FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

182.694 Microcontroller VU

FAKULTÄT FÜR !NFORMATIK

Martin Perner

SS 2017

Featuring Today:
TinyOS Part 2

Weekly Training Objective

- Already done
  - 3.4.1 Input capture
  - 3.6.3 SPI $*$
  - 3.7.2 Watchdog $*$
- This week
  - 4.2.1 UART to GLCD †
  - 4.2.2 Keypad
  - 4.2.3 Debounced Buttons $*$

TinyOS – Recap

TinyOS

- is a small operating system that is designed to run on low-power wireless sensor nodes, and networked embedded systems.
- provides a set of important services and abstractions.
- defines a concurrent execution model.
- is written in nesC, which is a C dialect.
- can be seen as a mixture of OOP and hardware wiring.

TinyOS first steps

Flash an example application

- in apps/Blink/ execute
  `$ make bigAVR6_1280`
- with connected board execute
  `$ make bigAVR6_1280 install`

A binary is generated and downloaded.

Hint

`$ make bigAVR6_1280 install,id`

Sets `TOS_NODE_ID` to the value of `id`.
This can be used, e.g., for defining the IP address of the node, or allow for different functionality (master/slave).

Interfaces

Enabling reusability

- Multiple interfaces can be used/provided.
- Interfaces are bidirectional.
  - **command**, implemented by provider of the interface, called by user.
  - **event**, implemented by user of the interface, signaled by provider.
- Bidirectionality is the basis for split-phase.

Example Interface

A generic timer interface – Timer.nc

```
interface Timer<precision_tag>
{
    command void startPeriodic(uint32_t dt);
    command void startOneShot(uint32_t dt);
    command void stop();
    event void fired();

    ...
}
```

Interface, and thus timer-usage, independent of microcontroller used.

Configurations

Wiring

TinyOS only allows to wire interfaces. This wiring is done in configurations.

- Connecting interfaces of components via -> resp. <-
- The arrow goes from user to provider!
- Connecting an interface of the configuration to an interface of a component is done with =

Example Configuration

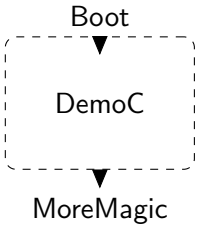
Configuration

```
configuration DemoC
{
    uses interface Boot;
    provides interface MoreMagic;
}

implementation
{
    components DemoP, MakeMagicC;

    Boot = DemoP.Boot;
    MoreMagic = MakeMagicC.MoreMagic;

    MakeMagicC.Magic -> DemoP.Magic;
}
```



Example Configuration

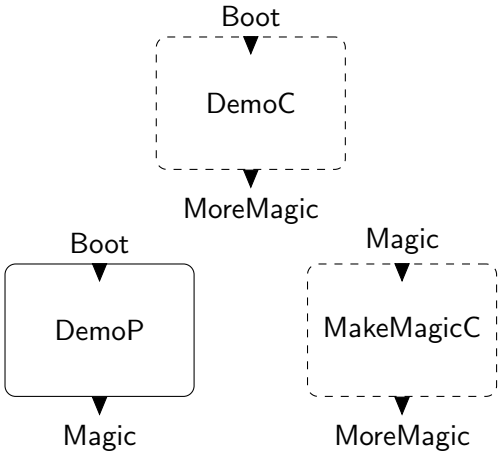
Configuration

```
configuration DemoC
{
    uses interface Boot;
    provides interface MoreMagic;
}

implementation
{
    components DemoP, MakeMagicC;

    Boot = DemoP.Boot;
    MoreMagic = MakeMagicC.MoreMagic;

    MakeMagicC.Magic -> DemoP.Magic;
}
```



