BAKKALAUREATSARBEIT

embedded operating systems / tinyOS

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Bakkalaureus der Technischen Informatik

unter der Leitung von

Dipl.-Ing Alexander Kössler
Institut für Technische Informatik 182

durchgeführt von

Harald Glanzer
Matr.-Nr. 0727156
Hardtgasse 25 / 12A, 1190 Wien

Wien, im September 2011 .....

embedded operating systems / tinyOS

Subject of this work is to give an introduction to the TinyOS Embedded Operating System, to explain the internal structure of this OS and to present its main components. Additionally, tinyOS is compared to another embedded OS, MicroC/OS-II. Because tinyOS will be used in future lectures at the TU Wien, there is also a chapter about installing tinyOS from scratch. Also, there is a chapter about new tinyOS modules for the bigAVR6 - developementplatform that have been written by me in the course of my project thesis.

Contents

1. Introduction

This work is about explaining tinyOS. tinyOS is an opensource operating system, designed for use with wireless embedded sensor networks. There are 2 major stable branches, v.1.x and v2.x, which are not compatible to each other. tinyOS 2.x introduced some major improvements, for example the task scheduler was completely redesigned. In it's new version, the whole project us now distributed under the new BSD license too.

Because of tinyOS's component-based architecture, a highlevel programmer does not have to care about microcontroller specifics, as long as the necessary modules are already exisiting. So, implementing new applications or changing exisiting ones is an easy and fast task. There are already existing implementations for a range of popular hardware notes, as for example the mica, iris and teleosa motes.

For developing applications in tinyOS, NesC is used. NesC stands for Network Embedded Systems C, which is very similar to C/C++. Components in NesC are related to objects in C++.

Embedded systems are designed for one or a few specific tasks, perhaps in combination with realtime constraints. Because embedded systems are often battery powered, one of the main requirements is low power consumption, so that high operation times can be achieved - flexibility is not that important.

This work has to be considered as part of another work, where tinyOS was ported to a new platform, the bigAVR6 development platform. So, a big part of this work handles about the bigAVR6 platform and explains some new written softwaremodules for highlevel-use of this platform's peripherals. It is also explained how to get a running buildenvironment for writing applications or extending the modules for an exisiting platform from scratch.

1.1. Motivation and Objectives

Ultimate goal of this work is to provide an easy-to-use manual for using tinyOS to the reader, so that a buildenvironement can be set up from scratch step-by-step without any necessary previous knowledge. By comparing tinyOS to another embedded operating system - MicroC/OS-II - the specific characteristics

of tinyOS will get much clearer. Finally, by guidung through the selfwritten software modules for the bigAVR6-board, the reader will get familiar with how to practically extend the tinyOS framework by using as much of the preexisting components and how to properly integrate the selfwritten code into tinyOS.

1.2. Structure of the Thesis

The thesis is structured as follows:

Chapter 2 gives a more detailed overview over tinyOS and explains its internals. It also compares tinyOS to another embedded operation system, MicroC/OS-II.

Chapter 3 gives an introduction to nesC.

Chapter 4 introduces the developement platform bigAVR6, for which tinyOS was ported to by the author, and explains the supported devices for this platform.

Chapter 5 introduces another embedded operation system, named MicroC/OS-II, and compares it's main characteristics to tinyOS.

Chapter 6 is a step-by-step howto for setting up a fresh tinyOS - environment.

z

2. tinyos

2.1. tinyOS Basics

2.1.1. What is tinyOS

tinyOS is a free and open operating system for hardware motes, which is written in nesC. A hardware mote is a microcontroller based node in a wireless sensor network, capable of reading sensory information, processing and exchanging of this data with other nodes. Communication typically goes on over wireless network, because so the cost for deployment and maintenance is reduced, and because wires are not feasible in many environments.

Development of tinyOS started as a collaboration of Berkeley University with Intel Research and Crossbow Technology, a california based company. The following requirements were defined for tinyOS:

- Require very few resources
- Allow fine-grained concurrency
- Adapt to hardware evolution
- Support a wide range of applications
- Robust Design
- Support a diverse set of platforms

One of tinyOS's most important features is that tinyOS applications are built out of components, which are connected or wired to each other by interfaces. Components can easily be developed, extended and reused. Another advantage is that a component can be built in software **or** hardware, increasing flexibility.

2.2. History

Developement of tinyOS started in 1999 at Berkeley university. The first supported platform was a mote called 'WeC', as shown in figure 2.1. For communication, this platform is equipped wit an radio device and SPI and UART - interfaces. As cpu an Atmel AVR AT90LS8535 microprocessor, clocked with 4MHz, is used. A major advantage of this mote is that it can be programmed over the wireless interface.

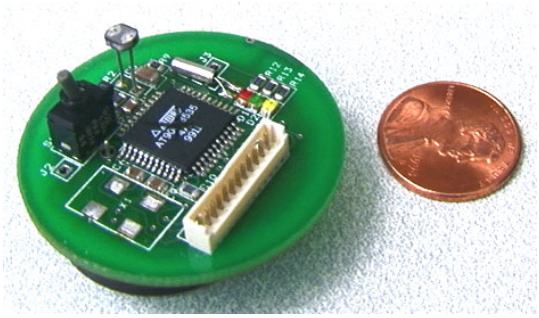

Figure 2.1.: WeC Mote

In the next years, the 'rene' and 'mica' platforms are developed. In the year 2002, work on the nesC programming language began. Until that, tinyOS consisted of a mixture of C files and Perl scripts. In the same year, tinsOS 1.0, the first tinyOS implemented in nesC, is released.

Improvement of tinyOS 1.x went on until february 2006, when tinyOS 2.0 beta1 was released. Version 2.1 was finished in april 2007. Among numerous bugfixes, a cc2420 wireless radio stack implementation was added in this release. tinyOS 2.02 was released some months later, which included an cc2420 stack reimplementation and bugfixes. After that, in august 2008, support for the 'iris' and 'shimmer' platforms was added by distribution of version 2.1. Clearly, bugfixes were included in this release too.

At the time of this writing, the latest tinyOS version is 2.1.1, which was released in April 2010. Most important add-ons are support for the 'mulle', 'epic' and 'shimmer2' - platforms.

2.3. Supported Platforms

With version 2.1.1, a range of hardware motes are supported out-of-the-box. A brief description of this platforms and its most important communication devices are given in the following list. Obviously, most of this platforms provide interfaces for UART, I2C, SPI or others peripherals, depending of the

microcontroller and extension boards used, too. These features are not listed here.

- btnode3: Atmega128L cpu and radio/bluetooth communication devices
- epic: MSP430 cpu and CC2420 radio chip
- eyesIFX: MSP430F149/F1611 and TDA5250 wireless transceiver
- intelmote2: PXA271 XScale cpu and CC2420 radio chip
- iris: Atmega1281 and AT86RF230 radio chip
- mica: Atmega103 and TR1000 radio chip
- mica2: Atmega128L and Chipcon 868/916 radio chip
- mica2dot: Atmega128 and cc1000 transceiver
- micaz: Atmega128 and CC2420 radio chip
- mulle: Renesas M16C and AT86RF230 transceiver
- sam3s_ek: SAM3S4C chip and cc2520 transceiver
- sam3u_ek: SAM3S4C chip and cc2420 transceiver
- shimmer / shimmer2 / shimmer2r : MSP430 cpu and CC2420 transceiver
- span: MSP430 cpu and CC2420 transceiver
- telosa / telosb: MSP430 cpu and CC2420 transceiver
- tinynode: MSP430 cpu and Semtech SX1211 transceiver
- ucmini: Atmega128RFA1 cpu(low power transceiver cpu integrated)
- z1: MSP430 cpu and CC2420 transceiver

2.4. tinyOS hardware abstraction

When looking at the supported hardware platforms, one can see that many platforms use the same cpu and/or communication hardware. To avoid rewriting of code, hardware abstraction is used. By introducing hardware abstraction, it is easier to port applications from one platform to another, and application development itself gets easier too. On the other hand, abstraction means generalisation, which is problematic because hardware motes only have very limited resources and strict energy-efficiency requirements. So, tinyOS uses a 3-level *Hardware Abstraction Architecture* to provide a flexible and performant framework to build applications on, as shown in figure 2.2.

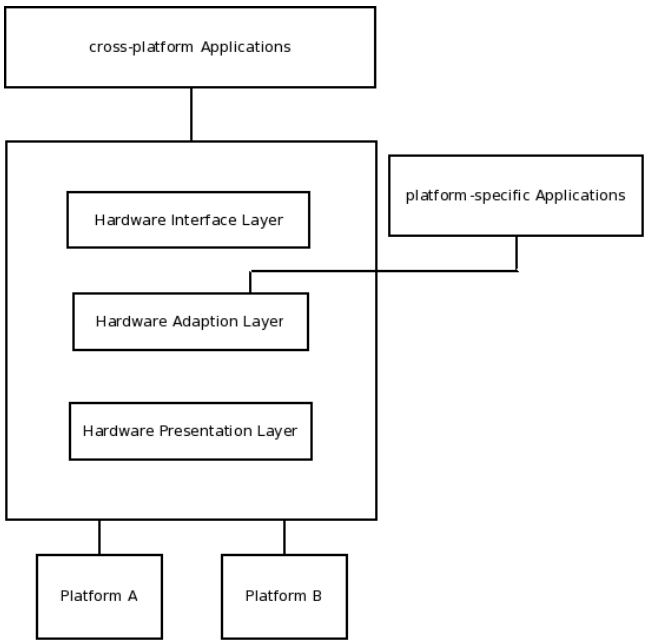

Figure 2.2.: Hardware Abstraction Architecture

In contrast to other embedded OS that use only 2 layer abstraction, the third tinyOS layer provides more flexibility. For maximum performance, a *Platform-Specific-Application* can directly hook into the *Hardware Adaption Layer*, circumventing the *Hardware Interface Layer*.

