Wireless Sensor Networks (WSN)

Operating Systems

M. Schölzel
Operating System Tasks

**Traditional OS**
- Controlling and protecting access to resources (memory, I/O, computing resources)
- Managing their allocation to different users
- Support of concurrent task execution and communication between tasks
- Hardware drivers

**Embedded OS**
- Support energy management (shut-down of various components, sleep modes or DVS)
- Handling of external components (sensors, radio, timer, serial communication, ...)
- Handling of interrupts
- All this at a minimal usage of memory and computing resources
Common Operating Systems

TinyOS
- Developed by UC Berkeley
- Uses nesC as adjunct “programming language”
- Very small memory footprint

Contiki
- Developed by Adam Dunkels (SICS)
- Supports protothreads
- Full-featured IP and 6loWPAN stacks

freeRTOS
- Open source real-time operating system
- Based on a microkernel architecture
- Support tasks and mutexes
OS Interfaces

- There is no standard for interfaces of OS
  - different OS have of course different interfaces
  - the same OS may has different interfaces on different HW-platforms
  - both makes porting of applications difficult

- no clear distinguishing between protocol stack and application
  - API of the OS allows for protocol stack implementations
  - protocol stack extends the API of the OS for applications
  - API of the OS depends (even under the same OS) on
    - available HW-resources
    - software/protocols implementation
Examples for possible Interfaces

- TinyOS supports the definition of interfaces for modules
- Interfaces can be defined arbitrary

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

start  stop  fired

Timer
Support of Concurrency

Concurrency in sensor motes arises due to multiple external events
- sensing with multiple sensors
- sending data
- receiving and forwarding data

Approaches for handling concurrent events
- Sequential handling with polling
- Process-based approach
- Event-based approach
Polling

- Polling is active waiting for an event
- Easy to implement for a single event
- Waiting for multiple events more difficult to implement; in particular if other tasks must be done concurrently
- Polling may cause long response times
- Processor is active during polling (consumes power)

```c
#include <msp430f5438A.h>

void main(void) {
    int k;

    WDTCTL = WDTPW+WDTHOLD;  // Stop WDT -
                             // WDTPW = 0x5a00,
                             // WDTHOLD = 0x80,
                             // WDTCTL = SFR_16BIT(WDTCTL)

    P8SEL = 0;
    P8OUT = 0;
    P8DIR = 0xFE;

    while(P8IN & 0x01);   // Polling
    // do something...
}
```
Polling of multiple events

- Use a loop
  - Check for each event, if something has to be done
    - sensor data available
    - radio data available
    - ...
  - non-blocking checking is required

- Could be a feasible solution
  - if no time-critical tasks
  - events cannot be missed or missing an event is not a problem

- System may even go into sleep mode
Process-based Approach

- OS switches between processes
- Resource consumption is typically a problem in small sensor nodes
  - each process has its own stack
- Polling within each process
- Better solution as usually done in OS:
  - Use of semaphore
  - ISR for each sensor is producer
  - process is consumer
  - use semaphore for blocking the process
  - avoids busy waiting
Interrupts

- Peripherals used to raise interrupts in case of an event in order to notify the CPU
- Polling can be avoided
- An interrupt interrupts the normal program execution and forces the CPU to execute an interrupt service routine (ISR)
Interrupt Handling

- **Interrupt Vector Table** $ivt$ contains addresses of ISRs

- **On an interrupt $i$:**
  - CPU pushes PC and SR onto the stack
  - Branches to $ivt[i]$
  - ISR must save the remaining processor state

- **At the end of an ISR:**
  - ISR must restore previously stored processor state
  - pop SR from stack
  - pop PC from stack (returning to previous program position)

```
0xFFFE
0xFFFC
0xFFF8
0xFFF6
0xFFF4
...  
0x0002
0x0000
```

```
addr of ISR 1
addr of ISR 2
addr of ISR 3
addr of ISR 4
addr of ISR 5
addr of ISR 6
```

Memory MSP430

Stack before interrupt

Stack after interrupt

- Saved by hardware
- done by IRET-statement

- SP
- topElem
- Free
- SR
- PC
- Stack
- SP
Details of Interrupt Handling in MSP430

- One or more interrupts occur

- Complete current instruction

- start MCLK if CPU was off (due to power saving)

- Push program counter on stack
- Push status register on stack

- Interrupt with highest priority is selected

- SR is cleared
  - Disables further maskable interrupts (GIE cleared)
  - Terminates low-power mode

- Interrupt vector is stored in the program counter

- Now ISR executes...
Interrupts on Digital I/O Ports

- Only port 1 and 2 support interrupts by providing various registers:
  - only transitions from low to hi or hi to low cause interrupts
  - Each pin of a port may cause an interrupt
  - Highest priority is given to pin 0
  - Only one interrupt handler per port is available

