

# GAiNe'08

## Outline

- **Introduction**
- Embedded Systems
- Embedded Systems Software architecture
- OS for Wireless Sensor Networks
- TinyOS
- TinyOS – Components
- Blink application Example
- Programming Hints
- Thank you



- A **wireless sensor network (WSN)** is a wireless network consisting of spatially distributed autonomous devices (Motes) using sensors to cooperatively monitor physical or environmental conditions.

- Mote is a microcontroller based embedded computer



# GAiNe'08

## Outline

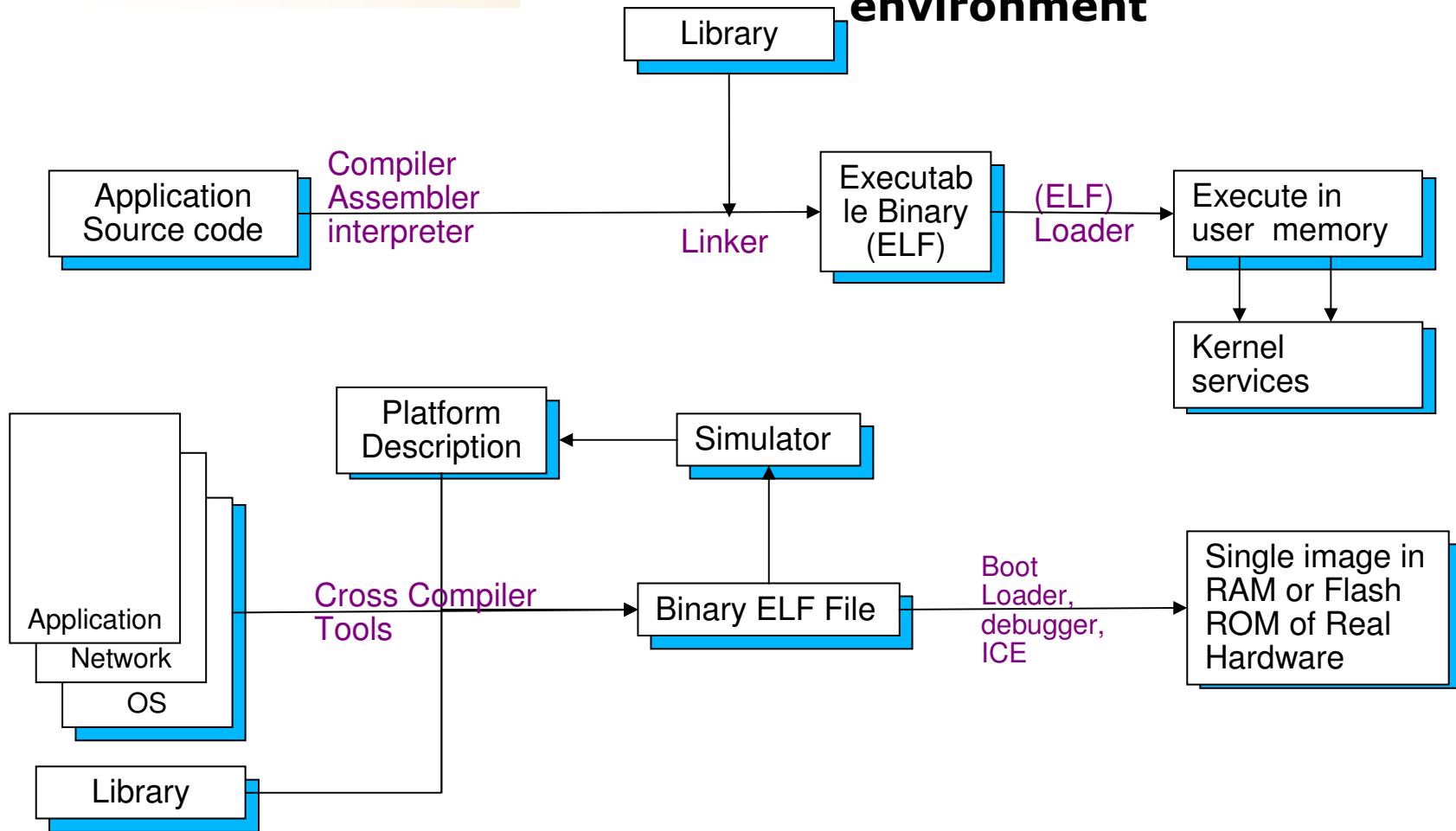
- Introduction
- **Embedded Systems**
- Embedded Systems Software architecture
- OS for Wireless Sensor Networks
- TinyOS
- TinyOS – Components
- Blink application Example
- Programming Hints
- Thank you



- An **embedded system** is a special-purpose computer system designed to perform one or a few dedicated functions
- Hardware and software is optimized for the particular applications
- Depend on the applications simple 4bit microcontrollers to network of 64bit processors are used.
- Cost becomes cheaper on Mass production
- The software for embedded systems are called as firmware, usually reside in Flash ROM



## Embedded systems – Development environment



# GAiNe'08

## Embedded Systems – Tools

- Compilers, Assembler, Linker Tool chain (C,C++, Java, Ada, Extension of basic)
- In Circuit Emulators, debuggers
- Often have interface with a PC for debugging purpose
- Target
- Host



Often Many boards do not support debugging facility

Only way to debug such systems are using Serial ports and LEDs

Or use simulator



# GAiNe'08

## Outline

- Introduction
- Embedded Systems
- **Embedded Systems Software architecture**
- OS for Wireless Sensor Networks
- TinyOS
- TinyOS – Components
- Blink application Example
- Programming Hints
- Thank you



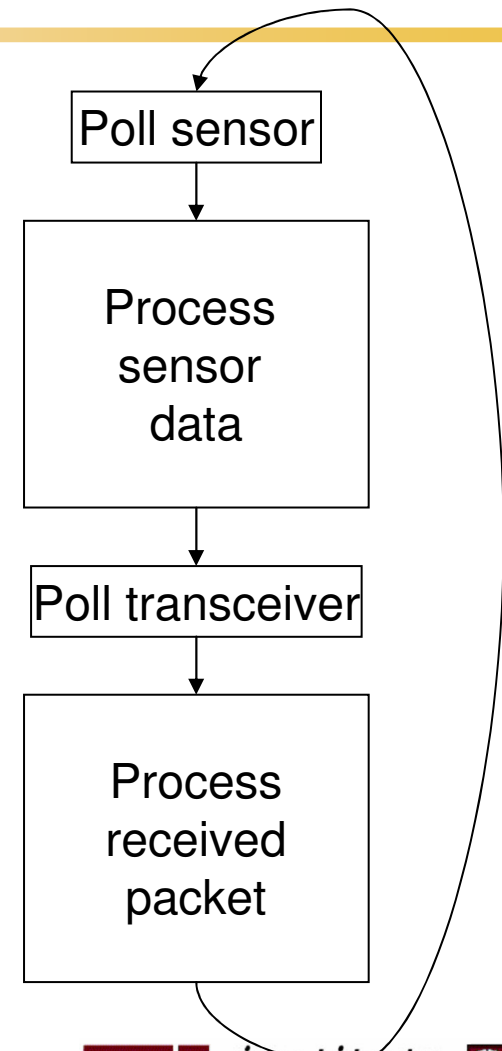
- **Simple control loop**

The software simply has a loop. The loop calls subroutines, each of which manages a part of the hardware or software

Many microcontroller (8,16bit) based systems runs in this manner.

