Lab Assignments in TDT4258 Energy Efficient Computer Systems

NTNU
Computer Architecture and Design Group
Department of Computer and Information Science

10th of December 2013

Contents

List of Figures

List of Tables

Abbreviations

ASIC Application Specific Integrated Circuit

DAC Digital to Analog Converter

ABDAC Audio Bitstream Digital to Analog Converter

DUT Design Under Test

EVBA Exception Vector Base Address

GCC GNU Compiler Collection

GDB GNU Debugger

GNU GNU's Not Unix

GPIO General Purpose I/O

GUI Graphical User Interface

I/O Input/Ouput

ICE In Circuit Emulator

INTC Interrupt Controller

IOCTL Input/Output Control

JTAGICE JTAG ICE

JTAG Joint Test Action Group

LCD Liquid Crystal Display

MMU Memory Management Unit

PC Program Counter

PIO Parallel I/O

RISC Reduced Instruction Set Computer

SD Secure Digital

SP Stack Pointer

SR Status Register

USB Universal Serial Bus

Chapter 1

Introduction

This document provides the information needed for the lab assignments in the course TDT4258 Energy Efficient Computer Systems. There are three assignments which are graded and the grades are part of the final grade in the course. The description of the three assignments is provided and the background information needed for the completion of the assignments.

There are four chapters: the introduction and a chapter for each assignment. The introductory chapter will introduce you to the lab setup. Each of the remaining chapters describes a single assignment. A background material for the assignments can be found across the chapters so that new information for the actual assignment lies in the corresponding chapter. At the end of each chapter devoted to individual assignments, there is a concise text which describes the assignment and a suggested approach to solving it. We recommend that you carefully read the background theory before beginning to work on the assignment. Assignments are thought up so that they build upon one another. Therefore, to solve the assignment 3, you need to understand a lot of the material relevant for assignments 1 and 2.

1.1 Practical goal: Scorched Land Defence

A practical goal of the lab assignments is to implement a variant of the Scorched Land Defence computer game on the development board by Atmel. The development board has a microcontroller, an LCD–display, a sound system, buttons and LED–diodes – all hardware components which are needed for making a computer game. Your task will be to program a microcontroller to control all I/O–components needed for the game and use these in the computer game you will implement yourselves.

Scorched earth defence is a military strategy which involves destroying anything that might be useful to the enemy while advancing through or withdrawing from an area. Although initially referring to the practice of burning crops to deny the enemy food sources, in its modern usage the term includes the destruction of infrastructure such as shelter, transportation, communications and industrial resources. Your final goal is to implement a simple version of the game where one player (the attacker) has to reach the position of the second player (the defender). The defender has to block the attacker by burning parts

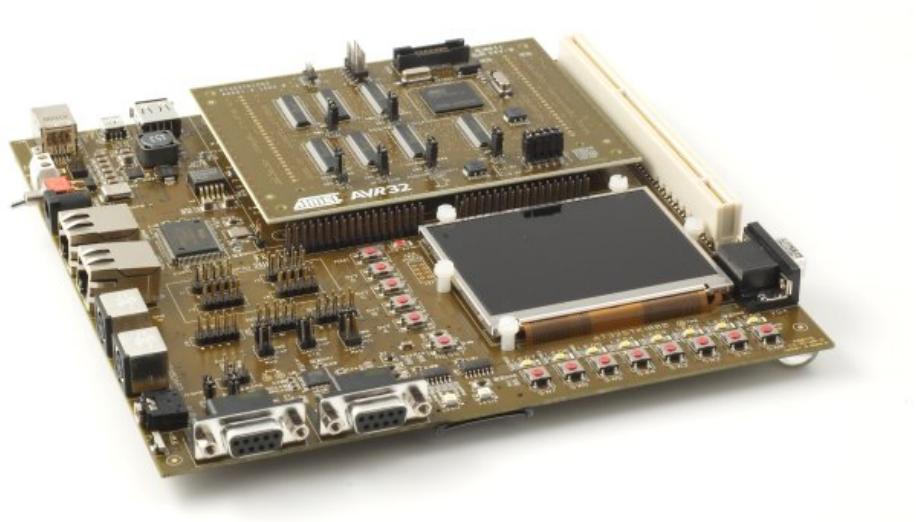


Figure 1.1: STK1000 development board

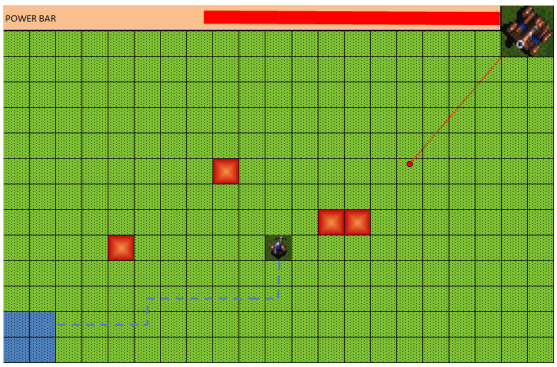


Figure 1.2: Scorched Game Defence

of the terrain. Other variants can be implemented.

1.2 Learning outcome

The setup for the lab assignments is made so that you will acquire practical experience in the development for microcontrollers in an embedded system. You will get experience with various kinds of programming for microcontrollers and experience with programming for hardware components by programming for a few often–used I/O–components.

You will get experience with the following:

- Programming for an AVR32–based microcontroller on the development board STK1000, both by Atmel.
  - Assembly programming
  - C-programming (with no operating system)
  - C-programming with operating system GNU/Linux on STK1000.
- hardware programming of I/O-components, both in assembly and C
  - Parallel I/O (buttons and LED–diodes)
  - Audio Bitstream Digital to Analog Converter (ABDAC) for generating sound
- Use of interrupts, both in assembly and C.
- Programming for Linux-kernel
  - Use of Linux functions call
  - Use of drivers in Linux
  - Making your own hardware drivers for Linux
    - Compiling of Linux-kernel
    - Making a kernel module
    - Hardware programming in Linux
- Cross–development from Ubuntu GNU/Linux
  - Use of development tools from GNU
  - General use of GNU/Linux on a PC.

1.2.1 Assignment 1

In the assignment 1, you will familiarise yourself with some of the widely used development tools from GNU. These will be used for programming in assembly code for AVR32. You will learn to find out which buttons are pressed on STK1000. In addition, you will learn how to handle interrupts in assembly.

1.2.2 Assignment 2

In the assignment 2, you will continue to use GNU toolchain but for programming in C. You will program in C for AVR32 but with no operating system to rely on. This means that you have to do most of the job yourselves. Here you will program the buttons, a simple task in C, but, in addition, you will program sound effects which will be used in the computer game. You will have to write a code which generates sounds and programs the ABDAC, a hardware component which converts the digital data to analog signals. The analog signals generated by ABDAC are further sent to analog output e.g. headphones. Use of ABDAC includes interrupts so that you will learn how to handle interrupts in C as well.

1.2.3 Assignment 3

This is the last assignment in the course. You will complete a computer game. Linux–kernel will be used on the development board and programming for Linux will make a large part of this assignment. You will learn to use the display and sound by directly programming drivers for these components. In addition, you will make your own driver for buttons. All this will be used for programming a computer game.

1.3 Practical information

1.3.1 Evaluation

Deliveries will be evaluated based on the delivered report, the code and a short presentation of your solution. During the presentation you will be asked some questions about the solution and the choices you made for it so that you show you worked on it independently.

The number of points for the solution will depend on the following:

- In what degree the requirements for the assignment have been met
- The quality of report
  - Clarity and readability
  - Complete yet concise
- Code quality and technical solutions
  - Well structured, clear and commented
  - The choice of technical solutions
- Testing
  - How the tests were performed
  - Test results
  - All material needed for running the tests must be attached to the delivery
- Presentation for the teaching assistant

- Solutions which stretch beyond requirements
  - You can get a maximum score for the assignment if the requirements are met in an appropriate and satisfactory way. Some extra points can be achieved if a relevant work has been done which goes beyond minimal requirements.

1.3.2 For late deliveries and copying

Deadlines for each assignment delivery are given in the timesheet on the course It's learning page. For late deliveries, 10% of the points scored for the assignment will be taken off for each day after the deadline. The only exception is the abscence because of illness. In such cases, you have to provide a written proof from your doctor or similar. As the grades given for the assignments are included in the final grade for the course, copying is not tolerated at all.

1.3.3 Lab and assistance

You are free to work on the assignments in the lab whenever it is convenient for you. However, during the lab hours designated for a three–hour weekly schedule with the help of student assistance, according to the course semester plan, the priority is given to the group which is assigned for those lab hours. If you want to get the most from the lab assistance, you are advised to attend the designated lab hours already well–prepared and acquainted with the assignment.

1.3.4 Ubuntu GNU/Linux

In this lab setup you will use a PC with an operating system Ubuntu GNU/Linux. Those who are not well acquainted with Linux from before will, therefore, have to learn about GNU/Linux in order to work on the assignments. For the beginners we recommend "Ubuntu Guide" (ubuntuguide.org) and "Kubuntu Desktop Guide"which can be found on the web. In particular, chapter 2 contains what you need to have an overview over Linux systems. In the assignment setup you will use shell (command line interface). Therefore, you need to learn how to use it. The information about shell can be found in section "Terminal" in the "Kubuntu Desktop Guide".

GNU/Linux will not be used on the PC but, later on (in the assignment 3), on the development board STK1000. So, there are more reasons for you to get to know Linux for this setup of lab assignments.

Further, you have to use a text editor to write your program code. Those who have used GNU/Linux before already have a favourite editor which can be used in this setup. For those who have no GNU/Linux–background, there is a short list of the often used text editors below. All these editors are powerful tools used for program development and they have possibilities for "syntax highlighting" and other smart features which are useful when programming.

- *Kate*: A text editor with a modern graphic user interface. This is presumably the editor easiest to use by those with Windows background. It is used as an editor for more advanced programming milieu.

- *Emacs*: A text editor which can be found in all operating systems and which can be endlessly developed further. A bit different user interface for those with Windows–background. In any case, we recommend this editor because, among other things, it can be used for program debugging. More information about Emacs can be found in section 2.11.1.

- *Vim*: An editor with (for most!) an unusual user interface. All the same, it has many devoted users.

1.3.5 Sources of documentation

This document does not provide enough information about everything you need. In the sections with background information you will be noted where you can find more information about the corresponding topic but there is a bit of general information sources in GNU/Linux which you need to know about.

man–pages

Luckily, all commands in Unix–based operating systems are documented with so–called "man–pages". You can read them by invoking the command `man` in shell:

`man <name_of_the_command_you_would_like_to_know_about>`

Try this now! Read the manual for the `man`–command itself:

`man man`

Man–pages are divided in various sections. The most important are the first three sections:

- Section 1: User commands (those which can be called from the command line)

- Section 2: Description of system calls

- Section 3: Description of functions in C–libraries

To specify that you would like to read a man–page for `printf`, you can provide the number of the section it belongs to.

`man 1 printf` will open a man–page for the `printf` command.

`man 3 printf` will open a man–page for the C–function `printf()`.

If you do not specify the section number, the first man–page with this name is shown. In the example with `printf`, a command `man printf` will open a printf–page which lies in section 1.

Mark that `man` is a very useful source of help.

info–pages

Some bigger documents and manuals are available as so–called "info–pages". This is a help–system which supports hypertext documents. Documents in the form of info–pages are, therefore, much easier to read and use than man–pages if they are of a larger size. In particular, the documentation of GNU software is often given as info–pages. Info–pages can be opened with info–command:

`info [name_of_the_document_you_would_like_to_read]`

Of course, info–command itself is documented on info–page. Write the following command to read info–documentation:

```
info info
```

To open a link (marked with the sign *) in the info–system, move the cursor over the link and press the **return**–key. To go back, press the **l**–key (**l** for "last", the same as "back" in a web browser).

It is actually the best to read info–pages through Emacs. Among other advantages, the links will be marked with different colour. Read about how to do it in section 2.11.1.

Other

If you wonder how a command works and you can't find either man– or info–pages, you can try to ask the command itself about how it works. It depends on the command how you can achieve that but typically one of the following arguments is given:

- `-?`
- `-h`
- `--help`
- `-help`

For example:

```
ls -help
```

1.4 Before you begin...

The lab setup is exciting and in a large degree independent. You will not be delivered complex frameworks which will do a large amount of work for you. Instead, the most will be done by yourselves. It is useful with respect to learning but also not an easy way to work. You must read the background information and the assignment description carefully before you set to work on the assignment. The background information is given at a somewhat general level so that you need to consult official documentation to get a detailed knowledge for completing the assignments. Where it is necessary, you will be given pointers to where to find more information. It is also to expect that you will spend some time searching for the information sources on the web if needed.

In the end of this chapter, a friendly advice: make a good estimate of the time needed for the assignments. It is better to finish one week earlier than to get stuck the last night before the deadline when there are no student assistants to help.

Chapter 2

Assignment 1

2.1 Introduction

In this assignment, you will write a program which enables a player to control a "paddle" on the row of LED–diodes. You will write your program in assembly and you need to know how to program buttons and LED–diodes on the STK1000 board.