Example Configuration

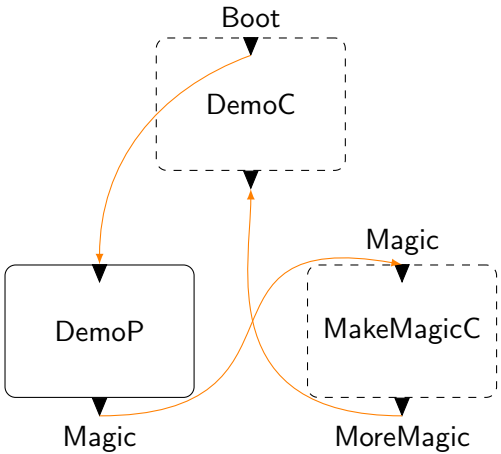
Configuration

```
configuration DemoC
{
    uses interface Boot;
    provides interface MoreMagic;
}

implementation
{
    components DemoP, MakeMagicC;

    Boot = DemoP.Boot;
    MoreMagic = MakeMagicC.MoreMagic;

    MakeMagicC.Magic -> DemoP.Magic;
}
```



Example Configuration

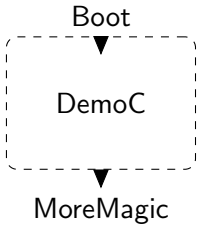
Configuration

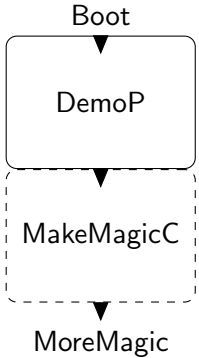
```
configuration DemoC
{
    uses interface Boot;
    provides interface MoreMagic;
}

implementation
{
    components DemoP, MakeMagicC;

    Boot = DemoP.Boot;
    MoreMagic = MakeMagicC.MoreMagic;

    MakeMagicC.Magic -> DemoP.Magic;
}
```





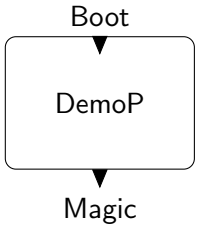
Example Module

Modules – Where code is placed

```
module DemoP
{
    uses interface Boot;
    provides interface Magic;
}

implementation
{
    event void Boot.booted() {
        ...
    }

    command int Magic.get() {
        ...
    }
}
```



Function Examples

- Required by Interface
  - Command function:

```
(async) command uint8_t f(uint8_t x);
y = call f(x);
```

  - Event function:

```
(async) event void am_ready(uint8_t x);
signal am_ready(x);
```

- To be used inside a module
  - Task:

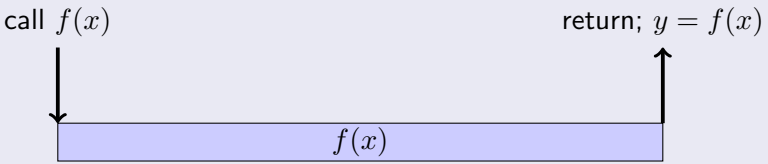
```
task void f();
post f();
```

  - Plain function:

```
uint8_t f(uint8_t x);
y = f(x);
```

Split-Phase

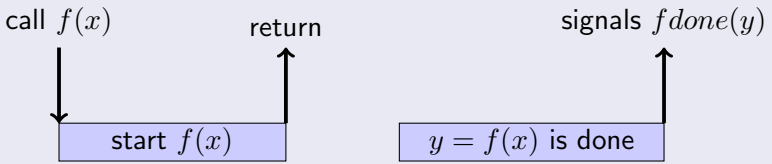
Classic Approach



$f(x)$ access some hardware, and has to wait for a signal by the hardware.
After calling $f(x)$ the exeuction blocks until completion (busy waiting ...)

Split-Phase

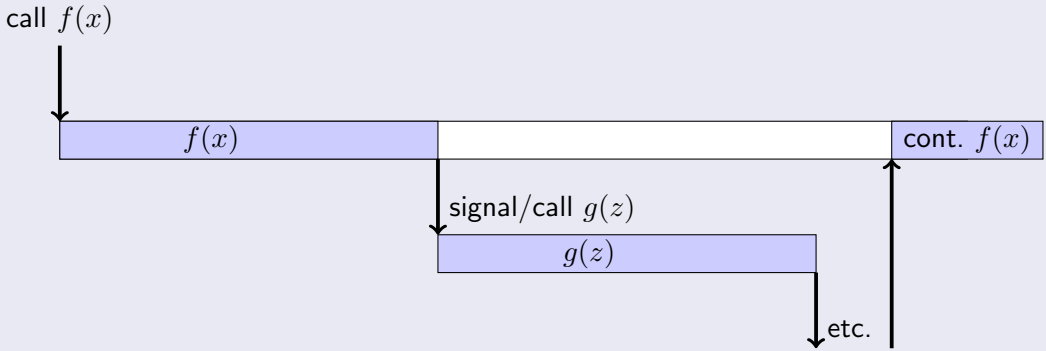
The Split-Phase Approach



After calling $f(x)$, the execution is started in the background. The caller receives a callback (event) upon completion of $f(x)$.

