Embedded Software: The Case of Sensor Networks

Graduate Student: Dimitrios Lymberopoulos
Instructor: A. Silberschatz
Outline

- Basic Concepts of Embedded Software – Black Box

- The case of Sensor Networks
  - Hardware Overview
  - Software for Sensor Networks
    - TinyOS
    - NesC
    - Demo using Berkeley’s Mica2 motes!
    - PaIOS
    - TinyGALS
  - Re-programmability Issues
    - Maté
    - SensorWare

- Conclusions – Open research problems
Basic Concepts

Main Features
- Timeliness
- Concurrency
- Liveness
- Heterogeneity
- Reactivity
- Robustness
- Low power
- Scaleable

User Input

Interaction with the physical world

Output

Embedded Software
Basic Concepts

- Embedded Software is not software for small computers.

- It executes on machines that are not computers (cars, airplanes, telephones, audio equipment, robots, security systems…)

- Its principal role is not the transformation of data but rather the interaction with the physical world.

- Since it interacts with the physical world must acquire some properties of the physical world. It takes time. It consumes power. It does not terminate until it fails.
Basic Concepts – More Challenges

- The engineers that write embedded software are rarely computer scientists.

- The designer of the embedded software should be the person who best understands the physical world of the application.

- Therefore, better abstractions are required for the domain expert in order to do her job.

- On the other hand, applications become more and more dynamic and their complexity is growing rapidly.
Outline

Basic Concepts of Embedded Software – Black Box

The case of Sensor Networks

- Hardware Overview
- Software for Sensor Networks
  - TinyOS
  - NesC
  - Demo using Berkeley’s Mica2 motes!
  - PaLOS
  - TinyGALS
- Re-programmability Issues
  - Maté
  - SensorWare

Conclusions – Open research problems
Why Sensor Networks?

- Sensor networks meet all the challenges that were previously described (Event driven, concurrent, robust, real time, low power…)

- In addition sensor nodes have to exchange information using wireless communication by forming a network.

- Communication is expensive.
What is a Sensor Network?

- A sensor network is composed of a large number of sensor nodes which are densely deployed in a region.

- Sensor nodes are small in size, low-cost, low-power multifunctional devices that can communicate in short distances.

- Each sensor node consists of sensing, data processing and communication components and contains its own limited source of power.

- Sensor nodes are locally carry out simple computations and transmit only the required and partially processed data.
Outline

- Basic Concepts of Embedded Software – Black Box

- The case of Sensor Networks
  - Hardware Overview
  - Software for Sensor Networks
    - TinyOS
    - NesC
    - Demo using Berkeley’s Mica2 motes!
    - PaiOS
    - TinyGALS
  - Re-programmability Issues
    - Maté
    - SensorWare

- Conclusions – Open research problems
## Hardware Platforms for Sensor Networks

### The Berkeley “Motes” family

<table>
<thead>
<tr>
<th>Mote Type</th>
<th>WeC</th>
<th>Renee</th>
<th>Mica</th>
<th>Mica2</th>
<th>Mica2Dot</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="WeC Image" /></td>
<td><img src="image2.png" alt="Renee Image" /></td>
<td><img src="image3.png" alt="Mica Image" /></td>
<td><img src="image4.png" alt="Mica2 Image" /></td>
<td><img src="image5.png" alt="Mica2Dot Image" /></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Microcontroller</th>
<th>WeC</th>
<th>Renee</th>
<th>Mica</th>
<th>Mica2</th>
<th>Mica2Dot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>AT90LS8535</td>
<td>Atmega163</td>
<td>Atmega128</td>
<td>Atmega128</td>
<td>Atmega128</td>
</tr>
<tr>
<td>CPU Clock (Mhz)</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>7.3827</td>
<td>4</td>
</tr>
<tr>
<td>Program Memory (KB)</td>
<td>8</td>
<td>16</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Ram (KB)</td>
<td>0.5</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>UARTs</td>
<td>1</td>
<td>1</td>
<td>2 (only 1 used)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SPI</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>I2C</td>
<td>Software</td>
<td>Software</td>
<td>Software</td>
<td>Hardware</td>
<td>Hardware</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonvolatile storage</th>
<th>WeC</th>
<th>Renee</th>
<th>Mica</th>
<th>Mica2</th>
<th>Mica2Dot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chip</td>
<td>24LC256</td>
<td></td>
<td></td>
<td>AT45DB041B</td>
<td></td>
</tr>
<tr>
<td>Size (KB)</td>
<td>32</td>
<td></td>
<td></td>
<td>512</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Radio Communication</th>
<th>WeC</th>
<th>Renee</th>
<th>Mica</th>
<th>Mica2</th>
<th>Mica2Dot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radio</td>
<td>RFM TR1000</td>
<td></td>
<td>Chipcon CC1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frequency</td>
<td>916 (single freq)</td>
<td></td>
<td>916/433 (multiple channels)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radio speed (kbps)</td>
<td>OOK</td>
<td>ASK</td>
<td>FSK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transmit Power Control</td>
<td>Programmable resistor potentiometer</td>
<td>Programmable via CC1000 registers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encoding</td>
<td>SecDed (software)</td>
<td>Manchester (hardware)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Objectives: Low idle time – Stay in inactive mode for as much time as possible
Hardware Platforms for Sensor Networks

