Embedded Software: The Case of Sensor Networks

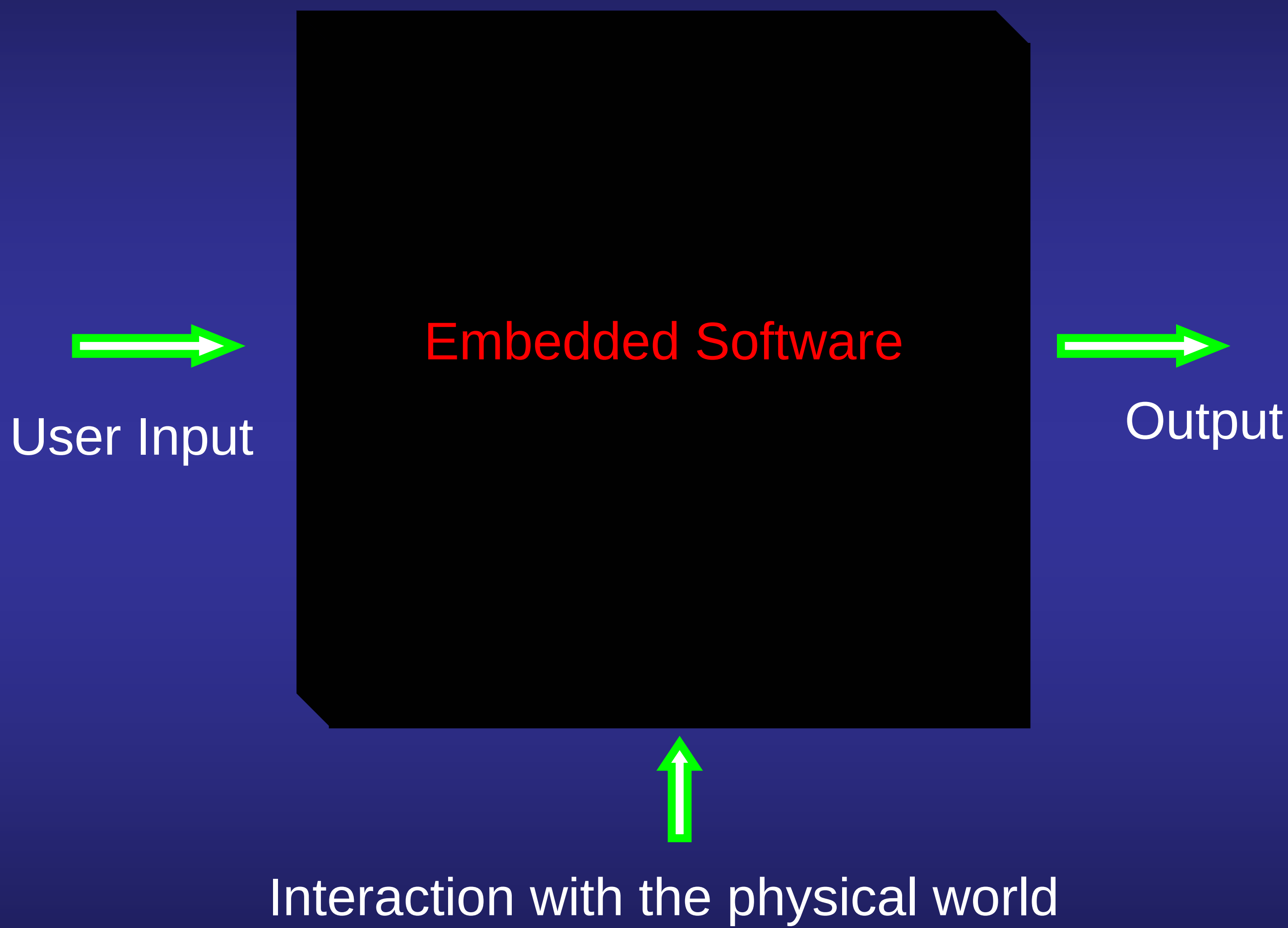
Graduate Student: Dimitrios Lymberopoulos

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!
    - PalOS
    - TinyGALS
  - Re-programmability Issues
    - Maté
    - SensorWare
- Conclusions – Open research problems

Basic Concepts



<u>Main Features</u>

- ✓ Timeliness
- ✓ Concurrency
- ✓ Liveness
- ✓ Heterogeneity
- ✓ Reactivity
- ✓ Robustness
- ✓ Low power
- ✓ Scaleable

Basic Concepts

- Embedded Software is not software fro 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?

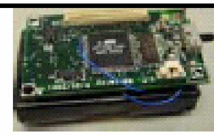
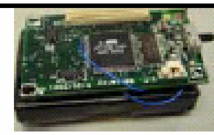
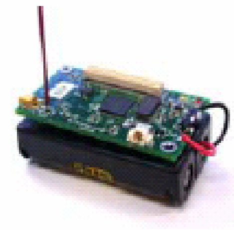
- A sensor network is composed of a large number of sensor nodes which are densely deployed in a region
- Sensor nodes are small in size, low-cost, low-power multifunctional devices that can communicate in short distances
- Each sensor node consists of sensing, data processing and communication components 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!
    - PalOS
    - TinyGALS
  - Re-programmability Issues
    - Maté
    - SensorWare
- Conclusions – Open research problems

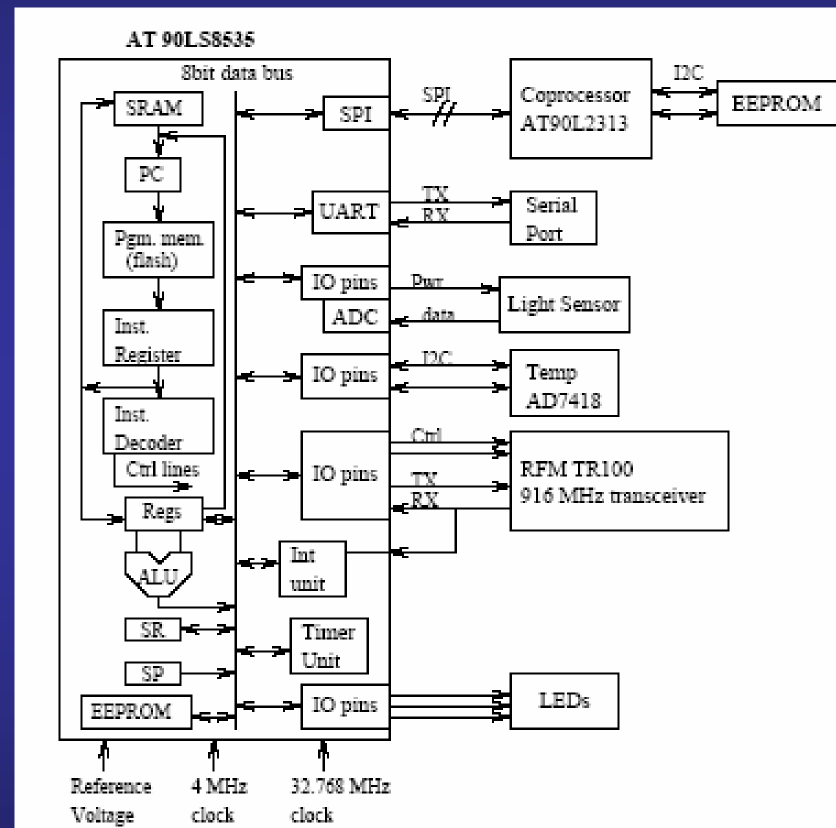
Hardware Platforms for Sensor Networks

The Berkeley "Motes" family

Mote Type	WeC	Renee	Mica	Mica2	Mica2Dot
					
Microcontroller					
Type	AT90LS8535	Atmega163	Atmega128	Atmega128	Atmega128
CPU Clock (Mhz)	4	4	4	7.3827	4
Program Memory (KB)	8	16	128	128	128
Ram (KB)	0.5	1	4	4	4
UARTs	1	1	2 (only 1 used)	2	2
SPI	1	1	1	1	1
I2C	Software	Software	Software	Hardware	Hardware
Nonvolatile storage					
Chip	24LC256		AT45DB041B		
Size (KB)	32		512		
Radio Communication					
Radio	RFM TR1000			Chipcon CC1000	
Frequency	916 (single freq)			916/433 (multiple channels)	
Radio speed (kbps)	OOK		ASK	FSK	
Transmit Power Control	Programmable resistor potentiometer			Programmable via CC1000 registers	
Encoding	SecDed (software)			Manchester (hardware)	

Hardware Platforms for Sensor Networks

WeC Berkeley "Mote" architecture

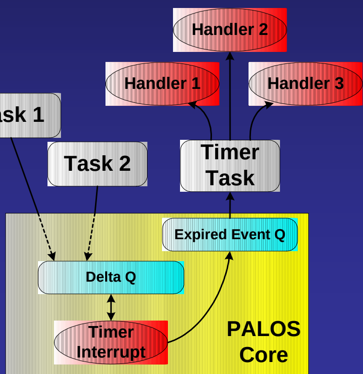


Component	Active (mA)	Idle (mA)	Inactive (μA)
MCU core (AT90S8535)	5	2	1
MCU pins	1.5	-	-
LED	4.6 each	-	-
Photocell	.3	-	-
Radio (RFM TR1000)	12 tx	-	5
Radio (RFM TR1000)	4.5 rx	-	5
Temp (AD7416)	1	0.6	1.5
Co-proc (AT90LS2343)	2.4	.5	1
EEPROM (24LC256)	3	-	1