2.5. tinyOS internals

2.5.1. Basic - Scheduler

By default, tinyOS 2.x uses a **non-preemptive FIFO** scheduler with a maximum of 255 parameterless tasks waiting for execution. A task can only be

scheduled once at a time, if periodic execution is needed the task has to repost itself just before finishing. The scheduler itself consists of an interminable for-loop, that pops (=Â executes) one task after the other, in sequence as this tasks got pushed. If no tasks are waiting for execution, the scheduler enters a powersaveing-mode immediatly.

Because the scheduler of tinyOS is implemented as a component it is possible to replace this default FIFO-scheduler with a selfwritten one. See [FIX] for details how to implement a selfwritten scheduler.

2.5.2. Microcontroller Power Management

To reduce power consumption, a microcontroller should always run in the lowest power state possible. For example, if you want to run an embedded system that is powered by two AA batteries with a capacitiy of 2.7Ah, for one year, you reach an average of about 1mW of power consumption. Clearly, this can only be achieved by saving energy whenever possible.

As mentioned above, tinyOS enters a low power mode if the task queue is empty. Normally, microcontrollers support a range of low power modes, the ATmega128 - for example - supports up to 6 different power saving modes. To decide what mode fits best, tinyOS uses the control- and statusregisters to find the proper lowpower-mode.

For example, on a ATmega128-based platform, the cpu-specific powersaving-mode *IDLE* is to be choosen if one or more timer, SPI, UART or I2C are in use. If the ADC-submodule is working, another mode, *ADC Noise Reduction*, is entered. If none of this modules are active and the task queue is empty, the cpu is set to *POWER DOWN*, from which it only can resume by some external interrupts/resets or a SPI address match interrupt.

Because entering powersaving modes always come with wakeup latency, problems can arise if some higher-level hardware modules have timeing constraints if the wakeup latency is too big. To solve this, the powersaving mode, found by examing the control- and statusregisters, can be overridden by a higherlevel module. For example, when going to a sleepmode just befor an alarm of a timer would occur, the wakeup latency could propably cause a miss of this alarm. Therefore, a component can *provide* an interface named *lowestState* that will be called when a powersave mode is requested, and can override the in principle valid powersaveing mode found by looking at the control- and statusregisters.

2.5.3. Boot Sequence

When tinyOS is booting up, it uses 3 interfaces:

- Init
- Scheduler
- Boot

Init has one command, called *init()*, that is responsible for initializing hardware components. Initialization must happen sequential, so a component is allowed to use a spin loop when - for example - waiting for an interrupt. **Scheduler** is responsible for initializing the task queue. Finally, **Boot** signales the completion of the bootup-process to the application.

2.5.4. Ressource Arbitration

One major task of every operating system is the management of available resources, like communication interfaces that are used by different components. tinyOS distinguishes between 3 different kinds of abstractions:

- dedicated resources
- virtualized resources
- shared resources

A **dedicated** resource is a resource which is allocated by one and only one subsystem all the time. Obvious, no sharing policy is needed here. Examples for such resources are counters and interrupts.

Virtualized resources are used by multiple clients through software virtualization. Here, every client interacts like using a dedicated reosource. Because virtualization is done in software, there is no upper bound on the number of clients(apart from memory/efficiency constraints), with all virtualized instances being multiplexed on top of the underlying resource. Clearly, this virtualization goes along with cpu-overhead.

For example, this concept is used for timers on ATmega128-based platforms. Every time a new timer is instantiated, tinyOS builds a virtual timer on top of the physical timer 0.

The **shared** concept is used for modules that need exclusive access to a resource for some time. An arbiter is responsible for multiplexing between the different clients that want to use the resource. As long as a client *holds* a resource, it has complete access to it. tinyOS arbiters assume that clients are cooperative, that means that a client only acquires a resource when needed,

and only holds it as long as needed, releasing it as soon as possible. No concept of preemption is used here, so client B that needs a resource cannot force Client A to release it.

The arbiter, the centralized place that knows whether a resource is busy or not, is an interface that must be instantiated by every client that wants to use the resource. After that, the client can *request* the resource. The request is queued by the arbiter if the resource is busy. As soon as the resource is available, a special event, *granted*, is signaled to the client, who has now exclusive access to it. It is the client's responsibility to *release* it so soon as possible. To avoid monopolizing a resource a *request* for a resource is only queued if there is no other request of this client queued yet.

An example for such a *shared* resource would be a bus, for example SPI, as show in codelistings 2.1 and 2.2. This listings are part of an implementation for sending and receiving IEEE8023 - frames with an 10MBit ENC28J60 chip. A microcontroller can communicate with this chip by SPI. Given that at least one other component could need independent access to SPI too, IEEE8023C *requests* the bus whenever data has to be exchanged between the microcontroller and the ethernet chip. Once the resource SPI is available, the arbiter signals this to IEEE8023C, which has implemented the proper handler for this situation, named *Resource.granted()*. Inside this handler, IEEE8023C is in full control of the SPI bus, regardless of other components that may be using SPI too. After finishing it's work, IEEE8023C **MUST** free the resource by calling *Resouce.release()*. From this time on, other components can use the resource.

Listing 2.1: IEEE8023P.nc

```
#include "IEEE8023.h"

module IEEE8023P
{
        provides interface IEEE8023;
        uses interface GeneralIO as ssMMC;
        uses interface GeneralIO as ssETH;
        uses interface GeneralIO as rstETH;
        uses interface HplAtm128Interrupt as intETH;
        uses interface SpiByte;
        uses interface Resource;
}

implementation
{

        command uint8_t IEEE8023.init()
        {
                /*
                        ... unimportant code is not shown here...
                */

                /*
                         queue a request for this resource now...
                */
```

```
                if(call Resource.request() == FAIL)
                {
                        return FAIL;
                }
                else
                {
                        stateETH = IEEE8023_INITIALIZING;
                        return SUCCESS;
                }
        }

        /*
                eventhandler will be called when request was queued and
                resource becomes available
        */
        event void Resource.granted(void)
        {
                /*
                        we are now in full control of the resource
                        read/write data to resource
                */

                call Resource.release();
                /*
                        NO MORE access to the resource now
                        queue another request to get in control again
                */
        }

}
```

Listing 2.2: IEEE8023C.nc

```
configuration IEEE8023C
{
        provides interface IEEE8023;
}

implementation
{
        components IEEE8023P;
        components Atm128SpiC as SPI;

        IEEE8023 = IEEE8023P;
        IEEE8023P.SpiByte -> SPI.SpiByte;
        IEEE8023P.Resource -> SPI.Resource[0];

        /*
                unimportant components and wiring not shown here
        */

}
```

3. nesC

3.1. Fundamental Programming Hints

Before diving into this C - dialect, the most important programming hints are listed, as given in [Lev]

- It's dangerous to signal events from commands, as you might cause a very long call loop, corrupt memory and crash your program.
- Keep tasks short.
- Keep code synchronous when you can. Code should be async only if its timing is very important or if it might be used by something whose timing is important.
- Keep atomic sections short, and have as few of them as possible. Be careful about calling out to other components from within an atomic section.
- Only one component should be able to modify a pointer's data at any time. In the best case, only one component should be storing the pointer at any time.
- Allocate all state in components. If your application requirements necessitate a dynamic memory pool, encapsulate it in a component and try to limit the set of users.
- Conserve memory by using enums rather than const variables for integer constants, and don't declare variables with an enum type.
- In the top-level configuration of a software abstraction, auto-wire Init to MainC. This removes the burden of wiring Init from the programmer, which removes unnecessary work from the boot sequence and removes the possibility of bugs from forgetting to wire.

- If a component is a usable abstraction by itself, its name should end with C. If it is intended to be an internal and private part of a larger abstraction, its name should end with P. Never wire to P components from outside your package (directory).
- Use the as keyword liberally.
- Never ignore combine warnings.
- If a function has an argument which is one of a small number of constants, consider defining it as a few separate functions to prevent bugs. If the functions of an interface all have an argument that?s almost always a constant within a large range, consider using a parameterized interface to save code space. If the functions of an interface all have an argument that's a constant within a large range but only certain valid values, implement it as a parameterized interface but expose it as individual interfaces, to both minimize code size and prevent bugs.
- If a component depends on unique, then #define a string to use in a header file, to prevent bugs from string typos.
- Never, ever use the 'packed' attribute.
- Always use platform independent types when defining message formats.
- If you have to perform significant computation on a platform independent type or access it many (hundreds or more) times, then temporarily copying it to a native type can be a good idea.