Deferred Task Execution

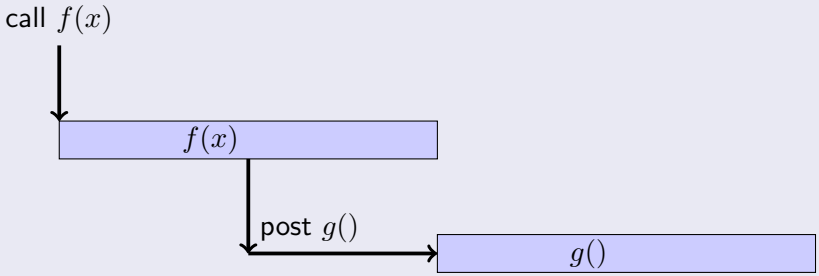
Function call



Assume that the return value and the side-effects of $g(\cdot)$ are not required by $f(\cdot)$. Why should we wait?

Deferred Task Execution

Post a Task



- Task have no parameters.
  Parameter passing needs to be done with state-variables in the module.
- A task can only be posted once onto the tasklist.

Booting

How is TinyOS booted?

- We have seen the interface *Boot* of **MainC** being used for initialization.
- It provides an event `booted`, which is called after start-up of the system.

Booting

tos/system/MainC.nc

```
#include "hardware.h"

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

implementation {
  components PlatformC, RealMainP, TinySchedulerC;

  RealMainP.Scheduler -> TinySchedulerC;
  RealMainP.PlatformInit -> PlatformC;

  // Export the SoftwareInit and Booted for applications
  SoftwareInit = RealMainP.SoftwareInit;
  Boot = RealMainP;
}
```

Booting

tos/system/RealMainP.nc

```
module RealMainP @safe() {
  provides interface Boot;
  uses interface Scheduler;
  uses interface Init as PlatformInit;
  uses interface Init as SoftwareInit;
}

implementation {
  int main() @C() @spontaneous() {
    atomic {
      platform_bootstrap();
      call Scheduler.init();
      call PlatformInit.init();
      while(call Scheduler.runNextTask());
      call SoftwareInit.init();
      while(call Scheduler.runNextTask());
    }
    __nesc_enable_interrupt();
    signal Boot.booted();
    call Scheduler.taskLoop();
    return -1;
  }
  default command error_t PlatformInit.init() { return SUCCESS; }
  default command error_t SoftwareInit.init() { return SUCCESS; }
  default event void Boot.booted(){}
}
```

How many instances of a Component are there?

Do we care?

- If we need some boot-up initialization, we use **MainC** and the *Boot* interface
- Do we get a new component every time we use it?
- No, components are “singletons” (there is only one)
- What can we do if we want two instances of a stateful component? (e.g., Queue)

Generic Components

- Add **generic** in front of signature of configuration/module
- Instantiate with **new** keyword
- We can pass parameters, and even types, at instantiation.

Generic Components – Examples

QueueC.nc

```
generic module QueueC(typedef queue_t, uint8_t queueSize) {
  provides interface Queue<queue_t>;
}
...
```

SomeThing

```
generic configuration SomeThing() {
  ...
}

implementation {
  components new QueueC(uint8_t, 5) as Queue;
  ...
}
```

Generic Component

Limitation

- Contrary to, e.g., C++, the type checking happens on declaration
- At this point, the used type is unknown
- Does $x + 1$ compile for very type used for x?
- The attribute @integer() can be applied to the declaration for some type assumptions.

WeirdC

```
generic module WeirdC(typedef foo_t @integer()) {
  ...
}
```

Abstract Data Type (ADT)

Does a type argument to a component helps?