Objectives: Low idle time – Stay in inactive mode for as much time as possible

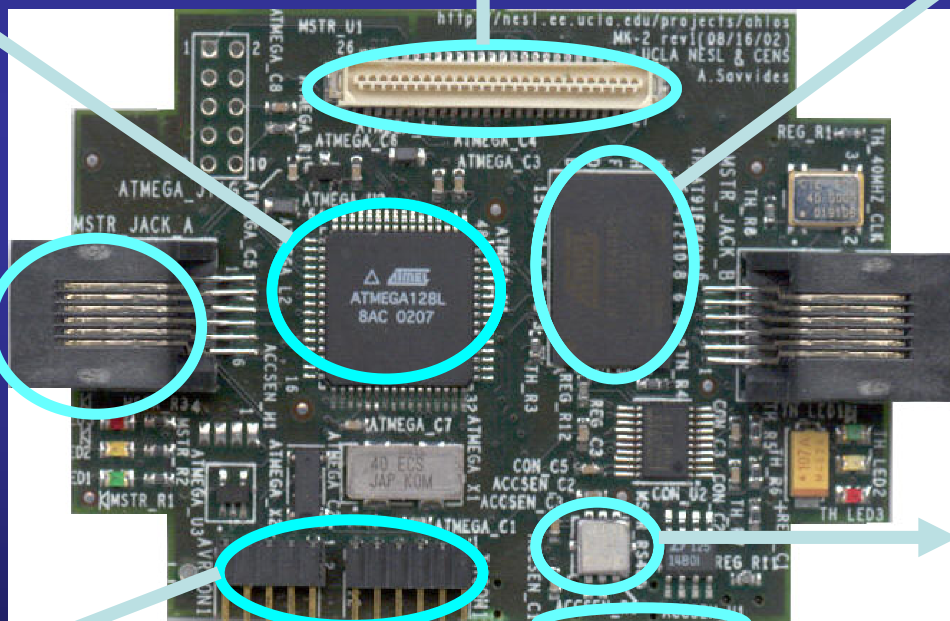
Hardware Platforms for Sensor Networks

UCLA's MK-II platform



PALOS

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!**
    - PalOS
    - 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

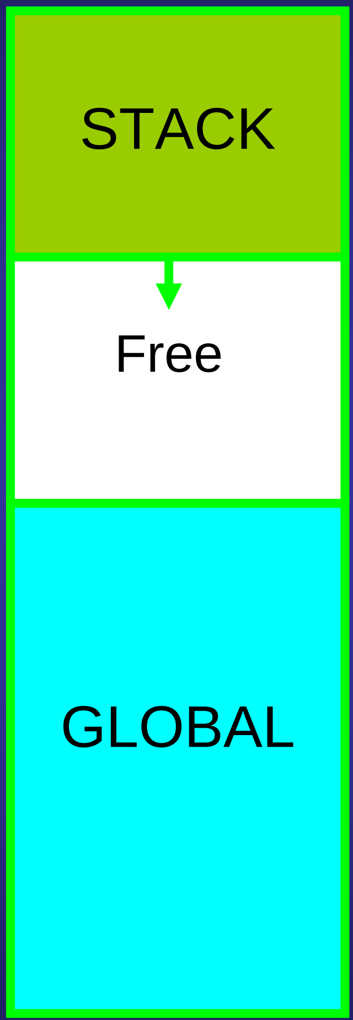
- Event oriented OS
- Multithreading
- Two-level scheduling structure

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

TinyOS

Tiny scheduler | Graph of components

- 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

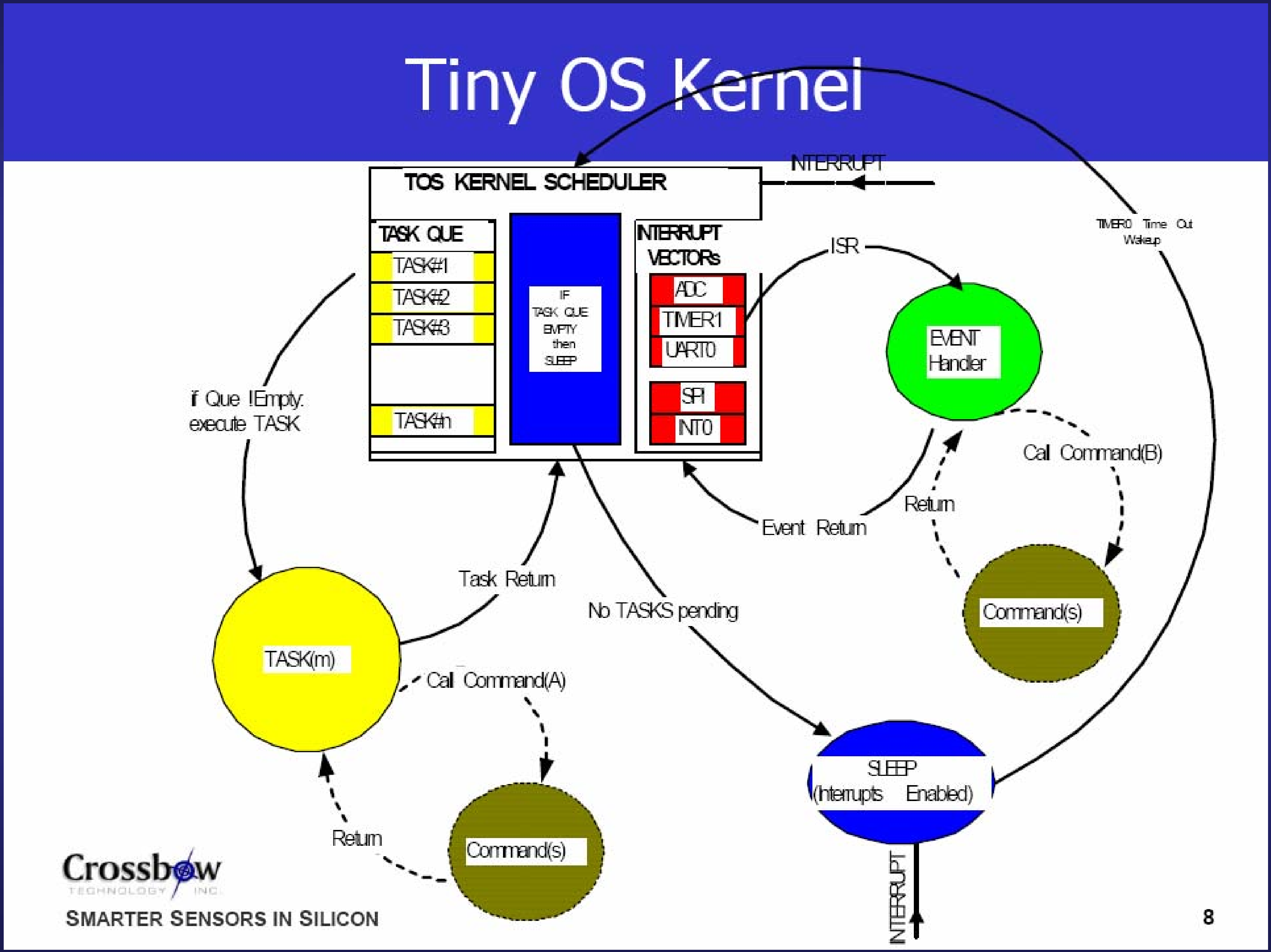
TinyOS Structure

- 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.