Simple & Good for doing a single job.

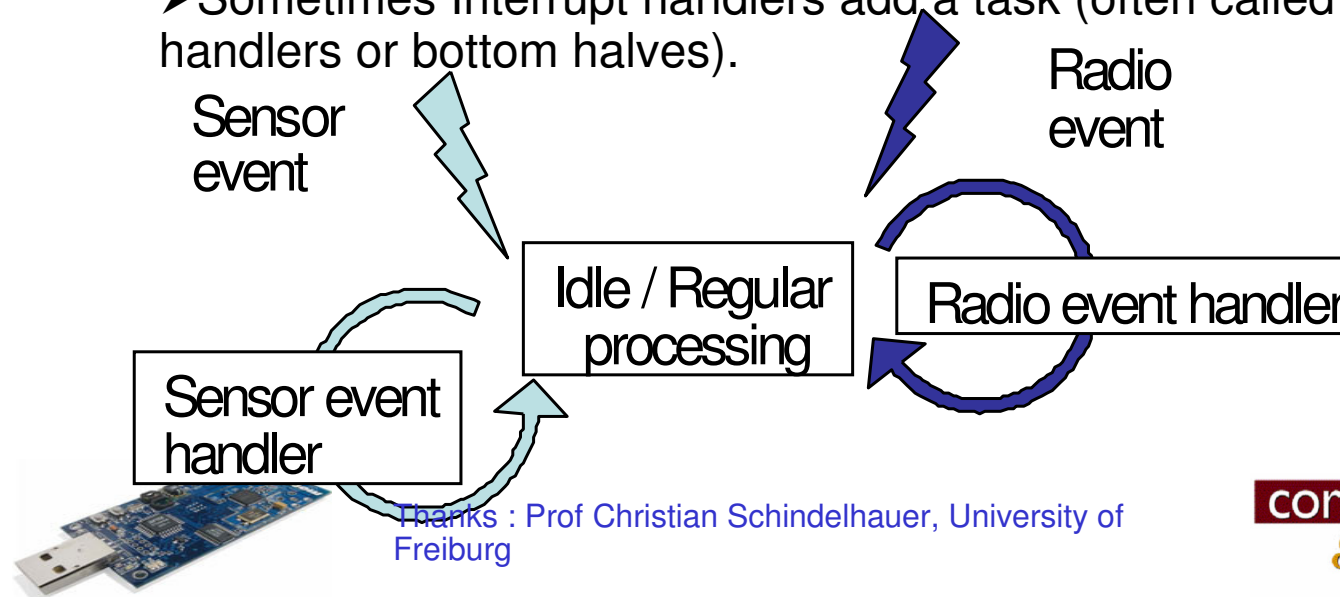
No concurrency



Thanks : Prof Christian Schindelbauer, University of Freiburg

### Interrupt controlled system

- Hardware events trigger the system to process
- Suitable for the events that need low latency and the event handlers are short and simple.
- Usually a main loop (time incensitive) or in idle
- Sometimes Interrupt handlers add a task (often called as deferred handlers or bottom halves).



- **Multitasking (Cooperative, pre-emptive)**

Multiple tasks share the CPU and they schedule themselves

"operating system" kernel.

As any code can potentially damage the data of another task (except in larger systems using an MMU)

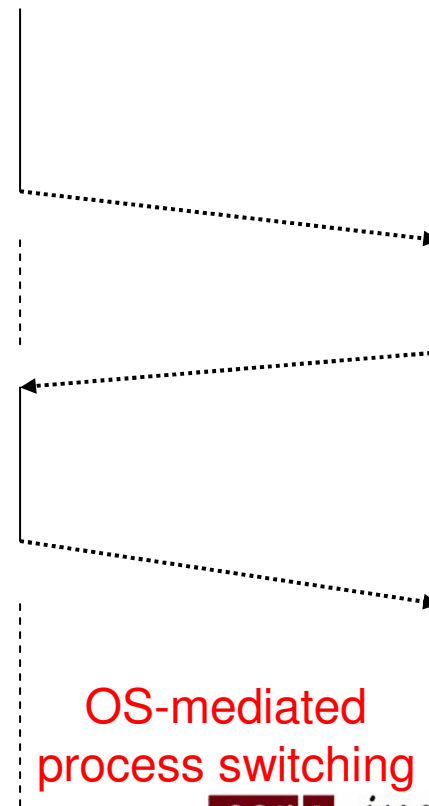
programs must be carefully designed and tested, and access to shared data must be controlled by some synchronization strategy (message queues, semaphores or a non-blocking synchronization scheme) .....Very complex, and require huge memory



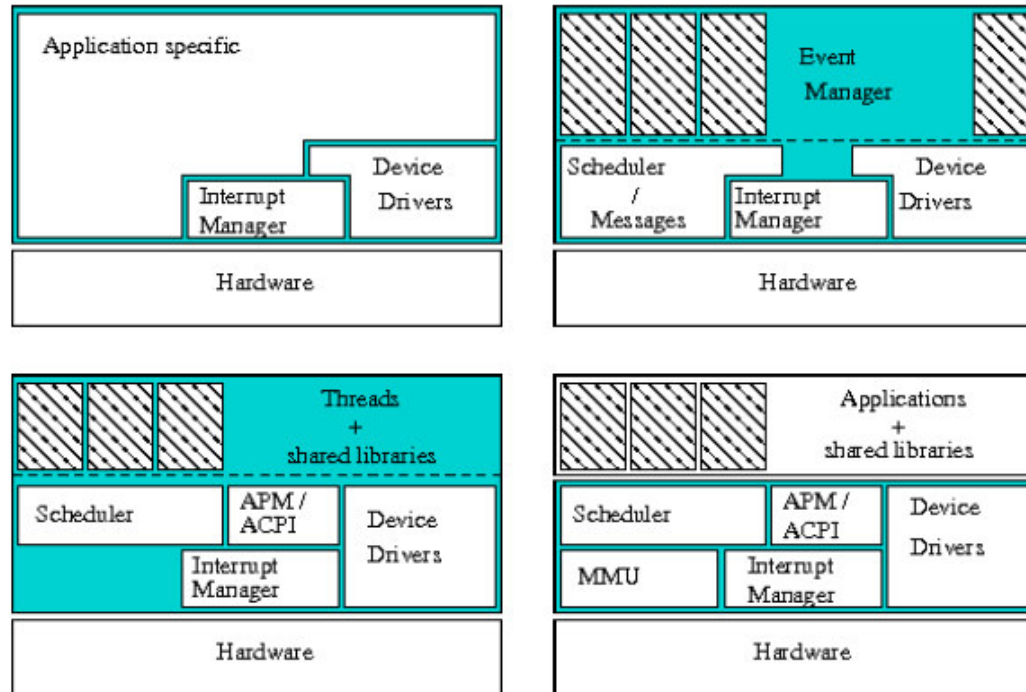
Thanks : Prof Christian Schindelhauer, University of Freiburg

