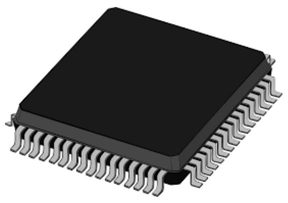
Digital Systems

Lab 1



EITF12

Introduction to Microcontrollers

Goals

- Get familiar with microcontroller development basics
- Be able to write simple programs interfacing with the world
- Be able to debug simple programs with Atmel Studio

Christoffer Cederberg
Jonathan Sönnerup
2022

Contents

Introduction

Before venturing into unfamiliar grounds, it is sensible to ask the question why it should be done. An answer to that will be given here.

For an engineer, computer-driven equipment and tools are essential, and for that reason it is also important to understand how a computer works. When you know this, potential problems regarding such tools will be more easily understood and thus less cumbersome to solve. This knowledge will be acquired through a series of lectures, exercises, and laboratory exercises. The lectures and the exercises treat this topic on a wider scale, while the laboratory exercises will give a hands-on experience of how to use a specific computer system, namely a microcontroller.

The following exercises will focus on how simple applications are created, compiled, and transferred to the microcontroller, and then executed. First, this will be done in a way that is common to the majority of processors (ARM, Intel, AMD, and so on). Later on, an integrated development environment (IDE) will be introduced.

Lab Equipment

During the laboratory exercises, the circuit board in Figure 1 will be used. It houses several components that will be used throughout the course. The essential parts are enclosed by red rectangles. Each rectangle is labeled with a number.

A simplified schematic of the circuit board can be viewed in Figure 2. For those who are interested in a detailed version, the full schematic can be found at `https://github.com/eit-lth/Computer-Organization`.

Below, a brief description is given. A more comprehensive explanation of each component will be given when they are used.

1 - Power supply Connect the external power supply here.

2 - Power switch This switch is used to turn on or off the electronics on the PCB (including the microcontroller).

3 - LED's Eight LEDs that are connected to Port B. By setting a pin high the corresponding LED emits photons with the energy 2.8^{-19} J ;).

4 - Potentiometers Two potentiometers are connected to Port A on the microcontroller (PORTA0 and PORTA1).

5 - Buttons Six buttons that are connected to Port A on the on the microcontroller (PORTA2 to PORTA7).

6 - OLED-display The OLED-display consists of 128x128 RGB pixels. It communicates over a serial bus which uses the UART (Universal Asynchronous Receiver Transmitter) protocol.

7 - CPU reset This button can be used to reset the microcontroller. When doing so, the execution is stopped, the RAM is cleared, and the program counter is set to its initial value. After this is done, the microcontroller is restarted.

8 - Atmega1284 This is the microcontroller.

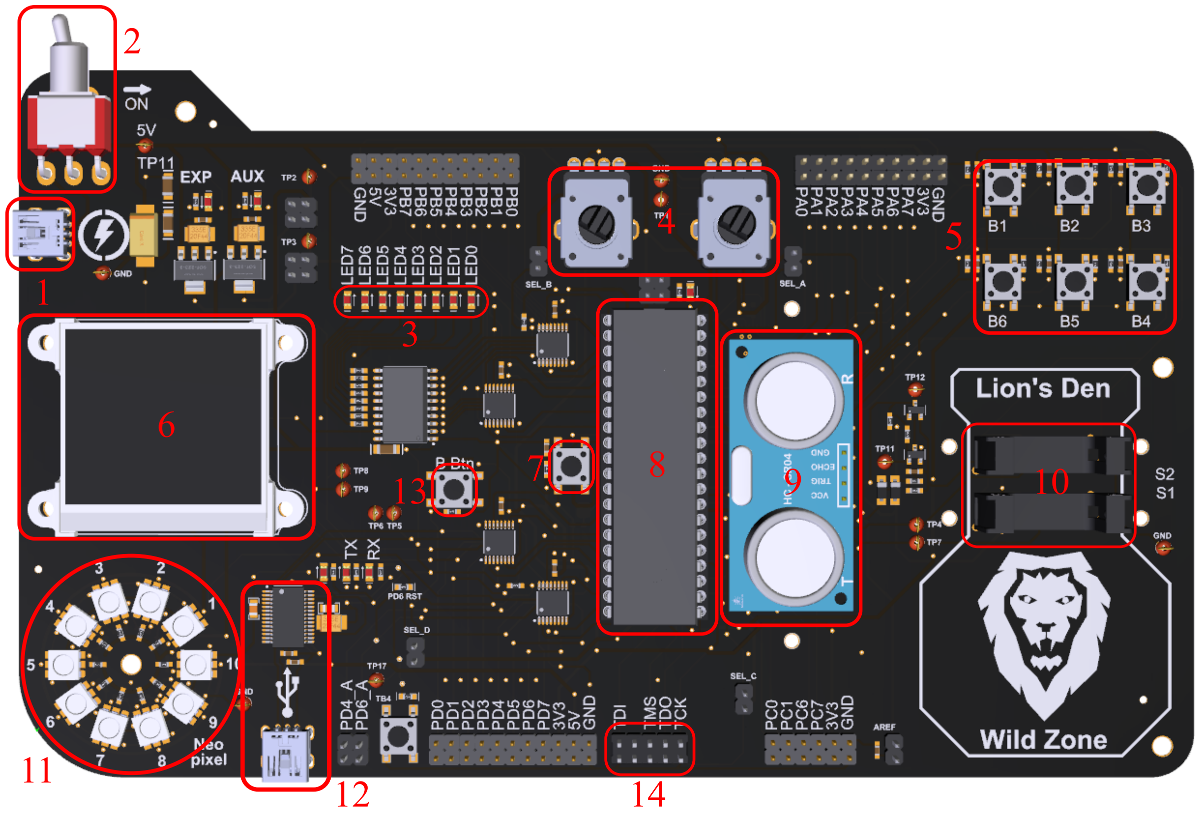


Figure 1: A picture of the PCB that will be used during this course.