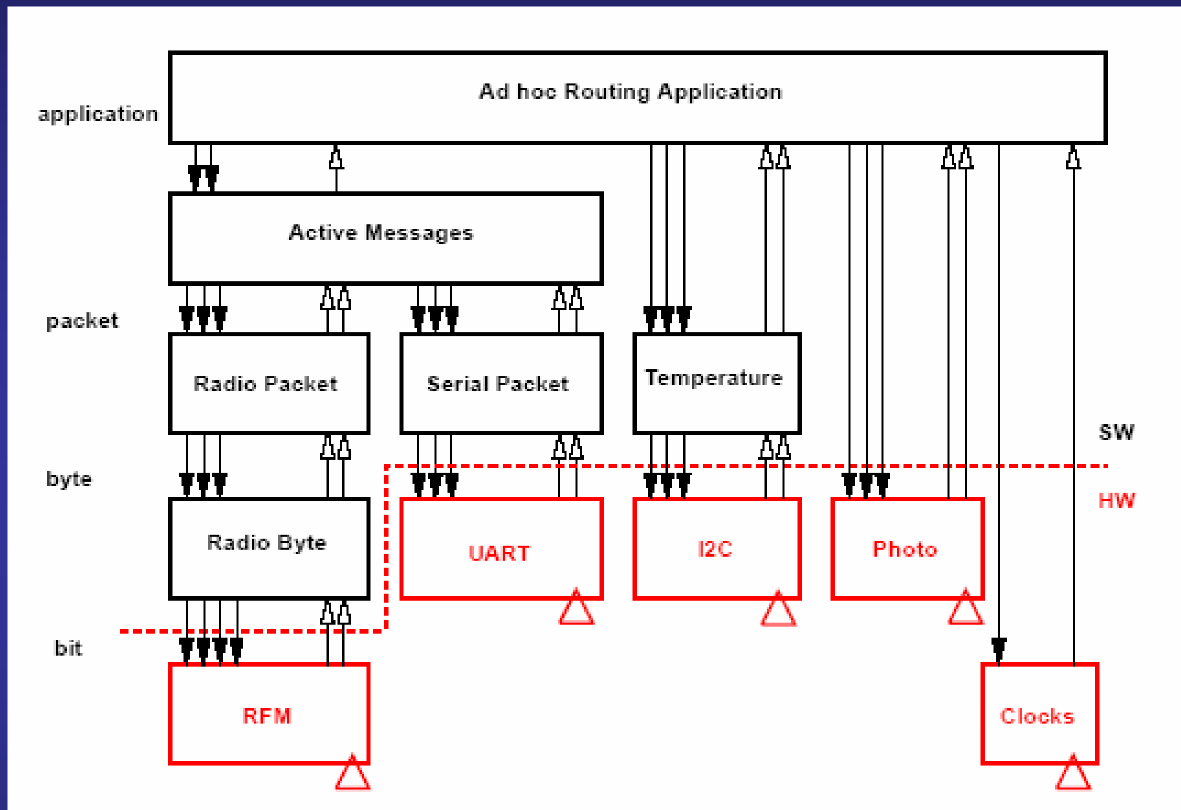
TinyOS Process Categories

- **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



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

Component Name	Code Size (bytes)	Data Size (bytes)
Routing	88	0
AM_dispatch	40	0
AM_temperature	78	32
AM_light	146	8
AM	356	40
RADIO_packet	334	40
RADIO_byte	810	8
RFM	310	1
Light	84	1
Temp	64	1
UART	196	1
UART_packet	314	40
I2C	198	8
Processor_init	172	30
TinyOS scheduler	178	16
C runtime	82	0
Total	3450	226

TinyOS Application Evaluation (2)

Operations	Cost (cycles)	Time (µs)	Normalized to byte copy
Byte copy	8	2	1
Post an Event	10	2.5	1.25
Call a Command	10	2.5	1.25
Post a task to scheduler	46	11.5	6
Context switch overhead	51	12.75	6
Interrupt (hardware cost)	9	2.25	1
Interrupt (software cost)	71	17.75	9

TinyOS

Advantages

- Multithreading and Event-driven operating system
- Low memory requirements (small footprint)
- Offers Modularity, Reusability

Disadvantages

- HW/SW boundary adjustment would significantly reduce power consumption and efficiency
- Programmers have to deal with the asynchronous nature of the system. Difficult to write programs

➢ 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

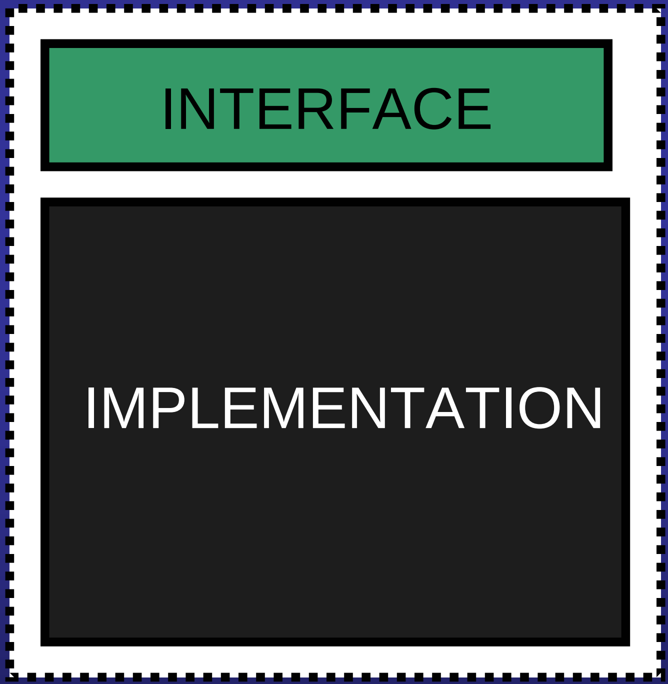
NesC – TinyOS

- **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

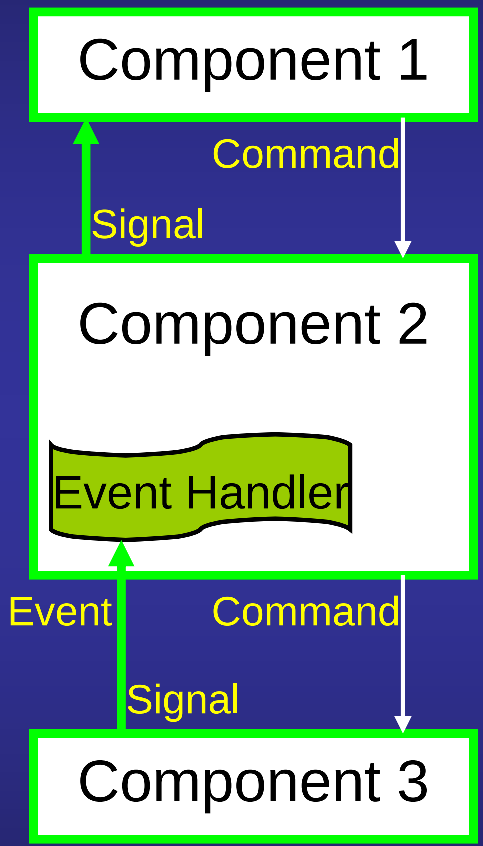


Interface Elements

- **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 **atomic** statements
- The NesC compiler reports potential data races to the programmer at compile time
- Variables can be declared with the **norace** keyword (should be used with extreme caution)

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.

```
#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 1sec.

BLINK: A Simple Application

Blink.nc

```
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;

  Main.StdControl -> BlinkM.StdControl;
  Main.StdControl -> SingleTimer.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
```

StdControl Interface

StdControl.nc