3.2. Namespace, Components and Interfaces

nesC applications consist of components that get wired together. A component can be a *configuration* or a *module*. Modules implement some functionality, while configurations describe how components are put together.

nesC uses a purely local namespace. This means that a component has to declare all the functions it calls. An advantage in this approach is that all the functions that get called are known at *compile time* - no RAM is used for storing function pointers, and no runtime allocation is required. This is possible because of the nature of nesC - applications: this applications run on embedded systems with well-defined and tightly specified uses, and it is much more important to save memory than to provide flexibility by dynamic linking.

This declaration is done in the *specification* of the component. In this codeblock, the component declares what function it *provides* and what functions it *uses*. This would be such a specification:

```
module doSomethingC {
  provides command <returnvalue> providedFunctionality(<arguments>);
  uses command <returnvalue> usedFunctionality(<arguments>);
}
```

In practice, instead of declaring the individual functions, a component specifies what *interfaces* is uses and it provides. When specifying that a certain interface is used, the component can use or *call* all of the functions that are implemented by this interface. But because this functions can be implemented by different components in different ways, a configuration is needed: there, the calling component is connected to another component that *provides* the needed functions through it's interfaces. This action is called *wiring*.

Consider this code example, taken from the tinyOS powermanagement subsystem:

```
interface StdControl {
  command error_t start();
  command error_t stop();
}
```

This interface named StdControl is *used* by components that that need to turn off or on other components, while StdControl is *provided* by components that represent an abstraction or a service.

For example, it should be possible to enable and disable UART-transmission. So, the UART-module provides the interface StdControl. Mind the keyword *as* in the following code snippet. This keyword just renames the interface for local use and can be ommited.

```
module HplAtm128UartP {
...
  provides interface StdControl as Uart0TxControl;
...
```

The actual work is done in the module's *implementation*. Keep in mind that the *providing* module **MUST** implement all of the provided functions:

```
...

command error_t Uart0TxControl.start() {
  SET_BIT(UCSR0B, TXEN);
  call McuPowerState.update();
  return SUCCESS;
}

command error_t Uart0TxControl.stop() {
  CLR_BIT(UCSR0B, TXEN);
  call McuPowerState.update();
  return SUCCESS;
```

```
}

...
```

The next listing shows how wiring works:

```
configuration Atm128Uart0C {
  provides interface StdControl;
  ...
}

implementation{

  components new Atm128UartP() as UartP;
  StdControl = UartP;
  UartByte = UartP;
  UartStream = UartP;
  UartP.Counter = Counter;

  components HplAtm128UartC as HplUartC;
  UartP.HplUartTxControl -> HplUartC.Uart0TxControl;
  ...
}
```

So, every higher-level component can switch on and off transmission on the Uart0 - interface, and the implementation also notifies the system that some hardware was started or stopped, an information that is used by the power-manangement system, as described earlier. Codelisting 3.1 demonstrates how an application can make use of such a interface.

At first, we define what components we want to use, and how they are put(*wired*) together.

Listing 3.1: MyPowersavingAppC.nc

```
/*                                                                    tab:4
        Harald Glanzer, 0727156 TU Wien

        Boot hardware, and turn on UART0 - TX when system is up

        this program uses the given components MainC and Atm128Uart0C
        component 'MyPowersavingC' is the app itself
*/

configuration MyPowersavingAppC
{
}
implementation
{
        components MainC;
        components Atm128Uart0C;
        components MyPowersavingC;

        /*
        wire our app to the interface 'Boot'
        this interface is provided by component 'MainC'
        */
        MyPowersavingC -> MainC.Boot;

        /*
```

```
        wire our app to the interface 'StdControl'
        this interface is provided by component 'Atm128Uart0C'
        */
        MyPowersavingC -> Atm128Uart0C.StdControl;
}
```

After that, we can use the interfaces provided by the used components. See listing 3.2 for how that works.

Listing 3.2: MyPowersavingC.nc

```
/*
        Harald Glanzer, 0727156 TU Wien

        interface 'Boot' provides the event 'bootet' to us,
        which MUST BE implemented by the user of 'Boot'

        Additionally, we are using the generic interface StdControl
        this interface is provided by various components - this app
        uses the interface provided by 'Atm128Uart0C', so we can
        enable / disable TX for UART0

        see file MyPowersavingAppC.nc for how the wiring works
*/

module MyPowersavingC @safe()
{
        uses interface Boot;
        uses interface StdControl;
}
implementation
{
        /*
        when system has 'bootet' up, we will receive this event and
        can take over from here...
        */
        event void Boot.booted()
        {
                call StdControl.start();
        }
}
```

But what to do if - for example - you want a similar functionality for the CC2420 radio chip, turning on the chip when traffic has to be sent, and turning it off afterwards to save power? Clearly, the way to do this is same as with the UART-subsystem. The only difference is to wire the application to another component - the lower level implementation of the CC2420 - chip will handle the rest.

3.3. Split-Phase Interfaces

As stated before, it should be possible in tinyOS to build a certain functionality either in software *or* in hardware. Because hardware is almost always non-blocking, the software must be non-blocking as well to guarantee this

exchangeability. A traditional approach for this problem is to use multiple threads: one thread requests an operation and is then put on a waiting queue, and another thread continues with some other work. The waiting thread is woken up as soon as the operation has finished and interrupts the other running thread.

A problem with threads is that they need RAM - a resource that is very precious on embedded systems. Every thread has it's own private stack that has to be untouched as long as the thread is suspended, so this part of memory is wasted storage.

So, instead of threads, tinyOS uses the split-phase - model: a program that wants to execute some functionality(sampling a sensor; sending a data packet) requests the operation, returning immediatly afterwards. When the operation has finished, this new state is *signaled* to the requester by an *event*. This is an important aspect of split-phase interfaces: they are birectional. There is always a *downcall*, generally a *command*, that starts an operation, and the corresponding *upcall* or *event* to signal the completion.

The following code snippet gives an example for such an interface:

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

A component that *uses* such a split-phase interface **MUST** implement all of the possible *events*. So, a higher-level component that wants to make use of the *Send* - interface by calling the *send()* - function must implement an *event* called *sendDone*, that will be *signalled* by the send-function.

When using split-phase interfaces, remembering the first programming hint is important: signaling events from commands is dangerous, because this can cause long call loops or corrupt memory caused by exploding stack size. So instead of using commands, the correct way is to use *tasks*.

3.4. Tasks

In tinyOS, a module can *post* a task to the scheduler, which will get executed at some point later. Because a task is a deferred procedure call, there is no return value. Additionally, there are no parameters for tasks because a task is always executed in the naming scope of a component. A declaration of a task has this form:

```
task void checkInterruptflag();
```

Adding this task to the scheduler is done with the *post* keyword:

```
post checkInterruptflag();
```

Tasks are non-preemptive, no task will ever get interrupted by the tinyOS scheduler. The advantage is that the programmer never has to worry about tasks corrupting each other's data. On the other hand, tasks never should be too long because otherwise other tasks maybe could get delayed. So, a task that has to do long computations should be broken into multiple tasks.

tinyOS needs about 80 microcontroller clock cycles to post and execute a task, so there's a tradeoff between lots of short tasks and fewer, but longer running tasks - no hard rule can be given here.

3.5. Async Functions

As stated before, a task will always run until it has finished. This is not true under all circumstances, because a task can be disrupted by an interrupt. Such functions that run preemptively, triggered by an interrupt and from outside the task context, are labeled with the *async* keyword. Consequently, all commands that are called and all events that get signalled by such functions are async as well and must be marked async too.

To get an async function into a syncronous context, the only way possible is by using tasks. For example, receiving UART bytes will normally be done interrupt driven, so the corresponding interrupt vector will be called asyncronously. If the received data are handled by a higherlevel component in a syncronous function, the interrupthandler can post a task to get from the asyncronous interrupt context to a syncronous task context.

3.6. Wiring

As said before, in nesC a component can only call functions and access variables whithin its local namespace. To call functions outside it's scope, a set of names in one component, generally an interface, must be mapped to a set of names in another component. This operation is called wiring.

3.6.1. Modules vs. Configurations

Components can either be *Modules* or *Configurations* - both use and provide interfaces. This set of *used* and *provided* interfaces defines the *signature* of the component.

The difference between *Modules* and *Configurations* lies in their implementation: while configurations are implemented by other components, which they wire, modules are executeable code. Modules are written for the most part in standard - C.

3.6.2. Operators

There are only 3 operators for wiring components together:

- <-
- ->
- =

The first two operators are interchangeable, for example this two lines are identical:

```
MyComponent.UsedInterface -> OtherComponent.ProvidedInterface;
OtherComponent.ProvidedInterface <- MyComponent.UsedInterface;
```

In short, the arrow always points from the *user* of an interface to the *provider* of that interface:

```
User -> Provider;
```

As soon as MyComponent is connected this way to OtherComponent, MyComponent can use all of the functions provided by ProvidedInterface. Additionally, UsedInterface **MUST** implement all the handler OtherComponent offers events for.