Handle sensor process

Handle packet process



OS-mediated process switching



Thanks : Antoine Fraboulet, Sensations 2007



# GAiNe'08

## Outline

- Introduction
- Embedded Systems
- Embedded Systems Software architecture
- OS for Wireless Sensor Networks
- TinyOS
- TinyOS – Components
- Blink application Example
- Programming Hints
- Thank you



Application specific (no os) firmware for wireless sensor network

Event driven model – Hardware events, simple context switching, optimum stack usage, relatively difficult programming.

Tinyos, Contiki

Executing Threads model - Shared memory applications with context switch Systems are more complex and less predictable, closer to traditional programming.

Examples: SOS, MantisOS, FreeRTOS. . .

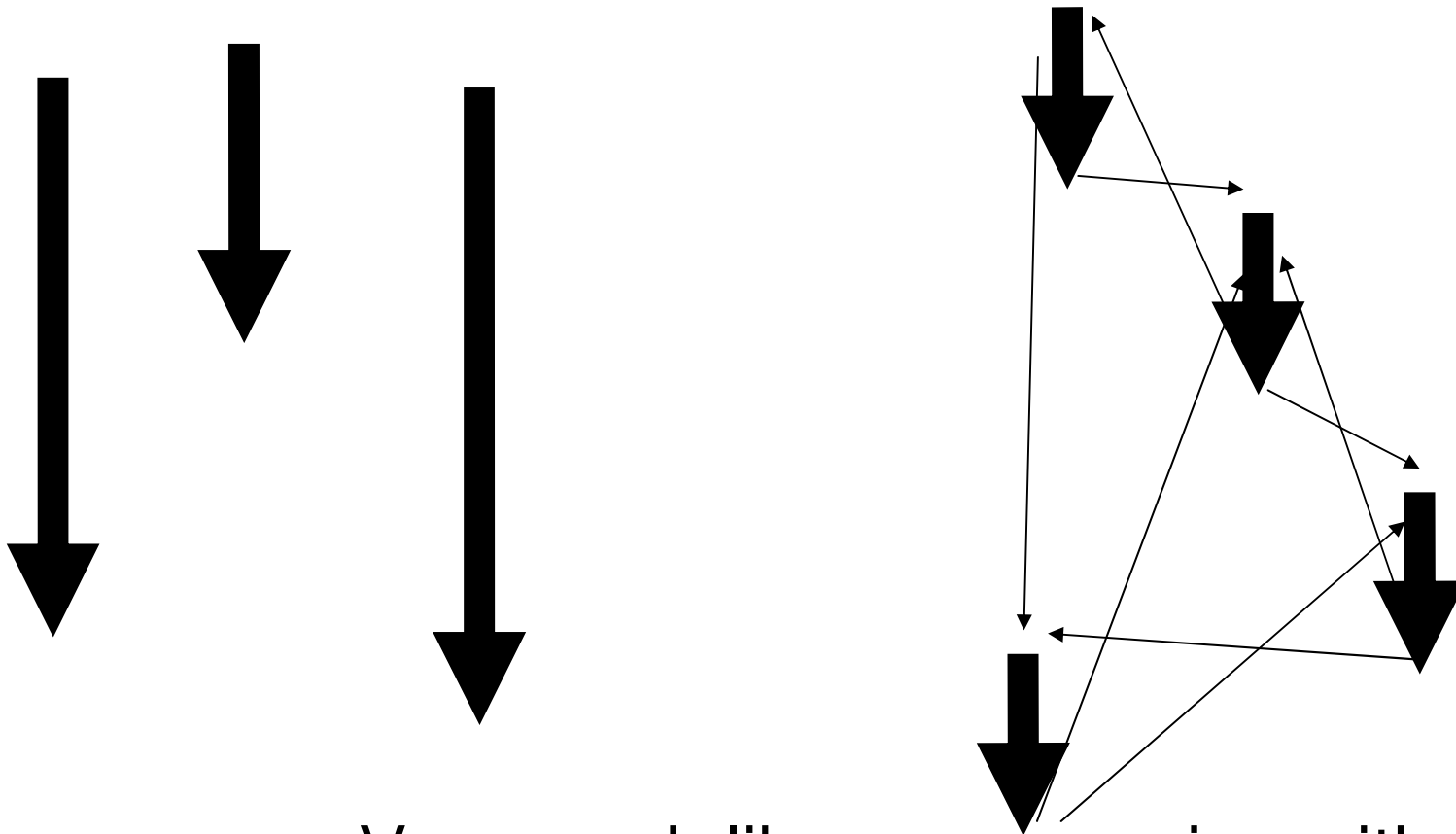
WSN applications are still simple. Both models are made for a single application tightly coupled and embedded with the OS.



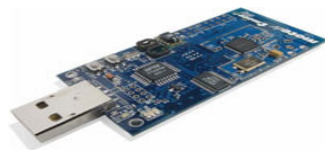
# GAiNe'08

## OS for WSNs

Threads: sequential code flow    Events: unstructured code flow



Very much like programming with GOTOs



# GAiNe'08

## OS for WSNs

- **Task/thread management**
- **HW interrupt management**
- **Network/communication management**
- Time and timers management
- Power management
- Sensors / actuators
- .....