```
interface StdControl {
  command result_t init();
  command result_t start();
  command result_t stop();
}
```

BLINK NesC Code

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

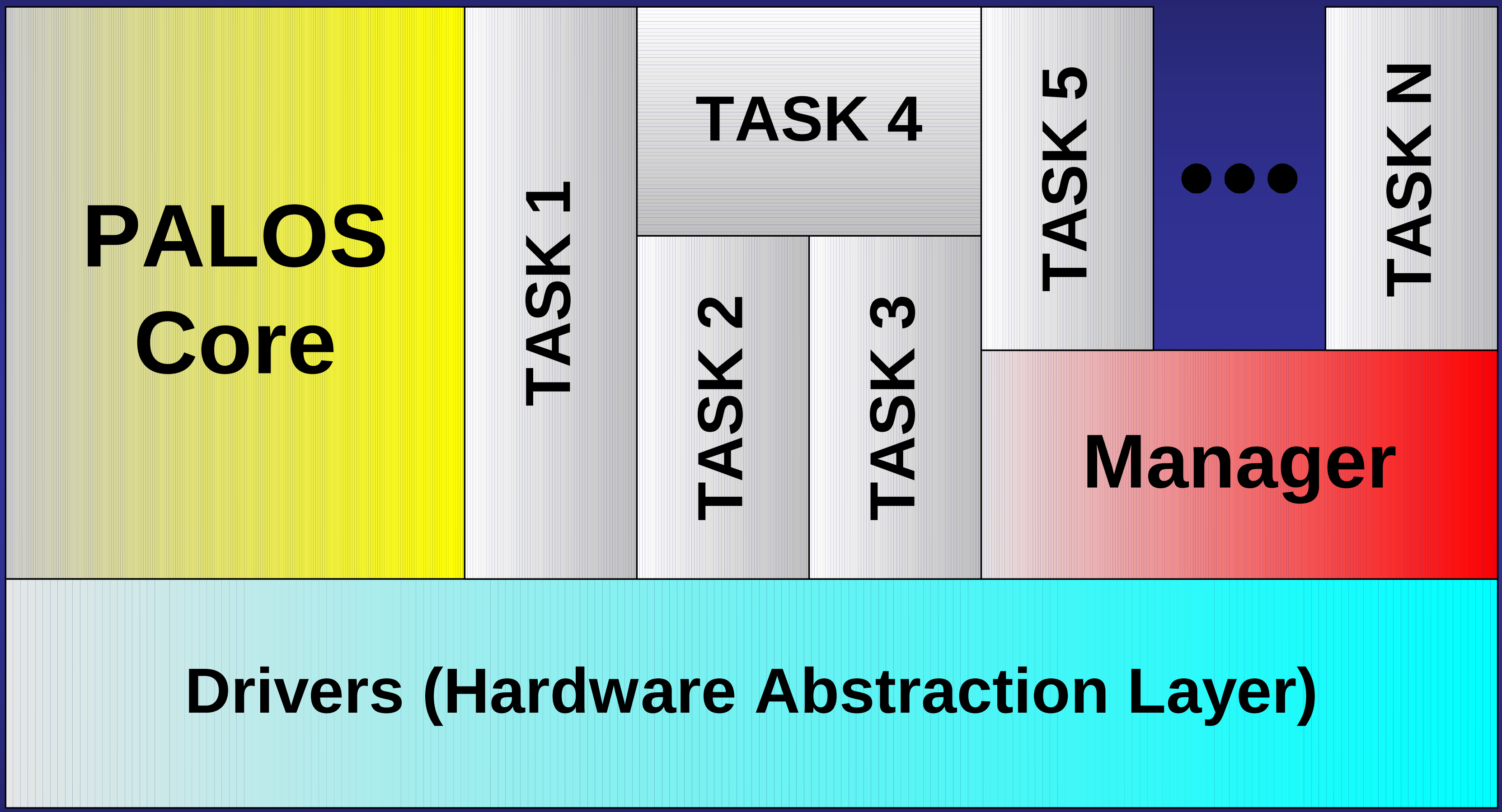
Demo: Surge

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

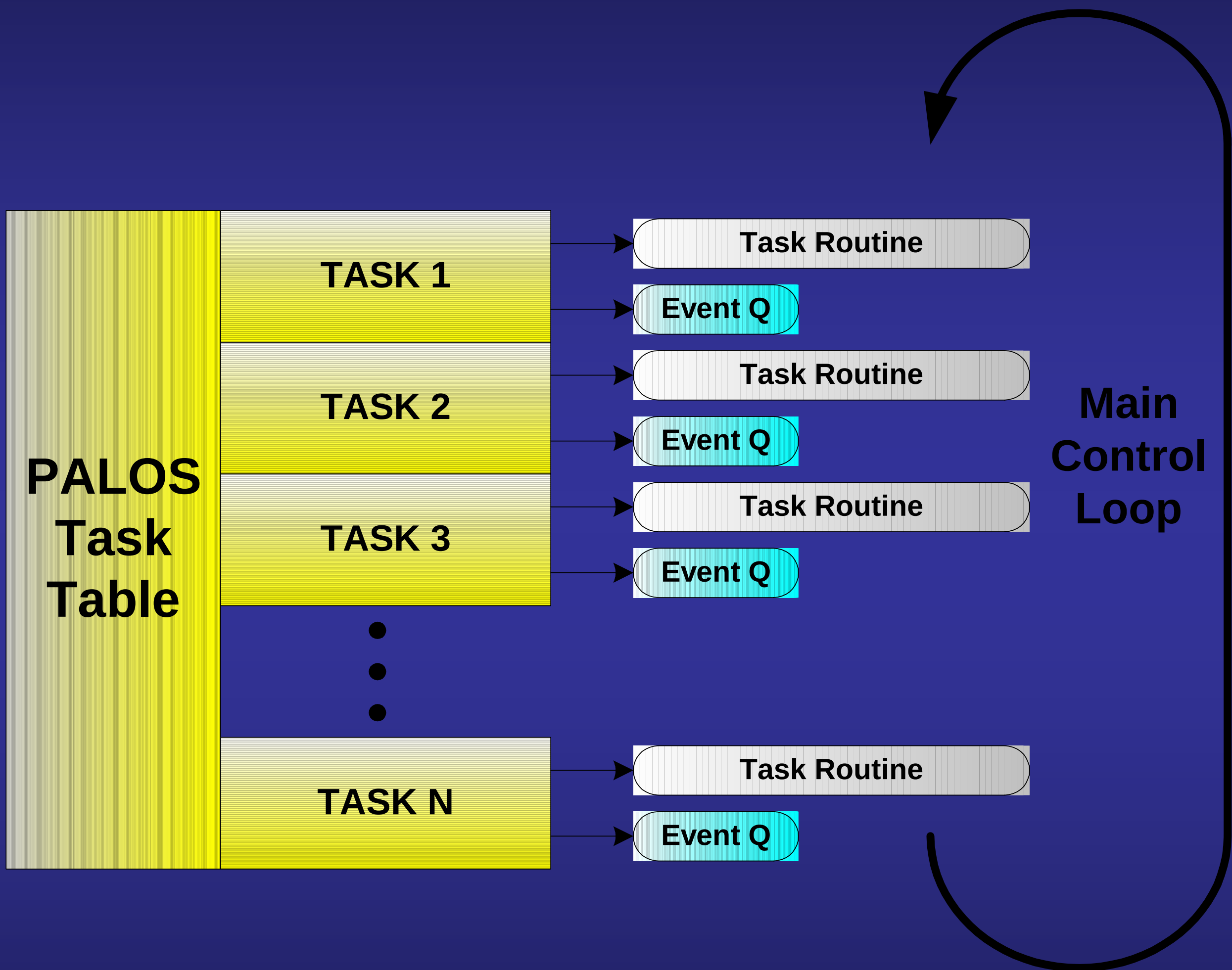
PalOS


PALOS
Core
TASK 1
TASK 4
TASK 2
TASK 3
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

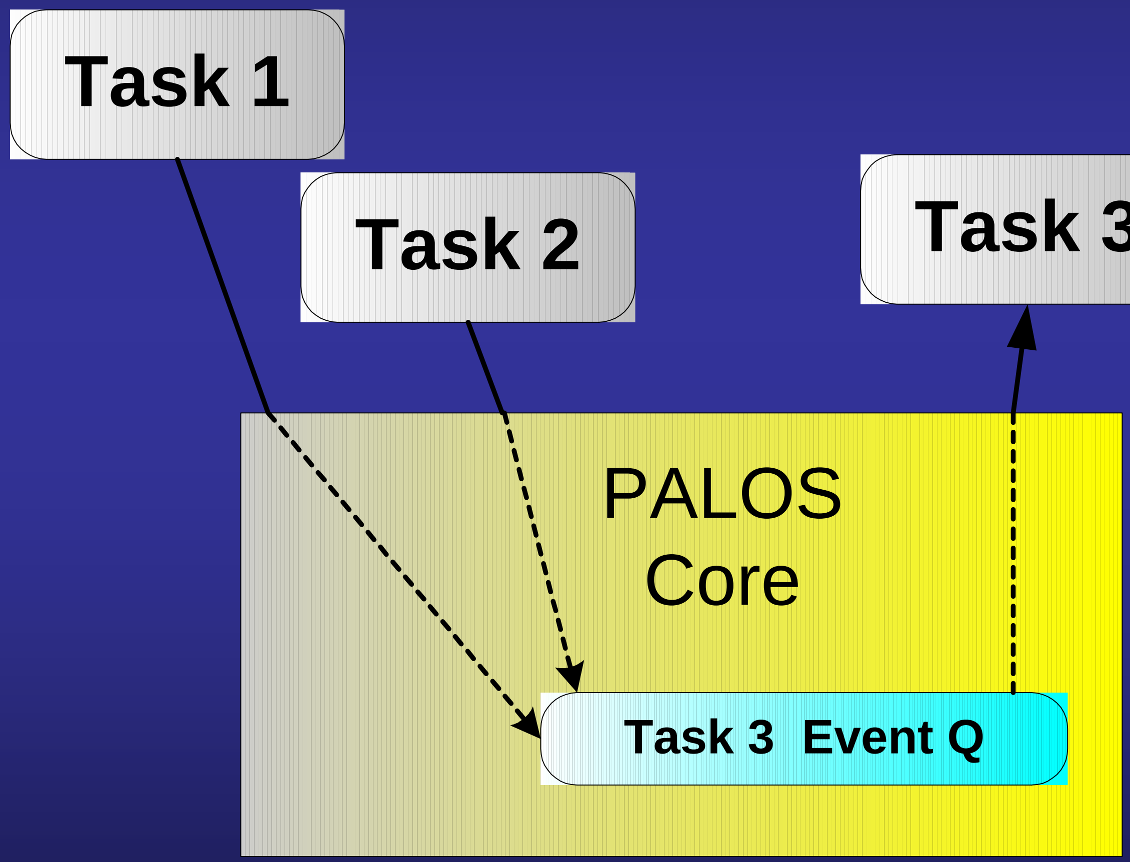
PalOS Tasks



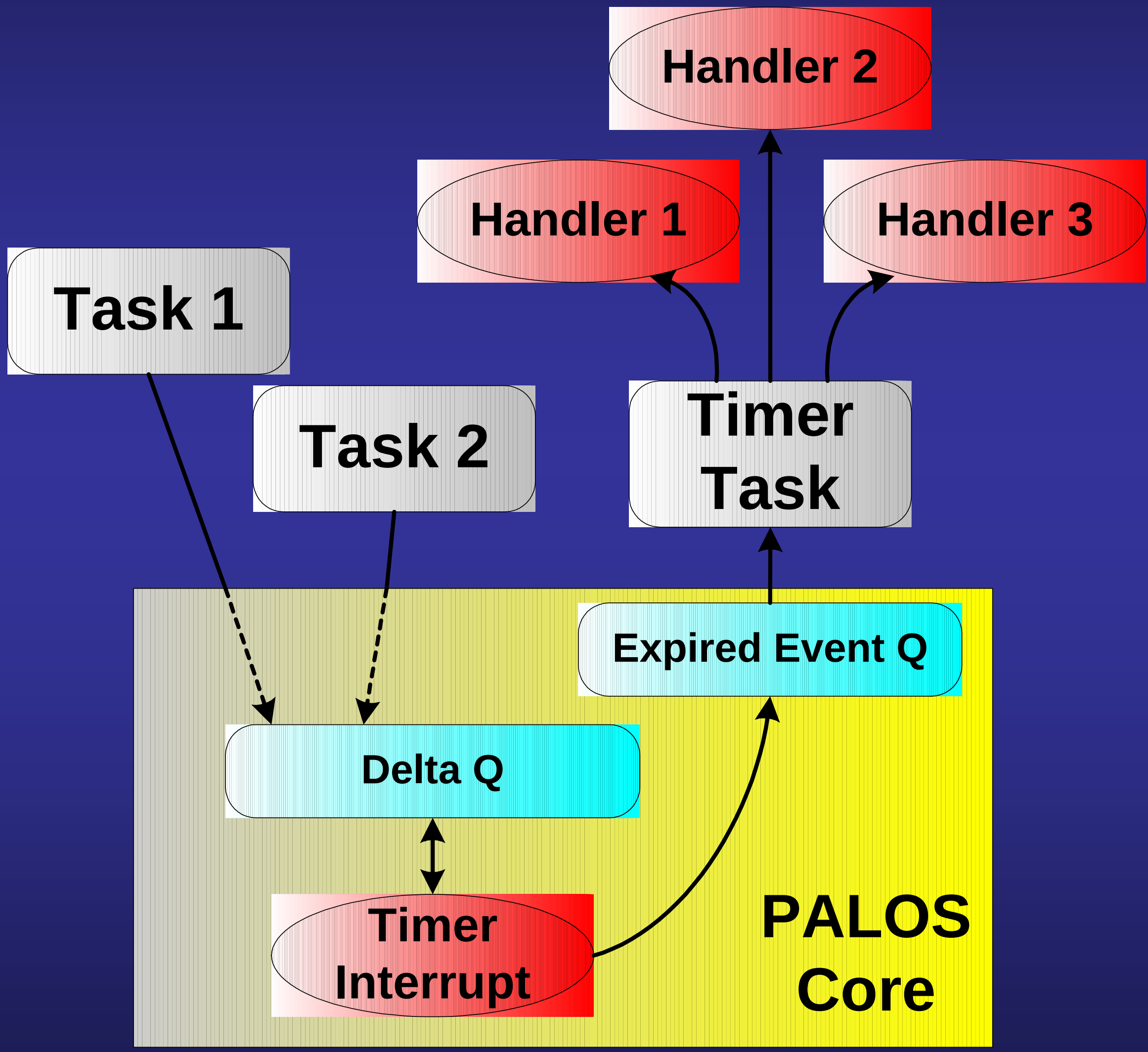
- A task belongs to the PalOS main control loop
- Each task has an entry in PalOS task table (along with eventQs)

PalOS Inter-task Communication

- Events are exchanged using the service provided by PALOS core



PalOS Core



- Periodic or aperiodic events can be scheduled using Delta Q and Timer Interrupt
- When event expires appropriate event handler is called

PalOS v0.1 Implementation – Main Control Loop