2.1.1 Learning outcome

The learning outcome of this assignment is:

- General architecture of AVR32–based microprocessors
- STK1000 development board
- Understanding of object files and the task of a linker
- Use of GNU-toolchain
  - GNU AS (assembler)
  - GNU LD (linker)
  - GNU Make (automatic use of assembler and linker)
  - GNU Debugger (GDB) (debugger)
- Programming in assembly for AVR32
- Parallel I/O, hardware I/O–components which control buttons and LED–diodes
- Interrupt handling in assembly for AVR32

2.2 AVR32

AVR32 is a processor architecture by Atmel which was launched in 2006. This is a 32–bit RISC–processor especially appreciated for the use in embedded systems with relatively high performance requirements. As shown in Figure 2.2,

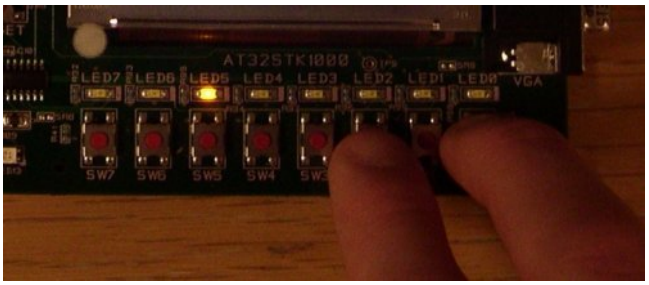


Figure 2.1: Assignment 1: Buttons and LED–diodes

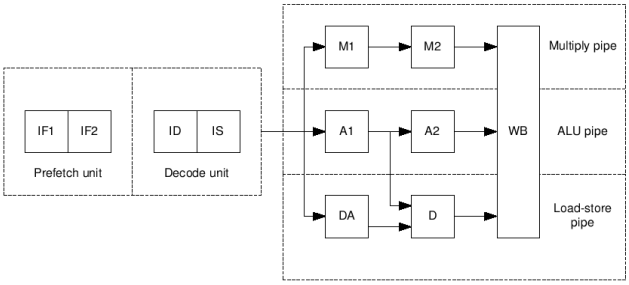


Figure 2.2: The AVR32–core pipeline [1]

an AVR32 has a seven stage pipeline architecture with the possibility to start the execution of three different instructions in parallel. The processor has a relatively high clock frequency compared to other processor cores which are used in microcontrollers.

In this lab setup, we shall use a specific microcontroller: AT32AP7000. It contains an AVR32 processor core, speed buffer, Memory Management Unit (MMU) and a variety of I/O–controllers which are integrated on the same chip. It is useful to mark the difference between a processor and a microcontroller. A processor is a part of a computer which executes instructions, while a microcontroller is a microchip which contains a processor but also a number of other useful components as, for example, various I/O–controllers.

A detailed description of AVR32–architecture can be found in AVR32 documentation [2]. There you can find a manual for the AVR32 instruction set, among other things. An AT32AP7000 datasheet [1] describes everything contained by an AVR32–based microcontroller as, for example, all I/O–components which are integrated on the chip.

2.2.1 Block diagram

An overview of the system is shown in Figure 2.3. It shows how different STK1000 parts are interconnected. The box “STK1000” corresponds to the whole development board and the box “AP7000” corresponds to the microcontroller. This figure shows only the modules which you need to know about. The official block diagram for the microcontroller includes all the modules and is shown in Figure 2.4.

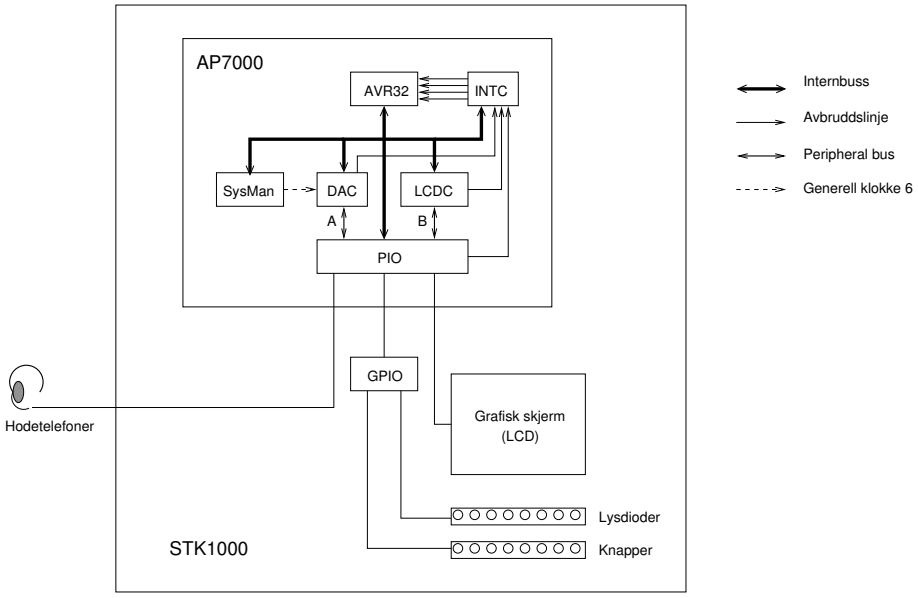


Figure 2.3: Overview of the system

It is useful to note that nearly all I/O–controllers which should have a possibility to send signals out from the microcontroller must use a so–called Parallel I/O (PIO) I/O–controller for that purpose. This is described in more detail in section 3.3.

2.2.2 Register file and system registers

A brief description of the AVR32 register file would be that it contains 16 32–bit registers which can be accessed by the program code. These are 13 general registers (r0 – r12), Link Register (LR) which contains a return address for a procedure call, Stack Pointer (SP) and Program Counter (PC). Besides, there is a number of system registers which are used for different functions of the AVR32. System registers can be read and written by special instructions. In particular, two of them which will be used in the assignments: Status Register (SR) og Exception Vector Base Address (EVBA). Status register with all its flags is documented in chapter 2.10 in the AVR32 architecture datasheet [2]. EVBA is related to interrupt handling about which you can read more in section 2.5.1.

2.2.3 Memory map

The memory map of the microcontroller is shown in Figure 2.5. Microcontroller has an MMU and, therefore, it has a possibility for mapping between virtual and physical addresses. MMU is used in the assignment 3 with Linux–kernel but there is no memory mapping in assignments 1 and 2.

As shown in the figure, different parts of the address space are used for different purposes by MMU and cache. Some addresses are never in cache and some are never translated by MMU.

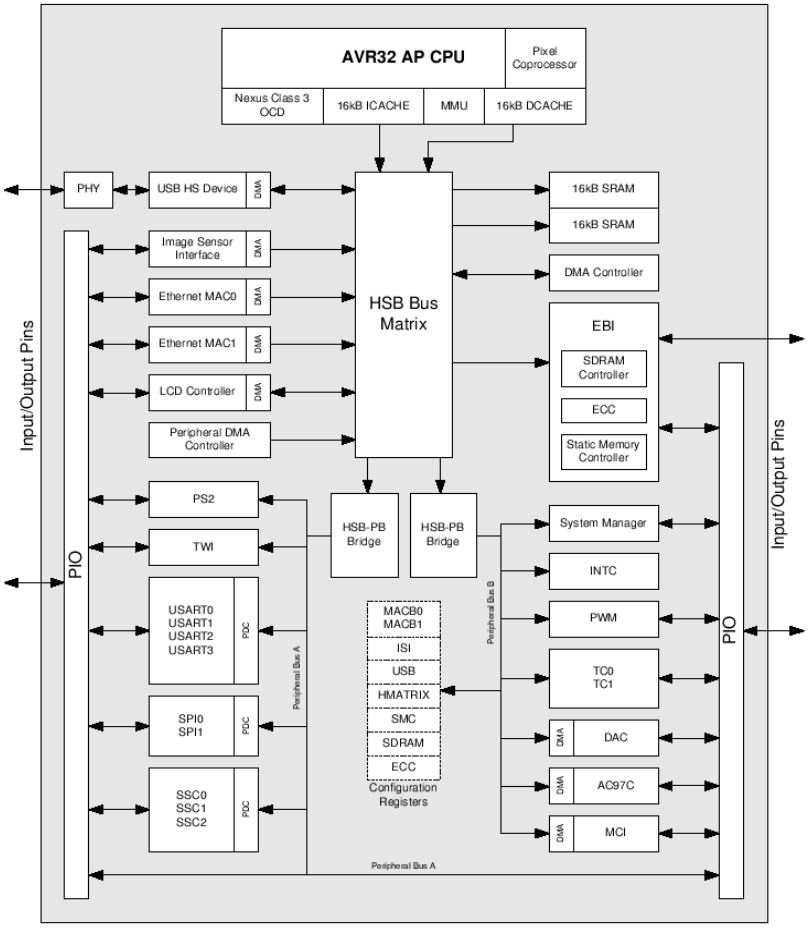


Figure 2.4: Block diagram for the microcontroller AT32AP7000 [1]

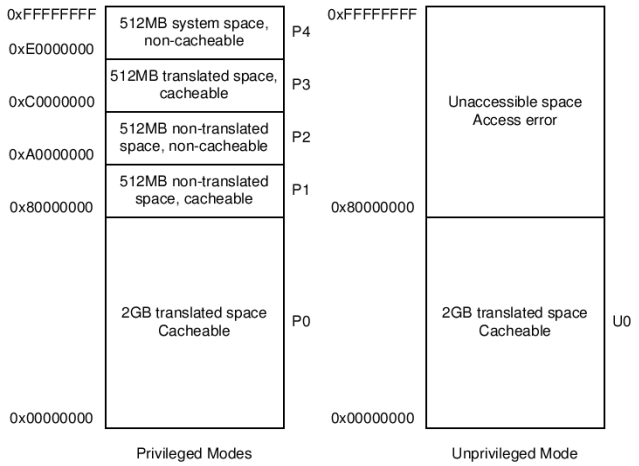


Figure 2.5: Memory map in AVR32 processor [2]

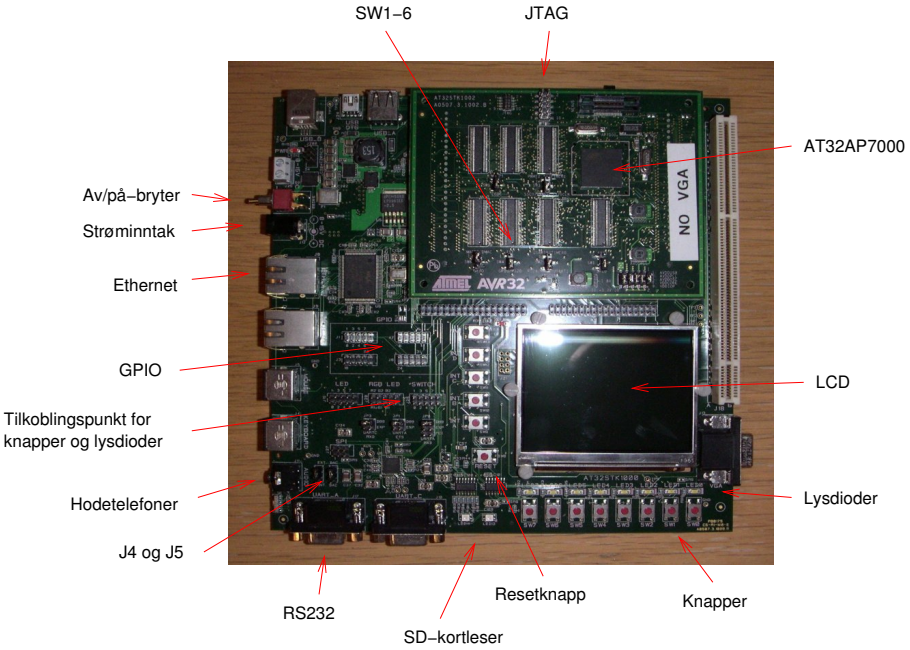


Figure 2.6: Overview of the STK1000 board

Here is a bit more detailed description of some areas in the memory address space which are important for the course assignments:

- `0x00000000-0x007fffff`
  A flash memory (and therefore the program memory) on STK1000 is mapped to this area. The area is cachable.

- `0x10000000-0x107fffff`
  SRAM (therefore data memory) on STK1000 is mapped here. It is a cachable area.

- `0xa0000000-0xa07fffff`
  A flash–memory is mapped here too. Therefore, this area is an identical copy of the area from the address 0 but it is not cachable. When the processor is powered on or reset, the program counter will start from the address 0xa0000000 so it is essential that a flash–memory is mapped here.

- `0xff000000-0xffffffff`
  I/O address space. AVR32 used memory mapped I/O so that all the registers which are related to I/O–controllers will be accessible at certain address in this area.

2.3 STK1000

STK1000 development board by Atmel is used for the demonstration of the AVR32 possibilities but also as a system which can be used by developers during the development of a new product. This board is used in all three assignments.

Figure 2.7: JTAGICE mkII

User documentation for STK1000 [3] can be found on the lab PCs. Open the file `/usr/local/stk1000/docs/html/AVR32_BSP_User_Guide.html` in a web browser to read the documentation.

Development board consists of two modules, STK1000 as a main board (mother board) with a variety of I/O–components and STK1002 as a daughter board which contains the processor itself.

See Figure 2.6 for an overview of various parts of the STK1000.

2.3.1 Configuration

There is a number of so–called "jumpers" on the STK1000. Jumpers are short wires which can bypass a break between two points in the circuit. By setting or removing jumpers, different properties of the STK1000 can be achieved. In the beginning of the text for each assignment, you will be given the setup of the jumpers which is needed for that assignment. Those with extra interest in what all the jumpers mean on the STK1000 can read about it in the STK1000 documentation.