If MyComponent wants to call a function provided by OtherComponent, the call would look like this:

```
call UsedInterface.FunctionName(<argumentlist>);
```

To implement the handler for the events that can be signaled by ProvidedInterface, a construct of this form in UsedInterface is neccessary:

```
event <returntype> UsedInterface.EventName(<argumentlist>)
{
        /*
        code to handle the new situation, as signalled by this event
        */
}
```

The third operator, '=', *exports* an interface. It defines how the configuration of an interface is implemented, by delegating it to another component. In contrast, the 'arrow' - operator combines components that already exist.

Listing 3.3: GLCDC.nc

```
/*
        Harald Glanzer, 0727156 TU Wien

        configuration for component GLCD
        needs components TouchScreenC and LCD128x64C for its work
*/

configuration GLCDC
{
        provides interface GLCD;
}

implementation
{
        components GLCDP, TouchScreenC, LCD128x64C;
        GLCD = GLCDP;

        GLCDP.LCD128x64 -> LCD128x64C;
        GLCDP.TouchScreen -> TouchScreenC;
}
```

The difference is explained on the basis of codelisting 3.3 - the top-level configuration for a component that provides functionality for a graphical display. This component provides the interface GLCD which can be used by a high-level programmer to easily use such a display. *GLCDC* consists of the private component *GLCDP* and 2 more components, *TouchScreenC* and *LCD128x64C*. *GLCD* is an interface of the configuration itself and is mapped to the component *GLCDP*. So, the interface *GLCD* gets *exported* by the configuration so that other, higher-level components can use this interface by specifying that component *GLCDC* with interface *GLCD* is used.

Opposite to that, the other 2 components *TouchScreenC* and *LCD128x64C* are used *inside* the component to get the work done - they don't export any functionality to a level higher than GLCDC. Therefore, the arrow-operator is used here.

3.6.3. Nameing Conventions

All components in tinyOS are ending in *C* or *P*: the first ending is used for *C*omponents, which means that they represent usable abstractions. *P*rivate components are ending in P. That means that generally, the programmer should not wire to it, because it's functionality is encapsulated by a *C*omponent.

4. bigAVR6

4.1. The Hardware

This developmentboard, as shown in figure 4.1, is produced by mikroElektronika and supports 64- and 100-pin AVR TQFP packages.

4.1.1. Onboard - Features

- 2 x UART
- highspeed CAN transceiver MCP2551
- USB 2.0
- MMC/SD card slot
- digital thermometer DS1820
- onboard USB 2.0 programmer
- real-time clock DS1307
- 86 LEDs to indicate logic state of all microcontroller-pins
- 86 push buttons to control microcontroller digital inputs
- IDC10 connectors for all microcontroller pins
- DIP - switches to separate port pins from pull-up/down resistors
- potentiometer for testing the ADC - channels
- 1KBit serial EEPROM 24AA01

4.1.2. Extensions

- 2x16 LCD, directly connectable via on-boad connectors
- 64x128 GLCD, directly connectable via on-boad connectors
- IEEE802.3 extensionboard 10MBit
- SmartMP3 decoder board

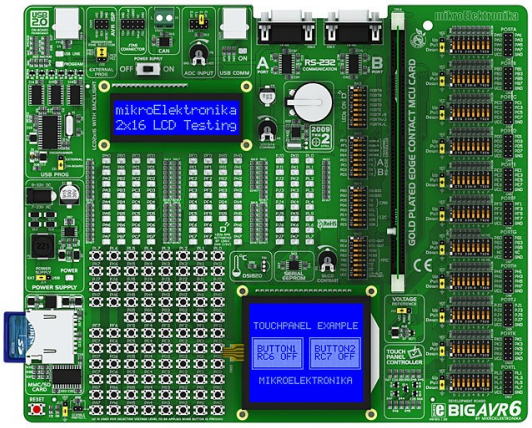

Figure 4.1.: bigAVR6 platform

4.2. Supported Modules

4.2.1. GLCD

The GLCD - extension consists of a KS0108B GLCD dot matrix liquid cristal graphic display system with a resolution is 128 x 64 pixel. As the schematics shows in figure 4.2, SW15/8 must be turned on to activate the GLCD-backlight, the contrast can be changed with potentiometer P3.

The GLCD is combined with a touchscreen foil, sticked onto the GLCD-surface. Activate SW13/1,2,3,4 to use it. To read the x and y coordinates, the component uses ADC channels 0 and 1, respectivly.

The logic level of the following pins **MUST NOT** be touched by the programmer when using the GLCD and the touchscreen:

- PORT A: all ports reserved for GLCD

- PORT E: PE2, PE3, PE4, PE5, PE6, PE7 reserved for GLCD
- PORT F: PF0, PF1 reserved for touchscreen
- PORT G: PG3, PG4 reserved for touchscreen

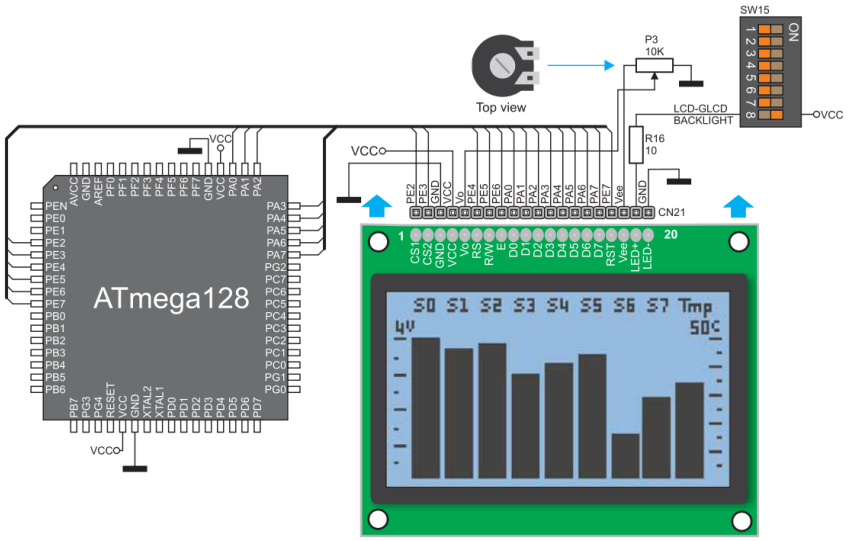

Figure 4.2.: GLCD schematics

To use the GLCD in an application, component *GLCDC* with interface *GLCD* must be used. This interface provides functions for initializing, clearing, writing strings, and drawing graphical objects on the the GLCD. Additionally, it provides functions for calibrating the touchscreen and for getting x/y - coordinates. See listing 4.1 for all the provided functions. Figure 4.3 shows the internal composition of the GLCDC: LCD128x64C is responsible for writing and initializing the GLCD, while TouchScreenC provides functionality related to determining the pressed positions. Therefore, TouchScreenC uses the parameterized interface Read<uint16_t>wired to component AdcReadClientC for getting the needed ADC - values from channels 0 and 1.

Listing 4.1: GLCD.nc

```
interface GLCD
{
	/*
	////////////////////////////////////////////////////////////////////
	                    TouchScreen - interfaces
	////////////////////////////////////////////////////////////////////
	*/
	command error_t calibrateTouchScreen();

	command void getCalibration( uint16_t *x_cmin, uint16_t *x_cmax, uint16_t *y_cmin, uint16

	command void getXY();
```

```
command void isPressed(bool on);

/*
        TODO: liefert die skalierten x/y - werte
        ADC liefert zwar prinzipiell 10 bit - werte, skalierung auf
        8 bit sollte jedoch reichen
*/
event void xyReady(uint16_t x, uint16_t y);
//event void xyReady(uint8_t x, uint8_t y);

event void calibrated();

event void tsPressed();


/*
#########################################################################
                        LCD128x64 - interfaces
#########################################################################
*/
command void copyByte(uint8_t x, uint8_t y);

command void initLCD(uint8_t pattern);


/*
        write byte to adress x(0...127) / y(0...8 = **PAGE**)
*/
command void writeByte(uint8_t x, uint8_t y, uint8_t data);

/*
        no posted task!
*/
command void writePixel(uint8_t x, uint8_t y, uint8_t on);

/*
        no posted task!
*/
command void setPixel(uint8_t x, uint8_t y);

/*
        if possible: write bar at 8bit-boundaries(in y-direction: 0, 8, 16, ...)
        and use width < 8 or width = multiple of 8 --> FASTER
*/
command error_t startWriteBar(uint8_t x, uint8_t y, uint8_t length, uint8_t width);

command error_t startWriteRectangle(uint8_t x, uint8_t y, uint8_t a, uint8_t b);

command error_t startWriteCircle(uint8_t xcenter, uint8_t ycenter, uint8_t radius);

/*
        print string pointed to by *data to coordinates x/y
        must be '\0' - terminated!

        x: range 0-127(pixel)
        y: range 0-7(lines)
*/
command error_t startWriteString(char *data, uint8_t x, uint8_t y);

command error_t startWriteLine(uint8_t x, uint8_t y, uint8_t xEnd, uint8_t yEnd);

command error_t startClearScreen(uint8_t pattern);
```

```
        event void initDone(void);
        event void circleWritten(void);
        event void stringWritten(void);
        event void rectangleWritten(void);
        event void lineWritten(void);
        event void barWritten(void);
        event void screenCleared(void);
}
```