```
// 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)();
        }
     }
   }
 }
```

- Code size: 956 bytes
- Memory size: 548 bytes

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!
    - PalOS
    - **TinyGALS**
  - Re-programmability Issues
    - Maté
    - SensorWare
- Conclusions – Open research problems

Operating Systems & Programming Models

TinyGALS

- **G**lobally **A**synchronous and **L**ocally **S**ynchronous 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

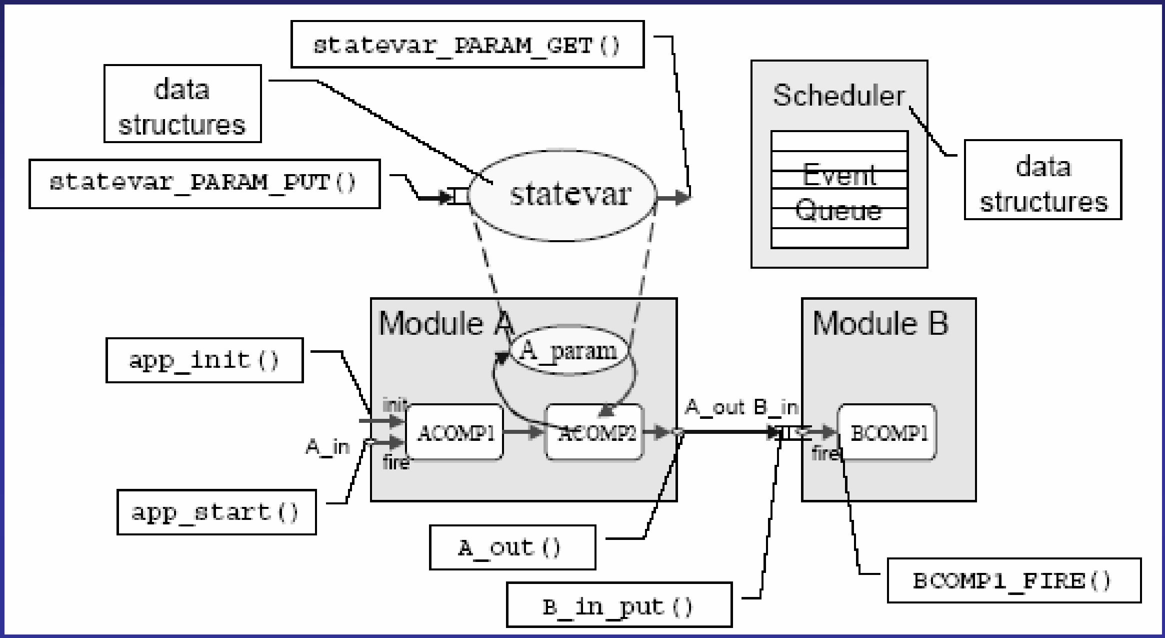
Operating Systems & Programming Models

TinyGUYS (**GU**arded **Y**et **S**ynchronous 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()**

Operating Systems & Programming Models

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!
