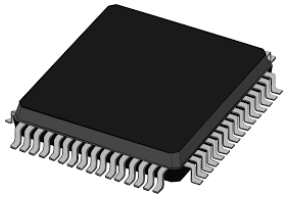
Computer Organization

Lab and Exercise 1



EITF70
EITA15

Introduction to Microcontrollers

Goals

- Get familiar with microcontroller development basics
- Get an understanding of how memory is organized
- Be able to write simple programs interfacing with the world
- Be able to debug simple programs with Atmel Studio

Christoffer Cederberg
Jonathan Sönnerup
2019

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 (PA0 and PA1).

5 - Buttons Six buttons that are connected to Port A on the on the microcontroller (PA2 to PA7).

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.

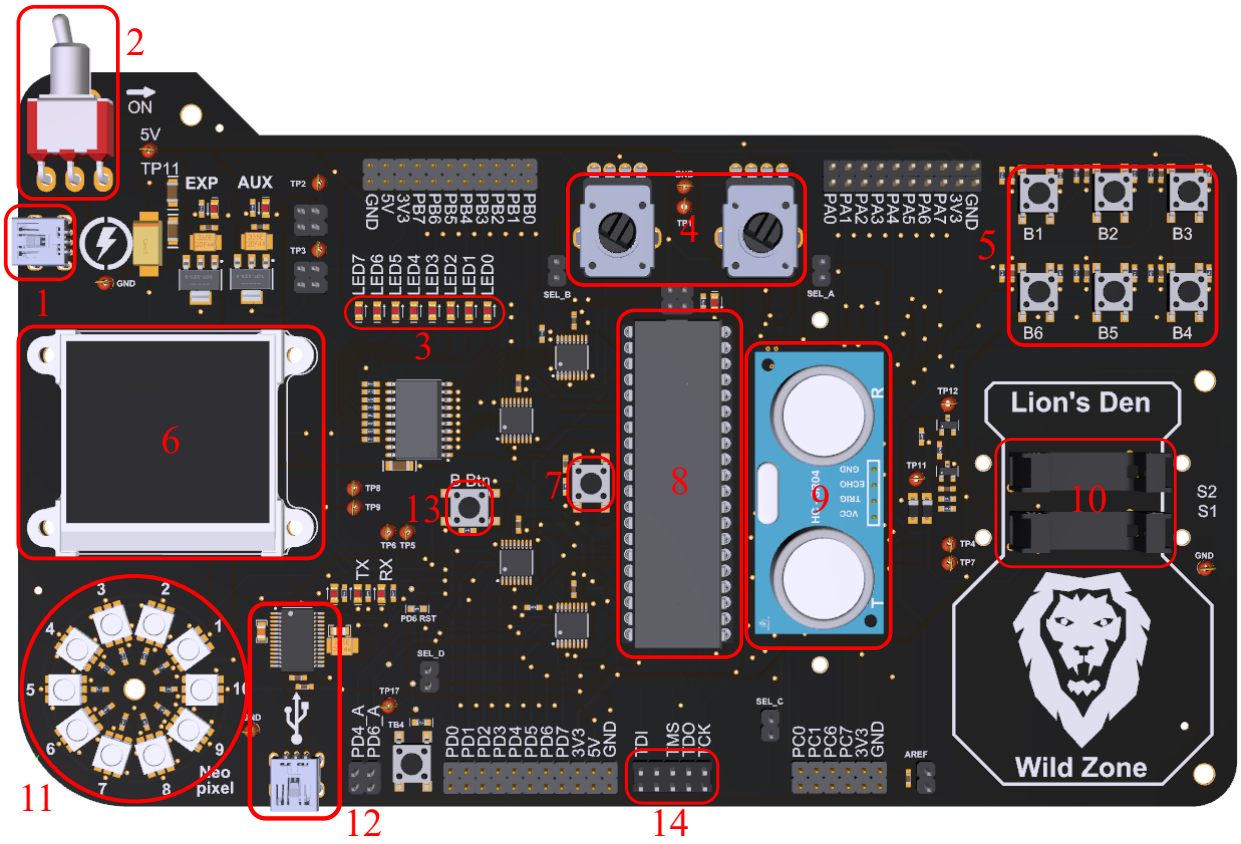


Figure 1: A picture of the PCB that will be used during this course.