- How can we use it in an interface?
- Use a typed interface!

QueueC.nc

```
interface Queue<t> {
  ...
  command t head();
  ...
}
```

Recap – Fan-In and Fan-Out

What happens if an interface is connected to multiple component

- If there are multiple modules connected to the same interface, then
  - All connected providers of a command receive the call.
  - All connected users of an event are signaled.
  - The order is not defined!
  - Generally, there is no possibility to determine whose signal caused an event.
- The last item can be circumvented by using parameterized interfaces.

Parameterized Interfaces

Adding caller/callee information to an interface

- Dirty hack: Additional parameter in command/event plus generic component to pass the parameter.
- Better: use parameterized interfaces
  - Adds “implicit” parameter to function call.
  - Interface definition does not need to be changed
  - use/provide clauses are extended.
  - Parameters are assigned in configuration.

Parameterized Interface

Default Cases!

- Implementationwise, parameterized interfaces are more or less a switch.
- There is no compile-time check if every called parameter is wired (could be data dependent!)
- The caller/callee must provide default implementations for parameterized calls.

Example Parameterized Interface 1

- Demo Application
- Count towards a generic value
- Increment after a timer has fired
- Output to a port interface

Example Parameterized Interface 1

CounterC

```
#include "Timer.h"
generic module CounterC(uint8_t top) {
  uses interface Boot;
  uses interface Timer<TMilli> as Timer;
  uses interface Port as CounterPort;
}
implementation {
  uint8_t counter;

  task void increment() {
    counter++;
    if (counter > top) {
      counter = 0;
    }
    call CounterPort.set(counter);
  }
  event void Boot.booted() {
    counter = 0;
    call CounterPort.makeOutput();
    call Timer.startPeriodic(500);
  }
  event void Timer.fired() {
    post increment();
  }
}
```

Example Parameterized Interface 1 – Port

Port

```
interface Port {
  command void makeOutput();
  command void set(uint8_t);
}
```

Example Parameterized Interface 1 – CounterAppC

CounterAppC

```
configuration CounterAppC {
}

implementation {
  components MainC;
  components new CounterC(6) as Counter;
  components new RawPortC((uint8_t) &PORTA, (uint8_t) &DDRA) as PortA;
  components new RawPortC((uint8_t) &PORTB, (uint8_t) &DDRB) as PortB;
  components new TimerMilliC() as Timer;

  Counter.Boot -> MainC.Boot;
  Counter.Timer -> Timer;
  Counter.CounterPort -> PortA.Port[3];
  Counter.CounterPort -> PortB.Port[4];
}
```

Example Parameterized Interface 1 – RawPortC

RawPortC

```
generic module RawPortC(uint8_t port_addr, uint8_t ddr_addr) {
  provides interface Port[uint8_t id];
}

implementation {
  #define PORT (*TCAST(volatile uint8_t * ONE, port_addr))
  #define DDR (*TCAST(volatile uint8_t * ONE, ddr_addr))

  inline command void Port.makeOutput[uint8_t id]() {
    DDR = 0xff;
  }

  inline command void Port.set[uint8_t id](uint8_t value) {
    if (id != value) {
      PORT = value;
    }
    else {
      PORT = 0xff;
    }
  }
}
```

Example Parameterized Interface 1 – Discussion

What have we achieved?

- Caller still unaware of the callees, as before
- Callees got a parameter passed. Not very impressive.
- Generic Components are better fitted for this scenario!

Example Parameterized Interface 2

- Weirder Demo Application
- Count towards a generic value
- Set current value to one port, on overflow set all ports to 0xff

selected.h

```
#ifndef __SELECTED_H__
#define __SELECTED_H__

#define UNIQUE_PORT "unique_port"

#endif
```

Example Parameterized Interface 2

SelectedAppC

```
#include "selected.h"

configuration SelectedAppC {
}

implementation {
  components MainC;
  components new CounterC(6) as Counter;
  components new RawPortC((uint8_t) &PORTA, (uint8_t) &DDRA) as PortA;
  components new RawPortC((uint8_t) &PORTB, (uint8_t) &DDRB) as PortB;
  components new TimerMilliC() as Timer;

  Counter.Boot -> MainC.Boot;
  Counter.Timer -> Timer;
  Counter.CounterPort[unique(UNIQUE_PORT)] -> PortA.Port;
  Counter.CounterPort[unique(UNIQUE_PORT)] -> PortB.Port;
}
```