- Registers of port x for interrupt control:
  - PxIFG .. Interrupt Flag registers
    - Bit = 0: no interrupt pending
    - Bit = 1: interrupt pending
  - PxIES .. Interrupt Edge Select reg
    - Bit = 0: PxIFG is set on low to high transition
    - Bit = 1: PxIFG is set on high to low transition
  - PxIE .. Interrupt Enable reg
    - Bit = 0: interrupt disabled
    - Bit = 1: interrupt enabled
  - PxIV .. Interrupt Vector reg
    - Pin i generates number 4*i in register P1IV

- Interrupt handler for port 1 could start like this:

  ADD &P1IV,PC ; Add offset to Jump table 3
  RETI         ; Vector 0: No interrupt
  JMP P1_0_HND ; Vector 2: Port 1 bit 0
  JMP P1_1_HND ; Vector 4: Port 1 bit 1
  JMP P1_2_HND ; Vector 6: Port 1 bit 2
  JMP P1_3_HND ; Vector 8: Port 1 bit 3
  JMP P1_4_HND ; Vector 10: Port 1 bit 4
  JMP P1_5_HND ; Vector 12: Port 1 bit 5
  JMP P1_6_HND ; Vector 14: Port 1 bit 6
  JMP P1_7_HND ; Vector 16: Port 1 bit 7
Example P1 interrupt

```c
void main(void)
{
    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
    P1DIR |= 0x01; // Set P1.0 to output direction
    P1IE |= 0x10; // P1.4 interrupt enabled
    P1IES |= 0x10; // P1.4 Hi/lo edge
    P1IFG &= ~0x10; // P1.4 IFG cleared
}

// Port 1 interrupt service routine
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
    P1OUT ^= 0x01; // P1.0 = toggle
    P1IFG &= ~0x10; // P1.4 IFG cleared
}
```
Event-Based Programming Model

- For every interrupt there is a fast ISR making all necessary data processing of an event
  - event can be: arrival of a packet, timer expired, ADC-conversion finished, etc.
- ISR (high priority) schedules a task (of lower priority) for a deferred processing (in Windows this is called deferred interrupt processing)
- OS schedules these tasks in the program-context
- tasks are not preemptive; run to completion
  - but can be interrupted by ISRs

```
Entering ISR
Put Task into queue
Exit from ISR and wake up

Any pending Task in queue?
no
  - Sleep

Execute Task
Scheduler of the OS
```

Asynchronous Program Flow

```
Interrupt Context

Interrupt

Program Context

Synchronous Program Flow
```
Some Consequences

- Code in the program-context must not be synchronized with any other code in the program context.
- Program-context code may be synchronized with ISR-context code; disabling interrupts, when PCC enters a critical section.
- ICC may be interrupted by other ICC.
- Real-Time behavior is hard to predict:
  - time between occurrence of an interrupt and processing of data matters.
  - if processing takes place in the PCC no bounds can be guaranteed:
    - potentially many cascading interrupts.
Example TinyOS

- Implements the concept of event-based programming
- Provides a way to flexibly define interfaces

- **Component**: Semantically related functions
  - its own state is comprised in a only locally accessible *frame*

- program code in tasks; represent PCC
- handlers for commands and events must be non-blocking; represent ICC
  - commands and events are exchanged between components
  - events originate in hardware and are propagated upwards
Purpose of Interfaces

- A component either uses or provides an interface

- **User:**
  - must implement events
  - can issue commands

- **Provider:**
  - must implement commands
  - can issue events
Configurations

- Configurations are used to establish connections (wiring) between components.

- For each interface of a component a configuration specifies:
  - which other component uses the interface
  - which component provides the interface
Blink Example in TinyOS

module BlinkC {
    uses interface Boot;
    uses interface Timer;
    uses interface Leds;
}

implementation {
    event void Boot.booted() {
        call Timer.startPeriodic(1000);
    }

    event void Timer.fired() {
        call Leds.led0Toogle();
    }
}

collection BlinkAppC {
    implementation {
        components MainC, LedsC, TimerC, BlinkC;
        BlinkC.Boot -> MainC.Boot
        BlinkC.Leds -> LedsC.Leds
        BlinkC.Timer -> TimerC.Timer
    }
}
Split-Phase-Programming

Problem:
- Issuer of a command may expects a return value, but execution of the command takes a while (in HW)
- Command handler/tasks should not block

Solution: Split-Phase-Programming
- Issue a command
- Corresponding Event Handler is notified later

LLC implementing an automatic request protocol

MAC layer
Summary

- Programming paradigms for WSNs:
  - Polling, Processes, Event Driven

- Broad range of operating systems available

- Older OS try to minimize resource consumption (TinyOS, Contiki)
  - support event driven approach

- Newer OS improve usability
  - support of process-based approaches and synchronization mechanisms

- Interfaces of the same OS differ a lot for different HW-platforms
  - Porting applications becomes difficult