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.

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

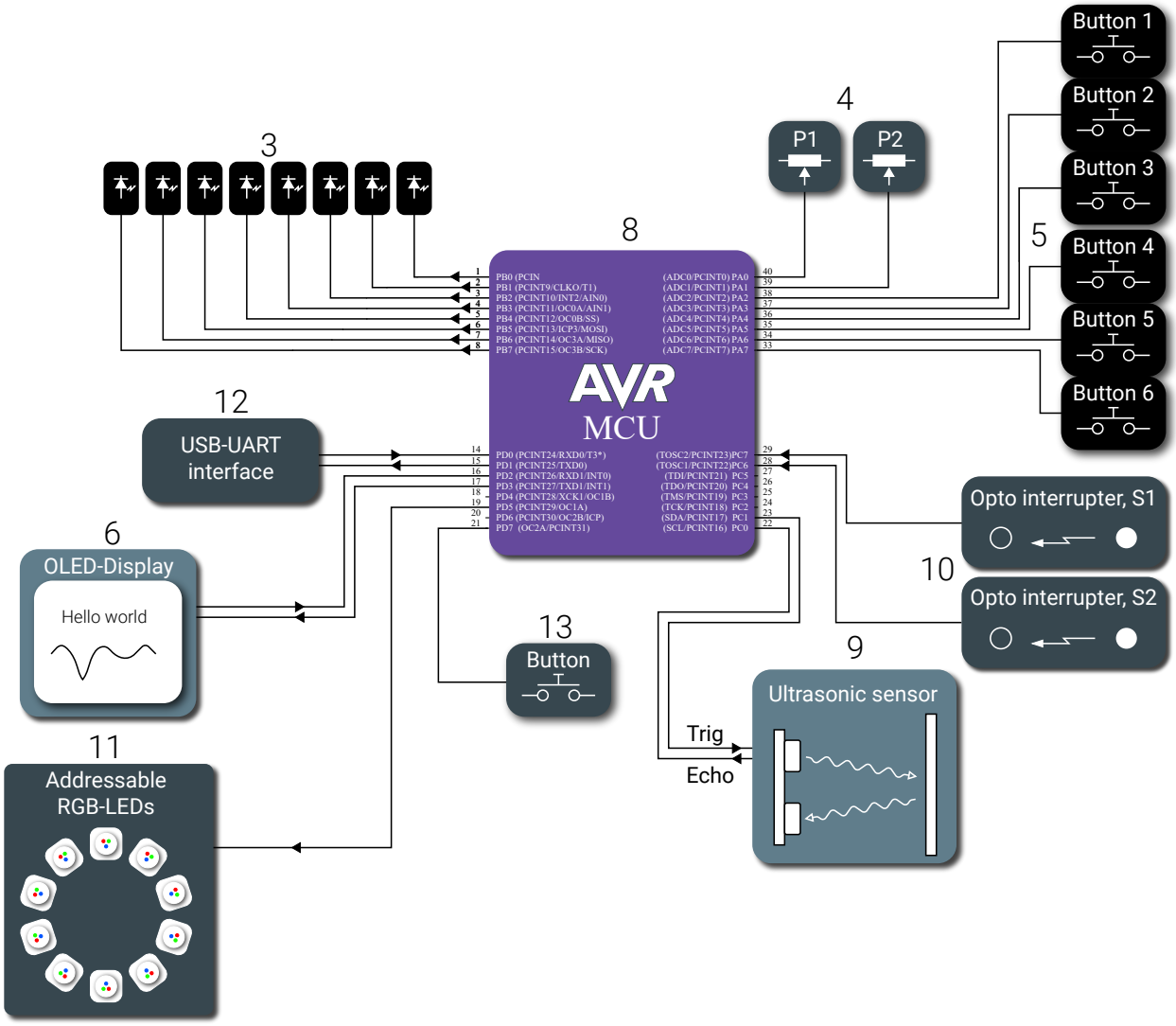


Figure 2: Block schematic of the circuit board.