### Programming environment and tools



# GAiNe'08

## Outline

- Introduction
- Embedded Systems
- Embedded Systems Software architecture
- OS for Wireless Sensor Networks
- **TinyOS**
- TinyOS – Components
- Blink application Example
- Programming Hints
- Thank you



# GAiNe'08

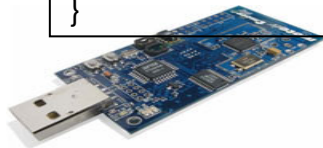
## TinyOS

- Event driven kernel
- Fixed frequency with low power modes in idle periods
- Provides abstractions for
  - Communications
  - Timers
  - Storage
  - Sensors
- Uses a component based programming model (nesC)



- TinyOS is completely non-blocking (split phase)
- Uses a single stack
- Any I/O operation that cannot return immediately become asynchronous and notified by a callback.
- It provides high concurrency
- For longer operations it also provides a Task.

```
//Blocking Example
void main()
{
    // Request Job 1
    sleep(1000);
    // Read Job 1 completion
    // Process Job2
}
```



```
//Split Phase Example
signal alarm_handler();
void main()
{
    // Process Job 1
    alarm(1000, alarm_handler);
    // Process Job2
}
signal timer()
{
    // Read Job 1 completion
}
```

# GAiNe'08

## TinyOS - NesC

- C – Purely functional with Global namespace
- nesC – Still functional but abstracted by Components, namespace is component/module level
- Almost implementation of module is same as C
- Connecting the modules (Components) is tricky – Wiring



	8 bits	16 bits	32 bits	64 bits
signed	int8_t	int16_t	int32_t	int64_t
unsigned	uint8_t	uint16_t	uint32_t	uint64_t



# GAiNe'08

## Outline

- Introduction
- Embedded Systems
- Embedded Systems Software architecture
- OS for Wireless Sensor Networks
- TinyOS
- **TinyOS – Components**
- Blink application Example
- Programming Hints
- Thank you



### Programming LEDs

```
PORTA |= 0x01; // Glow LED1  
PORTA &= 0xFE; //Off LED1
```

### Programming Timer

```
Void set_timer(int interval_ticks)  
{  
    TIMER0 = ticks //Program the interval  
    TIMERCRA |= ENABLE_INTR // Enable the interrupt  
}
```

```
Timer_Interrupt ()  
{  
    //Handle the interrupt  
}
```



# GAiNe'08

- A nesC **application** consists of one or more **components** linked together to form an executable.

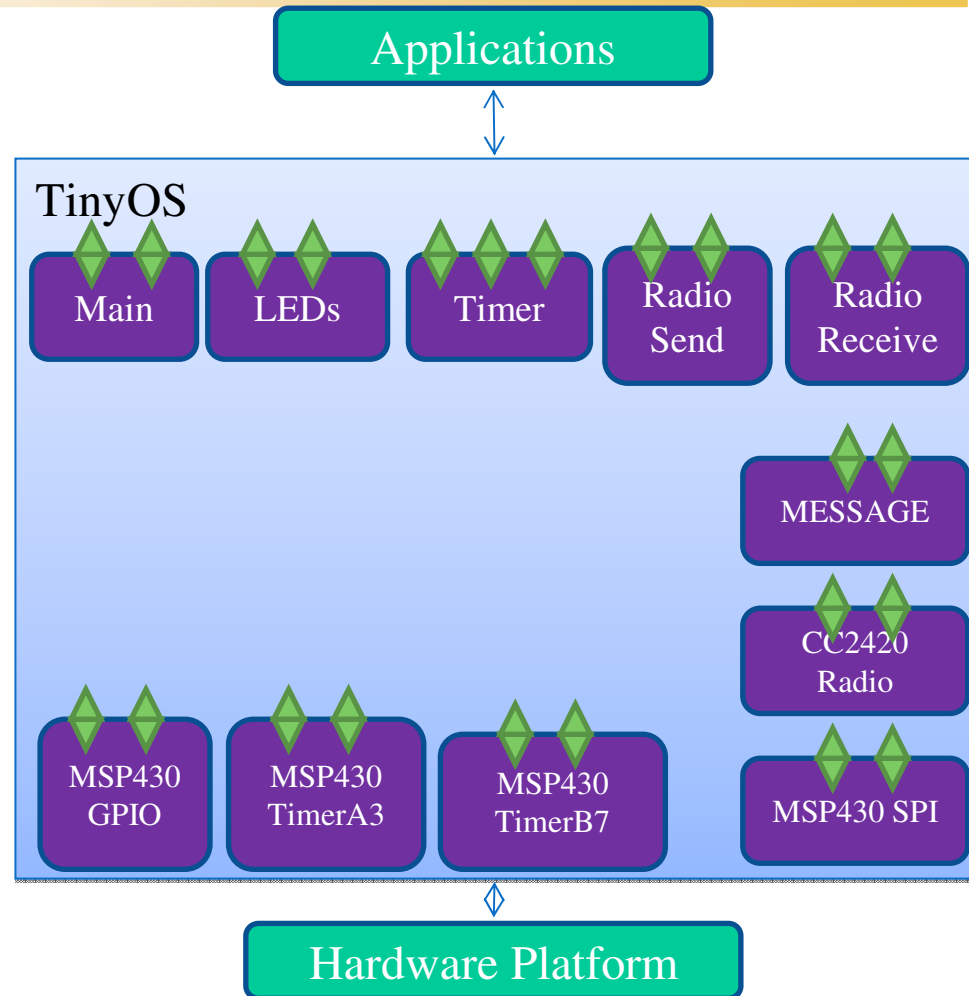
- Abstracts underlying functionality (hardware/software)

- A **component** *provides* and *uses* **interfaces**.

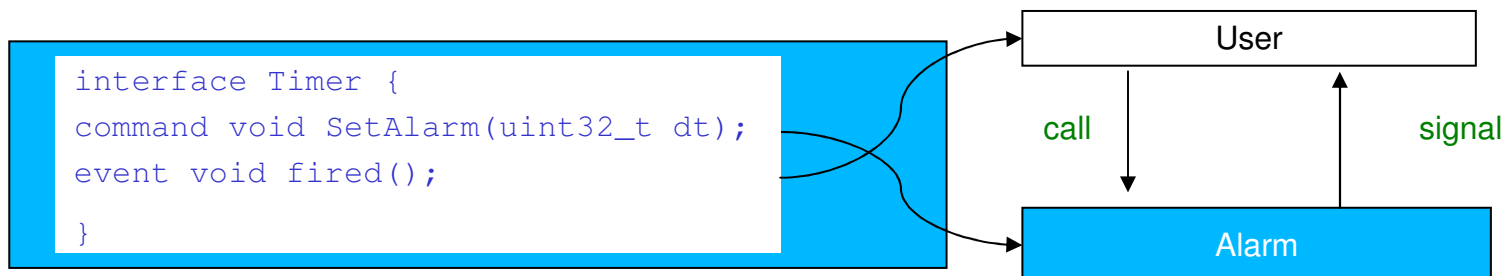
It looks like a library, those **components** are objects in the library and the **interfaces** are APIs. But it actually has more functions than just a library



## TinyOS - Components



- An **interface** *declares* a set of **functions** (written in **C** with some additional **nesC** keywords)
- **Interfaces** are the only point of access to the **component**
- **Interfaces** are bi-directional.
- Each interface may have **commands** that the **interface provider** must *implement* and another set of **functions** called **events** that the **interface user** must *implement*.