UCLA’s MK-II platform

- PALOS Core
- ARM/THUMB 40MHz
- Running uCos-ii
- RS-485 & External Power
- ADXL 202E MEMS Accelerometer
- MCU I/F Host Computer, GPS, etc
- UI: Pushbuttons
Outline

- Basic Concepts of Embedded Software – Black Box

- The case of Sensor Networks
  - Hardware Overview
  - Software for Sensor Networks
    - TinyOS
    - NesC
    - Demo using Berkeley’s Mica2 motes!
    - PaiOS
    - TinyGALS
  - Re-programmability Issues
    - Maté
    - SensorWare

- Conclusions – Open research problems
Hardware Platforms for Sensor Networks

- Sensor network hardware platforms are resource constrained but at the same time they must be very reactive and participate in complex distributed algorithms.

- Traditional operating systems and programming models are inappropriate for sensor networks (and for embedded systems).
TinyOS

- Designed for low power Adhoc Sensor Networks (initially designed for the WesC Berkeley motes)

- Key Elements
  - Sensing, Computation, Communication, Power

- Resource Constraints
  - Power, Memory, Processing

- Adapt to Changing Technology
  - Modularity & Re-use
TinyOS

- Event oriented OS
- Multithreading
- Two-level scheduling structure
TinyOS – Main Idea

- **Hurry up and Sleep**

- **Execute Processes Quickly**
  - Interrupt Driven

- **Sleep Mode**
  - Sleep (μWatt power) while waiting for something to happen
TinyOS Memory Model

- **STATIC**
  - No HEAP (malloc)
  - No FUNCTION Pointers

- **Global Variables**
  - Conserve memory
  - Use pointers, don’t copy buffers

- **Local Variables**
  - On Stack
TinyOS Structure

Each component has four interrelated parts:

1. A set of command handlers
2. A set of event handlers
3. Simple tasks
4. An encapsulated fixed-size frame

Each component declares the commands it uses and the events it signals (modularity)

Applications are layers of components where higher level components issue commands to lower level components and lower level components signal events to higher level components
TinyOS Structure

- Commands are non-blocking requests made to lower level components. They deposit request parameters into their frames and post a task for later execution.

- Event handlers are invoked to deal with hardware events.

- Tasks perform the primary work. They are atomic with respect to other tasks and run to completion. They can be preempted by events.

- Commands, events and handlers execute in the context of the frame and operate on its state.
TinyOS Process Categories

- **Events**
  - Time Critical
  - Interrupts cause Events (timer, ADC)
  - Small/Short duration
  - Interrupt Tasks

- **Tasks**
  - Time Flexible
  - Run sequentially by TinyOS Scheduler
  - Run to completion with other Tasks
  - Interruptible
TinyOS Kernel

Tiny OS Kernel

TOS KERNEL SCHEDULER

TASK #1
TASK #2
TASK #3

IF TASK QUE EMPTY then SLEEP

INTERRUPT

ISR

EVENT Handler

Call Command(B)

Event Return

Return

Command(s)

Task Return

No TASKS pending

Call Command(A)

COMMAND

CALL(S)

