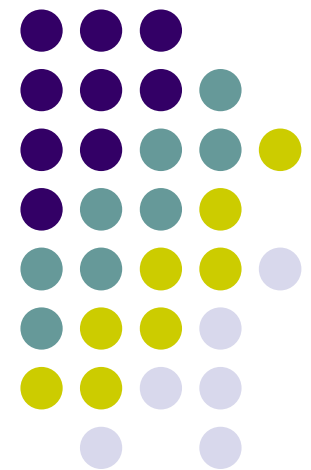


# Lab course: Programming Sensor Networks

## Lecture-1

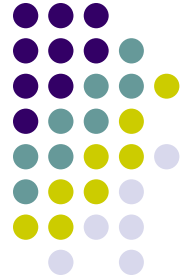
Programming Motes using TinyOS and NesC





# What is NesC?

University of Freiburg  
Institute of Computer Science  
Computer Networks and Telematics  
Prof. Christian Schindelhauer

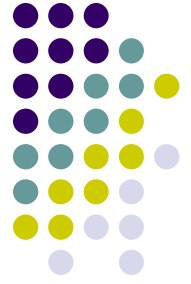


- NesC
  - A superset of C
    - One may generate an intermediate C file from a NesC project
  - Main feature:
    - Separation of declaration and definition



# What is TinyOS?

University of Freiburg  
Institute of Computer Science  
Computer Networks and Telematics  
Prof. Christian Schindelhauer



- TinyOS
  - An event-driven operating system
  - Developed using NesC
- Support for many types of motes
  - At least 15 Motes types are supported by NesC/TinyOS (source: [SNM](#))



# NesC Concepts

University of Freiburg  
Institute of Computer Science  
Computer Networks and Telematics  
Prof. Christian Schindelhauer



- Component
  - Module
  - Configuration
- Interface
- Command
- Event
- Split-Phase
- Task
- Sync Vs Async Commands



# Components



- NesC is a *component* based C dialect
- A component is similar to Java object
  - It provides encapsulated state and couple state with functionality
- A component is not really a Java object
  - No inheritance and usually Singleton
  - Components have only private variables
  - Only functions could be use to pass the variables between components
- Two types of components
  - Modules
  - Configuration



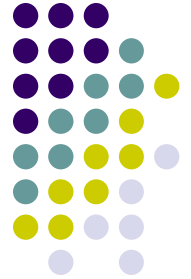
# Module & Interface



- Module has the implementation of functions
- It uses pure local namespace
  - Component has to declare function it *uses* and *provides*
- NesC Interface is very Similar to Java Interface
  - Declaration of functions



# NesC Concepts: Module & Interface



```
module fooC {  
    uses interface foobarInterface as fbi;  
}  
implementation {  
    void foo( ) {  
        call foobar( );  
    }  
}
```

```
module barC {  
    uses interface foobarInterface as foobi;  
}  
implementation {  
    void bar( ) {  
        call foobar( );  
    }  
}
```

```
interface foobarInterface {  
    command void foobar ( );  
}
```

```
module foobarC {  
    provide interface foobarInterface;  
}  
implementation {  
    command void foobarInterface.foobar( ) {  
        ...  
    }  
}
```

```
configuration foobarAppC {  
}  
implementation {  
    components fooC, barC;  
    fooC.fbi -> foobarC;  
    barC.foobi -> foobarC;  
}
```



# Configuration



- Recall: Components have two types
  - Module
  - Configuration
- Configuration
  - Wire components together
  - Has two operations
    - user -> provider (or provider <- user)
    - = (between two providers mostly)
      - Usually use to equate the interface provided by the configuration

```
Configuration ActiveMessageC {  
    provides interface Init;  
    provides interface SplitControl;  
}  
Implementation {  
    components CC240ActiveMessageC as AM;  
    Init = AM;  
    SplitControl = AM;  
}
```





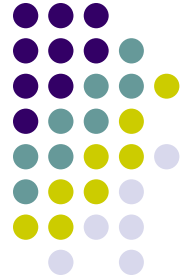
# Split-Phase



- Split-phase in hardware then split-phase in software
  - Two phase
    - Downcall : **Command** – start the operation
    - Upcall : **Event** – operation has been completed