- Components can be a Module or a Configuration
- Module – Actual implementation
- Configuration links one or more components together

```
module LedsP {
  provides {
    interface Init;
    interface Leds;
  }
  uses {
    interface GeneralIO as Led0;
    interface GeneralIO as Led1;
    interface GeneralIO as Led2;
  }
}
implementation {
  // implementaiton is omitted

  async command void Leds.led2Off() {
    call Led2.set();
  }

  // omitted
}
```

```
configuration LedsC {
  provides interface Leds;
}
implementation {
  components LedsP, PlatformLedsC;

  Leds = LedsP;

  LedsP.Init <- PlatformLedsC.Init;
  LedsP.Led0 -> PlatformLedsC.Led0;
  LedsP.Led1 -> PlatformLedsC.Led1;
  LedsP.Led2 -> PlatformLedsC.Led2;
}
```



```

module modA {
  provide interface interf_a
  provide interface interf_c
  use interface interf_b
}
Implementation
{
  uint8_t i=0;
  command void interf_a.start() {
    if (interf_b.isSet()) {
      i++;
      signal interf_a.fired();
    }
  }
  command void interf_a.stop() {
    .....
  }
  command void interf_c.get() {
    .....
  }
  event void interf_b.readDone() {
    .....
  }
}

```

```

configuration config {
  provide interface interfA
}
Implementation {
  component modA, configB;
  interfA = modA.interf_a;
  modA.interf_b -> configB.interf_b
}

```

A module **MUST** implement

- every command of interfaces it provides, and
- every event of interfaces it uses

It **should(must??)** also signal

- every event of interfaces it provides

```

configuration(or module) configB {
  provide interface interf_b
}

```

```

interface interf_b {
  command void isSet();
  event void readDone();
}

```

```

interface interf_c {
  command void get();
}

```

```

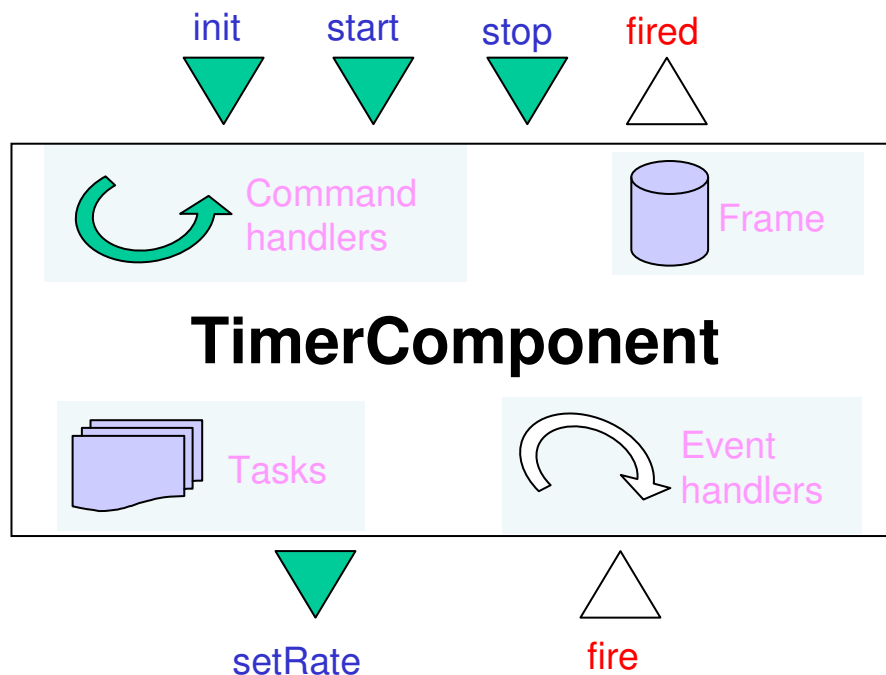
interface interf_a {
  command void start();
  command void stop();
  event void fired();
}

```

- It **must** implements the commands it provides
- It can use the command it **interf\_b** provided by configB because there are wired together
- It **must** implements the events it uses
- It **should(must??)** signal the events it provides

Another file specify the available commands and events in the interface

- Module contains the implementation of the provided interfaces
- It can have state variables
- Can have internal functions (C)



```
module LedsP {
  provides {
    interface Init;
    interface Leds;
  }
  uses {
    interface GeneralIO as Led0;
    interface GeneralIO as Led1;
    interface GeneralIO as Led2;
  }
}
implementation {
  // implementaiton is omitted
  uint8_t count;
  uint8_t myown_set_leds(uint_t a) {
    {
      if (a == 2)
        call Led2.set();
      else.....
    }

    return 0;
  }

  async command void Leds.led2Off() {
    myown_set_leds(2);
  }

  // omitted
}
```

**Configurations** are used to assemble other components together, connecting interfaces used by components to interfaces provided by others

This line export the interface provided by module *modA* through *interfA*

```
configuration config {
  provide interface interfA
}
Implementation {
  component modA, configB;
  interfA = modA.interf_a;
  modA.interf_b -> configB.interf_b
}
```

Specify the components you will *wire*

```
module modA {
  provide interface interf_a
  use interface interf_b
}
Implementation {
  (Your actual code is in here.)
}
```

**Modules** provide the implementations of one or more interfaces

The `->` operator maps between the interfaces of components that a configuration names,  
The `=` operator maps between a configuration's *own* interfaces and components that it names,

```
configuration(or module) configB {
  provide interface interf_b
  use interface interf_c
}
```



- Top level component must be a Configuration
- It should wire MainC component that provides two interfaces
- Name of the file is the top level application component
- The top level Configuration should not provide any interface

```
interface Boot {
  /**
   * Signaled when the system has
   * booted successfully. Components can
   * assume the system has been
   * initialized properly. Services may
   * need to be started to work,
   * however.
   *
   * @see StdControl
   * @see SplitControl
   * @see TEP 107: Boot Sequence
   */
  event void booted();
}
```

```
configuration BlinkAppC
{
}
implementation
{
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;

  BlinkC -> MainC.Boot;

  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
}
```



- Unix style file system – Use Cygwin
- The name of the nesc file will have .nc extension
- Following Convention

File Name	File Type
Foo.nc	Interface
Foo.h	Header File
FooC.nc	Public Module
FooP.nc	Private Module
App.nc	Application file (Configuration)



- Events must be shorter
- Signalling events from the commands could lead to recursion...
- Another aspect is how to do longer computation?

