FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

182.694 Microcontroller VU

FAKULTÄT FÜR !NFORMATIK

Martin Perner

SS 2017

Featuring Today:
TinyOS Part 1

Weekly Training Objective

- Already done
  - 3.4.4 PWM signals and glitches
  - 3.6.4 TWI (I^2C) $*$
  - 3.9.1 Keypad
- This week
  - 3.5.2 Noise
  - 3.5.3 Prescaler and accuracy $*$
  - 3.7.5 Dynamic memory analysis
- Next week
  - 3.4.1 Input capture
  - 3.6.3 SPI $*$
  - 3.7.2 Watchdog $*$

Submission Application 1

The Delivery Talk

- The registration is open!
- And will be open until 14.5.
- To enroll a code submission/upload is required!
- There will be a slot for everyone.
  Nonetheless, we urge you to register early!

For more informations regarding application 1 submission, please see the previous lecture slides.

TinyOS

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.

Benefits

Benefits of TinyOS over plain C

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

TinyOS first steps

Get a copy of TinyOS

After the introduction of App2

- from a PC in the TiLab:
  `$ git clone /opt/mcvl/tinyos_ss17.git`
- from home:
  `$ git clone ssh://user@ssh.tilab.tuwien.ac.at/opt/mcvl/tinyos_ss17.git`

TinyOS first steps

Build Environment

- is installed in the lab.
- Checkout the getting started guide on how to install the environment on Debian GNU/Linux. This is not officially supported by us, though.
- In the used version there is no need to set any environment variables.
  - The path to TinyOS has to be configured in the Makefile.
  - If you find documentation telling you to set, e.g., `TOSROOT`, ignore them; they are deprecated.
  - Mixing of the old and the new build environment is not possible!

The TinyOS compiler

How does TinyOS compile code?

- TinyOS's programming language is nesC.
- It is based on C, with a hint of OOP and hardware wiring analogy sprinkled on top of it.
- The used compiler is nescc, which basically performes preprocessing and semantic checks on the input files. The preprocessed input is then feed into `avr-gcc`.

The TinyOS compiler

```
$ make bigAVR6_1280
```

```
[INFO] compiling BlinkAppC to a bigAVR6_1280 binary
nescc -o build/bigAVR6_1280/main.exe   -Os -gcc=avr-gcc -Wnesc-all -fnesc-include=tos
-fnesc-scheduler=TinySchedulerC,TinySchedulerC.TaskBasic,TaskBasic,TaskBasic,runTask,postTask
-fnesc-cfile=build/bigAVR6_1280/app.c -fnesc-separator=__ -I../../tos/platforms/bigAVR6_1280
-I../../tos/platforms/bigAVR6 -I../../tos/platforms/micaz -I../../tos/platforms/mica2
-I../../tos/platforms/mica -I../../tos/chips_ecs/enc28j60 -I../../tos/chips_ecs/vs1011e
-I../../tos/chips_ecs/mrf24w -I../../tos/lib_ecs/keypad -I../../tos/lib_ecs/arp
-I../../tos/lib_ecs/async_queue -I../../tos/lib_ecs/glcd -I../../tos/lib_ecs/icmp -I../../tos/lib_ecs/ip
-I../../tos/lib_ecs/lcd -I../../tos/lib_ecs/llc -I../../tos/lib_ecs/packet_queue -I../../tos/lib_ecs/ping
-I../../tos/lib_ecs/rtc -I../../tos/lib_ecs/stdo -I../../tos/lib_ecs/touchscreen -I../../tos/lib_ecs/udp
-I../../tos/lib_ecs/sdcard -I../../tos/chips_ecs/atm1280 -I../../tos/chips_ecs/atm1280/timerSync
-I../../tos/chips_ecs/atm1280/pins -I../../tos/chips_ecs/atm1280/SoftSPI -I../../tos/chips_ecs/atm1280/spi
-I../../tos/chips/atm1281 -I../../tos/chips/atm1281/adc -I../../tos/chips/atm1281/timer
-I../../tos/chips_ecs/atm128 -I../../tos/chips_ecs/atm128/timerSync -I../../tos/chips/atm128
-I../../tos/chips/atm128/adc -I../../tos/chips/atm128/pins -I../../tos/chips/atm128/spi
-I../../tos/chips/atm128/i2c -I../../tos/chips/atm128/timer -I../../tos/lib/timer -I../../tos/lib/serial
-I../../tos/lib/power -gcc=avr-gcc -mmcu=atmega1280 -fnesc-target=avr -fnesc-no-debug -DPLATFORM_BIGAVR6_1280
-Wall -Wshadow --param max-inline-insns-single=100000 -Wno-unused-but-set-variable -Wno-enum-compare
-I../../tos/system -I../../tos/types -I../../tos/interfaces -DIDENT_APPNAME=\"BlinkAppC\"
-DIDENT_USERNAME=\"mperner\" -DIDENT_HOSTNAME=\"apps1.tilab.tuw\" -DIDENT_USERHASH=0xf3f37213L
-DIDENT_TIMESTAMP=0x5547ad40L -DIDENT_UIDHASH=0xfaa5dae6L -fnesc-dump=wiring
-fnesc-dump='interfaces(!abstract())' -fnesc-dump='referenced(interfacedefs,components)'
-fnesc-dumpfile=build/bigAVR6_1280/wiring-check.xml BlinkAppC.nc -lm
[INFO] script
            1816 bytes in ROM
              51 bytes in RAM
[INFO] size (toolchain):
   text    data     bss     dec     hex filename
   1816       0      51    1867     74b build/bigAVR6_1280/main.exe
[INFO] generating symbol table
[INFO] generating listing
[INFO] creating srec file
[INFO] creating ihex file
[INFO] writing TOS image
[INFO] writing TOS buildinfo
[INFO] running the wiring check
```