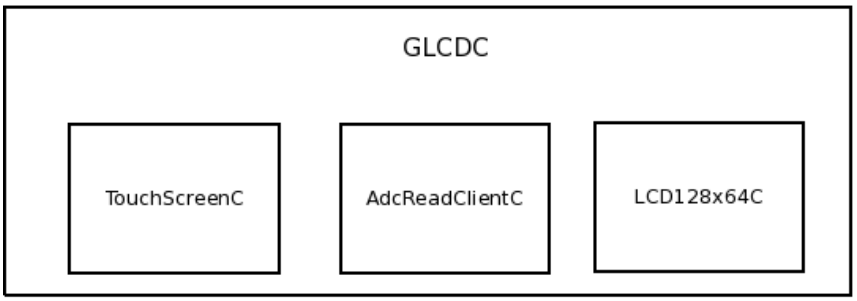

Figure 4.3.: GLCDC components

4.2.2. MMC

A MMC/SD connector is available onboard so that memory cards can be interfaced to the microcontroller via SPI. As shown in figure 4.4, activate SW15/3/4/5/6/7 to use it. Because the memory card is powered with 3.3V, whereas the microcontroller needs 5V supply, a 74LVCC3245 bus transceiver is used to obtain the needed voltage levels.

The logic level of the following pins **MUST NOT** be touched by the programmer when using the MMC:

- PORT B: PB1, PB2, PB3 reserved for MMC: SCK / MISO / MOSI
- PORT G: PG1, PG2 reserved for MMC: CS / CD

Component *MMCC* provides interface *MMC* to enable an application programmer to easily read **RAW** - data from a MMC, no write functionality is supported yet. The available functions are shown in listing 4.2.

To read data, *init()* must be called first to initialize the memorycard. If no MMC/SD is present, a corresponding errormessage is signaled to the application. Otherwise, completion is signaled by the event *initDone()*. Afterwards, data can be read from the memorycard by calling

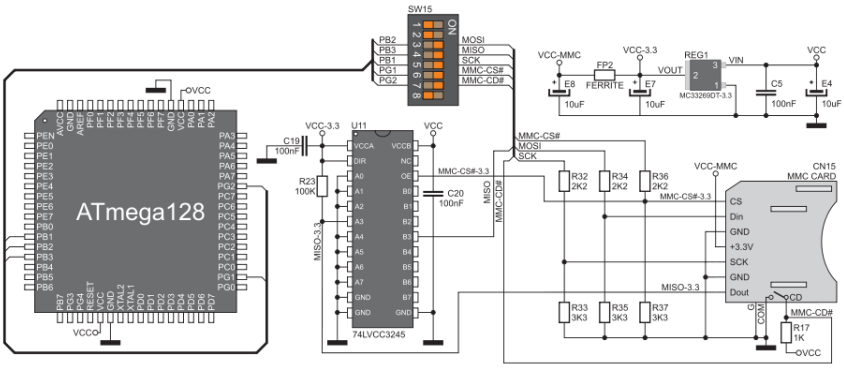

Figure 4.4.: MMC/SD Schematics

```
call command uint8_t readBlock(uint32_t addr);
```

After completion, event *blockReady* is signaled to the application and data is available by a pointer, in units of 32 byte blocks.

Listing 4.2: MMC.nc

```
interface MMC
{
        command uint8_t init(void);

        command uint8_t readBlock(uint32_t addr);

        event void initDone();

        event void blockReady(uint8_t *data);

        event void error(uint8_t *errStr);
}
```

The following limitations are to be noticed:

- Read - Only
- Raw data, no filesystem supported
- Data is always read in 32 byte blocks

4.2.3. Ethernetboard

This extension board, as shown in figure 4.5, can be used for sending and receiving IEEE802.3 frames, and therefore embedding the developmentboard into a network. The board features an ethernet controller ENC28J60 that exchanges data with a microcontroller via standard SPI. The board can be connected to dsPIC, AVR-8051 and PIC by three 10-pin IDC connectors. The

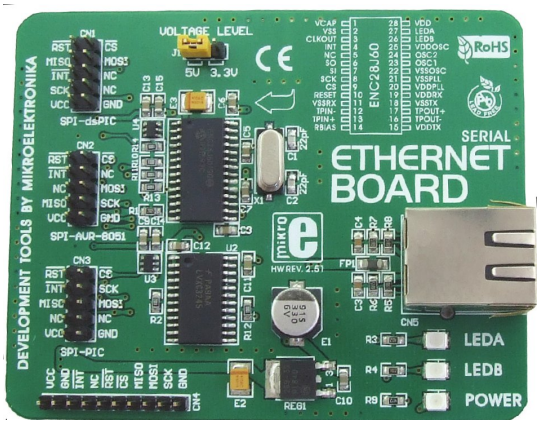

Figure 4.5.: Ethernet Board

board needs 3.3V voltage supply, but has a 74LVXc3245 bus transceiver chip for running the board with 5V logic levels. Therefore, jumper J1 has to be set correctly.

A basic udp/ip/arp - stack has been implemented, with the following limitations:

- No UDP - level checksum
- No IP - level options
- No IP - level fragmentation
- ARP request/reply only
- No ICMP

For using the extensionboard with the bigAVR6, the following requirements have to be provided:

- Jumper J1 on the Ethernetboard must be set to 5V
- SPI-AVR-8051 VCC must be connected to 5V
- SPI-AVR-8051 GND must be connected to ground
- SPI-AVR-8051 CS must be connected to PORT B0
- SPI-AVR-8051 SCK must be connected to PORT B1
- SPI-AVR-8051 MOSI must be connected to PORT B2

- SPI-AVR-8051 MISO must be connected to PORT B3
- SPI-AVR-8051 RST must be connected to PORT B4
- SPI-AVR-8051 INT must be connected to PORT D2

Obviously, the state of this connected pins **MUST NOT** be touched by the programmer.

IMPORTANT NOTE

RS-232 Port B CAN NOT BE USED together with this component because Port D, which is used as the external interrupt source from the ethernetboard, is also used as UART1(=RS-232 Port B) RX pin. Instead, RS-232 Port A **MUST** be used if the UDP-services are also needed.

Also, jumper J21 **MUST BE DEACTIVATED** for IEEE8023C to function!

To use the UDP - stack in an application, component UDPC with interface UDP has to be used. Interface UDP provides functions for initializing the stack and sending and receiving frames, as shown in listing 4.3.

Listing 4.3: UDP.nc

```
/*
        Harald Glanzer, 0727156 TU Wien

        interfaces for component UDPC
*/
interface UDP
{
        /*
                to receive packets, a listeningport must set first
        */
        command uint8_t createSocket(uint16_t socket);

        command uint8_t sendData(uint16_t *dataPtr, uint8_t *destPtr, uint16_t srcPort, uint16_t

        command uint8_t initStack();

        event void initDone();

        event void sendDone();

        event void hwInterrupt(uint16_t *info);

        event void gotDatagram(uint16_t len, uint8_t *dataPtr);
}
```

When the application wants to send data to or receive from a remote host, it initializes the stack by calling

```
command uint8_t initStack();
```

To send data, the corresponding function must be called:

```
command uint8_t sendData(uint16_t *dataPtr, uint8_t *destPtr, uint16_t srcPort,
                         uint16_t destPort, uint16_t len);
```

Afterwards, the data travels down the stack, as shown in figure 4.6. At first, an udp header is generated, and the payload, as assigned by the application, is attached to it. As mentioned above, **NO** checksum is generated at this level - if needed, a checksum must be calculated at application level. This udp packet(payload + udp-header) is then assigned to the next level, the ip level.

At this level, a checksum is generated, which includes the IP-header only. In the next step, a routing-decision has to be made: either the packet is addressed to a host *inside* the lan(the remote host has the same network-adress than the ip that has been assigned to the bigAVR6 board), or the packet must *hop* outside the lan. In the first case, the packet can be sent to the host directly, otherwise, it must be sent to the default gateway, as defined in the headerfile tinyos/tos/platforms/bigAVR6/ip/IP.h. When the proper destination ip has been found, an ARP - lookup is done. A local arp-cache is implemented as a ringbuffer, which saves ip/mac - address couples in a two-dimensional array. If no ARP - entry is found for the destionation ip, a lookup is sent automatically. Be aware that, if no MAC is found, the ARP-request - triggering packet is lost and must be resent by the application. If the MAC is known, it is extracted from the MAC cache, and component IEEE802.3 takes over control, where the frameheader and the payload(data + udp-header + ip-header) is copied into the transmit memory of the ENC28J60 by SPI. Afterwards, the communication is started, and the completion is signaled to the application.