SMARER SENSORS IN SILICON
TinyOS Application Example

**Drawback**: Concurrency model designed around radio bit sampling
TinyOS Application Evaluation (1)

- Scheduler only occupies 178 bytes
- Complete application only requires 3 KB of instruction memory and 226 bytes of data (less than 50% of the 512 bytes available)
- Only processor_init, TinyOS scheduler, and C runtime are required

<table>
<thead>
<tr>
<th>Component Name</th>
<th>Code Size (bytes)</th>
<th>Data Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing</td>
<td>88</td>
<td>0</td>
</tr>
<tr>
<td>AM_dispatch</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>AM_temperature</td>
<td>78</td>
<td>32</td>
</tr>
<tr>
<td>AM_light</td>
<td>146</td>
<td>8</td>
</tr>
<tr>
<td>AM</td>
<td>356</td>
<td>40</td>
</tr>
<tr>
<td>RADIO_packet</td>
<td>334</td>
<td>40</td>
</tr>
<tr>
<td>RADIO_byte</td>
<td>810</td>
<td>8</td>
</tr>
<tr>
<td>RFM</td>
<td>310</td>
<td>1</td>
</tr>
<tr>
<td>Light</td>
<td>84</td>
<td>1</td>
</tr>
<tr>
<td>Temp</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>UART</td>
<td>196</td>
<td>1</td>
</tr>
<tr>
<td>UART_packet</td>
<td>314</td>
<td>40</td>
</tr>
<tr>
<td>I2C</td>
<td>198</td>
<td>8</td>
</tr>
<tr>
<td>Processor_init</td>
<td>172</td>
<td>30</td>
</tr>
<tr>
<td>TinyOS scheduler</td>
<td>178</td>
<td>16</td>
</tr>
<tr>
<td>C runtime</td>
<td>82</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>3450</td>
<td>226</td>
</tr>
</tbody>
</table>
## TinyOS Application Evaluation (2)

<table>
<thead>
<tr>
<th>Operations</th>
<th>Cost (cycles)</th>
<th>Time (µs)</th>
<th>Normalized to byte copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte copy</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Post an Event</td>
<td>10</td>
<td>2.5</td>
<td>1.25</td>
</tr>
<tr>
<td>Call a Command</td>
<td>46</td>
<td>11.5</td>
<td>6</td>
</tr>
<tr>
<td>Context switch overhead</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Post a task to scheduler</td>
<td>51</td>
<td>12.75</td>
<td>6</td>
</tr>
<tr>
<td>Interrupt (hardware cost)</td>
<td>9</td>
<td>2.25</td>
<td>1</td>
</tr>
<tr>
<td>Interrupt (software cost)</td>
<td>71</td>
<td>17.75</td>
<td>9</td>
</tr>
</tbody>
</table>
## TinyOS

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Multithreading and Event-driven operating system</td>
<td>• HW/SW boundary adjustment would significantly reduce power consumption and efficiency</td>
</tr>
<tr>
<td>• Low memory requirements (small footprint)</td>
<td>• Programmers have to deal with the asynchronous nature of the system. Difficult to write programs</td>
</tr>
<tr>
<td>• Offers Modularity, Reusability</td>
<td></td>
</tr>
</tbody>
</table>

➢ Lack of communication among tasks.

**Note:** NesC programming model addresses most of these disadvantages!
NesC – The TinyOS Language

A programming language specifically designed for TinyOS
- Dialect of C
- Variables, Tasks, Calls, Events, Signals
- Component Wiring

A pre-processor
- NesC output is a C program file that is compiled and linked using gnu gcc tools
NesC – TinyOS

- **Component**
  - Building block of TinyOS
  - An entity that performs a specific set of services
  - Can be “wired together” (Configured) to build more complex Components
    - Implementation in a module (code)
    - Wiring of other components in a **Configuration**

- **Configuration**
  - A “Wiring” of components together
TinyOS Component Structure

- **Interface**
  - Declares the services provided and the services used

- **Implementation**
  - Defines internal workings of a Component
  - May include “wires” to other components

- **Component Types**
  - Modules
  - Configurations
Interface Elements

- **Commands**
  - Provides services to User

- **Events**
  - Sends Signals to the User

- **Mandatory (Implicit) Commands**
  - `.init` – invoked on boot-up
  - `.start` – enables the component services
  - `.stop` – halt or disable the component
Commands and Signals