    - PalOS
    - TinyGALS
  - **Re-programmability Issues**
    - **Maté**
    - SensorWare
- Conclusions – Open research problems

Why Re-programmability?

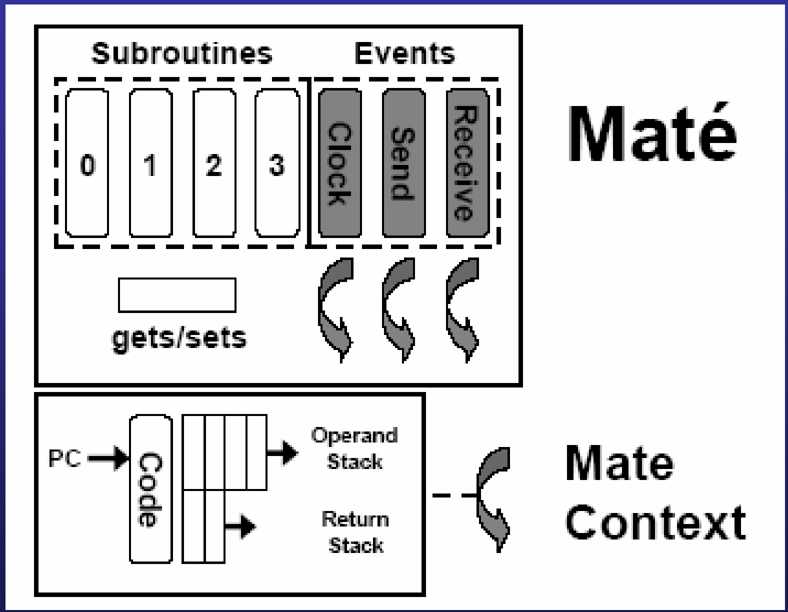
- What if there is a bug in the software running on the sensor nodes?
- What if we want to change the algorithm that the sensor network is running?
- Once deployed, sensor nodes cannot be easily collected. In some cases they cannot even be reached.
- Therefore, re-programmability should not require physical 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 *forw* instruction, to its neighbors.

- A capsule can forward other capsules using the *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

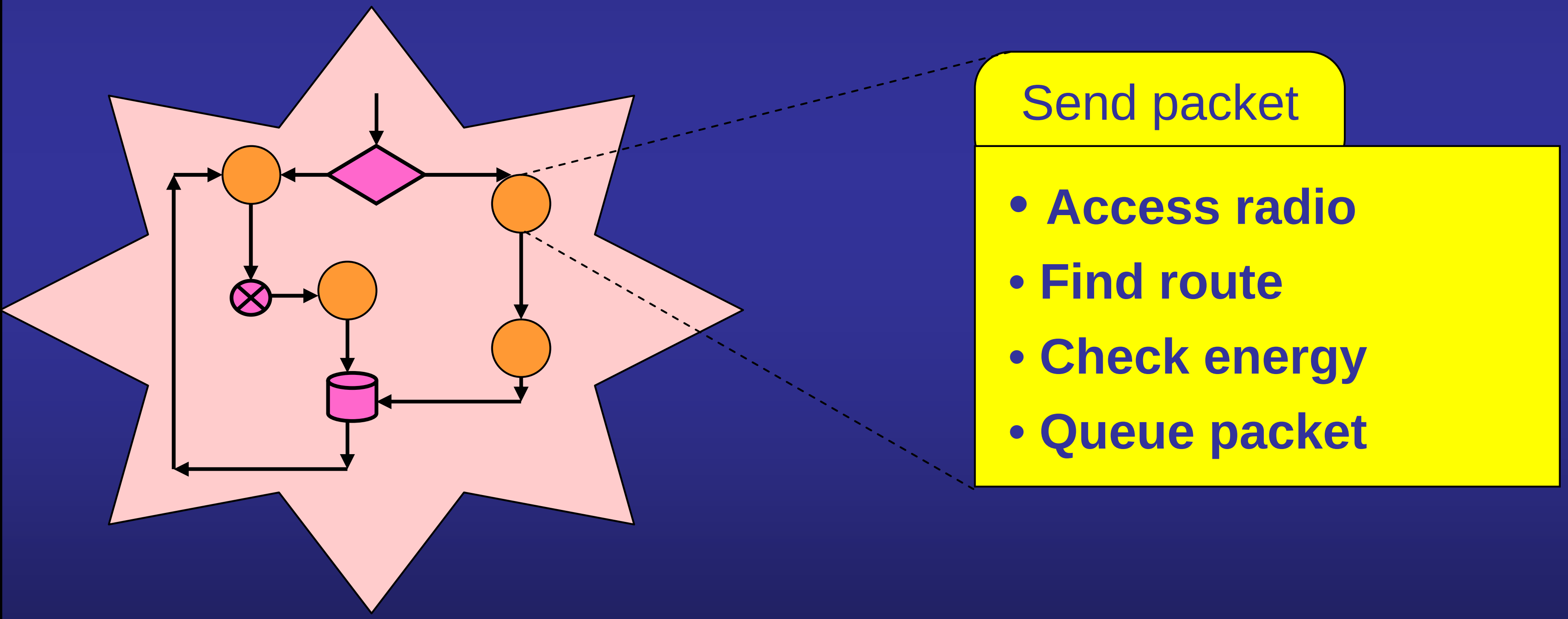