# Command & Event



- All commands are implemented by all providers of an interface
- All events are implemented by all users of an interface
- Example

```
interface Send {  
    command error_t send(message_t* msg, uint8_t len);  
    event void sendDone(message_t* msg, error_t error);  
}
```

```
module SendC {  
    uses interface Send;  
    uses interface Boot;  
}  
Implementation {  
    event void Boot.booted() {  
        call Send.send(NULL, 0);  
    }  
  
    event void sendDone(message_t* msg, error_t error) {  
        //do nothing  
    }  
}
```



# Task



- Task
  - Are deferred procedure call
  - Event are usually *signaled by posting* a task
  - Task are **strictly local** to a module
    - No parameters
    - No return type
    - No defined in any interface
  - Each task is **non-preemptive** and atomic with respect to other tasks
  - A task can post itself



# Async Vs Sync command



- Async are preemptable commands
- Unlike task Async commands are not atomic with respect to other commands
- Async command cannot call a Sync command
  - Can call other Async commands
  - Can post task which may call a Sync command
- Sync commands calls are blocking like normal function call



# Keywords



```
module FilterMagC {
  provides interface StdControl;
  provides interface Read<uint16_t>;
  uses interface Timer<TMilli>;
  uses interface Read<uint16_t> as RawRead;
}

implementation {
  uint16_t filterVal = 0;
  uint16_t lastVal = 0;
  task void readDoneTask();
  command error_t StdControl.start() {
    return call Timer.startPeriodic(10);
  }
  command error_t StdControl.stop() {
    return call Timer.stop();
  }
  event void Timer.fired() {
    call RawRead.read();
  }
  event void RawRead.readDone(error_t err, uint16_t val) {
    if (err == SUCCESS) {
      lastVal = val;
      filterVal *= 9;
      filterVal /= 10;
      filterVal += lastVal / 10;
    }
  }
  command error_t Read.read() {
    post readDoneTask();
    return SUCCESS;
  }
  task void readDoneTask() {
    signal Read.readDone(SUCCESS, filterVal);
  }
}
```



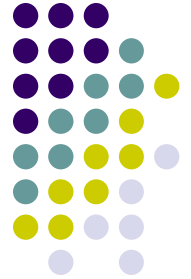
## NesC solution in detail



- Unlike C each NesC function had a unique local name
  - Component A calls command B then A\$B is the name of such call
- NesC component defines what it *uses* and *provides*
- A user is *wired* to a *provides* during compilation times (instead of linking) based on configuration
  - NesC has static linking
- Advantages of static linking
  - Better optimize codes by compiler
  - Less error prone
- Disadvantages
  - Less flexible
  - Configurations become cumbersome as the project grows



# TinyOS and NesC limitations

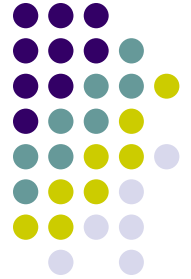


- NesC is a low-level languages
  - Have many disadvantages inherited from C
    - No automatic garbage collection
      - Memory leaks
    - No portability once code is compiled
- It is not object oriented languages
  - Limited design patterns application
- Configurations are difficult to change for a big program



# TinyOS and NesC limitations

University of Freiburg  
Institute of Computer Science  
Computer Networks and Telematics  
Prof. Christian Schindelhauer



- Thread Vs event driven
  - TinyOS is event-driven and not a thread base OS
  - Threads have better response time
  - Event drive OS has less memory requirements
  - Event driven model drawbacks:
    - requires manual configuration
    - Manual state handling
    - Difficult to change code without changing already written state handlers
- All Events have to be implemented by a user of an interface
  - Even if user of a interface is not interested in many of them





## References

University of Freiburg  
Institute of Computer Science  
Computer Networks and Telematics  
Prof. Christian Schindelhauer

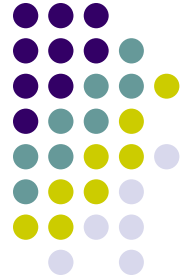


- Philip Levis, “*TinyOS Programming*”, 2006
- Kim et al, “*Multithreading Optimization Techniques for Sensor Network Operating Systems*”, EWSN 2007



# The End

University of Freiburg  
Institute of Computer Science  
Computer Networks and Telematics  
Prof. Christian Schindelhauer



- Thank you for listening