- **Commands**
  - Similar to C functions
  - Pass parameters
  - Control returns to caller
  - Flow downwards

- **Signals**
  - Triggers an **Event** at the connected Component
  - Flow upwards
  - Pass parameters
  - Control returns to Signaling Component
Events and Tasks

**EVENTS**

- Hardware event handlers are executed in response to a hardware interrupt and always run to completion.
- May preempt the execution of a task or other hardware interrupt.
- Commands and events that are executed as part of a hardware event handler must be declared with the `async` keyword.

**TASKS**

- Functions whose execution is deferred.
- Once scheduled (started)
  - Run to completion.
  - Do not preempt one another (executed sequentially).
Data Race Conditions

- Tasks may be preempted by other asynchronous code.

- Races are avoided by:
  - Accessing shared data exclusively within tasks.
  - Having all accesses within \texttt{atomic} statements.

- The NesC compiler reports potential data races to the programmer at compile time.

- Variables can be declared with the \texttt{norace} keyword (should be used with extreme caution).
Advanced Topics on Information Systems

TinyOS messaging

- A standard message format is used for passing information between nodes

- Messages include: Destination Address, Group ID, Message Type, Message Size and Data.

```c
#define TOSH_DATA_LENGTH 29
typedef struct TOS_Msg{
    uint16_t addr;
    uint8_t type;
    uint8_t group;
    uint8_t length;
    int8_t data[TOSH_DATA_LENGTH];
    uint16_t crc;

    //Extra
    uint16_t strength;
    uint8_t ack;
    uint16_t time;
    uint8_t sendSecurityMode;
    uint8_t receiveSecurityMode;
} TOS_Msg;
```

TOS Message
- 36 Bytes

Extension passed from MAC layer
- 12 Bytes
Active Messaging

- Each message on the network specifies a **HANDLER ID** in the header.
- **HANDLER ID** invokes specific handler on recipient nodes
- When a message is received, the **EVENT** wired that **HANDLER ID** is signaled
- Different nodes can associate different receive event handlers with the same **HANDLER ID**
BLINK: A Simple Application

- A simple application that toggles the red led on the Berkeley mote every 1 sec.
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;
  Main.StdControl -> BlinkM.StdControl;
  Main.StdControl -> SingleTimer.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
interface StdControl { 
  command result_t init();
  command result_t start();
  command result_t stop();
}
BlinkM.nc
module BlinkM {
    provides {
        interface StdControl;
    }
    uses {
        interface Timer;
        interface Leds;
    }
}
implementation {
    command result_t StdControl.init() {
        call Leds.init();
        return SUCCESS;
    }
    command result_t StdControl.start() {
        return call Timer.start(TIMER_REPEAT, 1000);
    }
    command result_t StdControl.stop() {
        return call Timer.stop();
    }
    event result_t Timer.fired() {
        call Leds.redToggle();
        return SUCCESS;
    }
}

Timer.nc
interface Timer {
    command result_t start(char type, uint32_t interval);
    command result_t stop();
    event result_t fired();
}
Demo: Surge

- Goal 1: create a tree routed at the base station
- Goal 2: Each node uses the most reliable path to the base station

- Reliability
  - Quality: Link yield to parent
  - Yield: % of data packets received
  - Prediction: Product of quality metrics on all links to base station
Demo: Surge

- Each node broadcasts its cost: Parent Cost + Link’s cost to parent
- Nodes try to minimize total cost
- Each node reports its receive link quality from each neighbor
- Data packets are acknowledged by parents
- Data packets are retransmitted up to 5 times
Demo: Surge

Does it work?
Outline

- Basic Concepts of Embedded Software – Black Box

- The case of Sensor Networks
  - Hardware Overview
  - **Software for Sensor Networks**
    - TinyOS
    - NesC
    - Demo using Berkeley’s Mica2 motes!
    - PalOS
    - TinyGALS
  - Re-programmability Issues
    - Maté
    - SensorWare