In addition, there is a general I/O–bus which is not connected to anything in particular but it can be used according to the users' (yours!) needs. It is called General Purpose I/O (GPIO) and it can be connected by flat cables to various components. You will typically connect GPIO–bus to LED–diodes and buttons.

2.3.2 JTAGICE

As software developers, you need to load new programs to flash–memory on the STK1000 and to run the processor in such way so that you can simply debug your program. Both uploading new programs and debugging are done through a JTAGICE device. This is a device which connects an STK1000 board with a PC and enables you to control STK1000 and AVR32–processor by software on

the PC. JTAGICE and its connection to with STK1000 is shown in Figure 2.7. JTAGICE is connected to a PC by usual USB–cable.

Program upload to microcontroller

There is a flash memory chip on the STK1000 which is used for the programs. As described in section 2.2.3, a flash memory will be mapped to both areas of address space: the area which starts at address `0` and the area which starts at address `0xa0000000`. The latter ensures that the processor will start to run the program which is in the flash memory on power on or reset.

So, to program a microcontroller, a flash memory needs to be written. This is done with the help of a JTAGICE and a program called "avr32program". Connect a JTAGICE both to the STK1000 and to the USB–port on your PC.

Before the program file is uploaded to the STK1000, it is wise to first stop the program which is already running on the processor. This can be done with the following command:

```
avr32program halt
```

Run this command on the PC before uploading a program to flash memory:

```
avr32program program -e -f0,8Mb <elf-programfil>
```

Here the "elf-programfil" is the program you want to upload to flash memory. After this, you only need to press the reset button on the STK1000 to start your program running.

2.4 Parallel I/O on AVR32

An AP32AT7000 microcontroller contains an I/O–controller which is called PIO. It controls general I/O–pins on the chip and it can be used for other purposes as well. A general I/O–pin can be configured as either input or output. The difference is that for an input pin the value can be read, while for an output pin the value can be written to. The buttons and LED–diodes on the STK1000 can be connected to I/O–pins which are controlled by the PIO–module.

As for the other I/O–controllers, PIO is controlled by reading and writing a certain set of registers. An overview of the PIO registers is given in table 2.1. All I/O–registers in the microcontroller are memory mapped. Therefore, all I/O–registers can be accessed by reading/writing certain addresses. In the table 2.1 every register is assigned an offset. This is because there are five I/O–ports, PIO A – PIO E, each a 32–bit port. Base addresses for their selection are given in table 2.2. To find the address of a certain register for a given port, the base address of the port must be added to the offset of the register. For example: If you want to write to the PUER register of the PIO B port, you have to use the address 0xFFE02C00 + 0x64 = 0xFFE02C64. Each PIO register is 32–bit wide and each bit controls the corresponding I/O–pin of the concrete PIO-port.

Physically, one part of PIO–ports will be connected to GPIO–bus which is available on STK1000 board (Figure 2.8). When PIO is to be used, it is here that you will physically connect the components you wish to control with PIO–controller (for example, buttons and LED–diodes). In general, the mapping between PIO–ports and GPIO–bus is as given in table 2.3.

Register	Offset	Description
PER	0x00	PIO Enable Register: The pins of the port which should be enabled are set here
PDR	0x04	PIO Disable Register: The pins of the port which should be disabled are set here
OER	0x10	Output Enable Register: The pins which should be used as output are set here (default is input)
PUER	0x64	Pull-up Enable Register: The pins which should have pull–up resistors are set here (this is needed for reading the status of the buttons)
PDSR	0x3c	Pin-Data Status Register: The values of the port pins can be read here
CODR	0x34	Clear Output Data Register: The output pins which should get the value 0 are set here
SODR	0x30	Set Output Data Register: The output pins which should get the value 1 are set here
IDR	0x44	Interrupt Disable Register: The pins which should be disabled to generate interrupt are set here
IER	0x40	Interrupt Enable Register: The pins which should be enabled to generate interrupt are set here
ISR	0x4c	Interrupt Status Register. It has several functions: (1) Reading what pins have generated interrupt. (2) Reading will erase interrupt request (cancel all pending interrupts). (3) After interrupt PIO can not generate a new interrupt of the same type. Reading of ISR will cancel this situation and allow new interrupts.
ASR	0x70	Peripheral A Select Register: The pins which will be controlled by peripheral A instead of PIO–controller are set here.
BSR	0x74	Peripheral B Select Register: The pins which will be controlled by peripheral B instead of PIO–controller are set here. Note that for this to happen, the same pins have to be disabled in PDR–register.

Table 2.1: PIO-registers

Port	Base address
PIO B	0xFFE02C00
PIO C	0xFFE03000

Table 2.2: PIO base addresses

PIO signal	GPIO-bus	Note	
Port B pins 0–10	GPIO pins 0–10	Only when SW4 is set to GPIO	
Port B pins 13–14	GPIO pins 11–12	Only when SW4 is set to GPIO	
Port B pins 15–16	GPIO pins 13–14		
Port B pins 30	GPIO pins 15		
Port C pins 0–15	GPIO pins 16–31	Only when SW6 is set to GPIO	

Table 2.3: Mapping between PIO-ports and GPIO-bus on STK1000

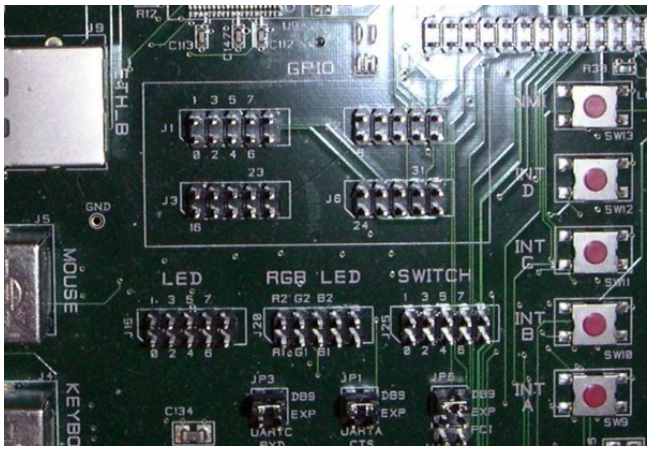


Figure 2.8: GPIO and connection to buttons and LED–diodes

More information can be found in the microcontroller datasheet [1] in chapter 19 . Turn to those pages when you need more information about the use of PIO.

2.4.1 Example: Use of buttons

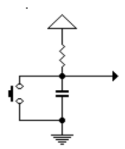
In order to use the buttons on STK1000, you need first to connect them to a PIO port. This is done by connecting GPIO–bus (for example, pins 0–7) and the contact marked as "SWITCH" by a flat cable. Then, PIO–controller has to be set up programmatically. This is done in following steps:

1. Activate the desired port pins by setting the relevant bits in the port's PER register to 1.
2. Activate pull–up resistors by setting the relevant bits in the port's PUER register to 1. This is necessary because of the way in which the buttons on the STK1000 board are implemented. If you want to read the value of a button, a pull–up resistor of a corresponding input pin must be activated.
3. Now you can read what buttons are pressed in the port's PDSR register. Note that the value 1 is read when the corresponding button is **not** pressed and value 0 when it is pressed.

Pay attention to a so–called *bouncing* phenomenon which is common for various types of buttons. It can influence the status of the buttons so that a wrong button status may be read. The main cause lies in the fact that each time a button is pressed, several contacts occur on its moving parts and this may result in reading different values of the button status. This phenomenon can be compensated by different methods. These methods can be divided in general in two large groups: hardware and software methods. Figure 2.9 shows how *debouncing* can be done with an RC–circuit (a) or in assembly code (b).

2.4.2 Example: Use of LEDs

In order to use LED–diodes on an STK1000, first you need to physically connect them to a PIO port. This is done by a flat cable which is connected between a



a

```
intr_sleep_start:
        sub r0, 1
        cp.w r0, 0
        breq intr_sleep_end
        rjmp intr_sleep_start
intr_sleep_end:
```

b

Figure 2.9: Example of debouncing in hardware (a) and in software (b)

GPIO–bus (for example, the pins 16–23) and the contact marked "LED". In the program you need to set up the PIO–controller. This is done in following steps:

1. Activate the desired port pins by setting the relevant bits in the port's PER register to 1.
2. Set the direction of the desired pins as output by setting the relevant bits in the port's OER register to 1.
3. Now you can turn the LEDs on and off by writing 1 to the port's registers SODR and CODR respectively.

2.5 Interrupt handling in AVR32

Processor core AVR32 has four general external interrupts: INT0–INT3. These are shown as four incoming interrupt lines to AVR32 in Figure 2.3.

When the processor receives an interrupt request (one of the four interrupt addresses is set to a high logical value), it will stop processing what it has been processing by this moment and jump to an interrupt routine. The processor obtains the address of the interrupt routine by putting together the address which lies in the system register "EVBA" and a so–called "autovector" which is provided by an external source of interrupt. The autovector is represented by the 14 least significant bits in the interrupt routine address. The following operation is performed in order to calculate the interrupt routine address (| denotes bitwise logical AND):

$$\text{interrupt\_routine\_address} = \text{EVBA} \mid \text{autovector}$$

The most of the I/O–controllers need to generate interrupt at some point of their usage so the four external interrupts are far from enough. Therefore, an AP32AT7000 microcontroller has its own interrupt controller named Interrupt

Bits:	31–30	29–14	13–0
Function:	Interrupt type	Unused	Autovector

Figure 2.10: The fields of IPR-register of the interrupt controller

Register name	Offset
IPR0	0
IPR1	4
...	
IPR14	56
...	
IPR63	252

Table 2.4: IPR-registers of the interrupt controller

Controller (INTC) which gathers interrupt requests from all possible sources and sends these furter to AVR32–core as one of the four possible interrupts.

Interrupt controller organises all the incoming interrupt requests into groups. There are 64 groups and each group has 32 individual interrupt sources (called number or line). So, there can be in total $64 \cdot 32$ different interrupts which can be received by the interrupt controller and this is more than enough to serve all the I/O–controllers which may generate interrupt.

For example: the port B of PIO–controller has interrupt line number 0 in group 14. When it should be set up so that it can generate interrupt, interrupt controller must be set up to send interrupt 0 in group 14 further to the AVR32. The information about which groups and numbers different I/O–controllers have can be found in the datasheet for the microcontroller [1] on pages 77–78.

In order to enable activation of interrupt from a specific I/O–controller, the following activities have to be performed programmatically:

1. Set up I/O–controller so that it can generate interrupt
2. Set up interrupt controller so that it can send this interrupt further to the processor
3. Set up the processor so that it will react to the interrupt

Point 1 depends on I/O–controller and you need to read the documentation for the specific I/O–controller to find out how this is done. This involves typically setting one or more bits in an “interrupt enable register” or similar.

Point 2 is done by setting the IPR register of the corresponding interrupt group. IPR register’s address can be found in table 2.4 and each of the IPR registers has a layout as shown in Figure 2.10. Each interrupt group must be set up with the type (INT0–INT3) of interrupt it will generate and which autovector is valid for the interrupt from this group.

Point 3 is done by setting up the processor’s EVBA system register to the base address for interrupt routine. Thereafter the interrupt must be globally enabled by setting the GM bit in the status register to 0.

The interrupt controller is documented in the datasheet for the microcontroller [1], chapter 13, and interrupt handling by the processor is described in the architecture manual for AVR32 [2], chapter 8.

2.5.1 Interrupt routine

An interrupt routine is a piece of code to which the processor will jump when interrupt request arrives. There are a few things to think of when you write an interrupt routine. Remember that an interrupt leads to a temporary break of the processor's job which runs the interrupt routine instead. It is, therefore, important to ensure that the interrupt routine does not ruin the program which has been run by the processor before.

Saving the registers temporarily If you use the registers in the interrupt routine which are also used at other places in the program, the values of these registers must be saved temporarily at some place so that they can be recovered at the end of interrupt routine.

For this purpose, you can use stack. Push the registers to stack in the beginning of the interrupt routine and pop them in opposite order at the end of interrupt routine. Read more about stack in section 2.9.

Returning from interrupt Interrupt routine must finish with the instruction `rete` (Return from Exception). This instruction will make the processor jump back to the place where it was before the interrupt occurred. AVR32 will automatically recover the status register to the state it was before interrupt routine was run.

2.5.2 Example: Setting up the interrupt from PIO port B

To set up the PIO port B so that it can generate interrupt, the following steps must be done:

1. Write an interrupt routine which will handle reading/writing of the data to PIO port B. Remember to read the PIOB register ISR even if you don't need this value at all. This is necessary to do so that PIO knows that interrupt has been handled and that it can generate new interrupts next time a button is pressed.

2. Set up PIO B so that it can generate interrupt at each change of the port:

   - Write value 1 to all bits which correspond to the pins which you want to get interrupt from, in PORT B register IER.
   - Write value 1 to all other bits in the IDR register in order to avoid receiving interrupts from other sources than those you would like.

3. Decide upon the address for EVBA and write this one to the processor's EVBA register. This is done with the instruction `mtsr 4,r1` (EVBA has a register address 4), with the assumption that r1 contains the wanted EVBA.

4. Calculate autovector for interrupt routine in relation to EVBA and write this in the IPR register number 14 of the interrupt controller (because PIO port B is in the interrupt group 14).

5. Enable interrupt globally by setting the bit GM (Global Interrupt Mask) in the processor's status register to 0. This is done with the instruction `csrf 16` (GM-flag is bit 16 in the status register).

A note about the choice of EVBA and autovector: in a simple program as it is the one for this assignment, it can be the easiest way to set EVBA to address 0 and set the autovector directly to the address of the interrupt routine.