9 - Ultrasonic sensor This piece of hardware can be used to measure the distance to an object. This is done by measuring the time of flight of a burst of sound waves (from the sensor to an obstacle and back). This time is proportional to the distance to the object on which the sounds reflects.

10 - Two opto interrupters A opto interrupter is a photo sensor that consists of two parts, an optical transmitter and an optical receiver. If there is an obstacle stopping the light beam from reaching the receiver, the output goes high, otherwise it is low.

11 - Addressable RGB LEDs There are ten addressable RGB LEDs available on the circuit board. They are connected in series and the microcontroller can target them individually, that is, set the intensity of the red, blue, and green LED.

12 - USB-UART interface This integrated circuit translates UART to USB (and the reverse) and enables the microcontroller to communicate with a PC. In order to send data back and forth to a PC, a USB cable needs to be connected between the USB mini connector on the circuit board and the PC.

13 - Button This is a spare button, nothing fancy.

14 - Connector for the Atmel ICE programmer and debugger This pin header connects a device named "Atmel ICE programmer and debugger" to the microcontroller. The device is used when transferring the program to the microcontroller and when an application is debugged.

During the first laboratory exercise the buttons (labeled with 5) and the LEDs (3) will be used.

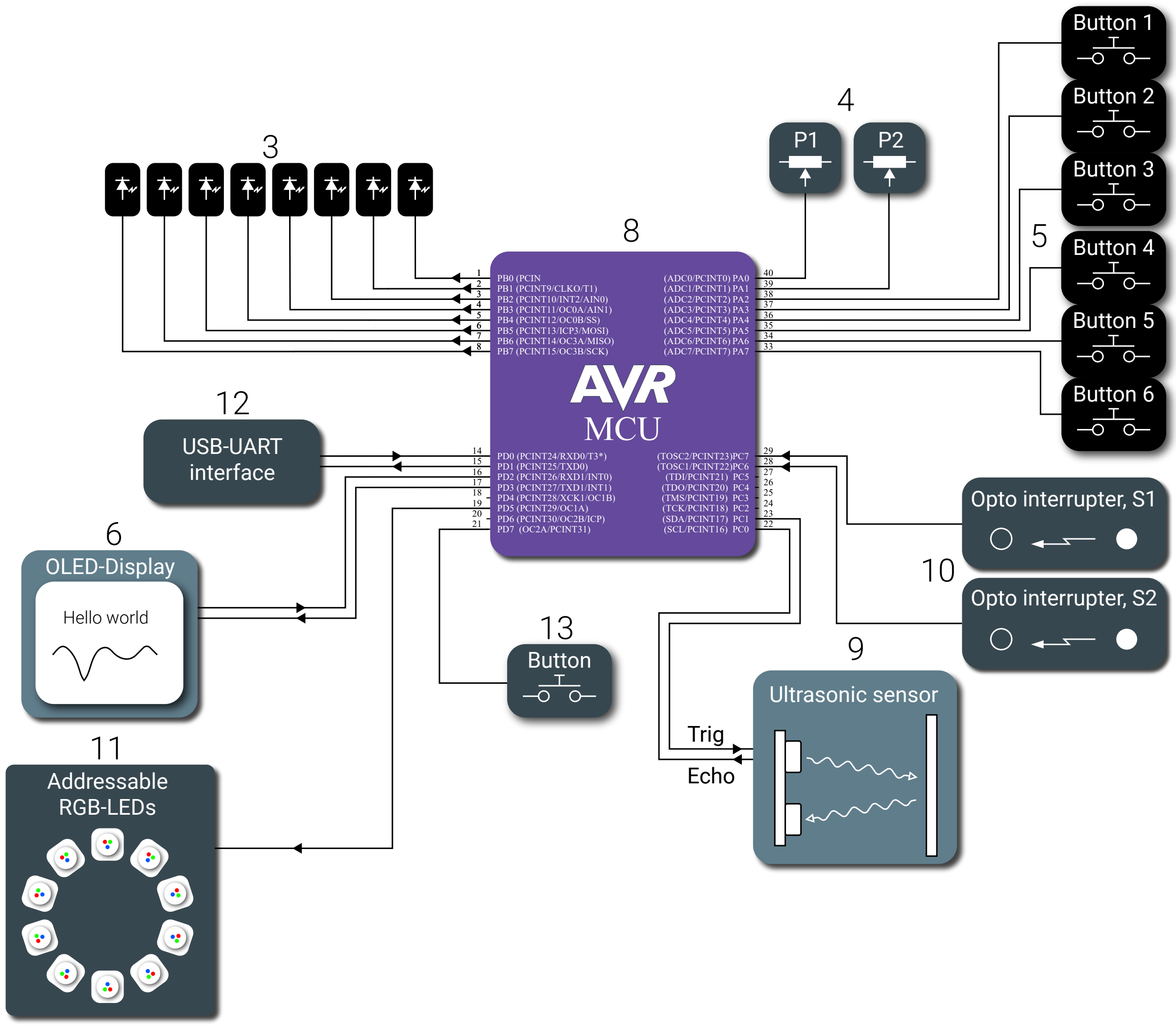


Figure 2: Block schematic of the circuit board.