- Conclusions – Open research problems
PalOS

PalOS Core

TASK 1

TASK 2

TASK 3

TASK 4

TASK 5

TASK N

Manager

Drivers (Hardware Abstraction Layer)
PalOS Core

- Processor independent algorithms

- Provides means of managing event queues and exchanging events among tasks

- Provides means of task execution control (slowing, stopping, and resuming)

- Supports a scheduler: periodic, and aperiodic functions can be scheduled
A task belongs to the PalOS main control loop

Each task has an entry in PalOS task table (along with eventQs)
Events are exchanged using the service provided by PALOS core.
Periodic or aperiodic events can be scheduled using Delta Q and Timer Interrupt

When event expires appropriate event handler is called
// main loop
while (1) { // run each task in order
    for (i=0; i< globalTaskID; i++) {
        isExact = qArray[i].isExactTiming;
        tmpCntr=qArray[i].execCounter;
        if ( tmpCntr != TASK_DISABLED) {/* task is not disabled */
            if ( tmpCntr ) { /* counter hasn't expired */
                if (!isExact)
                    qArray[i].execCounter--;
            }
            else { /* exec counter expired */
                if (isExact)
                    PALOSSCHED_TIMER_INTR_DISABLE;
                qArray[i].execCounter = qArray[i].reloadCounter;
                if (isExact)
                    PALOSSCHED_TIMER_INTR_ENABLE;
                /* run the task routine */
                (*qArray[i].taskHandler)();
            }
        }
    }
}
PalOS vs. TinyOS

- Notion of well defined tasks
- Inter-task communication through the use of separate event queues
- Multiple tasks can be periodically or not scheduled
- Easier to debug (minimum use of macros)
Outline

- Basic Concepts of Embedded Software – Black Box

- The case of Sensor Networks
  - Hardware Overview
  - **Software for Sensor Networks**
    - TinyOS
    - NesC
    - Demo using Berkeley’s Mica2 motes!
    - PaiOS
    - **TinyGALS**
  - Re-programmability Issues
    - Maté
    - SensorWare

- Conclusions – Open research problems
TinyGALS

- **Globally Asynchronous and Locally Synchronous** programming model for event-driven embedded systems.
- A TinyGALS program contains a single system composed of modules, which are in turn composed of components (two levels of hierarchy).
- Components are composed locally through synchronous method calls to form modules (Locally synchronous).
- Asynchronous message passing is used between modules to separate the flow of the control (Globally asynchronous).
- All asynchronous message passing code and module triggering mechanisms can be automatically generated from a high-level specification.
TinyGUYS (GUarded Yet Synchronous variables)

- Mechanism for sharing global state
- All global variables are guarded and modules can read them synchronously
- Writes are asynchronous in the sense that all writes are buffered
- The buffer is of size one, so the last module that writes to a variable wins
- TinyGUYS variables are updated by the scheduler only when it is safe
- TINYGUYS have global names which are mapped to the parameters of each module which in turn are mapped to the external variables of the components.
- Components can access global variables by using the special keywords: `PARAM_GET()` and `PARAM_PUT()`
TinyGALS code generation example

Advantages
- Application specific code is automatically generated
- Masks the asynchrony of the system
- Easier to write programs

Disadvantages
- Generated code is not optimized
- Use of FIFOS increases memory requirements
Outline

- Basic Concepts of Embedded Software – Black Box
- The case of Sensor Networks
  - Hardware Overview
  - Software for Sensor Networks
    - TinyOS
    - NesC
    - Demo using Berkeley’s Mica2 motes!
    - PaI0S
    - TinyGALS
  - Re-programmability Issues
    - Maté
    - SensorWare
- Conclusions – Open research problems
Why Re-programmability?

- What if there is a bug in the software running on the sensor nodes?
- What if we want to change the algorithm that the sensor network is running?
- Once deployed, sensor nodes cannot be easily collected. In some cases they cannot even be reached.
- Therefore, re-programmability should not require physical contact (recall that communication is expensive...)
Maté

- A tiny communication-centric virtual machine for sensor networks
- Instruction set was designed to produce more complex actions with fewer instructions (assembly like)
- Code is divided into 24 single-byte instructions (capsules) to fit into one tinyOS packet

Maté architecture

- 3 execution contexts (run concurrently)
- Shared state between contexts
Maté: Code Infection

- A capsule contains:
  1. 24 single-byte instructions
  2. Numeric ID: 0,1,2,3 (subroutines), 4,5,6 (clock, send, receive)
  3. Version Information

- If Maté receives a more recent version of a capsule, installs it and forwards it using the \textit{forw} instruction, to its neighbors.

