  - Radio frontends and controllers are much simpler than in conventional mobile networks
  - Energy supply and scavenging are still (and for the foreseeable future) a premium resource
  - Power management (switching off or throttling down devices) crucial
- Unique programming challenges of embedded systems
  - Concurrency without support, protection
  - De facto standard: TinyOS
  - TinyOS has some big drawbacks:
    - nesC: Special language; needs precompiler;
    - Portability
  - Other operating systems: Reflex, Kontiki
- Can we build an efficient OS for WSN applications with the used comfort of ordinary OS?

Acknowledgements

- Special thank’s for sharing their material goes to:
  - Thiemo Voigt, SICS
  - Karsten Walther, BTU Cottbus
  - Holger Karl, University Paderborn