Chapter 1

Introduction to the AVR Microcontroller

The microcontroller[1] used in this course is the ATmega1284. A block schematic of the microcontroller, labeled as 8 in Figure 1, is depicted in Figure 1.1.

Throughout the laboratory exercises, almost all of the peripheral units will be used, including the universal asynchronous/synchronous receiver and transmitter (USART X), the analog-to-digital converter (ADC), timers/counters (TC X), the IO-ports, and external interrupts (EXTINT). *Do not worry*, they will all be explained later on.

1.1 AVR Architecture

The ATmega family of microcontrollers is a modified Harvard architecture 8-bit RISC which is commonly referred to as AVR[2]. For comparison, The ARM Cortex A53 processor (which the raspberry pi 3 uses), is a 64-bit RISC architecture, with 4 cores.

In a Harvard architecture the program and data memory is separated. The main reason for this is to speed up the execution. With the memory divided into two parts, reading instructions and reading from / writing to the RAM can be done simultaneously. In Figure 1.2, a block diagram of the CPU is shown.

As seen in the block diagram, the CPU has 32 8-bit registers, see the block called "Register file." These registers are used as small and fast storage locations for the data that is currently used. The data may come from the larger data memory, from some of peripherals (that is, the USART unit, the analog-to-digital converter, etc.), or machine instructions. The data is often manipulated or examined by the ALU (Aritmetic Logic Unit). The output from the ALU can be stored in a register or in the data memory to be used later on, just like a variable in any programming language. The AVRs can be clocked from an internal R oscillator, by an external clock or with the help an external crystal. The microcontroller used in lab equipment is configured to use an external crystal at 16 MHz, thus the clock frequency of the processor is 16 MHz.

1.2 Memory Layout of a Program

To program the microcontroller, the C programming language will be used. When a C program is compiled, the code will be divided into the the following segments:

[1]A microcontroller is a small computer, including memory, a CPU, programmable input/output ports, etc., all fitted inside a single integrated circuit

[2]The acronym AVR is somewhat mysterious. There is no definite answer to what it stands for. The manufacturer, Atmel (which was bought by Microchip in 2016), has given different explanations over the years. "Advanced Virtual Risc" and the name of the creators "Alf (Egil Bogen) and Vegard (Wollan) 's Risc processor" are two of them.

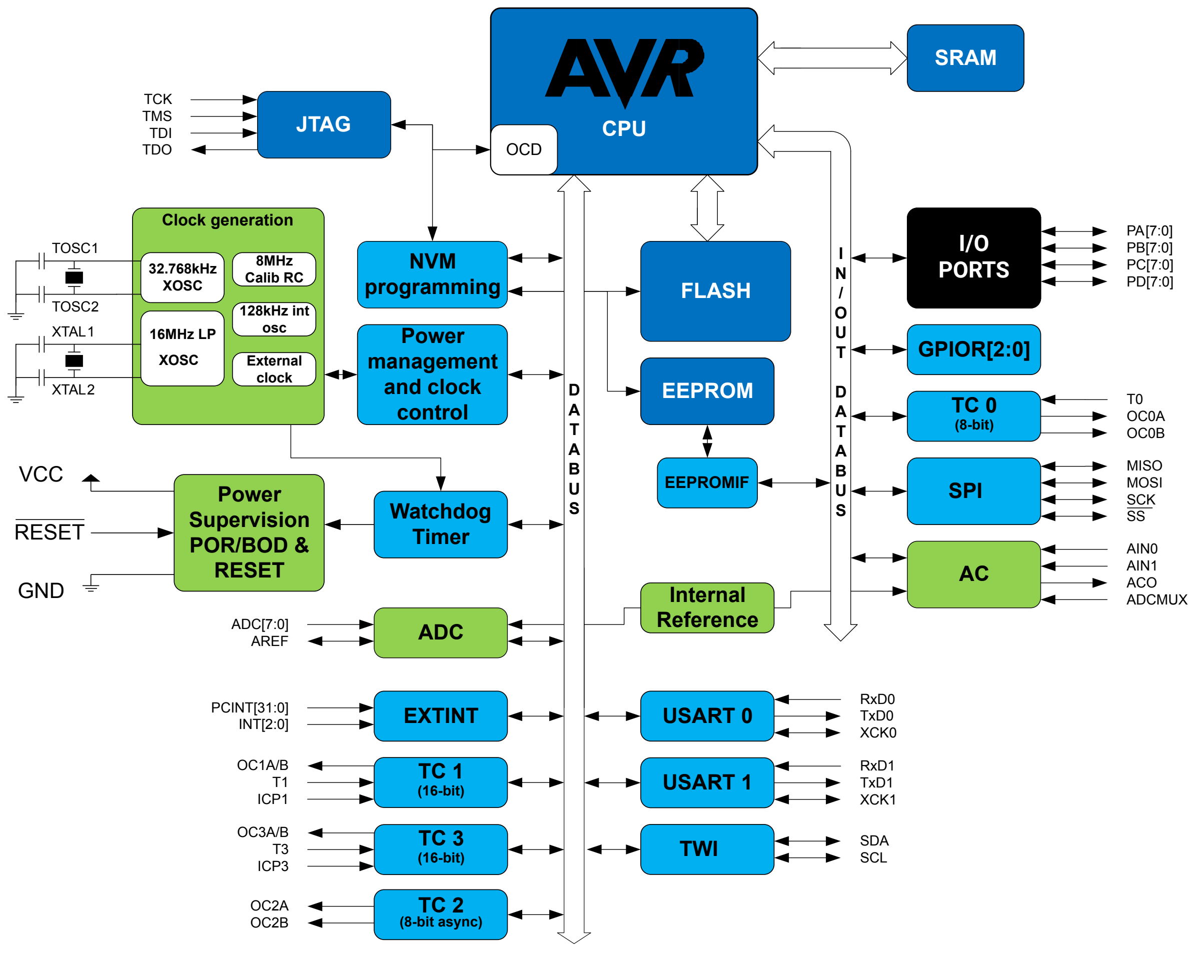


Figure 1.1: Block diagram for an ATmega1284.