To receive a packet from remote hosts, the packet must either has the proper destination MAC set, as defined in tinyos/tos/platforms/bigAVR6/eth/IEEE8023.h, or must be be a multicast frame(destination 0xFFFFFFFFFFFF), otherwise the ethernetcontroller chip ENC28J60 will not accept it. If a valid MAC is present, the ENC28J60 will autonomously copy the packet into the receive buffer and signal completion by an interrupt. The packet will be read out by component IEEE8023C and handed over to the IP-level component IPC. If the packet is a ARP - request, a corresponding reply will be sent, otherwise the payload(data + udp-header) will be handed over one level higher, which is the UDP - level. Note that the ip-checksum will **NOT** be checked. At UPD level, the actual payload will get extracted and is handed over to the application.

For testing purposes, the commandlinetool *netcat* can be used. As example, a constant string should be sent to the ip 192.168.1.100, port 4443, as shown in the following listing.

```
...
dstIP[0] = 192;
dstIP[1] = 168;
dstIP[2] = 1;
```

```
dstIP[3] = 100;
call UDP.sendData((uint16_t *)"hello world", (uint8_t *)dstIP, 80, 4443, sizeof("hello wo
...
```

To receive the string on the corresponding host, this commandline can be used:

```
nc -u -l 4443
```

This starts netcat in listening mode, and specifies that UDP instead of TCP should be used, and that it should bind to port 4443.

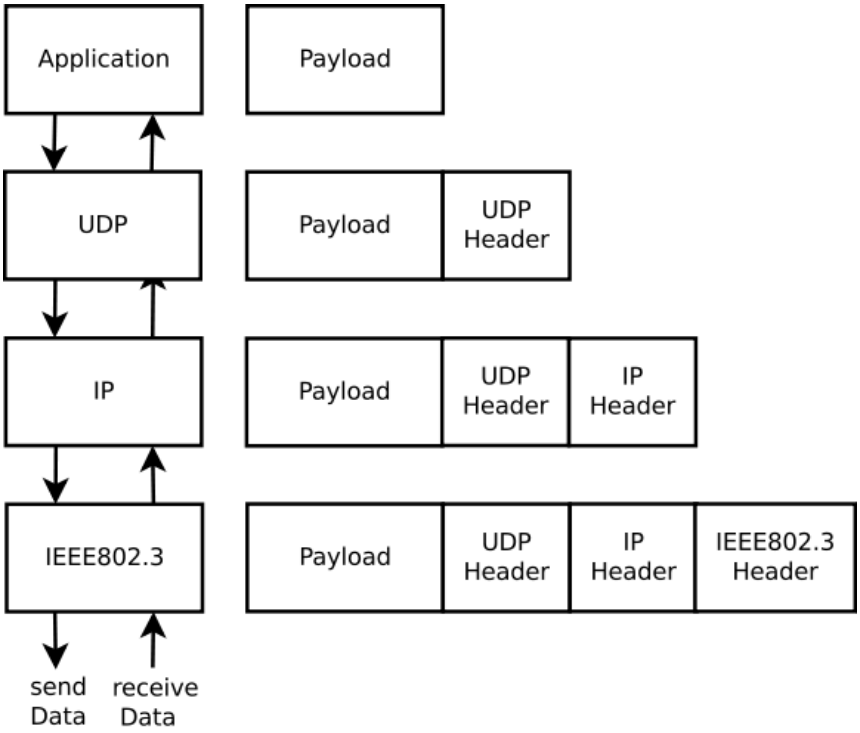

Figure 4.6.: UPD Stack

4.2.4. LCD 2x16

Additionally to the GLCD, a 2x16 alphanumeric LCD module is available for writing ASCII - characters. The display backlight can be turned on with switch SW15/8, as shown in schematic 4.7, the brightness can be adjusted with potentiometer P1

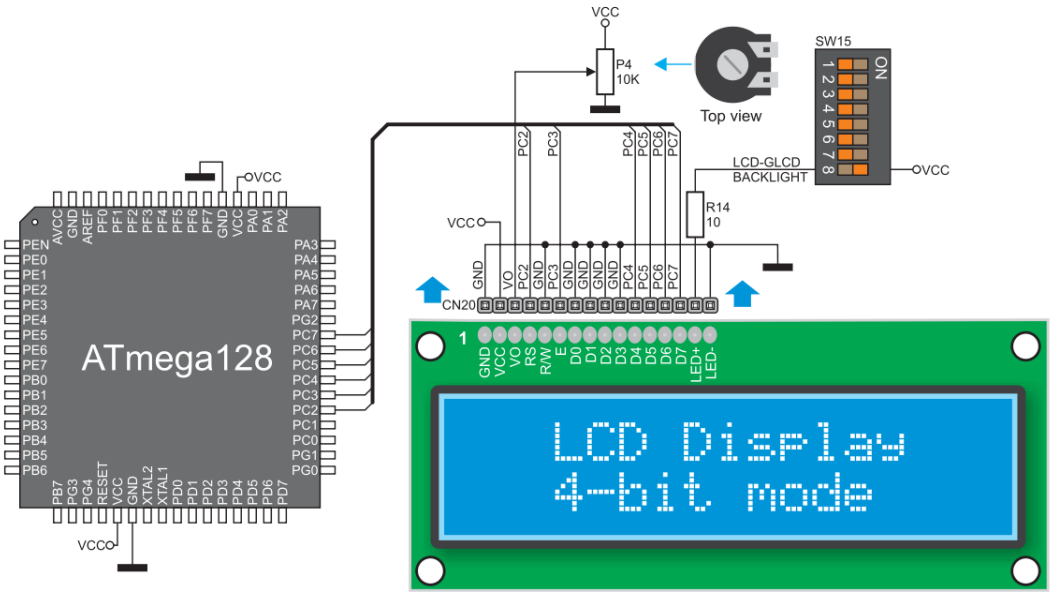

Figure 4.7.: LCD 2x16

The logic level of the following pins **MUST NOT** be touched by the programmer when using the LCD2x16:

- PORT C: PC2, PC3, PC4, PC5, PC6, PC7,

To use the LCD2x16, component *LCD2x16C* with interface *LCD2x16* must be used. Before writing data to the display, it must get initialized first. See listing 4.4 for the available functions. After initializing, data can be written to the LCD - memory, which will then get printed. It should be noticed that once-written data is preserved unless the corresponding position is overwritten by new text or the display is cleared.

When initializing, a parameter *mode* must be set, which determines wheter the cursor should be printed. Use the following constants, as defined in tinyos/-tos/platforms/bigAVR6/lcd2x16/LCD2x16.h:

- CURSOR_ON_BLINK_ON

- CURSOR_ON_BLINK_OFF
- CURSOR_OFF

Listing 4.4: LCD2x16.nc

```
/*
        Harald Glanzer, 0727156 TU Wien

        interfacefunctions for component LCD2x16C
*/
interface LCD2x16
{
        /*
                MUST BE CALLED before lcd can be used.
                returns FAIL if BUSY. can get initialized multiple times
                * uint8_t       mode

                available modes:
                        CURSOR_ON_BLINK_ON
                        CURSOR_ON_BLINK_OFF
                        CURSOR_OFF
        */
        command error_t init(uint8_t mode);

        /*
                splitphase command for writing data to LCD

                parameter:
                * char *        datastring: string to write to display
                * uint8_t       datalength: length of string to write
                * uint8_t       line: 'x' start posititon, valid values 0...15
                * uint8_t       column: 'y' position, valid values 0...1
        */
        command error_t sendString(char *str, uint8_t len, uint8_t line, uint8_t column);

        /*
                clear display by sending corresponding command
                returns FAIL if already busy, SUCCESS otherwise
        */
        command error_t clearDisplay(void);

        /*
                event for splitphase command sendString()
        */
        event void stringWritten(void);

        event void displayCleared(void);
}
```

When examing schematic 4.7, one can see that there is a design flaw existing: pin RW is bound to GND. Because this pin is low-active(low=write operation), **no read operation** is possible. Accordingly, it is not possible to read the corresponding register to determine whether a write-operation has already finished. To handle this problem and give the LCD controller enough time to process a write-operation(needed for initializing, clearing the display and writing characters to it), there are two options:

- use busywaiting - blocking
- use a timer - non-blocking

With the first solution - busy waiting - the time to wait can be minimized to the absolutely necessary time, but obviously the time spent in the spinloop is lost cpu time, which is bad.

The second solution - using the generic timer interface TimerMilliC - doesn't waste cpu time(apart from the extratime spent in the timer's interrupt vector), but problematic about this solution is that the existing timer-framework doesn't provide good granularity. The minimal periodic time that can be achieved is about 20msec, which is hereby the waiting time for every character when sending a string to the display. While this is not problematic when initializing or clearing the display, a considerable delay is introduced when text is written to the display.

By default, the blocking solution is used. This is achieved by setting a soft-link name **lcd2x16** in the directory tinyos/tos/platforms/bigAVR6. FIXME - ziemlich unsauber. funkt leider nicht mittels file tinyos/tos/platforms/bigAVR6/.platform.h - mÃ¶glicherweise mittels precompiler - direktive??

Because booth solutions are not optimal, it is recommended to **NOT** use the LCD2x16 - component to write long, fast-changing strings to the LCD. Because once written data are preservered unless the display is cleared, it is possible to update individual positions of the lcd by sending substrings, thus minimizing this problem.

