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

Harad 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 implementions 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, on 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 an explains some new written softwaremodules for highlevel-use of this platform's peripherals. Also, it is 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 other embedded OS FIXME 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 **??** gives an introduction into the basic terms and concepts used throughout the work.

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

Chapter 5 is a step-by-step howto for setting up a fresh tinyOS - environment.

Finally, the thesis ends with a conclusion in Chapter 11 summarizing the key results of the presented work and giving an outlook on what can be expected from future research in this area.

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. But because hardware is always non-blocking, the software must be non-blocking as well to guarantee this exchangeability. So, tinyOS uses the split-phase - model: a userprogram that wants to execute some kind of code *posts* a *task*, which is queued by the task scheduler, and gets executed later. The finished execution of the task is afterwards *signaled* to the caller of the task. This behavior can be compared to a hardware interrupt: some kind of operation is requested by a program, so the program initialises the needed datastructures and starts the operation. But instead of waiting until the hardware has finished, the userprogram returns immediatly and continues with other instructions, and gets discontinued by an interrupt as soon as the hardware has finished.

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.

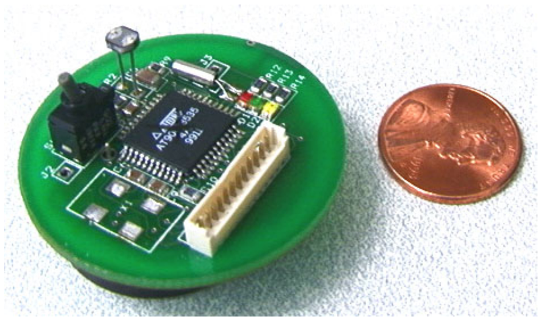


Figure 2.1.: WeC Mote

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.

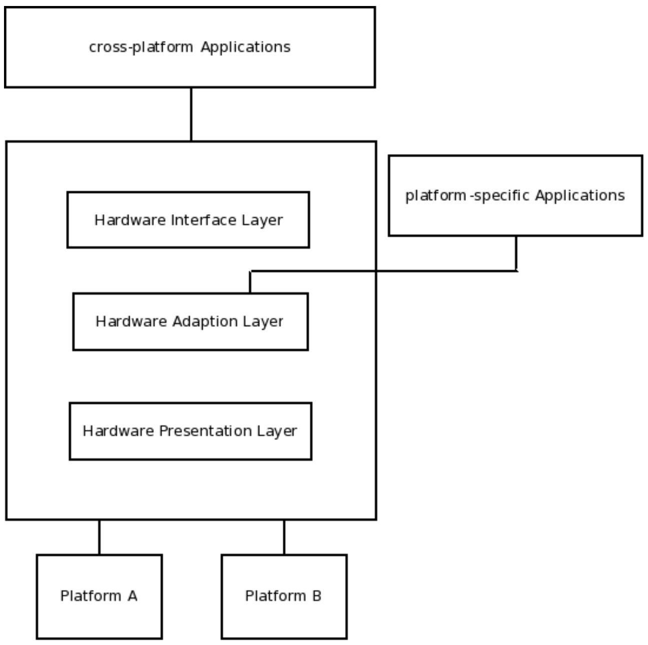
2.3. Supported Platforms

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
- mica: Atmega103 and TR1000 radio chip
- mica2: Atmeage128L 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


cross-platform Applications
Hardware Interface Layer
platform-specific Applications
Hardware Adaption Layer
Hardware Presentation Layer
Platform A
Platform B
Figure 2.2.: Hardware Abstraction Architecture

3-level *Hardware Abstraction Architecture* to provide a flexible and performant framework to build applications on, as shown in figure 2.2.

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 re-post 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.

3. nesC

3.1. Fundamental Programming Hints

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

- 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.
-

3.1.1. LCD 2x16

4. bigAVR6

4.1. The Hardware

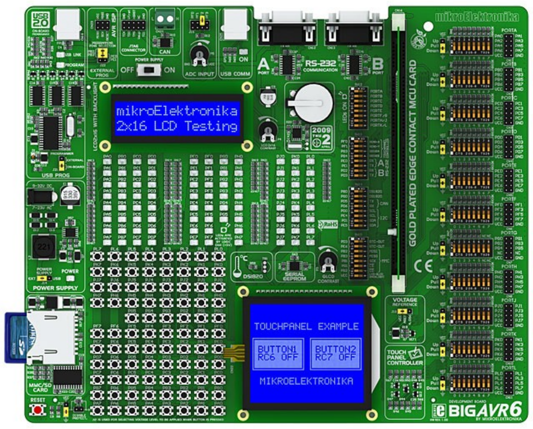


Figure 4.1.: bigAVR6 platform

4.2. Supported Modules

4.2.1. GLCD

4.2.2. MMC

4.2.3. Ethernetboard

4.2.4. LCD 2x16

5. buildenv

5.1. Prerequisites

6. 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

7. 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.

7.1. Types of Bachelor’s Theses

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

8. Design Approach

If you have derived some new concepts on your own, this is the place to present them. You can use generic names for this chapter like “Design Approach” or “System Architecture” or chose name accordingly to its contents (for example “Automatic Text Generator Algorithm”).

8.1. 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 from others, but no chapter as described here.

9. Implementation

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

9.1. 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.

10. 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".

10.1. 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 10.1.: Quality of calibrated infrared sensor data (from [Elm02])

10.2. 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

11. 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`.

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.