`.text` segment This segment contains the actual program, that is, the instructions in machine code.

`.data` segment The values of the initialized global or static variables reside here.

`.bss` segment In contrast to the `.data`, this is the location where the values of the uninitialized global or static variables will be stored.

`stack` and `heap` The stack and heap are regions in the RAM. The stack is used for temporary storage in a function (subroutine), and the heap is a region used for dynamically allocated memory.

When the program is transferred to the microcontroller, it is stored in the FLASH memory. On start-up the `.data`-segment is transferred to the RAM, see Figure 1.3. During execution, each instruction is fetched from the `.text`-segment in the FLASH, decoded, and executed.

The separation between `.data` and `.bss` is done in order to save memory. The values of each initialized global or static variable is stored in the flash. The required space for the `.data` section is equal to the sum of the individual sizes of the initialized variables. An uninitialized variable, on the other hand, does not need to have an initial value. This creates an opportunity for optimization, as they can all be initialized to the same value. Therefore the memory space needed for the initialization data is decreased.

1.3 AVR Toolchain

To develop an executable application for a target processor, numerous tools are needed. Together they form what is called a toolchain (since they are used sequentially). An overview of the tools that are

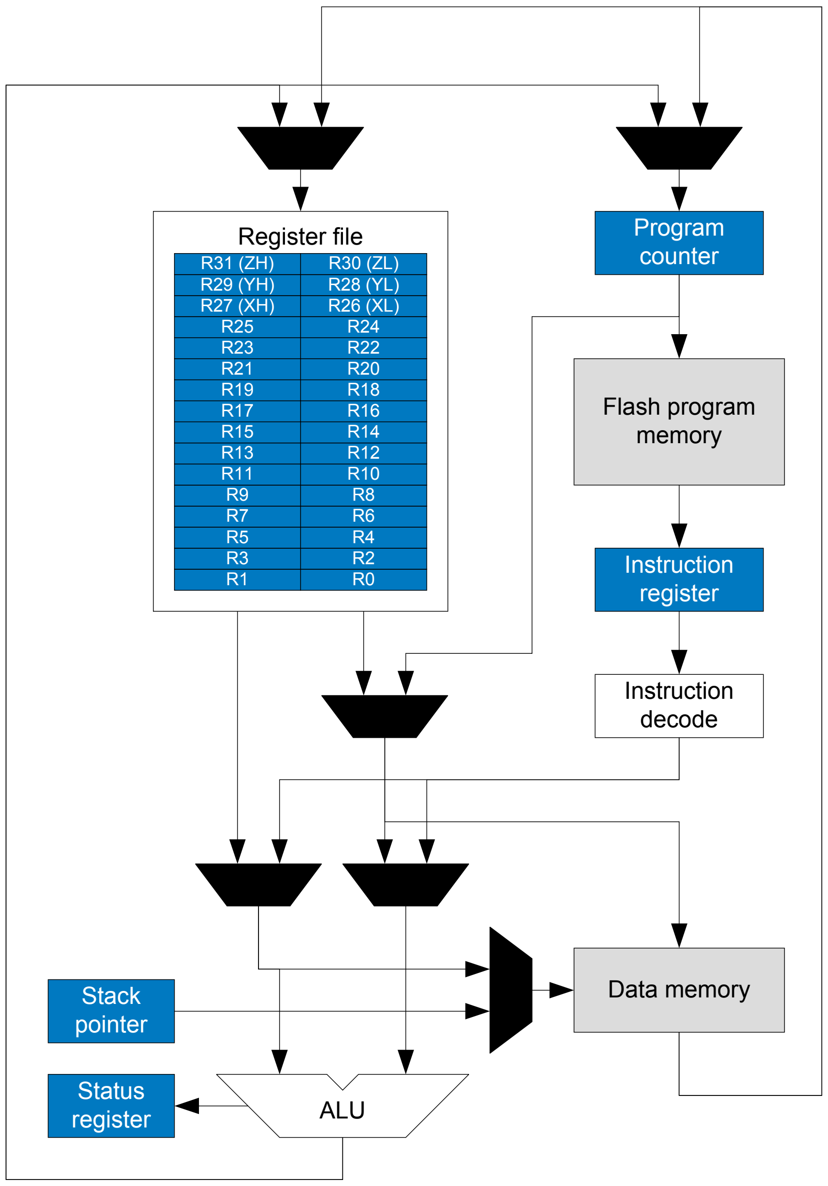


Figure 1.2: Block diagram for a AVR processor.

needed to generate an executable application for the AVR microcontrollers will be provided in this section. Note that this is very similar to other processors as well, such as Intel Core i9, AMD Ryzen, and ARM processors.

1.3.1 GCC - GNU Compiler Collection

The GNU Compiler Collection is a versatile compiler system. It is comprised of many different front-end compilers for various languages and has many back-ends, that is, it can produce assembly code targeting a variety of different processors. The front-ends and back-ends share some generic parts of the compiler which includes optimization.