4.2.5. UART

There are 2 UART ports available onboard, RS-232 port A and port B. To use them, component *bigAVR6UARTAC* with interface *bigAVR6UARTA* or component *bigAVR6UARTBC* with interface *bigAVR6UARTB* must be used. Switch SW15/5/6 respective SW15/7/8 must be used to active the ports, as shown in figure 4.8. Functions for sending bytes or streams and receiving a given number of stringcharacters are implemented, as defined in listing 4.5.

Listing 4.5: bigAVR6UARTA.nc

```
/*
        Harald Glanzer, 0727156 TU Wien

        interfacedefinitions for component bigAVR6UARTAC
        this is uart port A on the bigAVR6 board
*/
interface bigAVR6UARTA
{
        command void sendByte(uint8_t byte);
```

```
        command error_t send(uint8_t* str, uint16_t len);

        /*
                split-phase command for receiving UART bytes
                activate UART to receive len bytes
                data will get saved in memory *buf is pointing to
        */
        command error_t receive(uint8_t* buf, uint16_t len );

        //      EVENTS
        /*
                event for split-phase command send()
                will get signaled after len bytes have been received or if error occured
        */
        async event void receiveDone(uint8_t* buf, uint16_t len, error_t error );

        async event void sendDone(uint8_t* buf, uint16_t len, error_t error );

}
```

The logic level of the following pins **MUST NOT** be touched by the programmer when using the UART components:

- RS-232 **A**: Port E PE0, PE1
- RS-232 **B**: Port E PD2, PD3

Additionally, the following restrictions have to be noticed:

- RS-232 Port A: disable Jumper J21(external programmer)
- RS-232 Port B: Port PD3 servers as TX, but also connected to RTC. Disable with switch SW15/1
- BAUDRATE: use constant 115200/8/N/1

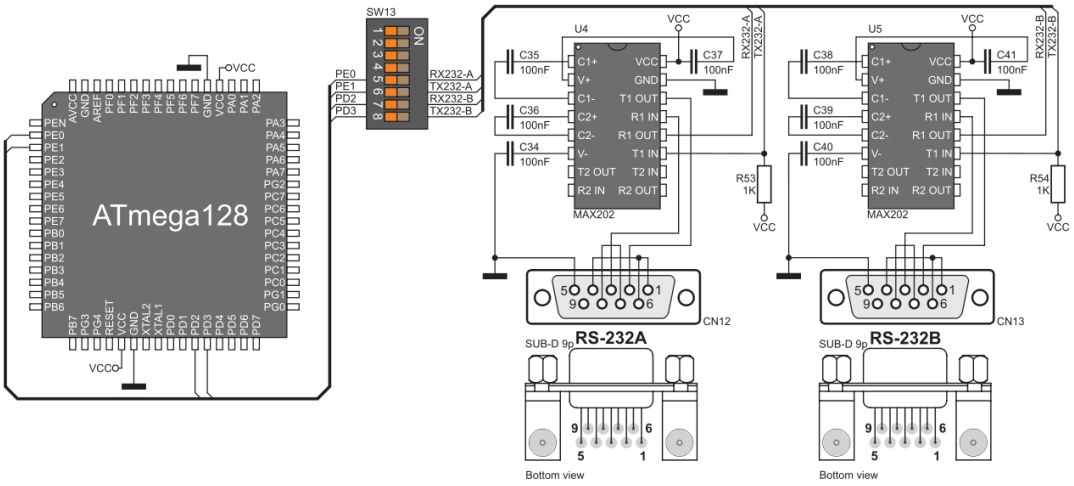

Figure 4.8.: RS-232 Ports

5. MicroC/OS-II

To emphasize the characteristics of tinyOS, another OS, called OS-II, is presented. The MicroC kernel was developed by Jean J. Labrosse.

5.1. Basics

uC/OS-II is a portable, ROMable, scalable preemptive real-time multitasking kernel, written in ANSI-C. Small portions of the sourcecode are written in assembler to adapt to different processor architectures. uC/OS-II is based on uC/OS, also called *The Real Time Kernel*, which was published in 1992. uC/OS v1.11 is compatible to its successor.

uC/OS-II is used in all kinds of embedded systems, ranging from engine controls to audio equiment.

5.1.1. Main Features

Important characteristics of this real-time multitasking OS are:

- Scaleable, with a minimum RAM footprint of 6KByte
- Maximum RAM footprint 24KByte
- Preemptive Scheduler
- 10 Kernel Services
- 80 System Calls
- Support for semaphores
- Support for event flags
- Support for Message Queues
- Support for Message Mailboxes

- Up to 250 Application Tasks
- Constant / Deterministic execution time for most services provided
- Support for large number of processor architectures
- Supported processors range from 8bit to 64 bit architectures
- Supports Multi-Threaded Applications

5.2. Differences tinyOS - MicroC/OS-II

5.2.1. Real-Time

A real-time system is a system where not only the correctness of a calculation is important, but also the instant **when** this calculation was finished. The most import features of a real-time kernel are minimal and predictable interrupt latency and task switching latency.

While uC/OS-II supports a timely determinstic behaviour, no such assertion can be given for tinyOS.

5.2.2. Scheduler

MicroC/OS-II has a *preemptive* scheduler - that means that every task gets only a limited time for using the CPU. The time a task gets for execution also depends on the task's *priority* - all tasks **MUST** have different priorities, because otherwise additional round-robing scheduling or similar would be necessary to handle tasks with equal priority. More important tasks get higher priority.

A task can be in 5 different states, as shown in figure 5.1:

- Dormant: task is loaded in memory but not ready to run yet
- Read: task is ready to run, but higher-priority task is running at the moment
- Running: task is in control of the cpu
- ISR: interrupt has occured, cpu is executing ISR instead of task
- Waiting: task waits for an event(completion of IO operation, ...)

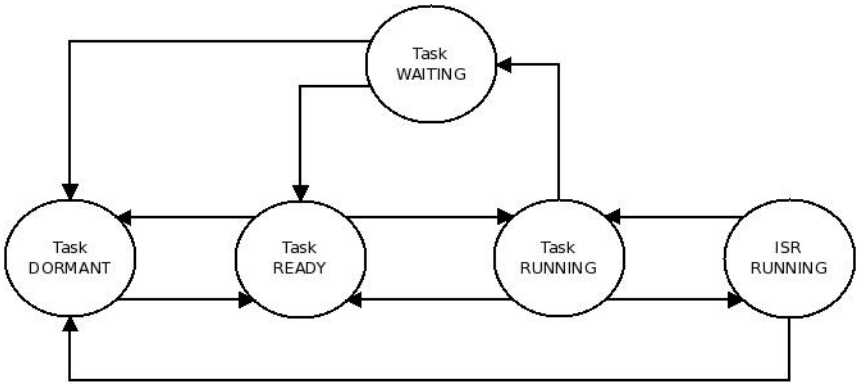

Figure 5.1.: Statemachine for Tasks

In contrast, tinyOS knows only 3 states:

- Queued: task is queued, but sooner queued task is running at the moment
- Running: task is in control of the cpu
- ISR: interrupt has occured, cpu is executing ISR instead of task

Also, as already said, tinyOS uses a non-preemptive scheduler, see chapter 2 for details.

5.2.3. Mutual Exclusion

Whenever different tasks need to share data with each other, some points must be ensured to avoid data corruption and resource contention. Problems arise, for example, if task A gets scheduled and accesses a dataelement that is also needed by task B. If task A gets preemted by task B before task's A calculations are finished, task B works on incomplete data, possibly invalidating it's own calculations. To prevent such situations, MicroC/OS-II provides the following mechanisms:

- Disabling/Enabling of interrupts with macros
- Semaphores

By *disabling interrupts*, a task or ISR can assure that it won't get preempted by an ISR or by a task, so this is an easy way to grant mutual exclusion. Drawbacks of this solution are that the system obviously can't react to other - possibly important - interrupts. Additionally, because the Micro/OS-II kernel

uses timeslicing, the task scheduler is deactivated because no timer interrupts are handled. So, a badly designed task can halt the whole system if it enters a deadlock or some kind of endless loop.

Semaphores, developed by Edsger W. Dijkstra, are softwareconstructs that can be used for the same goal. Additionally, they can be used for syncronizing tasks, as described in the following section.

Because tinyOS uses a non-preemptive scheduler, no such constructs are neccessary. Because a task cannot get suspended, apart from an interrupt, by another task, every task is responsible to keep it's data in a consistent state.

5.2.4. Intertask Communication

The process of sharing data between different tasks or ISRs is called *inter-task communication*. Therefore, MicroC/OS-II supports the following ways of sharing data between tasks

- Message Mailboxes
- Message Queues

MircoC/OS-II provides kernel services to send messages by depositing a pointervariable into a task's mailbox. A task that expects such a message is supended and put onto a waitinglist. If the message arrives within a timeout, the highest-priority task that is waiting for the message is woken up.

A message queue is basically an array of such message mailboxes.

In tinyOS, a task can share data by passing a pointer to the datastructure used, or by using global variables.