- Basic Concepts of Embedded Software – Black Box
- The case of Sensor Networks
  - Hardware Overview
  - Software for Sensor Networks
    - TinyOS
    - NesC
    - Demo using Berkeley's Mica2 motes!
    - PalOS
    - TinyGALS
  - **Re-programmability Issues**
    - Maté
    - **SensorWare**
- Conclusions

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

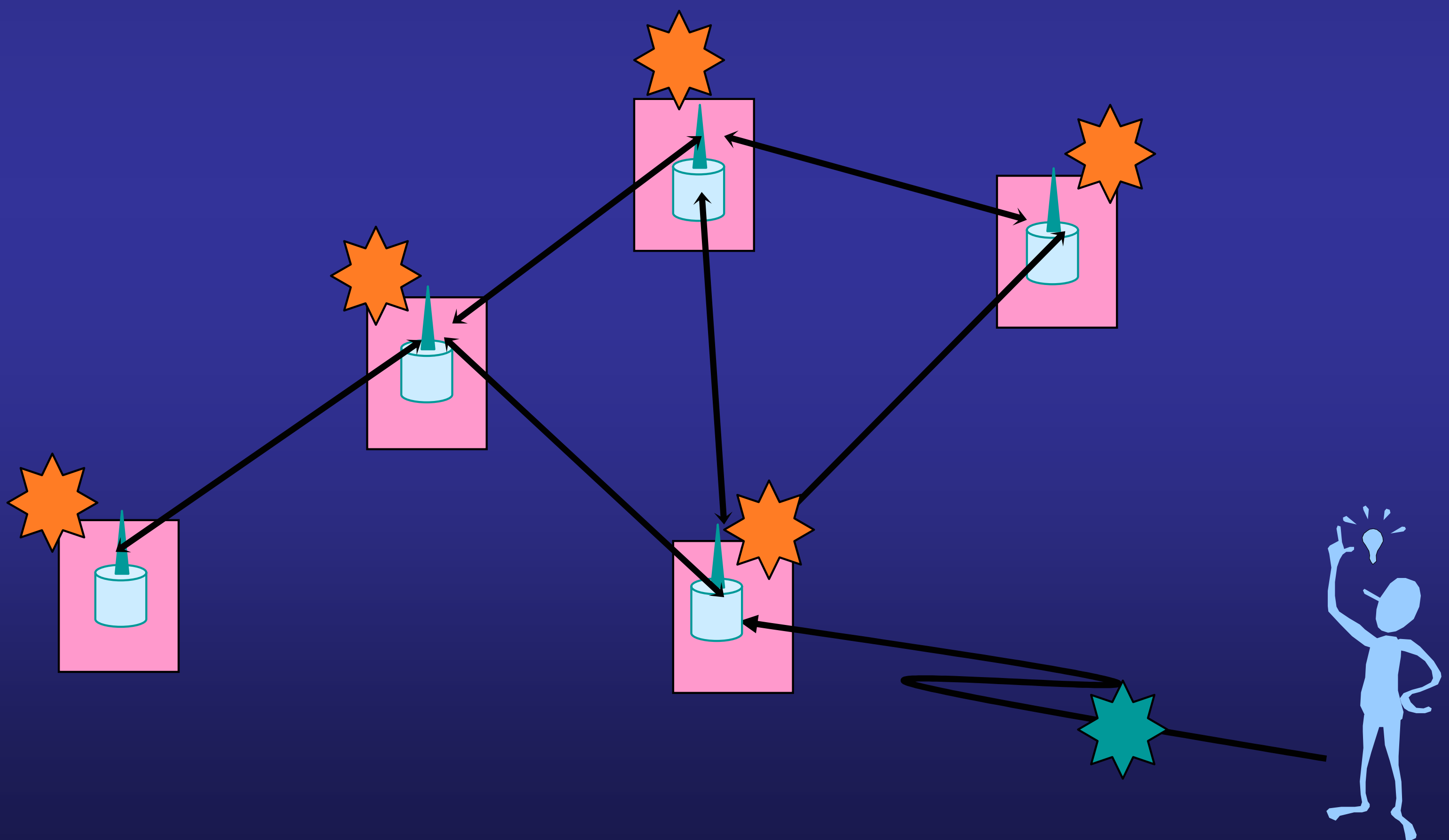


A script implementation of an algorithm

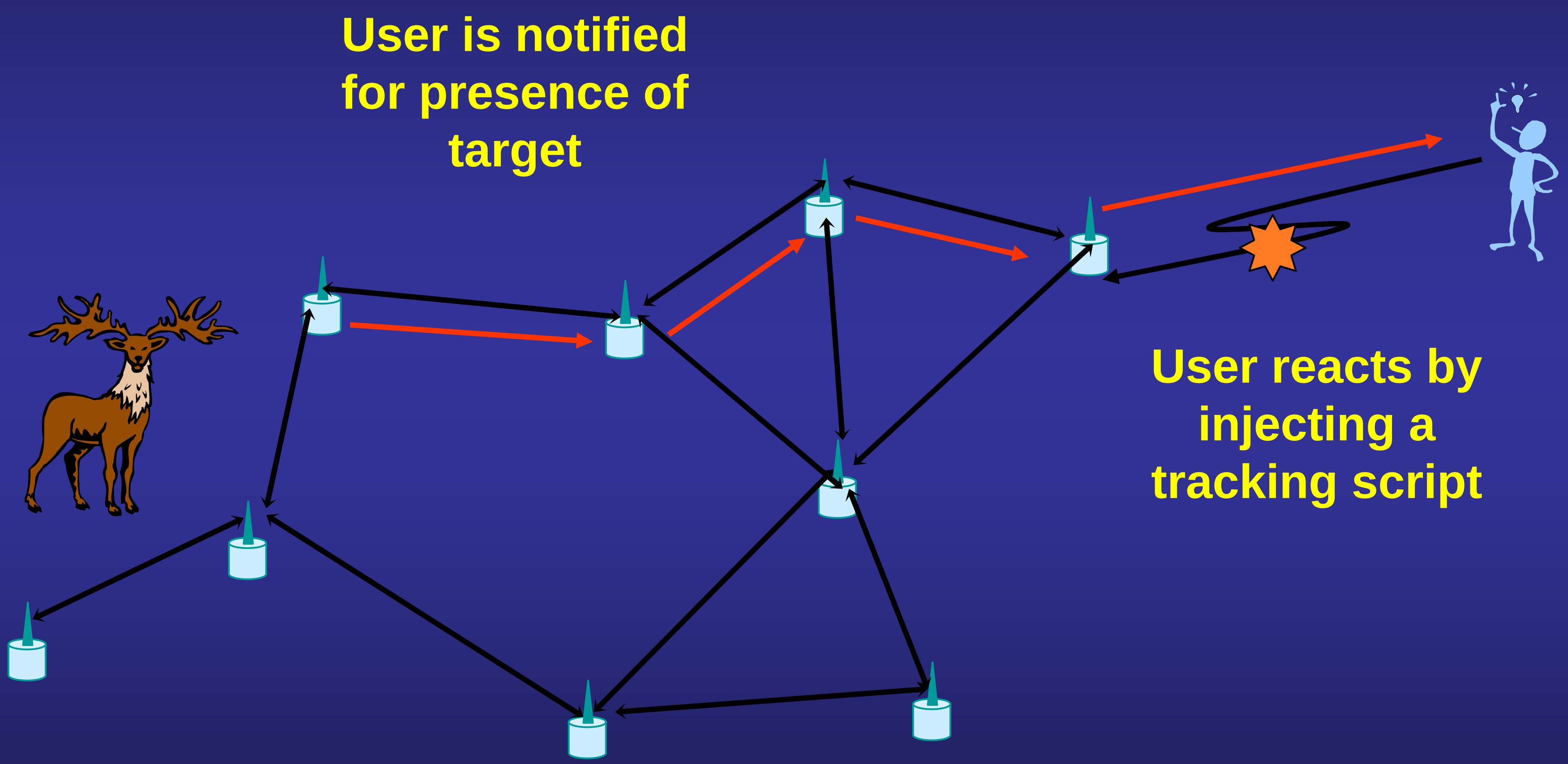
Corresponding low level tasks

SensorWare: Make Scripts Mobile

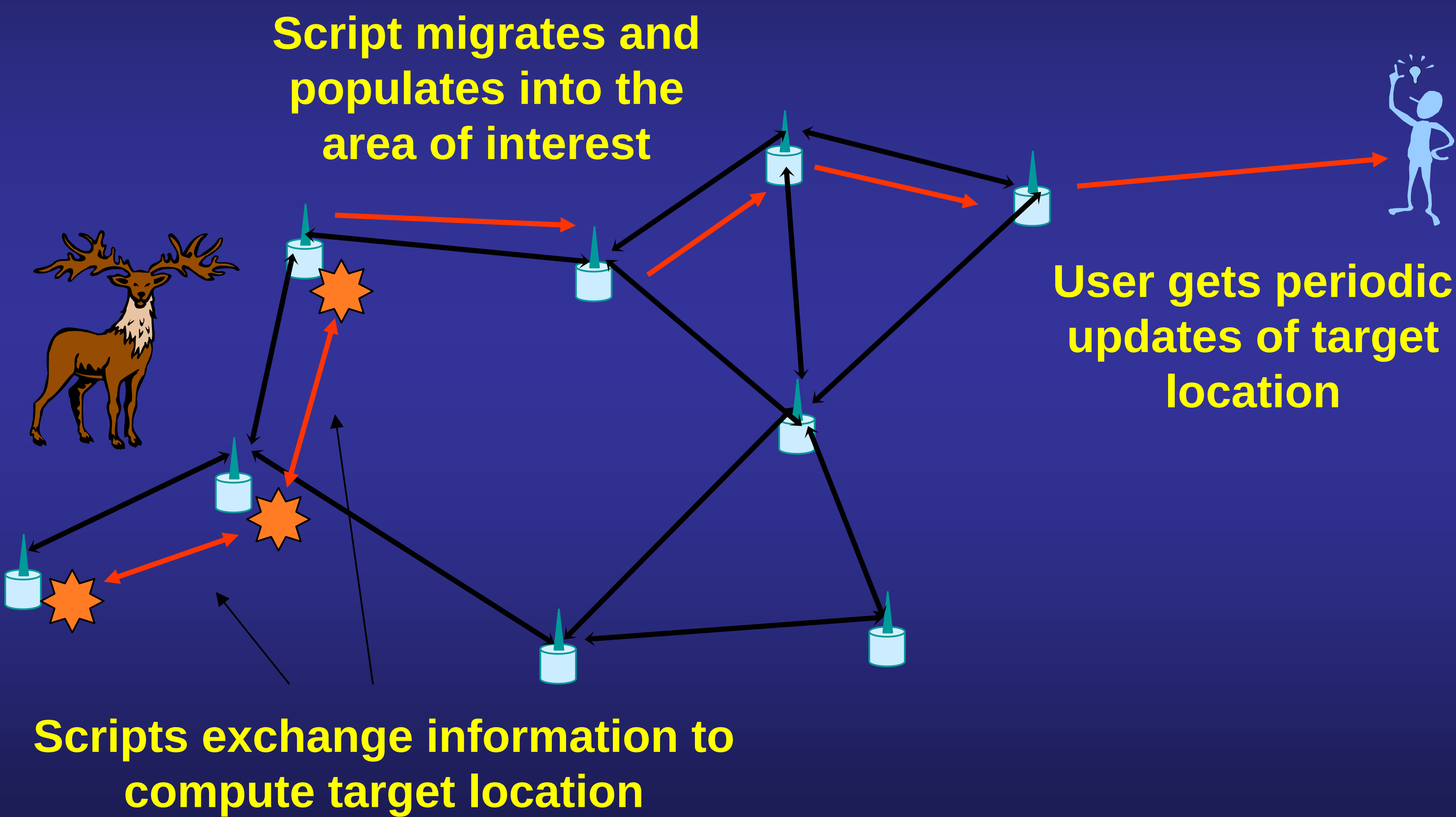
- Scripts can populate/migrate
- Scripts move due to node's state and algorithmic instructions and <u>NOT</u> due to explicit user instructions