If the host system that the compiler runs on differs from the target system, the compiler is a cross-compiler (if the host and target system is the same it is a native compiler). The version of GCC that will be used in this course is called AVR GCC and it is a cross-compiler, since it produces code for a different processor (you cannot run the executable on your own computer). AVR GCC supports three

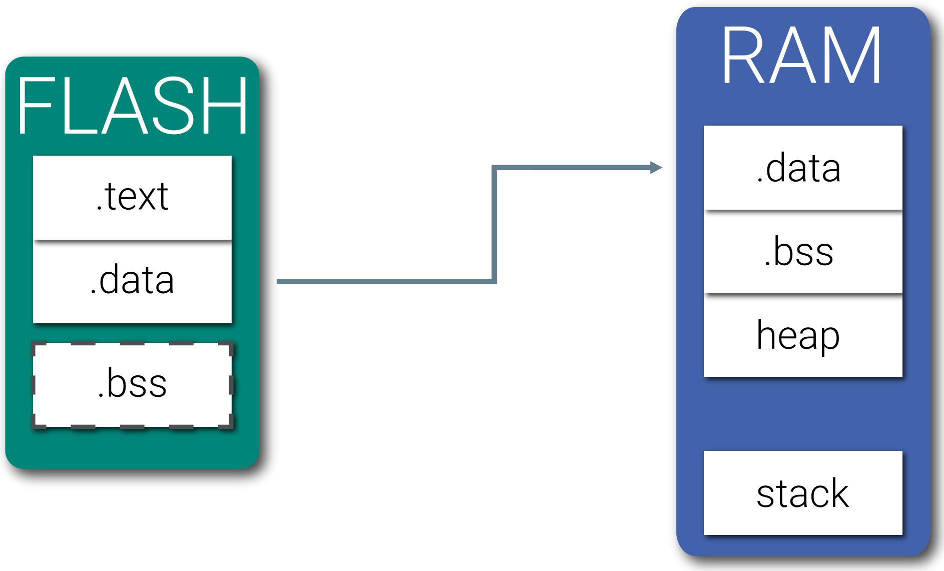


Figure 1.3: Memory layout and the content of the RAM and FLASH memory.

different languages, C, C++, and Ada.

In many cases, a compiler generates the actual machine code, but this is not the case for GCC since the output is in assembly language. Fortunately enough, AVR GCC is also a driver for other programs that are needed to produce an executable output. It uses an open source project called GNU Binutils (GNU Binary Utilities), which contains an assembler and a linker. Refer to Appendix **??** for a list of all included tools. The assembler translates the assembly code to machine code, and the linker links all of the object files to a executable file. The executable name of GCC, when it is configured to target the AVR microcontrollers, is `avr-gcc`. This is what is used in the command prompt when it is time to compile source code. Please refer to Appendix **??** for more details regarding this matter.

The output from avr-gcc is an `.elf`-file. The acronym stands for Executable and Linkable Format and it is similar to Windows `.exe`-file or a `.DMG` and `.APP` for Mac OS. The `.elf`-file contains an abundance of information which is designated for an operating system. Since there is no operating system on the microcontroller, this information serves no purpose and thus it is not needed in order to run the application on the microcontroller. The file that should be transferred to the program memory should just contain the program (instructions in machine code) and the data variables. With the program `objcopy`, which is part of the GNU Binutils, the parts that should be transferred can be extracted (`.text, .data` and `.bss`) from the `.elf`-file. This is known as a `.hex`-file.

There are two other tools that will be used in this laboratory exercise, and these are the `avr-objdump` and `avr-size`. With `avr-objdump` it is possible to displays detailed information about one or more object files and with `avr-size` the size of each memory section is displayed. See Appendix **??** for more details about `avr-objdump` and Appendix **??** for `avr-size`.

1.3.2 avr-libc

With only GCC and Binutils it is not possible to create an executable application. A key ingredient is missing, namely a standard C Library. There is a couple of different open source projects that provide just that. The AVR Toolchain commonly uses one of them, namely the avr-libc. It is a subset of the standard C language library and contains things like AVR-specific macros, AVR start-up code, files that contain the addresses of port and register names (header files), and a floating point library. All the functions that the standard C library contains are available in avr-libc. Some of the standard C functions have limitations or other problems which the user needs to be aware of before using them. Luckily enough, avr-libc is well documented (`https://www.nongnu.org/avr-libc/user-manual/pages.html`).

Additionally avr-libc contains many AVR-specific functions.

1.3.3 The Atmel-ICE Debugger and atprogram

To transfer the hex-file to the microcontroller the Atmel-ICE programmer and debugger, see Figure 1.4, and a software tool called `atprogram` is used.

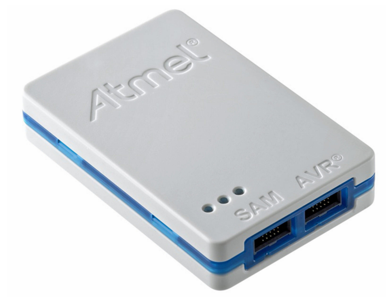


Figure 1.4: The Atmel-ICE programmer and debugger.

The Atmel-ICE programmer and debugger needs to be connected to the microcontroller, see Figure 1.5.