Note that every change of the status at the PIO port B will generate interrupt. That means that when a button is pressed, an interrupt will be generated when the button goes down but also when it goes down.

2.6 GNU-toolchain

GNU is a project sponsored by Free Software Foundation (FSF) with the main goal to provide a free and open operating system with all accompanying tools for software development. All Linux–based machines widely use GNU software for which they are rightly called GNU/Linux–machines meaning that the operating system is GNU with Linux kernel.

Although GNU-project provides its tools primarily for GNU operating system, they have been adapted for all of the most popular operating systems. This means that irrespective of the operating system, GNU tools can be downloaded from the internet and used within it.

The most popular GNU tools are software development tools and, therefore, primarily C–compiler, debugger and accompanying tools. You will use them for the set of lab assignments in this course.

All tools from GNU come with a detailed documentation which is available both as a book and as a hyperlink document. Use `info`–command to access these or search the web.

2.6.1 Cross–development

Cross–development denotes program development on one platform and running it on another. It is a usual form of development for embedded systems. The programming is done on a usual PC and the program is run and tested on a target system. You will do the following: develop programs on a GNU/Linux–machine and transfer binary files to STK1000 to see if they work as expected. Development tools which come with a GNU/Linux–machine cannot be used for cross–development so that Atmel has delivered its own variants of the tools which can be used instead. These are installed on the lab machines you will use. They can be downloaded from the Atmel homepage if you wish to install the tools on another machine (for example, the one you have at home). This is described in STK1000 manual [3].

2.7 Assembly

Assembly programming is the lowest level in which a machine can be programmed. We can imagine writing binary files in a machine language but that is never done. Instead, an assembler is used which takes a description of the program instructions and "assembles" them into a binary file. This must not be

mixed with compiling which takes a high level language and translates it into an executable binary file. Assembly language has (almost) direct mapping from the text in assembly file to the final binary file in machine language. Therefore, each processor has its own unique assembly but, for example, C–code will be the same for all processor types. You will learn AVR32 assembly. Even if it is unique for AVR32, the principles will be the same for all processors.

2.7.1 Instructions

An assembly file is a list of all the instructions which the program contains. Each instruction has the following form:

```
<mnemonic> <arguments>
```

Here, "mnemonic" is the name of the instruction which will be executed and arguments are a list of arguments separated by comma. The length of the list of arguments depends on the concrete instruction. Each argument is typically a register. Here is an example:

```
mov r1, r2
```

This instruction copies register 2 to register 1. The convention in the AVR32–assembly is that the first argument is the destination where the result of the instruction will be placed. In the example above, it is register 1.

All instructions are described in the architecture manual for AVR32 [2] in chapter 9.4. There is a nice overview of all instructions in chapter 9.3.

2.7.2 Numbers

There is often a need to specify a number in different numeral systems. To specify a number in a given numeral system, set the following in front of the number:

- Binary number: `0b`
- Octale number: `0`
- Decimal number: No prefix (default)
- Hexadecimal number: `0x`

Example: If you would like to write a hexadecimal number `5b`, you need to do it like this: `0x5b`.

2.7.3 Comments

Comments make assembly code easier to read. Writing comments is different from one assembly to another but in the assembly you are going to use comments are written in the same way as in C. For example:

```
/* this is a comment */
```

2.7.4 Symbols

In order to avoid hard–coding all addresses and values, you can use symbols. A symbol is the name you give to either address or constant.

Setting symbols explicitly

You can set a symbol value explicitly. It is done like this:

```
SYMBOLNAME = value
```

A concrete example in which a symbol "RETURNCODE" is introduced for the value 0x13 (hexadesimal 13):

```
RETURNCODE = 0x13
```

Labels

An important type of symbols are so–called labels. They can stand at any place in the code and the value of the label will be equal to the address of the instruction which follows it. This is highly appreciated for all kinds of jumps because you don't need to keep the track of all absolute addresses. Here is an example of a loop which counts down the register 1 until it is equal 0:

```
loop:   sub r1, 1
        brne loop
```

`brne`–instruction causes the execution to leave the loop if the previous instruction gave the result 0. `brne`–instruction takes one argument: the address to which the processor needs to jump. We write a label so as to avoid to write an address to which the program needs to jump and use this label as an argument for the instruction. Assembly will then take care of computing which address will be used when the code is assembled to a binary machine code.

2.7.5 Pseudoinstructions

There is a special type of instructions which are not proper instructions. They will not be translated to machine code in a binary file but, instead, they represent commands for the assembly itself. Such instructions are named pseudoinstructions. Pseudoinstructions are also known as directives.

Here is a list of useful pseudoinstructions for GNU–assembly:

- `.include "filename"`: Includes the file "filename". It is useful if you want to include a list of constants (explicitly set symbols) which you would like included in more than one source file. Then, they can be put in their own source file which is included by all the others. It corresponds to a header file in C.

- `.text`: Specifies that the code that follows will be placed in the text–segment. Read more about segments in section 2.8.1.

- `.data`: Specifies that the data which follow will be placed in the data–segment. Read more about it in section 2.8.1.

- `.globl symbol`: Specifies that a symbol "symbol" will be a global symbol, i.e. that it will be possible to refer to it from other object files. Read more about global symbols in section 3.2.3 about object files and linking.

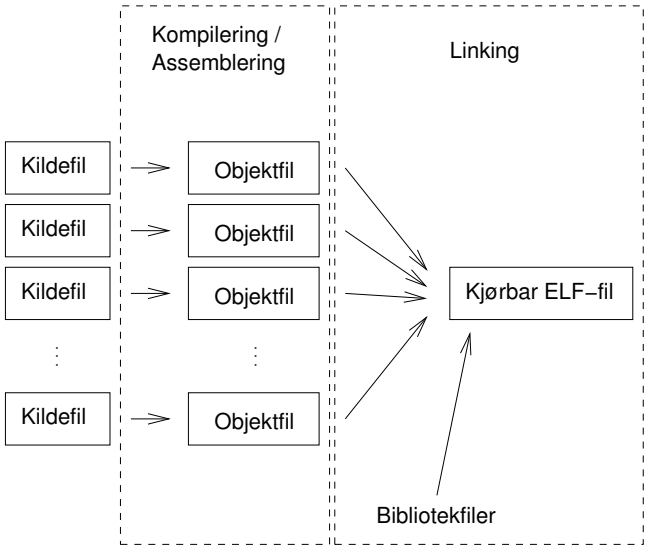


Figure 2.11: Link process, from source files via object files to an executable program.

2.7.6 GNU AS

The assembly you will use is called GNU AS. Its manual [7] can be obtained by the use of `info`–command. GNU AS is used like this:

```
as -gstabs -o <outputfile> <assemblyfile>
```

As you will work with cross–development, you need to use a special variant of as–command and the command line will be as follows:

```
avr32-as -gstabs -o <outputfile> <assemblyfile>
```

Arguments and options have the following meaning:

- `-gstabs`: Make debug–symbols so that program can be debugged with GDB.
- `-o <outputfile.o>`: Write the result of assembling to the file "outputfile.o". It is important to provide a file name with the extension ".o" because it is not an executable binary file but an object file. Read more about it in section 3.2.3.
- `<assemblyfile>`: The name of the assembly file which will be compiled by assembler.

2.8 Object files, libraries and linking

Many assemblies make an executable file directly. GNU AS is a more advanced assembler which makes an "object file" instead. It is not a ready executable file, rather something which must be converted to an executable file by the "linker". The same concept is used by a C–compiler which also compiles the code to an object file and to an executable file.

The advantage is that for a given program, there can be more different source files which assembler needs to compile, each for itself, so that linker can gather all different object files to an executable file. Often there are big parts of the program which are hardly ever changed and then it is not desirable to compile everything together anew because of a change in a small part of the programme.

Object files are binary files which contain machine code but where the symbols are still not set to certain value. The labels can not be computed before all object files are set together in a linking step because until then it is not known in which area of the address space the source code will be placed. Additionally, it is desirable to have the possibility to reference global symbols in other source files (marked with a `.globl`–directive in the assembly code) and these are not known before the details of the whole program are known in the linking stage.

A convenient thing with the use of the linking stage is that more languages can be used for the program development. The linker does not care if an object file was made by assembler, C–compiler or Pascal–compiler as long as it is in a correct format.

In addition, most programmers use one or more "libraries". A library is an already–compiled code which is made by someone else in order to be used by various programmers. Libraries are combined with your program in the linking stage.

2.8.1 ELF and segments

As mentioned, the result from the linker is an executable file. In a GNU/Linux–world this file is in the format which is called "elf". In addition to the ready machine code in binary format, it contains a bit of extra information.

An elf–file has more different segments. A segment is a portion of the program code. The most important segments from the perspective of an assembly programmer are text–segment and data–segment. Text–segment is assumed to contain program code, while data–segment contains variables used by the program code.

Important about assembly programming for STK1000

When programming in assembly, you need to specify in which segments your code will reside. This is done by directives `.text` and `.data`. All program code will come after `.text` directive and all (writable) variables will come after `.data` directive. There is often nothing to say if variables end in a text–segment but for AVR32 and STK1000 this is important. The program code (text–segment) is executed from a flash memory which is not writable from the program code. Therefore, it is essential to place variables which you'd like to write to in a data–segment as these data are placed in SRAM. This also holds for the stack: if you use stack, remember to initialise stack pointer in the beginning of your program so that it points to a certain position in SRAM.

In addition to taking care that correct segments are used, as an assembly programmer you need to define a global symbol `_start` which represents the address where the program execution will begin.

This is done simply by structuring the assembly code in this way:

```
/* here come all the explicitly set symbols */
```

```
.text

.globl  _start
_start:

/* here comes program code */

.data

/* here come all the data areas which can be written by program */
```

2.8.2 GNU LD

You will use linker which is called GNU LD. Use `info`–command to read the manual [8] but, in brief, it is used like this:

`ld <arg1> <arg2> ...  <argN>`

...or in our case where we code for avr32:

`avr32-ld <arg1> <arg2> ...  <argN>`

The arguments are:

- One or more object files
- `-o <outputfilename.elf>`: write the result of the linking stage to the file "outputfilename.elf". The suffix .elf is not necessary but it is useful for us because we do cross–development.
- `-l<library>`: include a library in the elf–file. Library file will be fetched from the system folder.

Here is a concrete example:

`avr32-ld -o program.elf file.o fileb.o -llib`

This will link together object files "file.o", "fileb.o" and library "lib" and write the result to the file "program.elf". Mark that the order of the arguments matters.

2.9 Stack

Stack can be useful in many cases. AVR32 has (as any other processor) a special hardware support for stack operations.

In AVR32 you can use the following instruction to push register data to the stack:

```
st.w --sp, r0  /* update the stack pointer and push r0 on stack */
```

To pop from the stack:

```
ld.w r0, sp++ /* pop r0 from the stack and update the stack pointer */
```

Both operations assume that a stack pointer is initialised to a reasonable value in the beginning of your program. An assembly trick you can use here is to set a stack pointer to the value `_stack`. This is a symbol which the GNU–linker will set to an appropriate place for a stack. Mark that `_stack` is a 32–bit value (full address). To load a 32–bit address to a register, you can use `lddpc`–instruction. It will load a 32–bit value from the memory location which is given relative to the program counter. Here is an example where we set a stack pointer:

```
        ... /* some code */

        /* loads a register with the value at the address "stackptr" */
        lddpc sp, stackptr  /* loads a stack pointer (sp) */

        ... /* some code */

stackptr:
        /* our 32--bit value which we want to load */
        /* this should lie in the same segment as lddpc-instruction */
        .int _stack
```

2.10 GNU Make

It is too cumbersome to compile a big project if all the commands are manually written. Therefore, there is a tool which automates this process: GNU Make. In brief, a so–called Makefile is set up where it is specified how to build the project (in other words, how to compile/assemble and link the files into an executable file) so that in the future it suffices to use the command "make" when a new version of the program needs to be built. Make is also wise enough to compile only object files which need to be compiled, while the source files whose code has not been changed since the last compilation need not be compiled anew.

2.10.1 How to make a Makefile

A Makefile lies in the same folder as source files and is always named "Makefile". A Makefile consists of "rules" which specify how to make, for example, an object file from a given source file. A rule has the following format:

```
target_file: dependencies
        command_line
```

- `target_file`: The name of the resulting file
- `dependencies`: A list of files (separated by a space caracter) on which the target file depends. This means in practice a list of the files where a target file must be made anew if there is a change in one or more of these files.
- `command_line`: The command line which must be executed in order to generate a target file. NB: Command line must be in a separate line and it MUST begin with a tab. If you set there a space character, it will not work, you have to use tab.

Here is an example:

```
example.o : example.c example.h
        gcc -c -o example.o example.c
```

This rule states that object file "example.o" depends on the files example.c and example.h, all the changes in these files implicate that example.o should be compiled anew. The command which will be executed to generate example.o is `gcc -c -o example.o example.c`.

Usually, there is such rule for each object file which makes a program and a rule which represents a linking stage which depends on all the object files. Let us see one typical linking rule:

```
example : example.o
        ld example.o -o example
```

Here we see that this rule depends on example.o, a file which is itself generated by a make–rule. When make tries to link the file "example", it will first check if example.o needs to be generated anew. In such a case, it will generate it before linking "example".

The first rule is the one which represents the executable program and, therefore, a linking stage. So, the linking stage is (almost) always over all other rules in a Makefile. If you would specifically like to execute some other rule from the top rule, you can give that on the command line:

`make example.o`

This command will build example.o but it will not execute a link rule because example.o is not dependent on the result of the linking stage.