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.

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] 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.

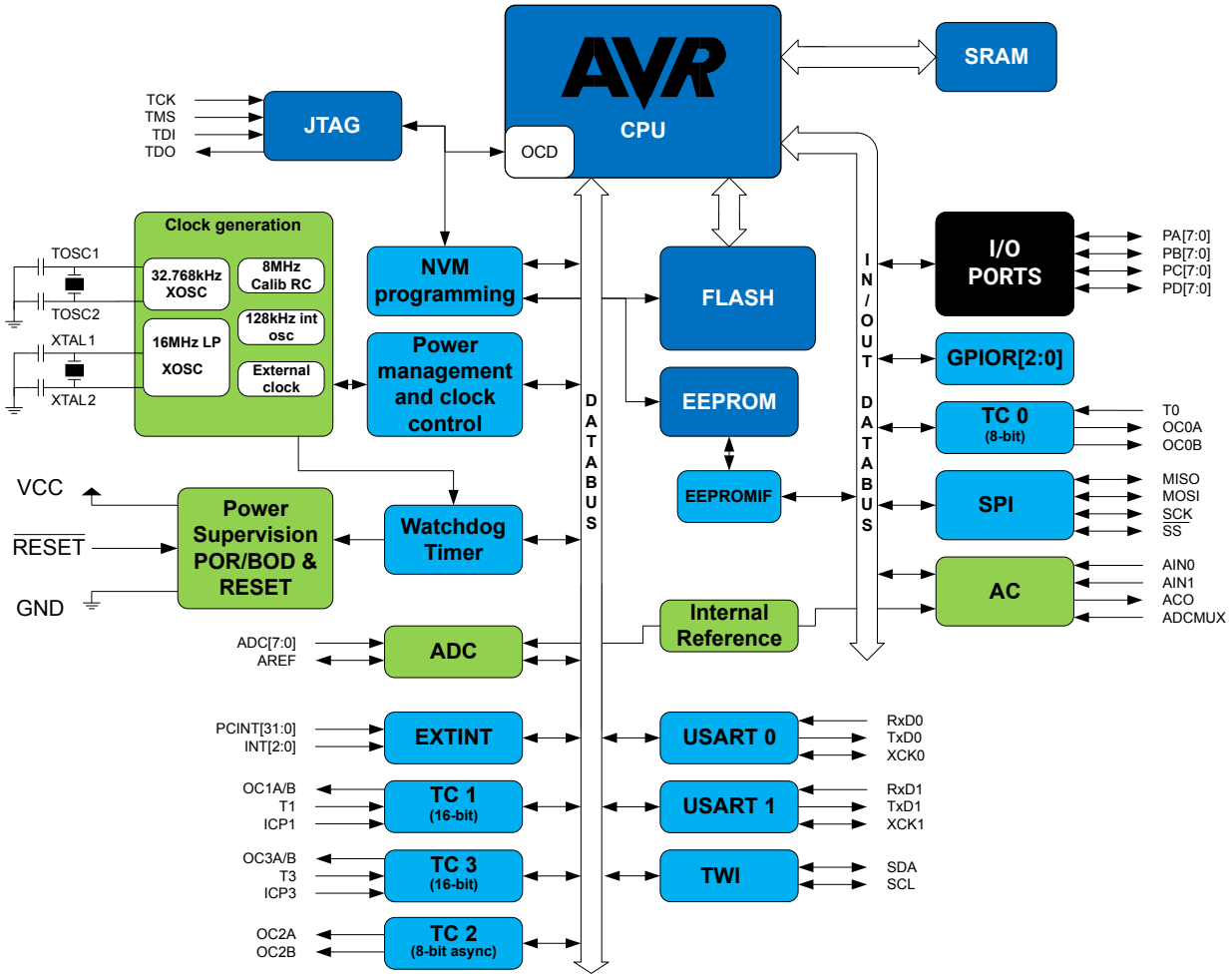


Figure 1.1: Block diagram for an ATmega1284.