- A capsule can forward other capsules using the \textit{forwo} instruction.
Maté: Execution Model

- Execution begins in response to an event (timer going off, send or received message)

- Control jumps to the first instruction of the corresponding capsule and executes until it reaches the *halt* instruction

- Each instruction is executed as a tinyOS task

### Advantages

- Masks the asynchrony of the system
- Easier to write programs

### Disadvantages

- Processing Overhead
- Complex applications cannot be built
- No multi-user support

Power Consumption is not always reduced!
Outline

- Basic Concepts of Embedded Software – Black Box

- The case of Sensor Networks
  - Hardware Overview
  - Software for Sensor Networks
    - TinyOS
    - NesC
    - Demo using Berkeley’s Mica2 motes!
    - PaiOS
    - TinyGALS
  - Re-programmability Issues
    - Maté
    - SensorWare

- Conclusions
SensorWare

- Dynamically program a sensor network as a *whole*, not just as a collection of individual nodes.

- SensorWare is a framework that defines, creates, dynamically deploys, and supports mobile scripts that are autonomously populated.

- Goals:
  1. How can you express a distributed algorithm?
  2. How can you dynamically deploy a distributed algorithm?
Idea: Make the node environment scriptable

- Define basic building commands (i.e., send packets, get data from sensors)
- Define constructs that tie these building blocks in control scripts

A script implementation of an algorithm

Corresponding low level tasks

- Send packet
- Access radio
- Find route
- Check energy
- Queue packet
SensorWare: Make Scripts Mobile

- Scripts can populate/migrate
- Scripts move due to node’s state and algorithmic instructions and **NOT** due to explicit user instructions
SensorWare: An example

User is notified for presence of target

User reacts by injecting a tracking script
SensorWare: An example

Script migrates and populates into the area of interest

User gets periodic updates of target location

Scripts exchange information to compute target location
SensorWare: An example

As target moves, scripts are migrating

User still is notified regularly
The Framework

Sensor node 1
Services
Scripts
SensorWare
RTOS
HW abstraction layer
Hardware

User can inject script

Code migration

Message exchanging

Sensor node 2
Scripts
Services
SensorWare
RTOS
HW abstraction layer
Hardware
SensorWare Language

SensorWare = Language + Runtime Environment

The glue core
The basic script interpreter (stripped-down Tcl)

Mobility API

Networking API

Timer API

Sensing API

wait command

id command

Optional GPS API

Unknown device API

Will the command set be expandable?
Execution Model

Initialization

Event handler a

Event handler b

Event handler c

Exit code

wait for event a or b or c

Example
SensorWare Trade-offs

- Capabilities-related
  1. Portability

- Energy-related
  1. SensorWare needs memory (180KB)
  2. Slower Execution
     → 8% slowdown for a typical application
  3. Compactness of code
     → 209 bytes for a typical application
     → 764 bytes the equivalent native code

- Security-Related
  1. Security problems
SensorWare - Overview

- Script-based framework
- Hide details from the programmer
- Implemented around the HP iPAQ 3670

Main Features

1. Distributed computational model for sensor networks
2. Simple multi-user taskable interface for sensor networks
Outline

- Basic Concepts of Embedded Software – Black Box
- The case of Sensor Networks
  - Hardware Overview
  - Software for Sensor Networks
    - TinyOS
    - NesC
    - Demo using Berkeley’s Mica2 motes!
    - PaLOS
    - TinyGALS
  - Re-programmability Issues
    - Maté
    - SensorWare

- Conclusions
Sensor Networks

What can be done?

- Only software optimization techniques have been proposed so far

  Hardware?

  Hardware/Software boundary?

- Develop domain specific hardware that can support a distributed computational model similar to SensorWare

- Adjust the hardware/software boundary to increase the performance of this distributed computational model
Sensor Networks

What can be done?

- TinyOS
  - improve the inter-task communication
  - Support on-the-fly component addition/removal

SensorWare

Development of a secure distributed programming model

Maintenance and tasking model to support experiments