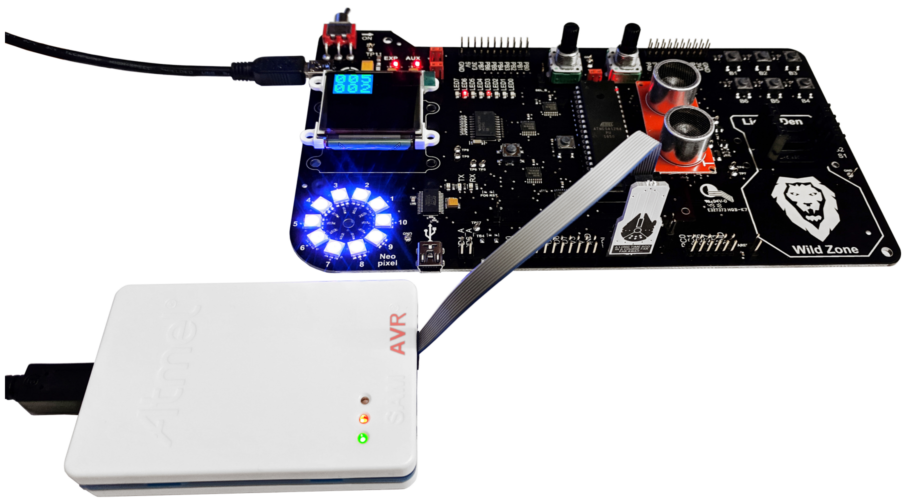


Figure 1.5: Connecting the debugger.

The end of the ribbon cable that is not connected to the pin header on the PCB should be connected to the connector labeled with “AVR” on the Atmel ICE programmer.

The steps from source code to a running application are summarized in Figure 1.6. Please refer to Appendix ?? for details regarding `atprogram` commands.

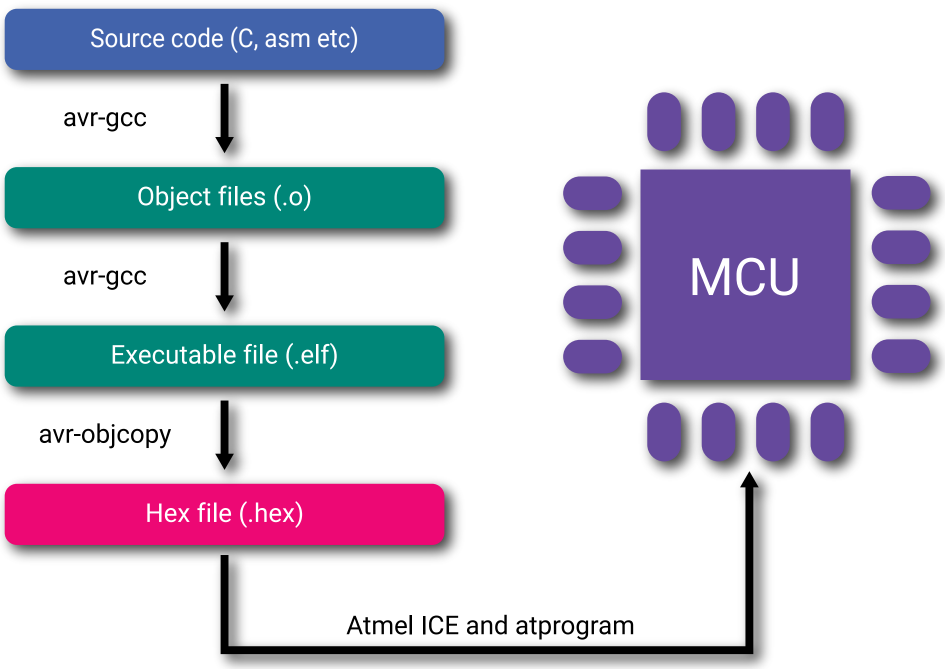


Figure 1.6: Summary of the steps from source code to transfer the binary to the microcontroller.

1.4 Programming Basics

There is a C compiler for almost every processor encountered. Thus, C is the natural language to use when dealing with low-level applications. Here, we introduce the basics of the C programming language and also show specific details regarding the AVR processor. The structure of a basic C program is shown in Listing 1.1.

```c
#include <snape.h>

char glob = 42;
int unknown;

int main()
{
    char a = 1;

    while (1) {
        perform_dark_arts();
    }
}
```

Listing 1.1: A basic C program.

First of all, we need a main function, just like Java's "`public static void main`." This is where code begins to execute. In the main function, we define a *local* variable and then we perform some dark magic. The `perform_dark_arts` function is written in another C-file or library. In order to use it, we must include it, just like java's `import` statement. The function is declared in `snape.h` and implemented in `snape.c`. To use the functions, the `.h` file is included, as seen at the top of the file. Above the main function, two *global* variables are defined, one initialized to 42, the other one uninitialized. These will end up in different memory segments, see Section 1.2.

1.4.1 C Data Types in an AVR

In Table 1.1 below the most commonly used data types in C are listed. In order to use the `xx_t` types, `stdint.h` must be included.

Table 1.1: C data types.

Name	**Size (byte)**	Min Value	Max Value
char	1	-128	127
unsigned char	1	0	255
int8_t	1	-128	127
uint8_t	1	0	255
int16_t	2	-2^{15}	$2^{15}-1$
uint16_t	2	0	$2^{16}-1$
int	2	-2^{15}	$2^{15}-1$
unsigned int	2	0	$2^{16}-1$

1.5 I/O Devices - Interfacing the world

An I/O device or peripheral unit is a piece of hardware connected to the processor either inside the same chip or externally. This device can perform tasks that the processor is not able to by itself. This could for instance be keeping track of time, convert analog signals to digital values or handle communication to external devices.