### Task!!!

```
event void Read.readDone(error_t err, uint16_t
val) {
    if (err == FAIL)
        return;
    buffer[index] = val;
    index++;
    if (index < BUFFER_SIZE)
        call Read.read();
}
event timer0.fired()
{
    call Read.read();
}
```

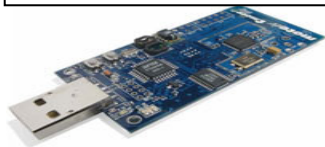


```
module ReadMessageP {
    provides interface StdControl;
    provides interface Read<uint16_t>;
}
implementation {
    uint16_t enable = 0;
    uint16_t Value;
    command error_t StdControl.start() {
        enable = SUCCESS;
        return SUCCESS;
    }
    command error_t StdControl.stop() {
        enable = FAIL;
        return SUCCESS;
    }
    command error_t Read.read() {
        if (enable)
            signal Read.readDone(SUCCESS, Value);
        else
            signal Read.readDone(FAIL, Value);
        return SUCCESS;
    }
}
```

- Events, Commands can post Task
- Tasks are non-preemptive
- Can post itself
- Long running task can create latency for other tasks
- Tasks are local to a Module (ie component)
- Maximum 255 Tasks in the queue

```
event void Read.readDone(error_t err, uint16_t
val) {
    if (err == FAIL)
        return;
    buffer[index] = val;
    index++;
    if (index < BUFFER_SIZE)
        call Read.read();
}

event timer0.fired()
{
    call Read.read();
}
```



```
module ReadMessageP {
    provides interface StdControl;
    provides interface Read<uint16_t>;
}

implementation {
    uint16_t enable = 0;
    uint16_t Value;
    task void Read_task();
    command error_t StdControl.start() {
        enable = SUCCESS;
        return SUCCESS;
    }
    command error_t StdControl.stop() {
        enable = FAIL;
        return SUCCESS;
    }
    command error_t Read.read() {
        post Read_task();
        return SUCCESS;
    }
    task Read_task() {
        if (enable)
            signal Read.readDone(SUCCESS, Value);
        else
            signal Read.readDone(FAIL, Value);
    }
}
```

# GAiNe'08

## TinyOS – Concurrency

- Only Tasks are non-pre-emptive
- interrupt handlers can pre-empt tasks
- Functions that can run asynchronously have to use **async** keyword
- Interrupt handlers must not call any synchronous functions
- race conditions

```
bool state;  
async command bool toggle() {  
    if (state == 0) {  
        state = 1;  
        return 1;  
    }  
    if (state == 1) {  
        state = 0;  
        return 0;  
    }  
}
```

```
toggle()  
state = 1;  
-> interrupt  
toggle()  
state = 0  
return 0;  
return 1;
```

Synchronous Code (SC): code (functions, commands, events, tasks) that is only reachable from tasks.

Asynchronous Code (AC): code that is reachable from at least one interrupt handler.



# GAiNe'08

## TinyOS – Concurrency

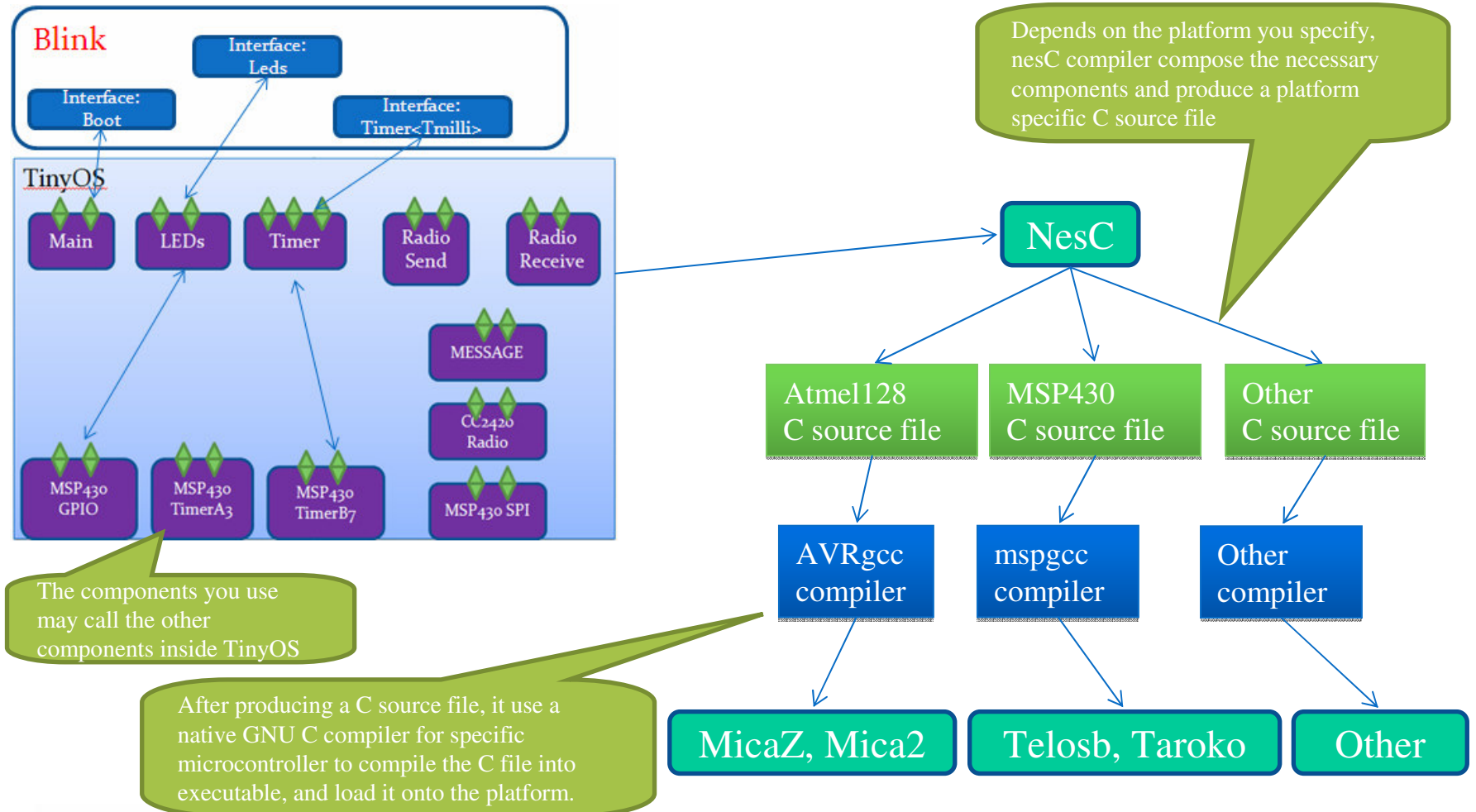
- Need to protect part of the code from interrupt
- atomic keyword

```
bool state;  
async command bool toggle() {  
  atomic {  
    if (state == 0) {  
      state = 1;  
      return 1;  
    }  
    if (state == 1) {  
      state = 0;  
      return 0;  
    }  
  }  
}
```



# GAiNe'08

## Compiling Tinyos application



- Build process is easier
- Uses Make Scripts
- For Tmote
  - `make telosb install,[nodeid] bsl,[serial port no - 1]`
- For Micaz
  - `make micaz install,[nodeid] mib510,[/dev/ttySN]`