Example Parameterized Interface 2

CounterC

```
#include "Timer.h"
#include "selected.h"

generic module CounterC(uint8_t top) {
  uses interface Boot;
  uses interface Timer<TMilli> as Timer;
  uses interface Port as CounterPort[uint8_t id];
}

implementation {
  uint8_t counter, i=0;

  task void increment() {
    counter++;
    if (counter > top) {
      for (i=0; i < uniqueCount(UNIQUE_PORT); i++) {
        call CounterPort.set[i](0xff);
      }
      counter = 0;
    }
    call CounterPort.set[counter](counter);
  }
  ...
```

Example Parameterized Interface 2

CounterC cont.

```
  ...
  event void Boot.booted() {
    counter = 0;
    for (i=0; i < uniqueCount(UNIQUE_PORT); i++) {
      call CounterPort.makeOutput[i]();
    }
    call Timer.startPeriodic(500);
  }

  event void Timer.fired() {
    post increment();
  }

   default command void CounterPort.makeOutput[uint8_t id]() {
    DDRF = 0xFF;
  }

  default command void CounterPort.set[uint8_t id](uint8_t value) {
    PINF = 0xFF;
  }
}
```

Example Parameterized Interface 2 – Discussion

Interface *Port* and module **RawPortC** as before.

What have we achieved?

- As the top-value of the counter is larger than 2, PORTF is toggled.
- Caller is aware of the callees.
- Caller needs to provide default implementation to prevent out-of-bound behavior.
- Callees unaware of the caller.

Unique

Why are we using unique?

- Numbers to parameterized interface need to be unique.
- Counting per hand is hard, ugly, and a path to hell ...
- If we need to pass the number of attached interfaces, we would need to pass a parameter to the component. Instead we can use `uniqueCount`.
- There are cases we non-consecutive numbering for the parameterized interface is used, e.g., port numbers.

Requirements

- Counting happens based on a unique string (use a define in a header!)
- `unique` returns a unique id. Numeric from 0 to $n - 1$
- `uniqueCount` returns n

Parameterized Interface at caller and callee

What needs to be changed?

- Add parameter to both sides in wiring.
- Add parameter to interface declaration.
- Add parameter at interface function declaration and usage.

Note: if the same value, generated by `unique`, is used multiple times: use an enum!

```
...
enum {
  COUNTER_PORT1 = unique(UNIQUE_PORT)
};
Counter.CounterPort[COUNTER_PORT1] -> PortA.Port[COUNTER_PORT1];
...
```

Execution Model

The Execution Model of TinyOS

TinyOS has two basic execution modes:

- synchronous
- asynchronous

Execution Model

Tasks in TinyOS

Every synchronous execution in TinyOS is a task.

As there is no preemptive scheduler (per-default), a task runs until it is finished.

- This is problematic for long running computations $\Rightarrow$ defer computation by posting another task.
- Can be interrupted by asynchronous event: interrupt.

Interrupts

- Interrupts are handled in TinyOS as asynchronous executions.
- As usual, they can happen at any time, given interrupts are enabled.
- Do the usual synchronization problems also arise?

Execution Model

Mixing Asynchronous/Synchronous Executions

TinyOS has checks to prevent obvious mistakes, but the programmer still has to take care.

- Variables used in asynchronous context must be accessed in **atomic** sections.
- Functions that can be called from asynchronous contexts must be declared **async**.
- To get from asynchronous to synchronous execution a task has to be posted.

Default Task Scheduling Policy

Non-Preemptive FIFO

- Small, Easy, Fast
- Every task is posted into the task queue
- Queue is processed in FIFO ordering
- Every task can only be posted once
- Every task consumes only 1 byte in the task queue
- Limited to 255 tasks
- Task queue length is evaluated at compile time
- Every task runs until completion
- Dispatch long operations in multiple separate tasks

What about context switches?