A common rule you are encouraged to make is a so–called "clean"–rule. This is a rule which can look like this:

```
.PHONY : clean
clean :
        rm -rf *.o example
```

Typically it is placed at the bottom of the Makefile and it will not be executed unless you specifically ask for it when you run make, (`make clean`). The intention is to clean up, remove all autogenerated files so that only source files remain. `.PHONY : clean` is a notification for Make which says that clean is not a proper rule, when it is executed, it does not generate a new file which is named clean.

These simple rules suffice to make Makefiles which work but there are many clever tricks which can make a Makefile more elegant and easier to write and which make the job easier for program developers (you). We recommend that you look into the GNU Make manual [9], it is accessible as an `info`–page. What you are especially encouraged to do is to look up the use of variables in Makefile and especially automatic variables.

2.11 GNU Debugger (GDB) – Debugging tools

All programs contain errors and that is why it is essential that you have a possibility to debug the program. You will use a debugger named GDB. It is a rather powerful program which offers many possibilities to monitor the program execution, stop the execution and inspect the contents of registers and memory. It also gives the possibility to "single step" the program execution,

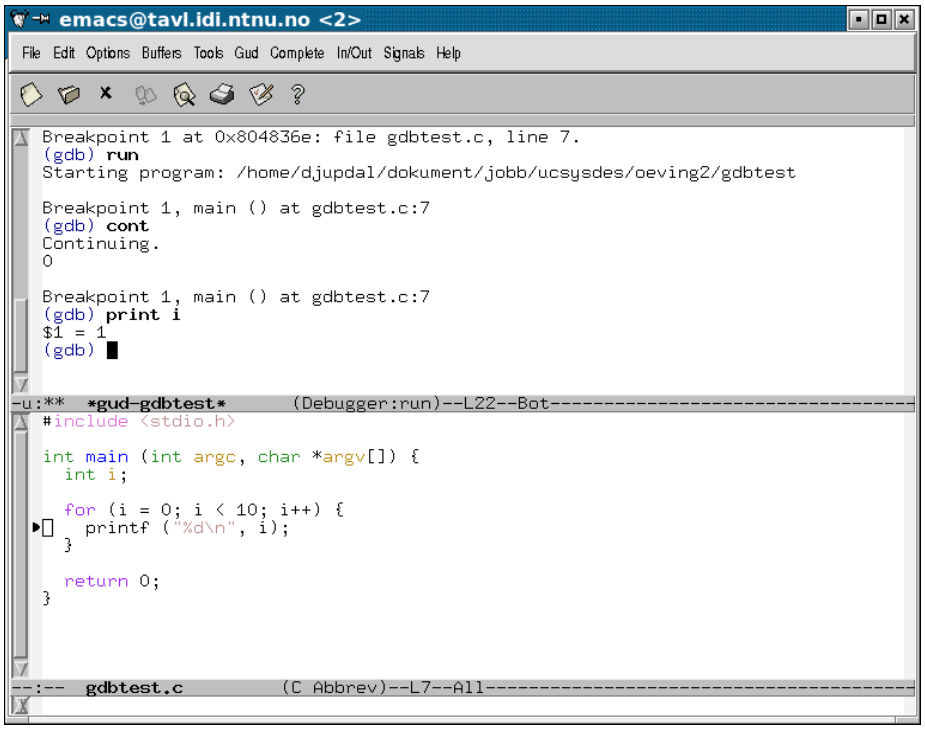


Figure 2.12: Debugging with GDB in Emacs

which presumes execution line by line from the original source code, so that the user can monitor what happens in the source code.

As we do cross–development, we must have a possibility to run debugger on a PC while the program we are debugging is being executed on the STK1000. To see to this, Atmel has made a program `avr32gdbproxy`. You can start it with:

```
avr32gdbproxy -f 0,8Mb -a remote:1024
```

Proxy works as a connection between JTAGICE and GDB. This enables GDB to drive AVR32 on STK1000 in detail.

GDB can be started as:

```
gdb <programfile>
```

As usual, we have our own variant for AVR32:

```
avr32-gdb <elf-programfile>
```

2.11.1 GDB in Emacs

In order to make debugging work efficiently, you need to combine GDB with a text editor. Often you wish to see which line in the source code is being executed, for example by single–stepping, and this presumes that GDB can communicate with the text editor. As no one can directly run GDB, the most run it as part of a bigger development environment.

We recommend that you run GDB through Emacs. Emacs is rather powerful and yet it does not hide away what happens on the bottom, for example, by

debugging with GDB. As this lab setup will introduce you to the tools for the work at the bottom, it suits the purpose well.

The main difference between running GDB in Emacs instead of running it directly from the command line is that you will have source files you are debugging in Emacs. The way in which GDB is used is the same in any case. This is contrary to one part of the GUI–tools which hide away what actually happens.

We can't provide a full description of Emacs at this place, thick books have been written about this editor. Here only the most important aspects for the use of GDB in Emacs will be mentioned.

Today a lot can be done with the mouse and menu and everything connected to file handling and simple text editing can be obtained through GUI, as in the majority of other text editors. Still, the biggest part of the Emacs functionality is available only from the keyboard. Emacs has many key combinations (key bindings). In Emacs–literature, the following notation is used to give a key combination: a key which should be pressed at the same time is written with a dash, '-', a key which should be pressed immediately afterwards is separated by a blank space. For example, `C-x k` stands for "hold Control pressed down while you press key x and then release all keys and afterwards press down key k". Small letters represent usual keys while capital letters represent special keys. "C" is a control key and "M" a left Alt–key (M stands for Meta, a key which is today replaced with Alt).

Those who would like to learn more about Emacs can open a tutorial by pressing `C-h t`. An Emacs manual [5] is opened as an `info`–page. Info–pages can be read also in Emacs, rather than using an `info`–command in shell. In Emacs you can open info–system by pressing `C-h i`.

To start debugging with GDB, first you need to open one of the source files of your program. Thereafter you start GDB with the following key combination `M-x gdb <RET>`. After you have given Emacs a notice about how GDB sould start (see the previous section), you get a GDB's command line. GDB–commands are described in the next section.

The last tip about Emacs: to get a "syntax highlighting" and certain part of other utilities, you need to configure Emacs so that it turns on these possibilities. We have made an Emacs start file for you. Copy ".emacs" from the delivered support files for the assignment 1 and paste it at the home folder (~/.emacs).

2.11.2 GDB–commands

GDB is a large program and it can not be fully documented at this place. GDB manual [6] can be found as an `info`–page. In addition, there is an on–line help system and a tutorial with FAQs on the web [11].

Here is the list of the most important commands but you are encouraged to read the manual and turn to the GDB help system to find out more about how these can be used.

- `target remote:1024`
  Connect to JTAGICE proxy. This must be done before you can do anything with AVR32 and STK1000.

- `set $reg = value`
  Set the register. A concrete example which sets the program counter to

the beginning of the program: `set $pc = _start`. As you can see, you can use symbols in your program freely in GDB.

- `bt`
  Print out stack trace.

- `regs`
  Show register file.

- `x/nx adresse`
  Show the memory contents where `n` is a number which specifies how many words will be shown

- `help`
  Online help. It is good, use it.

- `run`
  Run the program. Stop the program with Emacs key–combination `C-c C-c`

- `cont`
  Continue with running after stop.

- `si`
  Run a single instruction.

- `s`
  Run a single line of C–code.

- `print <expression>`
  Evaluate expression (for example, show the contents of a variable). Example: `print varA` will print out the value of the variable "varA".

- `display <expression>`
  The same as `print` but it will print out the value of the expression every time the program execution is stopped (for example, after each `si`).

- `break <place>`
  Set a breakpoint (the line in the source code at which the program will automatically stop). It can be run more easily with the help of Emacs: Go to the source file in Emacs where you would like to place a break point and press `C-x <SPACE>` to set a breakpoint on the line where the cursor is.

- `watch <expression>`
  Set a watch point. It will stop the execution when the expression (for example, a variable or a register) changes the value.

- `info break`
  Show all breakpoints.

- `delete <nr>`
  Remove break– or watchpoint.

- `quit`
  Exit GDB.

Mark that when using avr32-gdb, the command `run` is never used to start the program execution. This is because the program is already running on the STK1000 when you connect to avr32gdbproxy. The same effect can be achieved with the following commands:

```
set $pc = _start
cont
```

2.12 Description of the assignment

Write an assembly program which enables a user to move a "paddle" to the left or to the right on the row of LED–diodes by pressing buttons on the STK1000 board. The paddle can be represented as a single lighted spot (one LED–diode) and by pressing button 0 the lighted spot moves to the right while by pressing button 2, it moves to the left.

It is required that you write an interrupt routine (an interrupt handler) for reading the buttons, while the LED–diodes must be updated in the main loop of the program. It is also a requirement that you use a Makefile for the assignment and you can debug the program with GDB through JTAGICE. Remember that if you use a delivered Makefile instead of making your own, you need to look into it to see how it works.

2.12.1 Setup of the STK1000

The following jumpers must be set on the STK1000:

- SW1: Set to SPI0
- SW2: Set to PS2A/MMCI/USART1
- SW3: Set to SSC0/PWM[0,1]/GCLK
- SW4: Set to GPIO
- SW5: Set to LCDC
- SW6: Set to GPIO
- JP4: Set to "INT. DAC"
- JP5: Set to "INT. DAC"

In addition:

- Connect JTAGICE to STK1000 and PC
- Connect a flat cable between GPIO–connector and the connector for buttons.
- Connect a flat cable between GPIO–connector and the connector for LED–diodes.

2.12.2 Recommended Approach

1. Download support files for assignment1. They can be found in a .tgz–file named "oeving1.tgz" which can be unpacked with the following command line:

   `tar zxvf oeving1.tgz`

   You will find the following files:

   - `oeving1/Makefile`: A Makefile for assignment 1
   - `oeving1/oeving1.s`: An assembly file which can serve as a starting point for your assembly code
   - `oeving1/io.s`: An assembly file with useful constants
   - `oeving1/.emacs`: Start up file for Emacs. Copy this to the home folder, useful when you use Emacs.

2. Familiarise yourself with the tools by:

   - Use assembler and linker to compile and link delivered files by running a `make`–command
   - Upload the program to STK1000 by running `make upload`
   - Try out GDB by single stepping instructions, inspecting register file etc.

3. Write first a variant of the program without using interrupt.

4. After that, develop the program with the use of an interrupt routine.

2.12.3 Tips

- Do points 3 and 4 in stages (with GDB accompanying your work):
  - Begin with setting up the LED–diodes and make sure that you can turn them on and off.
  - Write the value on the LED–diodes.
  - Put the code for reading the button status to a subroutine.
  - Make a main loop which moves the "paddle" based on which buttons are pressed down.
  - Put the code for reading the button status into an interrupt routine and set up the interrupt.

Chapter 3

Assignment 2

3.1 Introduction

In this assignment you will make sound effects which can be used for the next assignment. You will make different sound effects will be played when different buttons are pressed. The code will be written in programming language C.

3.1.1 Learning outcome

The learning outcome in this assignment is:

- Programming in C
- I/O–control for AVR32 in C
- Use of the microcontroller's ABDAC for sound generation
- Interrupt handling in C for AVR32

3.2 C-programming

C–programming is probably something new for the majority of you who take up the course. A good news is that Java has borrowed a lot of its syntax from C so that there is not so much new to learn from scratch. The whole C language can not be described at this place so we just point to some of the differences with Java. We recommend that you buy and read the book "The C Programming Language" [10]. It will be useful, not only in this course.

3.2.1 Java and C: Similarities and differences

As said, Java and C have very similar syntax. The ways in which, for example, for–loops are written are the same. Here is the list of particularities of C which a Java programmer has to learn. You should look at this as a list of what you should look up in and read about in a proper book about C because our explanations are too brief to provide a good basis for learning C without additional help.

Compiling

C–code is compiled to object files, as assemly code was compiled to object files by assembler in the previous assignment (see section 3.2.3). Compiled source files need to be linked together into an executable file in the same way as in the assignment 1.

Object orientedness

C is not object oriented. Therefore, there are no methods which can be connected to an object, instead there is just a variety of functions which are no class members of any class. It is often wise to think in an object oriented way when programming, for example by relating a source file to an object in Java, but there is nothing in the language which is directly object oriented. C++ is an object oriented variant of C–language.

Structs

Instead of objects, C has so–called structs. **Structs** stands for a collection of variables and it can be viewed as an object whose all variables are "public" and which has no methods.

An example of the use of structs:

```
struct teststruct {  /* struct declaration */
  int a;
  int b;
};

int main (int argc, char *argv[]) {
  struct teststruct t; /* declares variable t to be of a type teststruct */
  t.a = 5;             /* assignment of variable i in struct */
  int c = t.b;         /* reading of a variable from struct */
}
```

Prototyping

In C, all the functions should have a so–called "prototype". It is a declaration of the function which tells what type of arguments the function takes and what type of value it returns, but without revealing how the function is implemented. All the code which will use a function must include a header file with the function's prototype. With that provided, a function can be implemented in some other source file.