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).

Where to find TinyOS code pieces?

Default code location

- example applications: `apps/`
- demos designed for our lab course: `apps_ecs/demos/`
- tinyos exercises stubs: `apps_ecs/exercises/`
- all system interfaces: `tos/interfaces/`
- major hardware independent code, implementing interfaces: `tos/system/`
- hardware dependent code: `tos/chips/`, `tos/chips_ecs/` and `tos/platforms/`
- TinyOS add-ons ,e.g., timer, ethernet, lcd, ...: `tos/lib/` and `tos/lib_ecs/`

Basic Concept

The roots of TinyOS

- TinyOS was designed for WSN (wireless sensor nodes).
- The idea was that every hardware chip on such a mote should be abstracted in its own module.
- There are three basic entities
  - modules, which provide code implementation
  - configurations, which connect modules to provide functionality
  - interfaces, which . . . well you guessed it
- Contrary to traditional programming languages, nesC's interfaces provide also a "direction" for a method.
- Thus, interfaces can be used or provided by configurations/modules.

Interfaces

Enabling reusability

- Interfaces are bidirectional.
  Certain types of functions require the user to implement an endpoint for it.
- Generic interfaces: think of templates in C++ or generics in Java.
  For example, a generic interface for a queue is instantiated with a type.
- Multiple interfaces can be used/provided.