Concurrency

- Execution model in nesC is “run-to-completion” tasks
  - No preemption
  - Atomic with respect to other tasks
  - Not atomic with respect to interrupt handlers
- Code divided into two parts:
  - Synchronous Code: functions, commands, events, tasks that are only reachable from tasks
  - Asynchronous Code: used in interrupt handlers (must be marked async)
- Race conditions:
  - No race conditions between tasks
  - Avoid race conditions by protection through an atomic statement
  - Calls to functions are only protected if every call is protected
  - Compiler detects race conditions

Tasks in TinyOS 2.x

How to use a task

```
command void thingy() {
  ...
  post processTask();
  ...
}

...

task void processTask() {
  //do work
  ...
  if (moreToProcess) {
    post processTask();
  }
}
```

The post will only fail iff the task is already posted in task queue and its execution has not started yet.

Task Example

Blink with Tasks

```
module BlinkTaskC
{
  ...
}

implementation
{
  task void toggle() {
    call Leds.led0Toggle();
  }
  event void Boot.booted() {
    call Timer0.startPeriodic(1000);
  }
  event void Timer0.fired() {
    post toggle();
  }
}
```

Splitting Computation via Tasks

Break-up long running computations for reduced latency

Instead of

```
task void computeTask(){
  uint32_t i;
  for(i=0;i<400001;i++){
  ....
  }
}
```

Try

```
task void computeTask(){
  static uint32_t i;
  uint32_t start=i;
  for ( ; i<start+10000 && i<400001; i++){
    ...
  }
  if (i>=400000) {
    i=0;
  } else {
    post computeTask();
  }
}
```

What Really Happens With Tasks

The Foundation in the Scheduler

```
command void Scheduler.init()
{
  atomic {
    memset( (void *)m_next, NO_TASK, sizeof(m_next) );
    m_head = NO_TASK;
    m_tail = NO_TASK;
  }
}

command void Scheduler.taskLoop()
{
  for (;;) {
    uint8_t nextTask;
    atomic {
      while ((nextTask=popTask()) == NO_TASK) {
        call McuSleep.sleep();
      }
    }
    signal TaskBasic.runTask[nextTask]();
  }
}
```

We see that every synchronous execution in TinyOS is based on a task.

What Really Happens With Tasks

Data-Structure usage

```
async command error_t TaskBasic.postTask[uint8_t id]()
{
  atomic{ return pushTask(id) ? SUCCESS : EBUSY; }
}

bool pushTask(uint8_t id){
  if (!isWaiting(id)) {
    if (m_head==NO_TASK) {
      m_head=id;
      m_tail=id;
    }
    else {
      m_next[m_tail]=id;
      m_tail=id;
    }
    return TRUE;
  }
  else {
    return FALSE;
  }
}
```

Threads

Deprecated!

As there was no one willing to maintain threads, they have been deprecated in the current TinyOS development version.

- Thread Based Priority Queues using Preemptive Jobs
  - Bigger
  - Not that easy
  - “Slow”
- Platform independent part:
  - Thread Queue
  - Thread
  - Thread context
- Platform dependent part:
  - the “real” context switching

Benefits of TinyOS

Benefits of TinyOS over plain C

- Predefined modules (hardware drivers)
- Inherent modularization
- Generic modules
- Tasks

Drawbacks of TinyOS

TinyOS Memory Requirements

ATmega1280 static memory:

- Flash: 128 kB
- EEPROM: 4 kB
- RAM: 8 kB

Usage of the Blink demo :

- 1816 B Flash, relates to 1.8 kB or 1.4% (.text segment size)
- 51 B RAM, relates to 0.05 kB or 0.6% (.bss segment size)

This is bad for a simple blinking application, but good for an “operating system”.

Command/Event and the Stack

Be aware of recursive behavior!

A module that wants to quickly sample multiple values from a sensor:

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

The sensor module, for some reason, caches the read values!

```
command error_t Read.read() {
  signal Read.readDone(SUCCESS, sensorVal);
}
```

This results in rapid growth of stack!

Therefore, it is dangerous to signal events from commands! Post a task.

Important Stuff

- Consider the programming hints!
- Use provided modules/configurations! (Crc, Queue, Pool(!), ...)
- Enjoy the benefits of the compostability: develop and test small units!

Questions?