It is common that the peripheral unit that resides inside the same chip as the processor contains one or several registers[3]. They are often divided into control, status and data registers. The control register is used to specify the behavior of the unit. The status register is used to obtain information regarding the state of the device while the data register simply contains data (e.g. the last converted analog value or the last received byte from an external unit). In the course project many of you will use several of these on-chip peripherals, but during this lab exercise we will get acquainted with the simplest type. These peripherals are described in Section 1.5.2.

1.5.1 Memory-mapped I/O

The peripheral units inside the microcontroller chip are memory-mapped. This means that the registers of the units shares the same address space as the memory of the processor. They can for this reason be accessed by the same instructions as used for writing and reading the data memory. In Figure 1.7 the data memory map is shown.

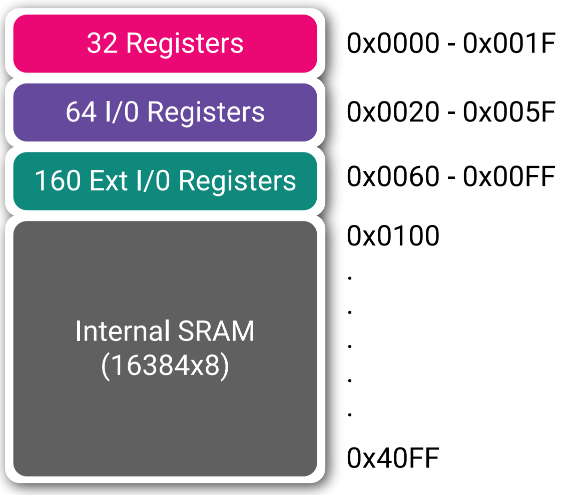


Figure 1.7: Address space of the ATmega1284.

[3]A register is a group of 1 bit memory elements. The majority of the registers in an AVR has the width of 8 bits (a byte), thus it can store an 8 bit word.

1.5.2 I/O Ports

In this laboratory exercises two out of four 8-bit, bi-directional I/O ports was used. The name of the ports are port A, B, C and D. With this piece of hardware it is possible to change the state (high or low) of an individual I/O pin. By doing so it is, as an example, possible to turn an LED, which is connected to one of the ports I/O-pins, on or off. The I/O ports can also be configured to be an input.

Each port has three different register. For port A the name of these registers are `DDRA`, `PINA` and `PORTA`. The register names is a label that corresponds to the address of the given register and can be used in the code. Each port contains 8 pins and thus the corresponding registers have the same length. The DDR (Data Direction Register) register is used to specify if a pin is set to be an input or an output. The PIN register is used to read the state of the port and the PORT register is to set pins of the port high or low.

Table 1.2: I/O port registers and their corresponding functionality.

Register	**Description**
DDRx	Specifies the data direction
PINx	Read the state of the port
PORTx	Set the state of the port

To set the first pin on port B, `PORTB0`, as an output and the pin high, follow the steps below.

- Set the data direction register to `0x01` (hexadecimal) or `0b00000001` (binary).
- Set the pin `PORTA0` to high by writing `0x01` or `0b00000001` to the port register.

When writing to a I/O port register (or any other register for that matter) it is good practice to use the logic operator |= instead of the equal sign (=). By using the |= operator a bitwise OR operation is done with the content of the register and the value on the right hand side of the operator. By doing so it is ensured that only the specified bit is changed. This is good since in the majority of cases there will be different types of devices connected to the same port. When clearing a bit (set the value to 0) one should apply the methodology, i.e. do nate change the state of the other bits in the reister. This can be done by &= operator, bitwise AND, in conjunction with the bitwise NOT operator, ˜. How to do this in C program is shown in Listing 1.2

```
// Setting the first bit to 1
// The content of PORTC is 0b00100100

PORTC |= 0b00000001;            //      0b00100100
                               //  OR  0b00000001
                               //---------------------
                               //      0b00100101

// Clearing the first bit
// The content of PORTC is 0b00100100

PORTC &= ~0b00000001;           // The bitwise inverse of 0b00000001 is 0b11111110

                               //      0b00100101
                               //  AND 0b11111110
                               //---------------------
                               //      0b00100100
```

Listing 1.2: Bitwise operations.

As mentioned earlier the microcontroller has four I/O ports. See Table 1.3 for their addresses. In Listing 1.3 C code that turns on an LED if the state of a given I/O pin is high is shown.

Table 1.3: I/O registers and their corresponding addresses.

Address	Register
0x20	PINA
0x21	DDRA
0x22	PORTA
0x23	PINB
0x24	DDRB
0x25	PORTB
0x26	PINC
0x27	DDRC
0x28	PORTC
0x29	PIND
0x2A	DDRD
0x2B	PORTD

```c
// Turning on an LED connected to pin 2 (starting from zero) on port C
// if the state of pin 0 on port D is high


int main(){

    uint8_t pin_state;       // Declaring a 8-bit variable to hold the pin state

    DDRD &= ~0b00000001;     // Setting the data direction to input for pin 0
                             // on port c

    DDRC |= 0b00000100;      // Setting data direction to output

    while(1) {

        pin_state = PINC & 0b00000001;  // Reading the PINC register and
                                        // filtering out pin 0

        if (pin_state != 0) {
            PORTC |= 0b00000100 // Turn on LED
        } else {
            PORTC &= ~0b00000100 // Turn off LED
        }

    }


}
```

Listing 1.3: Bitwise operations.

Chapter 2

Introduction to Atmel Studio

The manufacturer of the microcontroller used in this course has developed an integrated development environment (IDE) that simplifies the development process, which consists of writing code, compilation, programming the controller, and debugging. The IDE is called Atmel Studio. In this part of the laboratory exercise, this IDE and some of its tools will be introduced.