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:

`.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

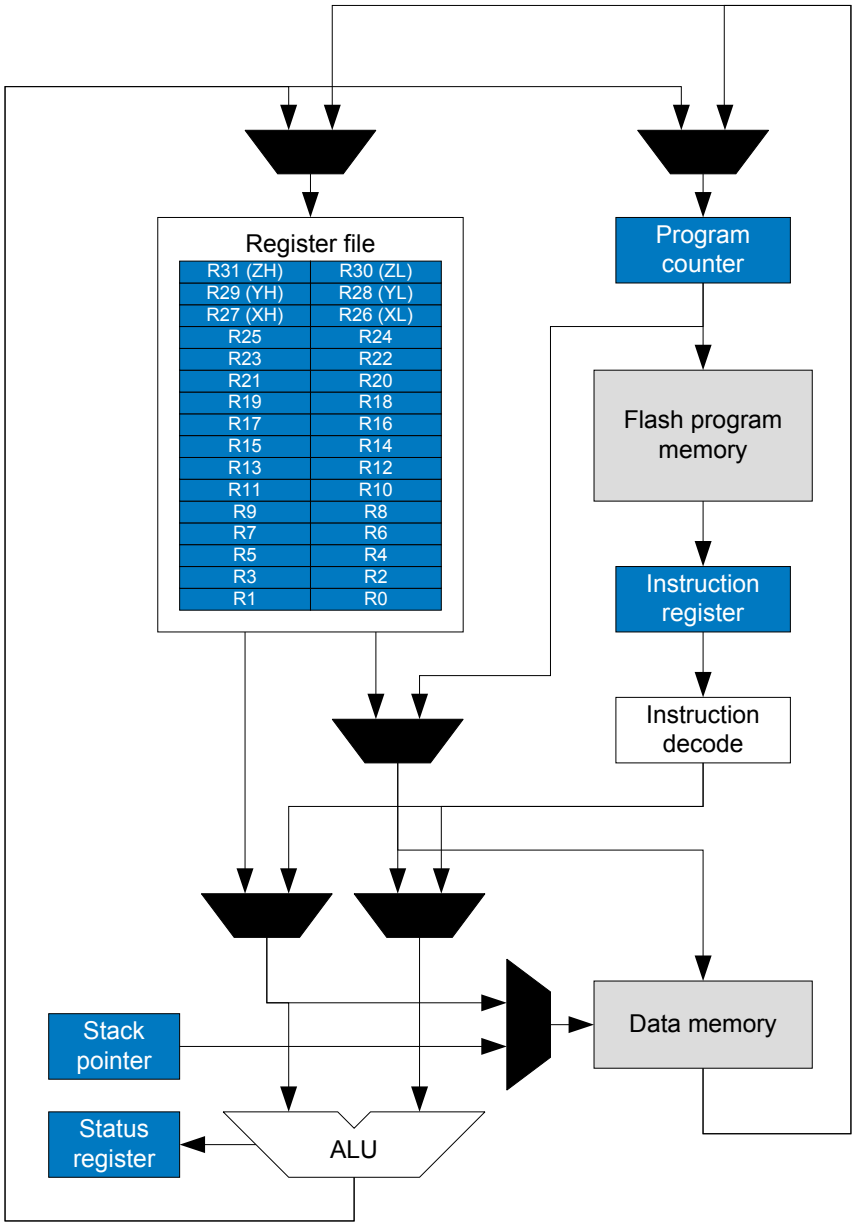


Figure 1.2: Block diagram for a AVR processor.