- Copying existing sample applications
- For new applications Create a Makefile in the application directory with

```
COMPONENT=BlinkToRadioAppC  
include $(MAKERULES)
```



# GAiNe'08 **Frequently used interfaces, Components**

Component	Interface	Description	Interface location
MainC	Boot, SoftwareInit	Must be wired in top level configuration, Boot.booted is the entry point for applications	/tinyos-2.x/tos/interfaces
TimerMilliC (Generic)	Timer<TMilli>	command void startPeriodic(uint32_t dt), command void startOneShot(uint32_t dt); event void fired();	/tinyos-2.x/tos/lib/timer
LedsC	Leds	async command void ledNOn(); ledNoff,ledNToggle,ledSet	/tinyos-2.x/tos/interfaces
DemoSensorC (Generic)	Read<uint16_t>	command error_t read(); event void readDone( error_t result, val_t val );	/tinyos-2.x/tos/interfaces
ActiveMessageC	SplitControl, AMSend, Receive, Packet, AMPacket, PacketAcknowledgements	command error_t start(); event void startDone(error_t error);	/tinyos-2.x/tos/interfaces
AMReceiverC (Generic)	Receive, Packet,AMPacket		/tinyos-2.x/tos/interfaces
AMSenderC (Generic)	AMSend, Packet, AMPacket, PacketAcknowledgements as Acks		/tinyos-2.x/tos/interfaces



# GAiNe'08

## Outline

- Introduction
- Embedded Systems
- Embedded Systems Software architecture
- OS for Wireless Sensor Networks
- TinyOS
- TinyOS – Components
- **Blink application Example**
- Programming Hints
- Thank you



Blink application /opt/tinyos-2.x/apps/Blink

Displays a counter value in 3 LEDs

- Toggle LED1 at .25Hz (bit0)
- Toggle LED2 at .5Hz (bit1)
- Toggle LED3 at 1Hz (bit2)

Todo

- Initialize
- Initialize the timers
- On timeout toggle the LEDs



## Review of the Blink application - Module

```
module BlinkC {
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
  // implementation code omitted
}
```

module keyword indicate this is a module file

Interface's name

It's parameter

Alias name

In the module, you use the interfaces you need to build the application



## Review of the Blink application - configuration

- Every nesC application start by a top level configuration
  - *wire* the interfaces of the components you want to use
- You already know what components to reference
- In configuration of Blink
  - apps/Blink/BlinkAppC.nc

Configuration keyword indicate this is a configuration file

```
configuration BlinkAppC {  
}  
implementation {  
  components MainC, BlinkC, LedsC;  
  components new TimerMilliC() as Timer0;  
  components new TimerMilliC() as Timer1;  
  components new TimerMilliC() as Timer2;  
  
  BlinkC -> MainC.Boot;  
  BlinkC.Timer0 -> Timer0;  
  BlinkC.Timer1 -> Timer1;  
  BlinkC.Timer2 -> Timer2;  
  BlinkC.Leds -> LedsC;  
}
```

In the configuration, you specific the components you want to reference. This configuration references 6 components



# GAiNe'08

## Review of the Blink application - Wiring

- A full wiring is **A.a->B.b**, which means "interface **a** of component **A** wires to interface **b** of component **B**."
- Naming the interface is important when a component uses or provides **multiple instances of the same interface**. For example, BlinkC uses three instances of Timer: Timer0, Timer1 and Timer2
- When a component provides an interface name

BlinkC component has **one instance** of *Boot* and *Leds* interface, but it has **three instances** of *Timer* interface. So, it can elide the interface name *Boot* and *Leds*, but cannot elide *Timer*.

```
configuration BlinkAppC {  
  implementation {  
    components MainC, BlinkC, LedsC;  
    components new TimerMilliC() as Timer0;  
    components new TimerMilliC() as Timer1;  
    components new TimerMilliC() as Timer2;  
  
    BlinkC -> MainC.Boot;  
    BlinkC.Timer0 -> Timer0;  
    BlinkC.Timer1 -> Timer1;  
    BlinkC.Timer2 -> Timer2;  
    BlinkC.Leds -> LedsC;  
  }  
}
```

```
configuration LedsC {  
  provides interface Leds;  
}  
implementation {
```

```
generic configuration TimerMilliC() {  
  provides interface Timer<TMilli>;  
}  
implementation {
```

```
configuration MainC {  
  provides interface Boot;  
  uses interface Init as SoftwareInit;  
}
```

```
BlinkC.Boot -> MainC.Boot;  
BlinkC.Timer0 -> Timer0.Timer;  
BlinkC.Timer1 -> Timer1.Timer;  
BlinkC.Timer2 -> Timer2.Timer;  
BlinkC.Leds -> LedsC.Leds;
```



# GAiNe'08

## Review of the Blink - Implementation

This module didn't provide interface, it use five interfaces

```
module BlinkC {
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }

  event void Timer0.fired()
  {
    call Leds.led0Toggle();
  }

  event void Timer1.fired()
  {
    call Leds.led1Toggle();
  }

  event void Timer2.fired()
  {
    call Leds.led2Toggle();
  }
}
```

A module **MUST** implement

- every command of interfaces it provides, and
- every event of interfaces it uses

Timer0.startPeriodic(250)

= BlinkC.Timer<TMilli>.startPeriodic(250)

= Timer0.Timer<TMilli>.startPeriodic(250)

= TimerMillic.Timer<TMilli>.startPeriodic(250)

In module, Timer0 is an interface. In configuration, Timer0 is a component

What it says here is pretty straight forward. After the system booted, start the timer periodically. When the timer fired, toggle LED.

```
implementation {
  components MainC, BlinkC, LedsC;
  components new TimerMillic() as Timer0;
  components new TimerMillic() as Timer1;
  components new TimerMillic() as Timer2;

  BlinkC -> MainC.Boot;
  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
}
```



# GAiNe'08

## TinyOS - NesC

declaration

```
int read(int fd, void* buf, size_t count);  
main() {  
  
    :  
    result = read(fd, &val, 1);  
}
```

reference

definition

```
int read(int fd, void* buf, size_t count)  
{  
  
    :  
    return 0;  
}
```



# GAiNe'08

```

module BlinkC
{
  uses interface Timer<TMilli> as
  Timer0;
  uses interface Timer<TMilli> as
  Timer1;
  uses interface Timer<TMilli> as
  Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic(250);
    call Timer1.startPeriodic(500);
    call Timer2.startPeriodic(750);
  }
  event void Timer0.fired()
  {
    call Leds.led0Toggle();
  }
  event void Timer1.fired()
  {
    call Leds.led1Toggle();
  }
  event void Timer2.fired()
  {
    call Leds.led2Toggle();
  }
}