Bugs are not uncommon during the development of an application. An IDE is a great tool when debugging code, as it lets you pause execution, step through code one line at a time, analyze variables in real time and more.

2.1 Lab Exercises

2.1.1 Hello, World

Blinking an LED is the microcontroller equivalence of the common "hello, world" program. This is usually done as a sanity check that the microcontroller works. In Figure 1, the LEDs are encapsulated with a red rectangle labeled 3. To create a project targeting the microcontroller used in this lab follow the steps in this video:

`https://youtu.be/G7i1gU27d70`

It is important to chose the correct device during the creation of the project. Furthermore make sure that "GCC Executable Project" is selected.

Tasks:

Home Assignment 1.1
Write a program that blinks a LED with the frequency of 1 Hz. See Appendix A for a very helpful delay function.

- This video will show you how to program the microcontroller: `https://youtu.be/GlCbV565vNA`
- This video will show you how to program the microcontroller:

Lab Question 1.1
In general, why is a `while`-loop needed? What would the main function return to? What happens if you remove the loop from your code?

2.1.2 Debugging in Atmel Studio

During this exercise, a program will be debugged using Atmel Studio. The debug capabilities of the IDE is a powerful tool. It helps a developer to analyze the behavior of the program in real-time, while it runs on the microcontroller. The program can also be halted by inserting a breakpoint at a given line of code. Another convenient feature is to monitor variables content in a "watch".

Tasks:

- Create a new project (select "GCC C Executable Project.")

Home Assignment 1.2
Write a program that ***toggles*** (!) a LED when a button is clicked, see the state diagram in Figure 2.1.

- Compile, program the microcontroller, and run your code from the home assignment. Do not forget to include the course library.

- The buttons, B1 to B6, are labeled as 5 in Figure 1.
- This video will show you how to program the microcontroller: `https://youtu.be/GlCbV565vNA`

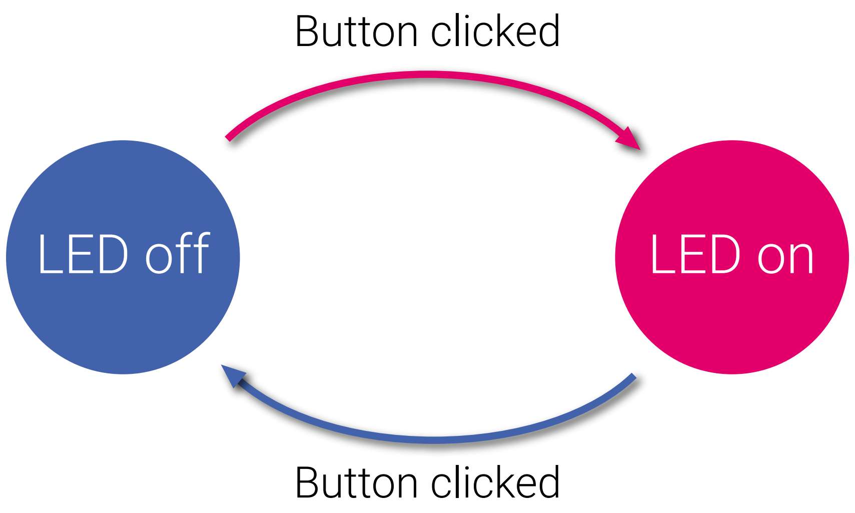


Figure 2.1: State diagram of the program.

- Add a global variable of the type `uint8_t` in your code and initialize it to 3. The variable should be decremented on every button press.
- Change the code so that you only change the state of the LED if your global variable is less than zero (`var` < 0).

Lab Question 1.2
How many times is it required to press the button until the LED is toggled? Why?

- You will now investigate the behavior of the program. Press the "Start debugging and break"-button. Place the breakpoint on a line of code that is executed on every button press.

- Here is a video showing how to start and debug your application `https://www.youtube.com/watch?v=Rx7aLKsiyRA`
- It is always good to turn off the optimization when debugging. Otherwise, the compiler may optimize away some variables. Watch this video to learn how to turn off optimization: `https://youtu.be/aElLa6ujruk`.

- Add your global variable to a watch.
- Use the debugger to understand what the code does.

Lab Question 1.3
What is the problem with the code? How do you solve it?

2.1.3 Knight Industries Two Thousand

The Knight Industries Two Thousand, or KITT, is a world famous intelligent, self-driving vehicle. If this sadly does not ring a bell, watch the following clip, `https://www.youtube.com/watch?v=oNyXYPhnUIs`.

Michael Knight accidentally crashed with KITT, damaging the LEDs in the front. Being an engineer, it is your job to program the moving LED sequence, seen in the beginning of the video.

Tasks:

- Create a new "GCC C Executable" project.
- Using the functions in the course library, implement the moving LED sequence.

It does not have to be a perfect match between your sequence and the sequence in KITT, but one should be able to see the similarity.

Lab Question 1.4
What would be required in order to get the fading effect seen in KITT?

You are now done with this part, show your work to a lab assistant! ____________ ☑

Appendix A

Libraries

A.1 Atmel Library

`void _delay_ms(int val)` - Delays the execution by `val` milliseconds. This value can **not** be a variable, it must be defined at compile time.

Dependency: `<util/delay.h>`, the macro `F_CPU` needs to be defined before the include statement. The value it should be set to is the clock frequency. During the lab it is 16 MHz. See below for an example on how to do that.

```
#define F_CPU 16000000UL
#include <util/delay.h>
```

These two lines should be at the top of your source file. See example 1.1 for something similar.