5.2.5. Task Syncronisation

- Semaphores
- Event Flags

Aside from using semaphores for granting mutual exclusion, this software constructs can also be used to syncronize a task with an ISR or other tasks. Additionally, MicroC/OS-II offers kernel services to *set*, *clear* and *wait* for event flags.

Syncronisation in tinyOS is implemented by *events*.

5.2.6. Memory Footprint

When using a kernel to manage tasks, the total memory that is needed consists of the application code size + the kernel code size. A minimum ram footprint of 6kByte is given by the developer of MicroC/OS-II.

In tinyOS, the ram needed for the kernel FIXME

5.2.7. Supported Platforms

6. buildenv

6.1. Prerequisites

7. resourcesused

- http://www.youtube.com/watch?v=j6hRsue5b30 / lecture about tinyos
- http://www.tinyos.net/tinyos-2.x/doc/html/tep2.html
- http://www.tinyos.net/tinyos-2.x/doc/html/tep106.html
- http://www.tinyos.net/tinyos-2.x/doc/html/tep108.html
- http://www.tinyos.net/tinyos-2.x/doc/html/tep112.html
- http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf

8. todos

- POST tasks after startWrite* @ glcd, NO COMMANDS!
- split-phase for UDP-stack
- arp-request: first (triggering) datapacket is lost! returncode!!
- IEEE8023 / MMC: booth via SPI / arbiter. CHECK

9. Related Work

This chapter should give an overview over existing work that is related to your work. Instead of "Related Work", this chapter can also be named specifically to the topic of the thesis.

.

Each related approach should be described by a section of about 100-500 words.

9.1. Types of Bachelor's Theses

If you write a plain report on some implementation, you might have no chapter on related works.

10. Implementation

Call this “Implementation” or “Case Study”, here you will describe you actual hands-on part of your work.

10.1. Types of Bachelor’s Theses

If you write a Bachelor’s thesis in form of a survey, you might have several chapters on existing work in the literature, but no chapter as described here.

11. Results and Discussion

Name this chapter "Results and Discussion", "Experimental Results", "Evaluation" or "Experiments and Evaluation". First present your measurements here, usually in form of graphs and tables. Second, discuss it. Explain for example missing or misplaced data points[1].

If this chapter grows too large, you might split it into two separate chapters, for example "Results" and "Discussion".

11.1. Tables

Sensor	Mean squared error (cm^2)	Mean absolute error (cm)	Estimated variance (cm^2)	Respective confidence
IR 1 ($d \leq 80$ cm)	228.38	5.97	212.98	4
IR 1 (hybrid)	860.87	14.35	686.08	2
IR 1 ($d > 110$ cm)	1880.50	28.27	1078.30	2
IR 2 ($d \leq 80$ cm)	242.04	7.25	233.15	4
IR 2 (hybrid)	226.04	7.15	206.56	4
IR 2 ($d > 110$ cm)	162.13	6.24	127.92	5
IR 3 ($d \leq 80$ cm)	108.45	7.35	104.82	5
IR 3 (hybrid)	795.57	18.59	538.91	3
IR 3 ($d > 110$ cm)	1945.08	37.65	505.68	3

Table 11.1.: Quality of calibrated infrared sensor data (from [Elm02])

11.2. Types of Bachelor's Theses

If you write a Bachelor's thesis in form of a survey, you might have several chapters on existing work in the literature, but no chapter as described here.

[1] By the way, do not refer to colored elements in a figure or graph, assume the user prints out your thesis in black and white

12. Conclusion

The chapter "Conclusion", sometimes also named "Summary" should contain two things:

Main contribution(s) of this work – Why is the world a better one now ;-)

When you write the conclusion, assume that some quick readers might not have gone through the whole thesis but are merely peeking into the conclusion, therefore avoid complicated nomenclature here.

Outlook – what could be the next steps or possible extensions for this research?

Bibliography

[Elm02] W. Elmenreich. *Sensor Fusion in Time-Triggered Systems*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

[FIX] Philip Levis / Cory Sharp FIXME. *Scheduler and Tasks*. Treitlstr. 3/3/182-1, 1040 Vienna, Austria. Available at `http://www.tinyos.net/tinyos-2.x/doc/html/tep106.html`.

[Lev] Philip Levis. *TinyOS Programming*. Treitlstr. 3/3/182-1, 1040 Vienna, Austria. Available at `http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf`.

A. Setup Guide

You have implemented some system others will use? They will surely acknowledge a short setup guide here. Describe the system requirements and setup steps in a precise way here, like for example:

A.1. System Requirements

In order to use the *thesis template*, you need to install LaTeX, furthermore an editor supporting LaTeX is recommended.

A.1.1. Required Software for Windows

For compiling this template, we used the actual version of MikTeX, which is a an up-to-date implementation of TeX and LaTeX for all current variants of Windows on x86 systems. MikTeX is freely available at `http://www.miktex.org`.

As an editor, we recommend the free *TeXnicCenter* (available at `http://www.toolscenter.org`). Both, MikTeX and TeXnicCenter are published under the Gnu Public License (GPL).

TeXnicCenter comes with an integrated spell checker, otherwise you are recommended to install the Windows version of *aspell*, an open source spell checker under the GPL, which is available at `http://aspell.net/win32/`.

If also want to do grammar checking, try Queequeg (`http://queequeg.sourceforge.net/index-e.html`) or see guides like the one at `http://www.physics.usyd.edu.au/guides/spell-grammer-latex.html`.

A.1.2. Required Software for Linux and BSDs

The standard distributions for Linux already come with a LaTeX system (typically `tetex`).

As an editor, we recommend the Kile editor (available at `http://kile.sourceforge.net/` under GPL).

As spell checker we recommend *aspell*, an open source spell checker that replaces the older *ispell* checker. *aspell* is included in most distributions, otherwise it can be downloaded from `http://www.gnu.org/software/aspell/`. If also want to do grammar checking, try Queequeg (`http://queequeg.sourceforge.net/index-e.html`) or see guides like the one at `http://www.physics.usyd.edu.au/guides/spell-grammer-latex.html`.

A.1.3. Required Software for Apple Mac OS X

The *darwin ports* (`http://darwinports.opendarwin.org/`) provide a port of *teTeX* that can be installed under Apple Mac OS X.

As an editor, we recommend TeXShop (available at `http://www.uoregon.edu/~koch/texshop/` under GPL).

As spell and grammar checker we recommend Excalibur (`http://www.eg.bucknell.edu/~excalibr/`).

A.2. Installing the Thesis Template

The thesis template comes in a zip-archive. Simply extract the archive into a directory of your choice and start working.

A.3. Types of Bachelor's Theses

Of course, not all Bachelor's theses require a setup guide.

B. User Guide

You have implemented some system others will use? If you were in their place what kind of documentation would you like to have in order to start working?

For complex programs, a tutorial that guides the user through a typical task showing screenshots of the intermediate states is desirable.

B.1. Types of Bachelor's Theses

Of course, not all Bachelor's theses require a user guide.

B.2. How to use the Thesis Template

B.2.1. Files to Edit

You should edit the following files (unless you remove some of them, see Section B.2.2):

- `title.tex`
- `abstract.tex`
- `acronyms.tex`
- `introduction.tex`
- `concepts.tex`
- `relatedwork.tex`
- `designapproach.tex`
- `implementation.tex`
- `results.tex`

- `conclusion.tex`
- `setupguide.tex`
- `userguide.tex`

B.2.2. Removing Chapters

Open the file `thesis.tex` and remove (or insert a comment) at the lines with the include-commands, for example:

Before	After
`\include{concepts}`	`\include{concepts}`
`\cleardoublepage`	`\cleardoublepage`
`\include{relatedwork}`	`%\include{relatedwork}`
`\cleardoublepage`	`%\cleardoublepage`
`\include{designapproach}`	`\include{designapproach}`
`\cleardoublepage`	`\cleardoublepage`

Table B.1.: Removing the Chapter `Related Work`

B.2.3. Adding Chapters

Open the file `thesis.tex` and add an `\include{`*filename*`}` command (followed by a `\cleardoublepage`) at the respective line. Then create a new file *filename*`.tex` in the same directory and write the chapter's contents into it.

B.2.4. Adding References

Whenever you `\cite` something, there must be a respective entry in any of the included .bib files. To add such an entry, open the respective bibfile (e. g., `bibfile.bib`) with a text editor and add the entry according to the bibfile syntax. The reference will be added after running latex/bibtex/latex on your project.

B.2.5. Removing References

Remove all `\cite` commands to this reference in the text and the citation will not be listed in the bibliography after running latex/bibtex/latex on your project. There is no need to remove the reference from the .bib file.

B.2.6. Changing the .bib Files to be used

In the file `thesis.tex` there is a line with a `\bibliography` command, that lists all used .bib files without file extensions separated by commas.

B.2.7. Troubleshooting

If you make something wrong, you should get an error at compile time, except for citation problems, like:

Adding or removing references does not work: Perhaps there is a spelling error in some .bib file, this causes the program `bibtex` to abort execution leaving the old bibliography unchanged.

Changing a reference does not work: Some systems only perform a new bibtex run if there are missing references, delete the `.aux` file in order to overcome this problem.

References show a question mark: Bibtex could not find a matching entry in the bibfile for this reference, check the label name.