Example:

```
int test (void);  /* prototype of the function test */

int test (void) { /* implementation of the function test */
  /* code */
}
```

void

A function in C which does not take any arguments must explicitly declare that. Here is a prototype of one such function:

```
void test (void);
```

This function takes no arguments and returns no value.

main()

The function which is called when the program starts is called main() and its prototype should be like this:

```
int main (int argc, char *argv[]);
```

Modifiers

As in Java, C has a number of key words which tell something about a variable and which are called "modifiers". The following modifiers are important:

- `const`: It will be impossible for the program code to change the value of the variable.
- `static`: If used for a local variable in a function: the variable will keep its value between different function calls. If used for a global variable: the variable will not be visible for the linker, it will be visible only for the source file in which it is defined.
- `extern`: The variable is a global variable which is defined in some other source file (and therefore it will placed in some other object file). The linker will account for making it possible to use such variable anyway.
- `volatile`: The variable will be kept in RAM. If a variable is not declared "volatile", compiler will be able to perform optimisation by removing it if it finds it be for the better. Variables which represent I/O–locations and variables which are shared between, for example, an interrupt routine and other functions must, therefore, be set to "volatile" in order to make sure that they are not removed as a result of optimisation.

Pointers

A pointer is a variable which contains a memory address. It often refers to another variable and, therefore, it has a type which says what type of variable it points to. This makes possible type checking also for the code which uses the data to which the pointer points even if the pointer itself always contains a memory address.

Example:

```
int a;    /* declare variable a */
int *b;   /* declare a pointer to an int */

b = &a;   /* set pointer b to point to a variable a */
          /* Actually: &a means "address of a" */

*b = 5;   /* set the value of the variable to which b points to 5 */
          /* Actually: *b = 5 means "write 5 to the memory address to which b points */

          /* a will now have the value of 5 */
```

In case the pointer points to a struct, somewhat different syntax is used for struct (mark the notation `->`):

```
struct teststruct t;  /* declare a teststruct--variable */
struct teststruct *p; /* declare a pointer to a teststruct--variable */

p = &t;   /* set p to point to t */
p->a = 5; /* the same as t.a = 5 */
```

There are also general pointers which don't have any types associated to themselves. These are declared as void–pointers:

```
int a;          /* declare variable a */
void *b;        /* declare void--pointer */

b = &a;         /* set pointer b to point to variable a */

*(int*)b = 5; /* set the value of the variable to which b points to 5 */
              /* when void--pointers are used, explicit cast must be invoked */
              /* as it is here (cast to an int--pointer) */

              /* a will now have value 5 */
```

Macros

C has so–called macros. Typically, they are used for constant definitions in a header file.

For example:

```
#define RETURN_CODE 13  /* RETURN_CODE can now be used instead of 13
                           later in the code */
```

Header files

In C there is a difference between two types of source files: C–files and header files. All program code should be placed in a C–file (file suffix ".c"). All definitions (constants, structs, prototypes) should be in a header file (file suffix ".h"). Header file is included like this:

```
#include <stdio.h>  /* include a system header file */
#include "test.h"   /* include a header file which is local for the project */
```

stdlib

C has a standard library with a set of functions which are called from all programs. Contrary to Java API, this is a rather little and limited library. The reasoning behind is that a C–programmer should turn directly to an operating system when more functionality is needed than that which is found in the standard library. Therefore, C–programs are typically specific with respect to an operating system and not so easy to port to another operating system.

In order to be able to use a function from the library, a corresponding system header file has to be included in the source code. This is because the compiler needs to know that the code for the actual function exists at some other place so that the function can be used.

Tips: All C–functions have a Unix man–page, so when you would like to learn about some specific C–function, you can open its documentation by writing `man 3 <function_name>`.

3.2.2 Code organisation and conventions

It is a common practice to organise functions and global variables in different C–files and compile them each for itself. Organise the files so that the functions which belong together are placed in the same source file, in about the same way you would organise the code into different classes in Java.

The functions and variables which need to be global because they will be used in a different C–file from the one where they are declared must be declared in a header file which is included by all C–files which need to use them. In other words: global functions must have their prototypes in a header file which is included in all C–files which use these functions. Global variables must be declared as "extern" in a header file which is included in all source files where they are used.

A typical program can then have a certain number of C–files and one header file which includes all C–files. In this header file, all function prototypes are gathered, side by side with "#includes", "#defines" and "externs". But, remember: do not write function code or variable assignment in a header file, that will only cause problems and it is also considered as a bad programming style.

3.2.3 GNU Compiler Collection (GCC)

You will use a GNU C–compiler: GCC. Use info–command to read the documentation for GCC. Here is a brief description of how GCC can be used for compiling a C–file but the compiler certainly has many more options which we can not go into at this place. Read the documentation.

```
gcc -Wall -g -c -o <outputfile.o> <inputfile.c>
```

As you will do cross–compiling for AVR32, you have to use the following command:

```
avr32-gcc -Wall -g -c -o <outputfile.o> <inputfile.c>
```

The arguments for this option have the following meaning:

- `-Wall`: Turn on all the warnings, this will generate warnings for those things which are allowed in a C–code but considered a bad programming style
- `-g`: Enable the use of GDB for debugging (all symbols are included in an executable file)
- `-c`: Make an object file
- `-o <outputfile.o>`: Write the results into a file outputfile.o
- `<inputfile.c>`: C–file which will be compiled.

Linking

In the assignment 1 you have already learned about linking of object files which were generated by assembler. The same linker is used for linking object files generated by C–compiler but for C–programs there is a bit of extra object files and libraries which have to be linked in. This is, among the others, a start up code which sets a stack pointer and which calls the main() function in your program. In addition, a standard C–library should always be linked in.

To skip the necessity to specify all this, you can again use a `gcc` command for linking. If `gcc` is called as if it were a `ld` (without providing C–files in the list of arguments but instead giving one or more object files), it will run `ld` for your object files but with all correct arguments for linking C–programs. In other words: you can perform linking as in the assignment 1 but do it with `avr32-gcc` instead of `avr32-ld` directly.

If you would like to find out how `gcc` calls `ld` (which arguments it gives), you can provide `gcc` in the linking stage with the arguments `-v -Wl,-v`. Then, the full `ld` command which is being run will be written in the console.

3.2.4 I/O–control from C–code

C–programs can also work directly on hardware and control I/O–controllers as in assembly. As AVR32 has a memory mapped I/O, this is done with the use of pointers. Atmel has given out header files which include macros in which the addresses for different I/O–controllers are defined.

To get an access to these macros, you have to include a system file "avr32/ap7000.h". Here is an example:

```
#include <avr32/ap7000.h>

int main (void) {
  /* each I/O-controller is represented as a struct */
  /* set a struct pointer "pio" as a pointer to the address area of PIO C */
  volatile avr32_pio_t *pio = &AVR32_PIOC;

  pio->per = 0xff; /* set PIO C PER-register to 0xff */

  return 0;
}
```

In the same way, there are structs for all I/O–controllers in a microcontroller. Look into a header file `ap7000.h` for more information about different structs, their names and how they get access to individual registers in I/O–controller.

In this lab setup, you will need to handle the registers of PIO B, PIO C, Power Manager and ABDAC. Further, it is shown how you can assign variables with wanted structs for handling these I/O–controllers:

```
volatile avr32_dac_t *dac = &AVR32_DAC;
volatile avr32_pio_t *piob = &AVR32_PIOB;
volatile avr32_pio_t *pioc = &AVR32_PIOB;
volatile avr32_sm_t *sm = &AVR32_SM;
```

Each of these structs has members which correspond to the registers of the I/O–controller. For example, piob->per means PER–register of PIO port B.

3.2.5 Handling interrupt from C-code

There is no standard way in which an interrupt is set in C. You use GCC and AVR32 and therefore you must learn the method for handling interrupts specified by Atmel for the use in GCC.

To use an interrupt, you must include the following header file:

```
#include <sys/interrupts.h>
```

You have to write an interrupt routine with the following prototype:

```
__int_handler *int_handler (void);
```

Mark that the name of the function can be whatever you choose, it is only important that it returns type `__int_handler` and that it has no arguments.

To initialise interrupt, you have to call the following functions (in addition to initialisation of I/O–controller which will generate interrupt):

- **void** set_interrupts_base (**void** *base): This notifies the system about where in the address space the interrupt controller is placed. In your case, its argument will always be `(void *)AVR32_INTC_ADDRESS`.

- ```
  __int_handler register_interrupt(__int_handler handler,
                                   int_grp, int line, int priority)
  ```

  This registers the function as an interrupt routine so that it is called whenever interrupt occurs. The arguments are:

  - `handler`: Name of your interrupt routine
  - `int_grp`: Interrupt group (see section 2.5.1)
  - `line`: Number of interrupt line within interrupt group
  - `priority`: Interrupt level which will be used

- **void** init_interrupts (**void**): This function initialises interrupt.

Example

Here is an example. Mark that in this example, interrupt group and interrupt number are calculated from the macro `AVR32_DAC_IRQ`. All I/O–controllers have similar macros defined and the group number can be found by dividing with 32 and interrupt number within the group can be found by taking a modulo 32 division (% 32).

```
__int_handler *int_handler (void) {
  /* interrupt code ... */

  return 0;
}


int main (int argv, char *argv[]) {

  /* code which sets up I/O-controller for interrupt ... */

  set_interrupts_base ((void *)AVR32_INTC_ADDRESS);

  register_interrupt ((__int_handler)(int_handler),
                      AVR32_DAC_IRQ/32, AVR32_DAC_IRQ % 32,
                      INT0);
  init_interrupts();
}
```
```

3.3 Sound generator: Audio Bitstream Digital to Analog Converter (ABDAC)

For the sound generation, you will use an internal ABDAC in the microcontroller. An ABDAC is a unit which takes a sequence of digital "samples" (a sample is a numeric value which describes the amplitude of the sound wave at a given point in time) and converts the sequence to an analog signal which can be amplified and further sent to headphones/loudspeakers.

In order to use internal ABDAC, jumpers 4 and 5 must be set to "INT. DAC". Then the sound from the ABDAC will be sent out to the audio output where you connect your headphones.

ABDAC is rather easy to use, there are just three registers you need to know: SDR, CR and IER. These registers can be accessed in C in the same way as described in section 3.2.4. Read about the ABDAC registers in the microcontroller datasheet [1] in chapter 26.

You need to do the following to make an ABDAC generate sound:

- Set up the clock signal which will be used to clock in samples for ABDAC (determines the speed of the sound sequence).
- Set up PIO–controller to allow ABDAC to send the sound to the microcontroller output.
- Activate ABDAC by writing 1 to the "en"–bit in the CR–register.
- Activate interrupt by writing 1 to the "tx_ready"–bit in the IER–register.
- In interrupt routine: Write the next sample of the sound data to the SDR–register (a new sample every time the interrupt routine is called).

What makes the use of the ABDAC somewhat more complicated is the fact that it is dependent on two other controllers which must be set up correctly. ABDAC uses some of the same physical pins as PIO port B so PIO–controller must be set up not to use these pins but give control to ABDAC. Further, a controller called "Power Manager" must be set up to provide a clock signal for the ABDAC. ABDAC needs its own clock signal because it can then play the sounds with different sample rates.

The clock signal for the ABDAC can come from various available sources: PLLs, OSC0 and OSC1 [1]. It is this signal which causes the ABDAC to generate interrupt. Mark that the frequency of OSC0 and OSC1 is 20MHz and 12MHz respectively.

3.3.1 Setting up PIO for the use by ABDAC

Physical pins on the microcontroller can be controlled by up to three different units. PIO–controller decides which unit will control them. The three possibilities are:

- PIO itself, for example for the use of LED–diodes and buttons
- Peripheral A
- Peripheral B

Peripherals A and B are I/O–controllers integrated in the microcontroller. What is peripheral A and what is peripheral B depends on which pins will be used. A list over what different I/O–controllers are connected to (which pins and if they are peripheral A or B) can be found in the microcontroller datasheet [1] in chapter 9.7. For example, ABDAC is connected as peripheral A to PIO port B pins 20 and 21. Therefore, PIO must be set up accordingly so that ABDAC will function:

1. Bits 20 and 21 in PIO B PDR–register must be set to 1 in order to avoid that PIO drives these outputs itself.
2. Bits 20 and 21 in PIO B ASR–register must be set to 1 in order to choose that peripheral A (therefore the ABDAC in our case) controls these pins.

In C this can be written as:

```
volatile avr32_pio_t *piob &AVR32_PIOB;

piob->PDR.p20 = 1;
piob->PDR.p21 = 1;

piob->ASR.p20 = 1;
piob->ASR.p21 = 1;
```

3.3.2 Setting up Power Manager to be used by ABDAC

Power Manager provides power and clock signals in the microcontroller [1], see Chapter 10. You have to set up Power Manager so that it provides the clock signal for ABDAC. There are different clock signals and ABDAC is connected to clock number 6.

Power Manager has a PM_GCTRL–register for each clock and this register turns the clock on and sets the clock speed. Therefore, you must set up the sixth PM_GCTRL–register in order to get the ABDAC going. In C, the sixth PM_GCTRL–register can be accessed in this way:

```
volatile avr32_sm_t *sm = &AVR32_SM;

sm->pm_gcctrl[6] = data; /* set PM_GCTRL to "data" */
```

3.3.3 Setting up the ABDAC

The following must be done in order to set up the ABDAC itself so that it can take sound data:

- Activate ABDAC by writing 1 to “en”-bit in the CR–register.

  ```
  volatile avr32_dac_t *dac = &AVR32_DAC;

  dac -> CR.en = 1;
  ```

- Activate interrupt by writing 1 to “tx_ready”-bit in the IER–register.