Example Interface

A generic timer interface

```
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

Connection the dots

- We have modules which implement interfaces
- TinyOS only allows to wire interfaces
- How do we connect implemented interfaces?

Configurations

Connection the dots

- We have modules which implement interfaces
- TinyOS only allows to wire interfaces
- How do we connect implemented interfaces?

Wiring

- 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

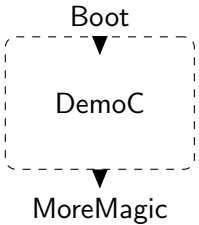
Configuration

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

implementation
{
    components DemoP, MakeMagicC;

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

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



Example Configuration

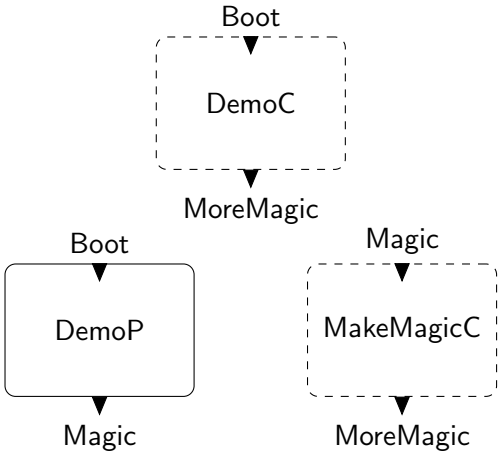
Configuration

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

implementation
{
    components DemoP, MakeMagicC;

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

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



Example Configuration

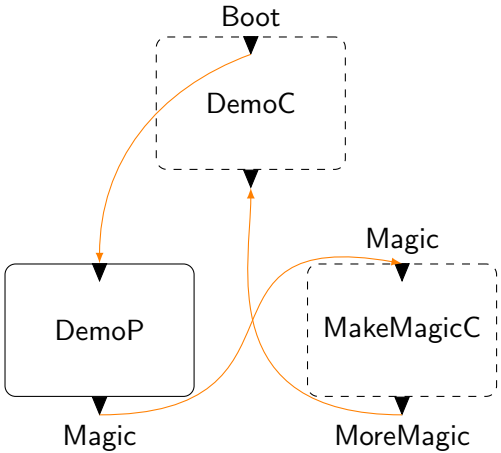
Configuration

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

implementation
{
    components DemoP, MakeMagicC;

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

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



Example Configuration

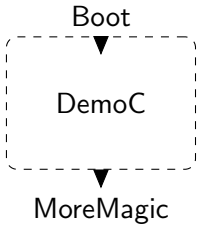
Configuration

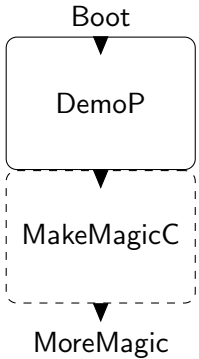
```
configuration DemoC
{
    uses interface Boot;
    provides interface MoreMagic;
}

implementation
{
    components DemoP, MakeMagicC;

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

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





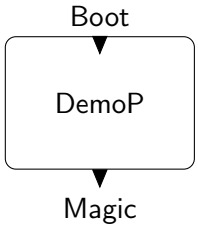
Example Module

Modules – Where code is placed

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

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

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



Functions

Functions in TinyOS

There are of four "types" of functions

- a plain function, called inside a module.
- **command**, implemented by provider of the interface, called by user.
- **event**, implemented by user of the interface, signaled by provider.
- **task**, synchronous execution, posted in a function.

Function Examples

- Plain function:

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

- 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);
```

- Task:

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

Why the Distinction Between Commands and Event?

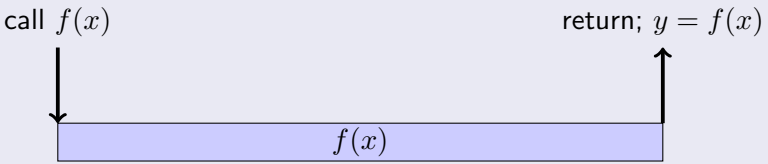
Origins of TinyOS

- A function inside a chip is started via a command, over a few wires.
- The completion of this function is signaled back to user/controller via another wire.

This approach has been implemented in TinyOS, and is called split-phase.

Split-Phase

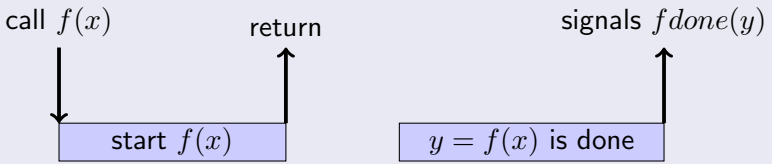
Classic Approach



$f(x)$ 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

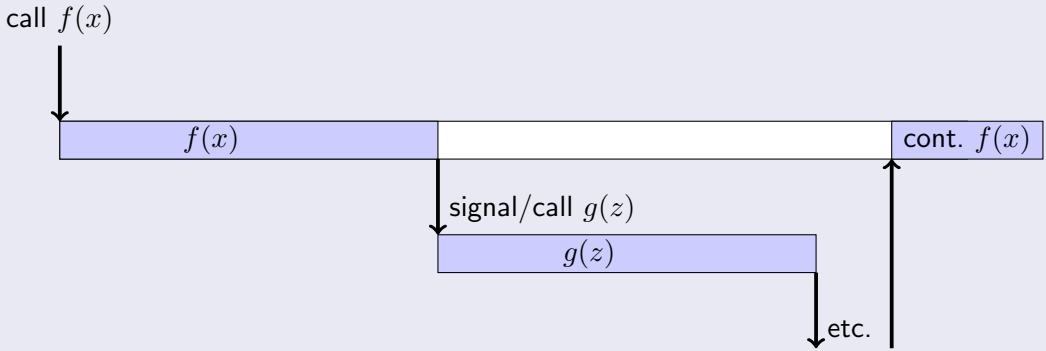
The Split-Phase Approach



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

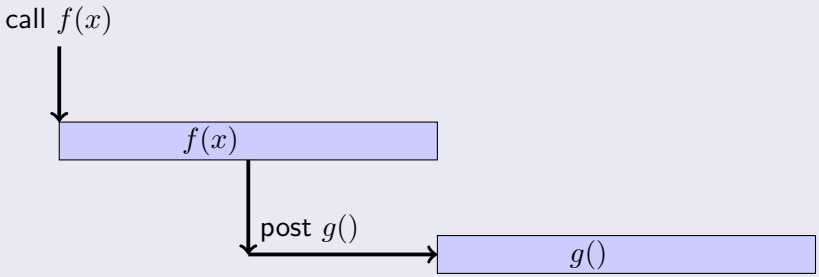
Function call



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.

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.

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.

Fan-In and Fan-Out

What happens if an interface is connected to multiple component

- Again, the hardware analogy comes into play.
- Although, not that literally anymore. 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.

Example Blinker – Module

```
#include "Timer.h"

module BlinkerC
{
  uses interface Boot;
  uses interface Timer<TMilli> as Timer;
  uses interface GeneralIO as BlinkPorts;
}

implementation
{
  event void Boot.booted() {
    call BlinkPorts.makeOutput();
    call Timer.startPeriodic(500);
  }

  event void Timer.fired() {
    call BlinkPorts.toggle();
  }
}
```

Warning: Officially, the TMilli timer fires 1024 times per second!
Although, the port in the lab is tuned to 1000 times per second.

Example Blinker – Top-Level Configuration

```
configuration BlinkerAppC
{
}

implementation
{
  components MainC, BlinkerC;
  components HplAtm1280GeneralIOC as IO; // Plattform-dependent!
  components new TimerMilliC() as Timer0;

  BlinkerC.Boot -> MainC.Boot;
  BlinkerC.Timer -> Timer0.Timer;
  BlinkerC.BlinkPorts -> IO.PortA0;
  BlinkerC.BlinkPorts -> IO.PortA3;
  BlinkerC.BlinkPorts -> IO.PortG2;
}
```

Alternatively, LedsC provides the platform independent Leds interface. Provides 'numbered' leds, so not necessarily what you need.

Naming Convention

Notation for (file)names:

- Postfix C for public usable and stable configurations/modules, e.g., `HilTimerMilliC.nc`.
- Postfix P, for private, unstable configurations and modules.
- No postfix for interfaces.
- The main configuration has AppC as postfix.
- The filename is equal to the name of the configuration/module/interface defined in the file.

Structured Development in TinyOS

Common Structure

- Modules are hidden behind configurations, thus the are usually private.
- In general, more configurations than modules are used.

The Main Configuration (_____AppC.nc)

- Uses and provides no interfaces.
- Only connects components.

Things to remember

Remember

- Read the “manual”. [3]
- There are programming hints for a reason. [3]
- Keep atomic sections short.
- Task are non-preemptive.

Resources

David Gay, Philip Levis, David Culler, Eric Brewer
nesC v1.1 Language Reference
nescc.sourceforge.net/papers/nesc-ref.pdf, 2003.

TinyOS
www.tinyos.net.

Philip Levis
TinyOS Programming
www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf offline!,
/opt/mcvl/tinyos-programming.pdf, 2006.

Questions?