SensorWare: An example

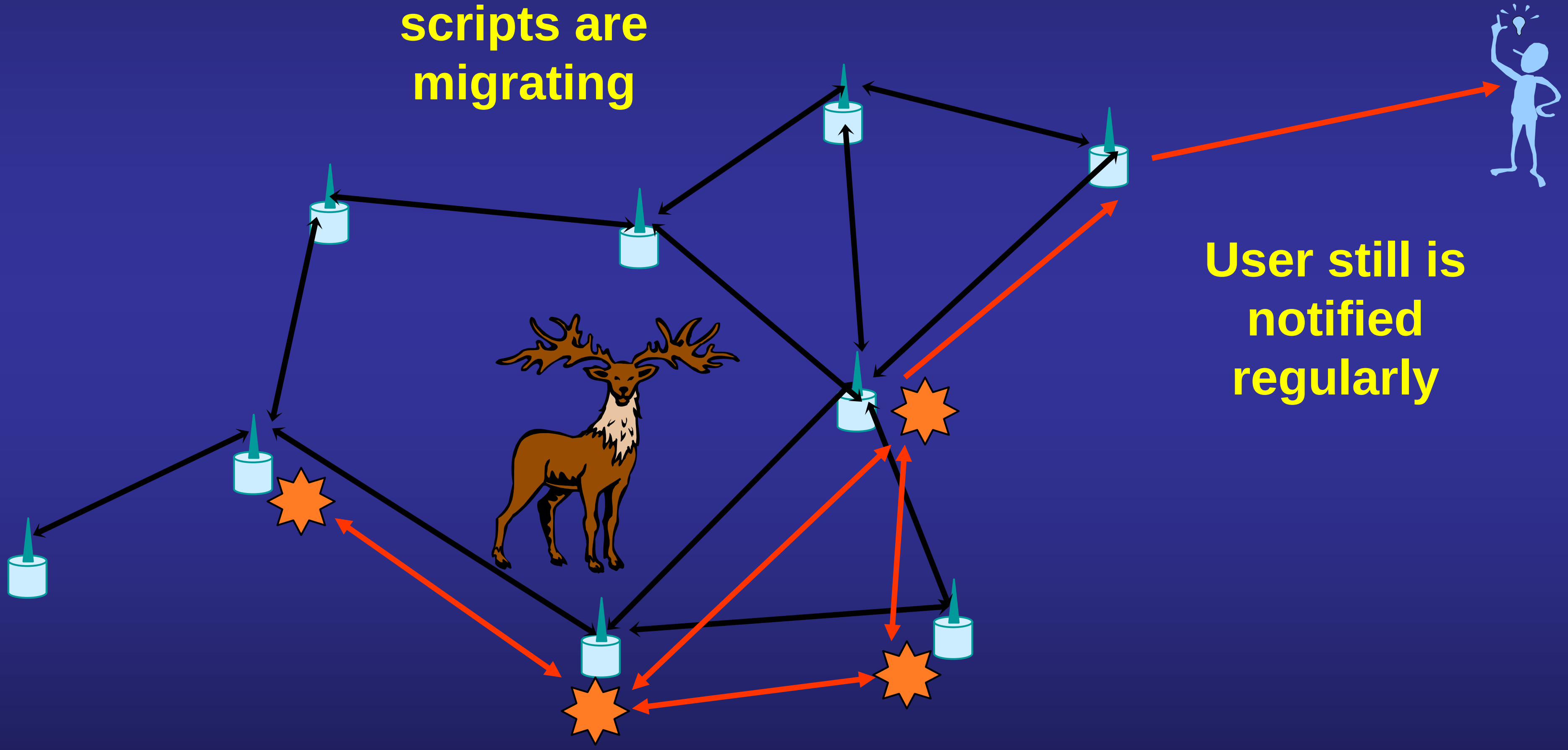


SensorWare: An example



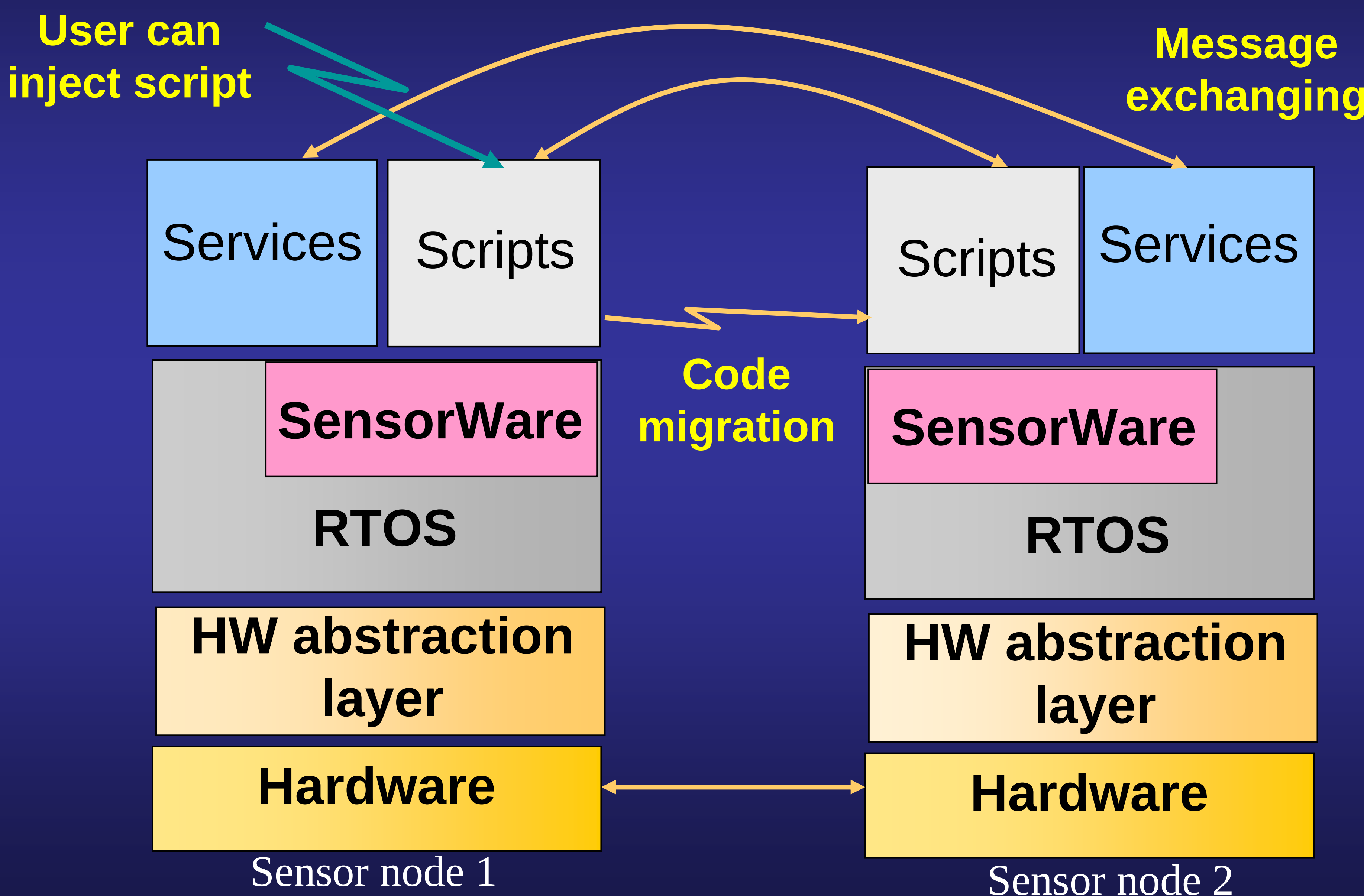
SensorWare: An example

As target moves, scripts are migrating



User still is notified regularly

The Framework



SensorWare Language

SensorWare = Language + Runtime Environment

Extensions to the core

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

`wait` command

`id` command

Mobility API

Timer API

Networking API

Sensing API

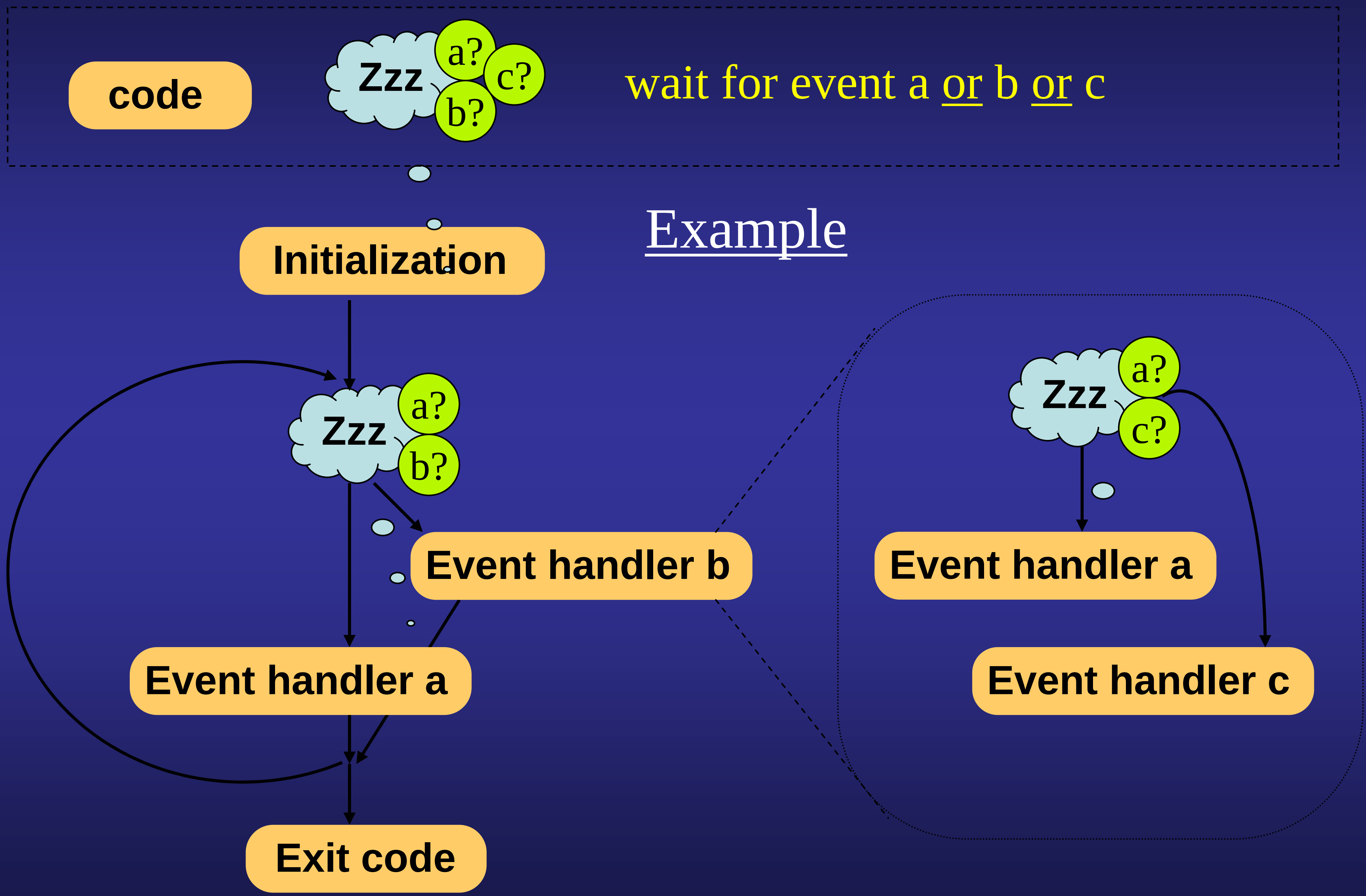
Optional GPS API

Unkown device API

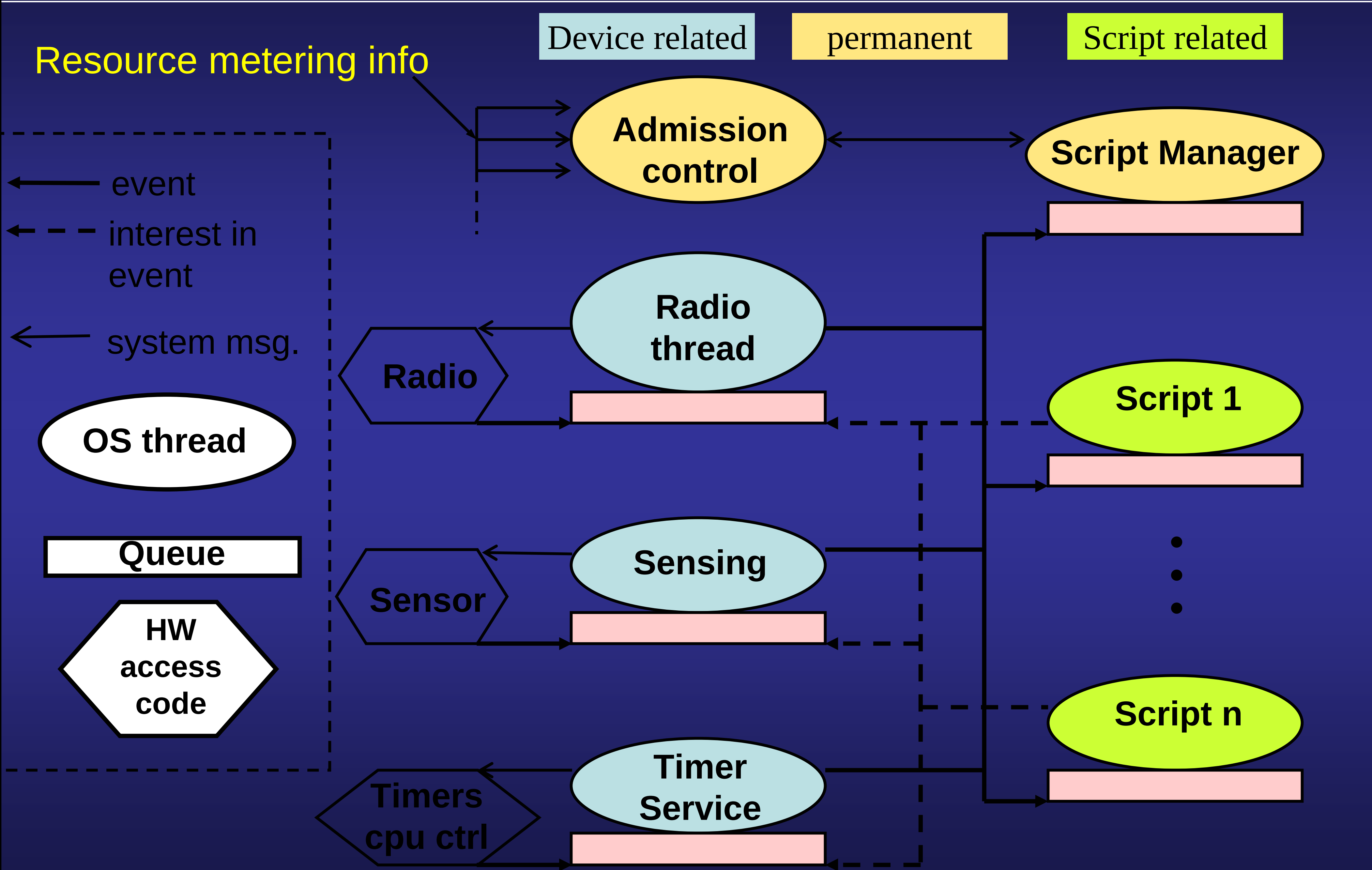
•
•
•

Will the command set be expandable?

Execution Model



SensorWare Run Time Environment



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