  ```
  dac -> IER.tx_ready = 1;
  ```

- In interrupt routine: write the next sample of the sound data to the SDR–register (a new sample each time an interrupt routine is called)

In addition, you have to set up the interrupt system which is described in section 3.2.5.
Mark that the interrupt controller must be programmed before the ABDAC. Read about it in chapter 26.5.4 in [1]. Therefore, we recommend that you set up the interrupt registers in the beginning of the main method.

Synthesiser

Physical basis for sounds lies in the longitudinal waves which are created by oscillations of pressure in some media. Therefore, the properties of the sound are the properties of the wave: frequency, period, amplitude, for example. Average hearing limits for the sound are 20HZ lower and 20 kHz upper but it varies from one person to another.

So, how to generate a sound? There are more possibilities but our suggestion is to make a synthesiser in software. This means that you will generate sound waves 100% artificially. The sound is nothing else but the repetition of oscillating waves of certain frequency and amplitude. The frequency defines the tone of the sound and the amplitude defines the strength of the sound. See Figure 3.1.

To make a synthesiser, you need to decide upon the waveform you would like your synthesiser to generate. Three periods of some of the common waveforms are shown in Figure 3.2. You need to repeat this period many times, as long as you would like the tone to last. Each period must be divided in a certain number of samples. The number of samples in the period defines the length of the wave and, therefore, the frequency and tone. Each sample must be written to the SDR–register of the ABDAC and you will write one sample after another each time an interrupt routine is called. In the C–code this can be written as following:

```
volatile avr32_dac_t *dac = &AVR32_DAC;

dac->SDR.channel0 = (short)channelData;
dac->SDR.channel1 = (short)channelData;
```

3.4 Description of the assignment

Write a C–program which runs directly on the STK1000 board (without support of an operating system) and which plays different sound effects when different buttons are pressed. Each generated sound effect will be a sound effect you may use in the final game. You have to make at least three different sound effects (for example, cannon shot, target hit, player win etc.). Make a start up melody which can be played when the game begins.

We recommend that you use the microcontroller's internal ABDAC. Those who are tempted to learn more may read about the external ABDAC and use it in order to achieve better quality of the generated sound but this is not a requirement in this assignment.

The requirement is that you use an interrupt routine to pass the samples to the ABDAC.

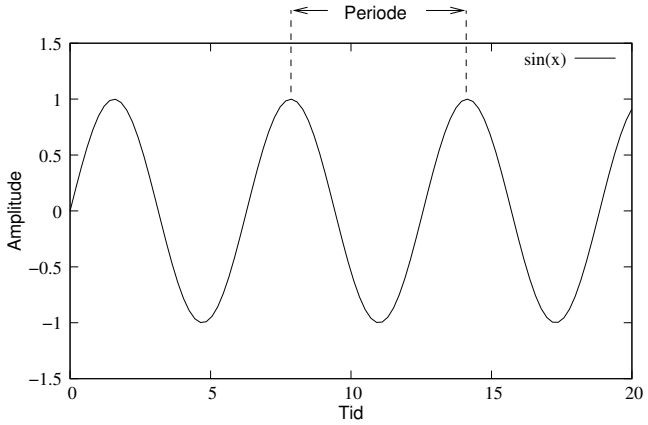


Figure 3.1: Sound wave

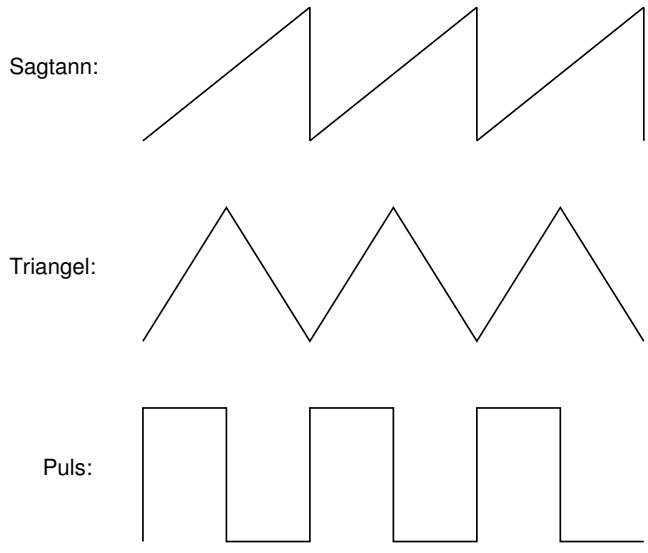


Figure 3.2: Various sound waves

3.4.1 Setup of the STK1000

The following jumpers must be set on the STK1000:

- SW1: Set to SPI0
- SW2: Set to PS2A/MMCI/USART1
- SW3: Set to SSC0/PWM[0,1]/GCLK
- SW4: Set to GPIO
- SW5: Set to LCDC
- SW6: Set to GPIO
- JP4: Set to "INT. DAC"
- JP5: Set to "INT. DAC"

In addition, you need to:

- Connect JTAGICE to STK1000 and PC.
- Connect headphones to the Audio contact.
- Connect GPIO–connector and buttons connector by a flat cable.
- If you intend to use LED–diodes in the assignment, they also need to be connected to GPIO.

3.4.2 Recommended Approach

1. Download the support files for the assignment 2 ("oeving2.tgz") and unpack them in the same way as in the assignment 1. The following files can be found:
   - `oeving2/oeving2.c`: An example of a C–file
   - `oeving2/oeving2.h`: An example of a header file
2. Write a Makefile which compiles and links the given C–file. You may start from the Makefile given for the assignment 1.
3. Make sure that you can use the tools correctly:
   - Test whether the Makefile works and everything compiles and behaves as it should.
   - Upload the executable file to the STK1000.
   - Try debugging in GDB by single–stepping C–lines, inspecting the variables etc.
4. Write the solution for assignement 1 in C-code
5. Begin programming for the assignment.

3.4.3 Tips

- Don't forget to test and debug while you work on point 5.
- Before you make any advanced sounds or melodies, you can send some random data (rand()–function in C) to the ABDAC to test if it works and is set up correctly. Then, you will hear some noise in the headphones and make sure that it works.
- Think of the code reuse. Organise the hardware–dependent code into separate functions so that your code for sound generation can be easily taken out and reused at a later stage. You will use it in the assignment 3 but within a Linux driver instead of directly for hardware. You may see the description of the sound driver in the assignment 3 so that you can learn more about how the code can be written in order to work for both assignment 1 and assignment 2.

Chapter 4

Assignment 3

In the last assignment in the course, you will make a computer game. This time you will not program hardware directly but have a Linux kernel as a support so that all the code related to hardware entities will be written in a form of device drivers for Linux kernel.

4.1 Introduction

In this assignment you will make a computer game The Scorched Land Defense. The name of the game comes from a warfare term - **scorched earth** which stands for:

In warfare, the policy of burning and destroying everything that might be of use to an invading army, especially the crops in the fields. It was used to great effect in Russia in 1812 against the invasion of the French emperor Napoleon and again during World War II to hinder the advance of German forces in 1941.

(from http://encyclopedia.farlex.com/Scorched+Land)

All your programme code will be written in C (you are encouraged to reuse the code from exercise 2) and you will use Linux as an operating system on the STK1000 board.

There are two parts of the assignment. First, you will make a Linux driver for the buttons and LED diodes on the STK1000 board. This will involve Linux-hacking at a low level with, among other things, compilation of the Linux kernel. In the second part, you will complete the game, but in such way that all I/O goes through Linux device drivers, including your own.

4.1.1 Learning outcome

The learning outcome of this assignment is:

- C-programming for Linux
- Programming of Linux device drivers:
  - Use of Linux shell
  - Compilation of the Linux kernel
  - How to make kernel modules in Linux

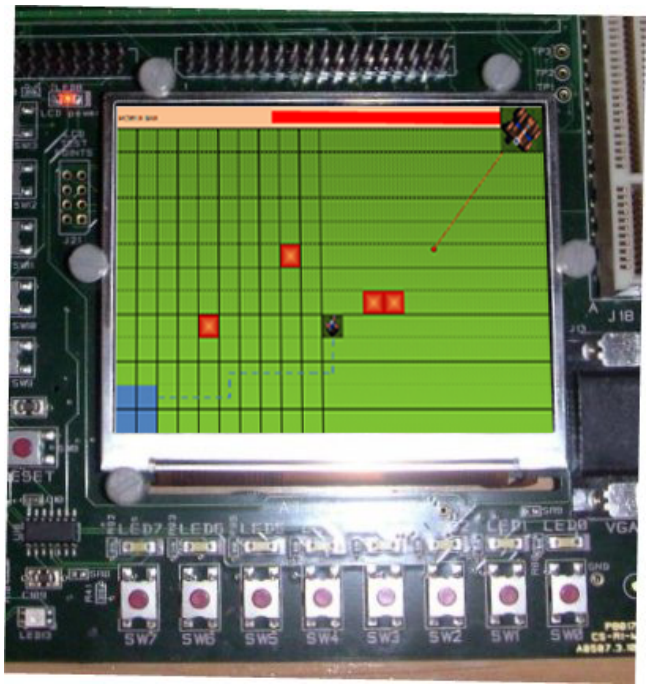


Figure 4.1: Assignment 3: the game

– Programming hardware in Linux
– How to make your own device drivers in Linux

4.2 GNU/Linux

STK1000 board does not necessarily have to be programmed directly like you did in assignments 1 and 2. It is also possible to run an operating system on the STK1000 board. With the help from Atmel, Linux has support for AVR32 and STK1000. Linux-variant which works for STK1000 is a full Linux 2.6 kernel and even if support for AVR32 comes as a patch for Linux kernel, it will be included in Linux from the version 2.6.19.

In order to run the Linux kernel on STK1000 board, its flash memory must be programmed with the bootloader which loads Linux from the external media. This can be done over net or from SD-card which is plugged in STK1000. Bootloader is named `u-boot` and it is provided as part of the supporting files for this exercise. Use `avr32program` to program STK1000 with the bootloader in the same way as you have programmed STK1000 with your own programs in assignments 1 and 2.

You should load Linux system from the SD-card. Every STK1000 has an SD-card reader and accompanying SD-card with already installed Linux system. Therefore, an SD-card functions as a hard disk for STK1000 and it contains all files which are needed for a working Linux system. So, you will copy your program for the assignment 3 on SD-card. SD-cards have also a USB plug so that you can take an SD-card out from STK1000 and plug it in your PC and

read/write files on it. You can also reach your files over net as STK1000 has a network cable plug–in. For this purpose you can use program `wget` on STK1000.

To make it possible for Linux to boot and to have access over net, the following jumpers must be set on STK1000:

- SW1: Set to SPI0
- SW2: Set to PS2A/MMCI/USART1
- SW3: Set to SSC0/PWM[0,1]/GCLK
- SW5: Set to LCDC
- SW6: Set to MACB0/DMAC

To get access to shell (command line) on STK1000, you need a terminal connected to STK1000. Connect STK1000 to PC with RS232 cable (use port "UART_A" on STK1000) and use some terminal program to communicate with STK1000. This can be, for example, "minicom" which is installed on the lab machines. Start minicom from the command line like this:

```
minicom
```

We have set up minicom so that it should work with STK1000. If you want to use some other terminal, you need to set it up with the following communication parameters:

- 115200 bps
- 8 databits
- 1 stop bit
- no parity (8N1).

When Linux boots, you will get a command line terminal and you will have a full access to Linux running on STK1000. You will notice that the number of programs following Atmel's Linux variant is limited, but when the full version of Linux kernel 2.6 runs, it will take only to compile the programs you need. This is certainly not a part of this assignment but only information for those of you who find it interesting.

You can find more information on Linux on STK1000 in the STK1000 manual. [3].

4.2.1 Compiling Linux Kernel

To be able to develop your own Linux drivers, you need to compile your own version of Linux kernel. Source code for Linux kernel is delivered as a support file. Unpack it and go to the newly unpacked directory. The kernel is configured so that it complies with your needs for assignment 3, but you are free to make your own version of the kernel configuration with the following command:

```
make xconfig
```

After running it, you will get a window with different kernel properties which can be adjusted.

To compile the kernel, run:

```
make
```

When the kernel is compiled, the newly compiled version can be found at the location (relative to the root of the source code):

`arch/avr32/boot/images/uImage`

Copy this file to the Linux system on STK1000 board and boot STK1000 anew. If everything went as it should, you would be running now a newly compiled kernel.

4.2.2 Kernel modules

Linux runs in two different modes, as the majority of modern operating systems: user mode and kernel mode. All usual programs are running in user mode and they have limited rights. Linux kernel runs in kernel mode and it has all the rights. Drivers need to run typically in kernel mode and therefore they need to be programmed as one part of the kernel. Fortunately, Linux kernel is built out of modules so that it is possible to make so called kernel modules which can be dynamically linked together with the kernel when the module is loaded.

So, one of the useful properties of Linux is that the modules can be dynamically loaded and removed while the kernel is running. Kernel modules have extension ".ko". They can be loaded by the use of the following command:

`insmod <kjernemodul.ko>`

To remove the module which is running as a part of the kernel:

`rmmod <kjernemodul>`

To get the list of all the loaded modules:

`lsmod`

Kernel module is actually a little program which runs in kernel mode. Therefore, kernel modules have to follow certain rules, they can't have the same behaviour as common programs. Here is the list of limitations:

- Firstly, the kernel module must implement a strictly defined interface (i.e. a set of functions) so that the kernel knows exactly how the module should be used
- Secondly, a kernel module cannot call other functions from those which are defined in the Linux kernel itself. That means that none of the functions from the C standard library can be used.
- Thirdly, kernel modules have to be programmed with parallelism in mind – one module must always function for itself even if more different processes are trying to use it at the same time.
- Finally, all kernel modules are event–based. They can't have loops which are running to eternity because that would cause the kernel to be 'hanging'. Instead, they have only functions which are called every now and then when other programmes need access to the module.

This compendium gives a brief introduction to how the kernel moduless should be made, but this topic is explained in detail in Chapter 2 of the book "Linux Device Drivers" [4] which can be found in Tapir or downloaded for free from the net. You have to read parts of this book if you want to complete assignment 3. The compendium will only point towards what you have to find out, not explain everything in detail.

Interface between kernel and modules

A kernel module must implement two functions which are called by the kernel:

- **static int** __init my_init (**void**): This is the function which is called when the module is loaded, you should do everything necessary for the setup here, for example allocate and initialise hardware.

- **static void** __exit my_exit (**void**): This function is called when the module stops being used. Here you will deallocate everything you have allocated in init-function.

You can name these functions as you wish but the kernel needs a piece of information about the module in order to be able to use it and, among the other things, it needs to know the names of the init- and exit functions. This is achieved by calling the following macros at one or another place in the module's source file (typically at the bottom of it):