```

```

configuration
{
  implements
  {
    component BlinkC;
    component BlinkC;
    component BlinkC;
    component BlinkC;
  }
}

```

reference

```

configuration
  provides
  {
  }
implementation
  components
  {
    Leds =
    LedsP.Led0;
    LedsP.Led1;
    LedsP.Led2;
    LedsP.Led2 -> PlatformLedsC.Led2;
  }
}

```

```

module LedsP {
  provides {
    interface Init;
    interface Leds;
  }
  uses {
    interface GeneralIO as Led0;
    interface GeneralIO as Led1;
    interface GeneralIO as Led2;
  }
  implementation {
    command error_t Init.init() {
      atomic {
        call Led0.makeOutput();
        call Led1.makeOutput();
        call Led2.makeOutput();
        call Led0.set();
        call Led1.set();
        call Led2.set();
      }
      return SUCCESS;
    }
    async command void Leds.led0On() {
      call Led0.clr();
    }
    async command void Leds.led0Off() {
      call Led0.set();
    }
    async command void Leds.led0Toggle() {
      call Led0.toggle();
    }
    async command void Leds.led1On() {
      call Led1.clr();
    }
    async command void Leds.led1Off() {
      call Led1.set();
    }
  }
}

```

Definition

## LedsC

```



implementation of Leds {
  command void Leds.led0On() {
  };
  command void Leds.led0Off() {
  };
  command void Leds.led0Toggle() {
  };
  command void Leds.led1On() {
  };
  command void Leds.led1Off() {
  };
  command void Leds.led1Toggle() {
  };
  command void Leds.led2On() {
  };
  command void Leds.led2Off() {
  };
  command void Leds.led2Toggle() {
  };
}

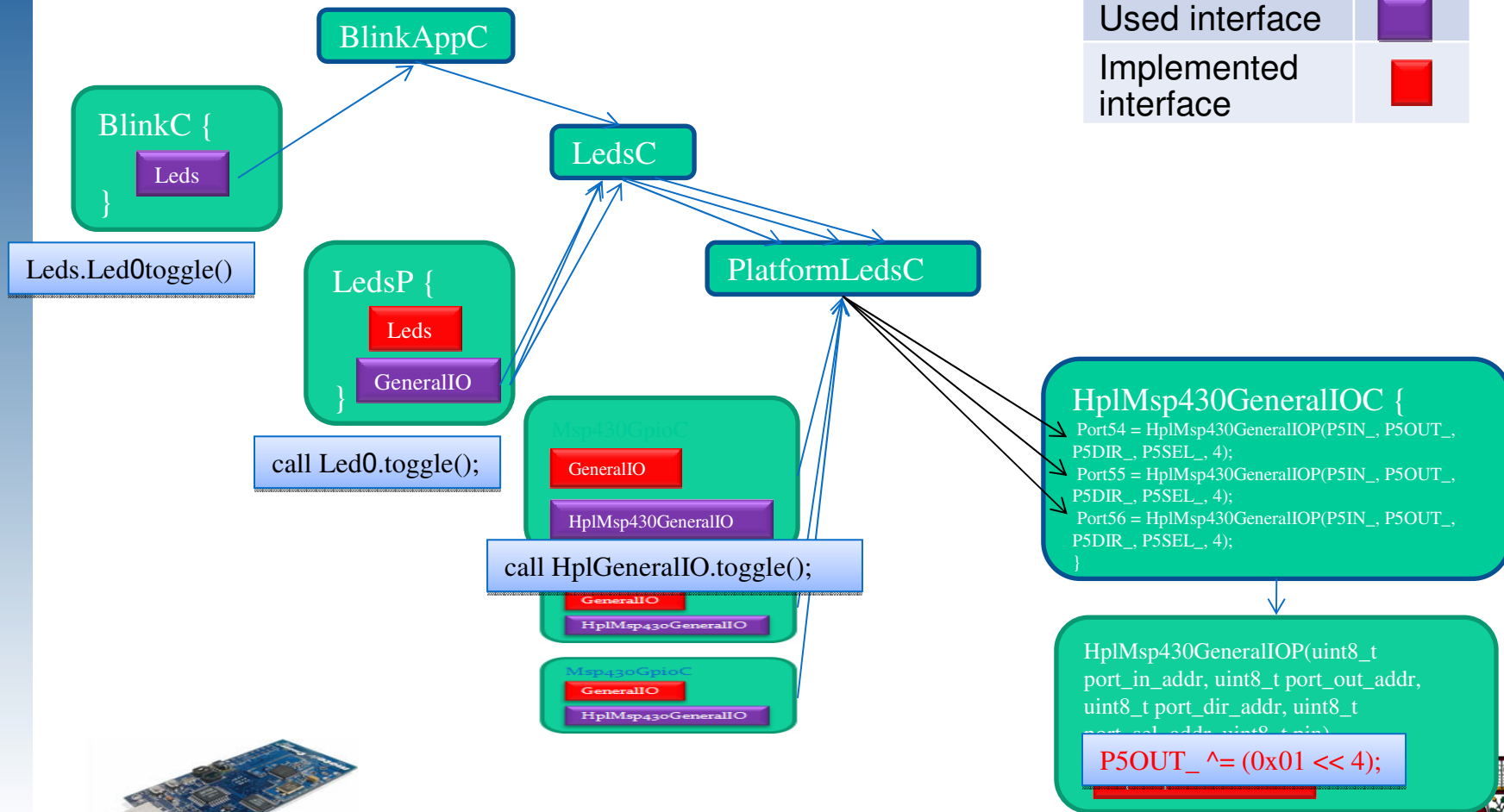
```

Declaration



## Call graph of Leds

Name	colo
Configuration	
Module	
Used interface	
Implemented interface	



Hint 1: It's dangerous to signal events from commands, as you might cause a very long call loop, corrupt memory and crash your program.

Hint 2: Keep tasks short.

Hint 3: Keep code synchronous when you can. Code should be async only if its timing is very important

Hint 4: Keep atomic sections short, and have as few of them as possible.

Thanks, Phil Levis, Tinyos Programming



# GAiNe'08

## References

[http://docs.tinyos.net/index.php/TinyOS\\_Tutorials](http://docs.tinyos.net/index.php/TinyOS_Tutorials)

TinyOS Programming, Philip Levis October 27, 2006

(<http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf>)

<http://cone.informatik.uni-freiburg.de/teaching/lecture/wsn-w06/>

Tinyos Enhancement Proposals, TEPs



# GAiNe'08

Thank you!

- Go raibh maith agaibh
- Thank you!
- शु क् रिया (SUKRYA)
- ந ண் றி (NANDRI)
- Dêkuji
- Dziekuje
- Muchas gracias
- ありがとうございます (ARIGATOU GOZAIMASU)
- 谢谢