```
module_init (my_init); /* specifies which function will be used as init */
module_exit (my_exit); /* the same, but for exit function */
MODULE_LICENSE ("GPL"); /* specifies the license which stands for the code */
```

Printing

Kernel modules cannot call other functions from those which are defined in the kernel. Therefore, you cannot use printf() as in common C programs. Rather, there is a corresponding function in the kernel. Here is an example of its use:

```
printk (KERN_INFO "Variable value %d\n", i)
```

It prints out the value of variable i. KERN_INFO means that this message is only an info message. Mark that there is no comma after KERN_INFO.

Compiling modules

Kernel modules have to be compiled by a make-system of the Linux kernel itself. That means that a Makefile for a kernel module is not a common Makefile. Here is an example:

```
MODULE = modulnavn

ifneq ($(KERNELRELEASE),)
obj-m := ${MODULE}.o

else

KERNELDIR := ../linux/
PWD := $(shell pwd)

default:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

endif

.PHONY : clean
clean:
	rm -rf *.o *.ko .*.cmd *.mod.* .tmp*
```

What you need to change here in order to make it functional for your module is variable "MODULE" which is set to the name of your module (the same as a C–file but without the extension). Besides, you need to specify where the source code of the Linux kernel is placed and that is specified by the variable "KERNELDIR". Those of you who would like to know more about how the Makefile functions, you can read more about it in "Linux Device Drivers" [4], Chapter 2.

4.2.3 Files

In Unix (and Linux) nearly everything is represented in a form of a file. This holds for drivers as well. Each driver has a corresponding file in the directory `/dev`. In order to use the driver, you need to open the driver file as if it was a common file and the access to the driver's functionality is provided by the usage of the common functions for the file reading and writing.

Common user programs can call the following Linux functions in order to get access to the files:

- open(): Open file
- close(): Close file
- read(): Read from file
- write(): Write to file
- lseek(): Search in file (change the position in the file)
- ioctl(): Give special command (typical for a driver)

These functions are documented as man pages. Check for yourself how they are used by reading man pages (for example `man 2 open`).

A driver must implement support for these operations if the user program should have the possibility to use the functions for driver access. This is described in section 4.2.4.

4.2.4 Device drivers

In Linux (and other Unix systems) there are two main types of drivers: block drivers and char drivers. These two have some different properties and different interfaces towards the kernel. In this assignment, we shall only work with char drivers.

A driver is identified by two numbers: a "major" number and a "minor" number. For example, the sound driver can have major number 14 and minor number 0. Drivers must be accessed as if they were files on a hard disk so there must exist a way to associate a file name with a driver. This is done by a `mknod` command. Go to the directory `/dev` and perform the following command to establish a driver file.

```
mknod <drivername> c <major> <minor>
```

- `<drivername>`: Name of the file which will be a point of contact for the driver

- `c`: The driver is a char driver
- `major`: The driver's major number
- `minor`: The driver's minor number

After the file is established, it can be read or written with the aim of accessing the driver, if the driver with the given major and minor number is loaded and at in place.

The following has to be done in order to make a driver:

1. Make the driver as a kernel module
2. Allocate a major and minor number for the driver
3. Allocate access to I/O registers in hardware which will be used
4. Initialise hardware
5. Implement a set of functions which perform file operations (open/close/read-/write) on the driver and register these in the system
6. Activate the driver

We shall briefly see how each of these points is performed, but you will get a better and more thorough explanation by reading Chapters 2, 3 and 9 in "Linux Device Drivers" [4].

Allocation of the major and minor number

Allocation of the device number is performed by the function call alloc_chrdev_region().

This is explained in the section "Allocating and Freeing Device Numbers" in Chapter 3 of the "Linux Device Drivers" [4].

Asking for access to I/O ports

The driver cannot just use the hardware without asking for the access first. This is performed by the function call request_region() and it is described in the section "I/O-Port Allocation" of the Chapter 9 in "Linux Device Drivers".

Registration of file functions and activation of the driver

To make it possible for the user program to handle the driver as a file, the driver has to implement support for access. This is done by the following four functions:

```
/* user program opens the driver */
static int my_open (struct inode *inode, struct file *filp);

/* user program closes the driver */
static int my_release (struct inode *inode, struct file *filp);

/* user program reads from the driver */
static ssize_t my_read (struct file *filp, char __user *buff,
            size_t count, loff_t *offp);

/* user program writes to the driver */
static ssize_t min_write (struct file *filp, const char __user *buff,
              size_t count, loff_t *offp);
```

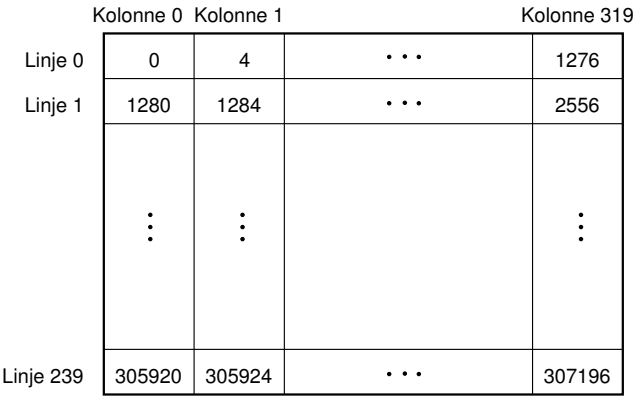


Figure 4.2: Organisation of framebuffer. Every square corresponds to one pixel and the address of this pixel (relative to the first pixel) is written in the square.

Byte:	0	1	2	3
Contents:	Red	Green	Blue	Not used

Figure 4.3: Organisation of each pixel in the framebuffer

To register these functions so that the kernel knows how they are invoked, the following structure must be set:

```
static struct file_operations my_fops = {
  .owner = THIS_MODULE,
  .read = my_read,
  .write = my_write,
  .open = my_open,
  .release = my_release
};
```

After all initialisation is completed, the driver is registered in the system by establishing the structure which represents the driver and registering the driver with the function cdev_add(). Read about how this is done in the Chapter 3 in "Linux Device Drivers" [4].

I/O-programming

AVR32 has memory mapped I/O. Therefore, special functions are not needed for writing or reading I/O registers. If you have got access to I/O registers you would like to use, you can perform I/O operations exactly as you did in assignment 2. The only difference is that Atmel's header files are not accessible for Linux. We have delivered a header file together with the support files for assignment 3 which can be used in the same way as header files from Atmel in assignment 2.

4.2.5 Framebuffer device

To get access to LCD on STK1000 you have to program for the so–called "Framebuffer device". This is a device (`/dev/fb0`) which represents LCD's graphic memory. You can use it by opening the driver `/dev/fb0`. It is possible to ask

the driver about the type of display it is connected to (for example, the size of the screen), but you don't have to do that. You can just assume the following for the screen:

- Screen size: 320x240
- Number of bits per pixel: 32 (8 for each colour + 8 not used).

The screen is organised as shown in Figure 4.2. Each pixel in the framebuffer is organised as shown in the Figure 4.3. To write to one pixel on the screen, you can search the right byte by using the function lseek() followed by the function write() to write the right value.

Easier way to do this is to map the driver to an array in the memory. Then you will be able to write pixels by writing directly to a usual C array which can be done by the function mmap(). This is something we strongly recommend, check the man page: `man 2 mmap`.

4.2.6 Sound

Use of the sound driver in Linux is very simple. You need only to open the file `/dev/dsp` and write the sound data to this file.

The sound driver is set up like this:

- 8 bits pr. sample (not 16 as in assignment 2)
- only one channel
- sample rate: 8000Hz

To change the sample rate you can use IOCTL command. This command can be given to the driver with the help of ioctl() function and for the sample rate of the sound driver it is done in the following way:

```
#include <linux/soundcard.h>

int dsp_rate = 44100;
ioctl (fd_dsp, SOUND_PCM_WRITE_RATE, &dsp_rate);
```

You can also use IOCTL command to set other parameters, as well as to read out how the driver is set up at the moment.

4.2.7 Compiling for AVR32-linux

Compiling and linking is done in exactly the same way as in assignment 2. The only difference is that all development tools will now be named `avr32-linux-<program>`, (for example `avr32-linux-gcc`) instead of `avr32-<program>`.

4.3 Description of the assignment

The assignment consists of two parts:

4.3.1 Part 1: The driver

Make a driver for the use of buttons and LEDs on the STK1000. It should be implemented as a kernel module. You are free to make a driver as you wish, but the minimum requirements are to support your needs for the game to work.

4.3.2 Part 2: The Scorched Land Defense

Complete the game. Use /dev/fb0 directly for writing to LCD screen. Use /dev/dsp for producing the sound. Use your own driver for reading the status of the buttons on STK1000. Use also your own driver for the control of the LED diodes. These can be used, for example, to show how many lives a player has left or some other status information about the game. Or you can just blink in some nice repetitive way.

4.3.3 Necessary setup of STK1000

The following jumpers are to be set on STK1000:

- SW1: Set to SPI0
- SW2: Set t0 PS2A/MMCI/USART1
- SW3: Set to SSC0/PWM[0,1]/GCLK
- SW4: Set to GPIO
- SW5: Set to LCDC
- SW6: Set to MACB0/DMAC
- JP4: Set to ”EXT. DAC”
- JP5: Set to ”EXT. DAC”

Beside that, also:

- JTAGICE should be connected to STK1000 and PC
- STK1000 (UART_A) and PC should be connected by an RS232-cable
- GPIO connector and buttons connector should be connected by a flat cable
- If LED diodes are used in the assignment, they also need to be connected to GPIO.

4.3.4 Recommended steps

- Download the files for assignment 3 (”oeving3.tgz”) and unpack them:
  - `oeving3/driver/Makefile`: Makefile for your kernel module
  - `oeving3/driver/driver.c`: A C file which contains empty versions of the functions you have to implement in your kernel module so that the interface towards the Linux kernel is in accordance with the specification.

  - `oeving3/driver/ap7000.h`: A header file which corresponds to avr32/ap7000.h in assignment 2.

- As for all other assignments, it is recommended that you do the work in stages. For the kernel module, a convenient procedure could be:
  - Make a simple ”hello world” module
  - Check that module compiles and works
  - Program it as needed for a simple char-device
  - Set up the device by making a suitable node in /dev
  - Test whether the device works by writing and reading it
  - Implement the support for the control of buttons and LED diodes
- For the game, the following procedure is convenient:
  - Open your own driver and make sure that you can read the buttons and handle the LED diodes
  - Open /dev/fb0 and /dev/dsp and make sure you can use them
  - Implement the game.

Appendix A

The list of versions

Here is the overview of the revisions of this document.

- 2007-01-15: The first version for students
- 2007-02-01: A bit more about GDB (2.11.2). A bit more about interrupts in assembly (section 2.5.1 and 2.5.2). Section on stack (2.9).
- 2007-12-21: New version for 2008
- 2010-01-11: New version for 2010
- 2011-07-28: New version for 2012, in English for the first time
- 2011-12-09: New version for 2012, exercises updated
- 2013-01-10: New version for 2013

Bibliography

[1] Atmel. *AT32AP7000 Datasheet*, 2006. `http://www.atmel.com/dyn/resources/prod_documents/doc32003.pdf`.

[2] Atmel. *AVR32 Architecture Guide*, 2006. `http://www.atmel.com/dyn/resources/prod_documents/doc32000.pdf`.

[3] Atmel. *STK1000 BSP Manual*, 2006. `http://www.atmel.com/dyn/products/tools_card.asp?tool_id=3918`.

[4] Jonathan Corbet, Allesandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 3rd edition, 2005. `http://lwn.net/Kernel/LDD3/`.

[5] Free Software Foundation. *Emacs Manual*, 2006. `infonode: emacs`.

[6] Free Software Foundation. *GDB Manual*, 2006. `infonode: gdb`.

[7] Free Software Foundation. *GNU As Manual*, 2006. `infonode: as`.

[8] Free Software Foundation. *GNU Ld Manual*, 2006. `infonode: ld`.

[9] Free Software Foundation. *GNU Make Manual*, 2006. `infonode: make`.

[10] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Software Series, 2nd edition, 1988.

[11] Richard Stallmann. GDB debugger tutorial. `http://www.unknownroad.com/rtfm/gdbtut/`, 2006.

Index