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.

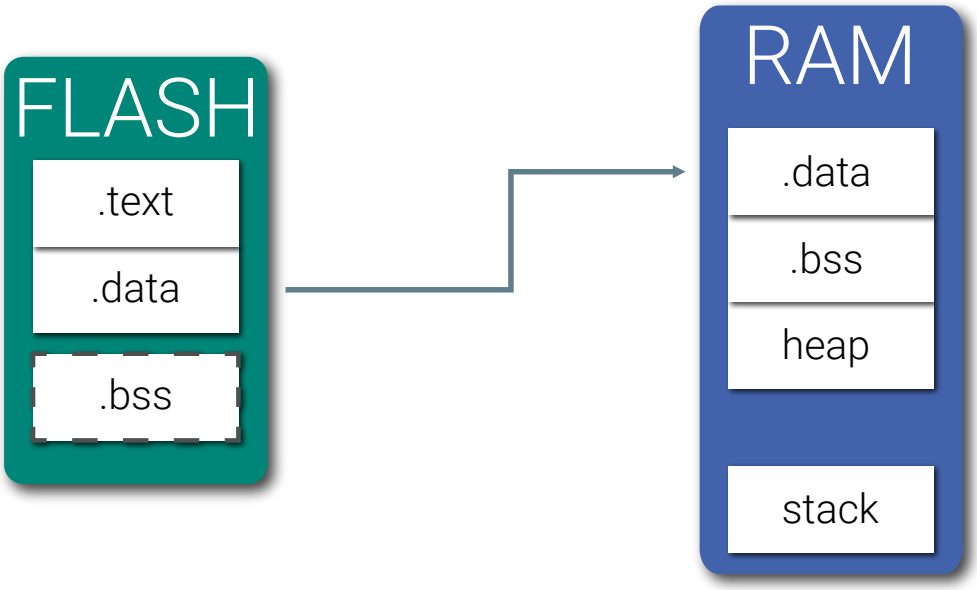


Figure 1.3: Memory layout and the content of the RAM and FLASH memory.

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 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 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 B.1 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 B.2 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 B.4 for more details about `avr-objdump` and Appendix B.5 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.

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 B.6 for details regarding `atprogram` commands.

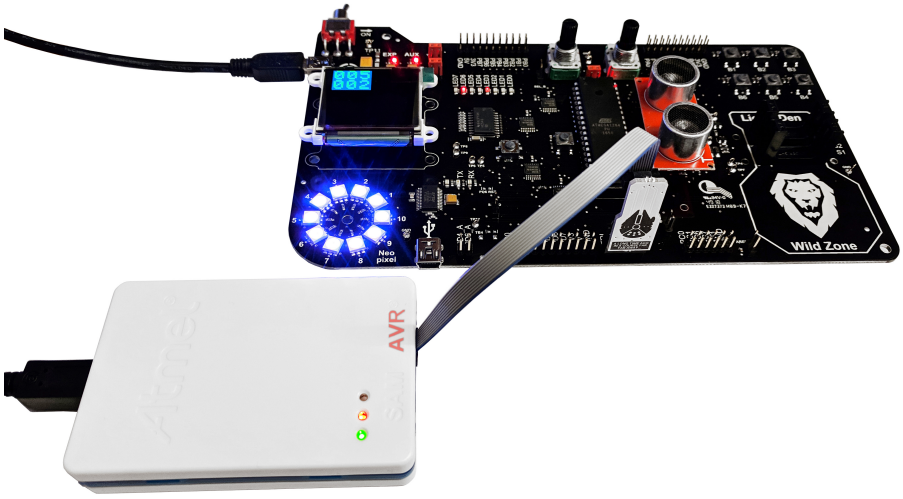


Figure 1.5: Connecting the debugger.

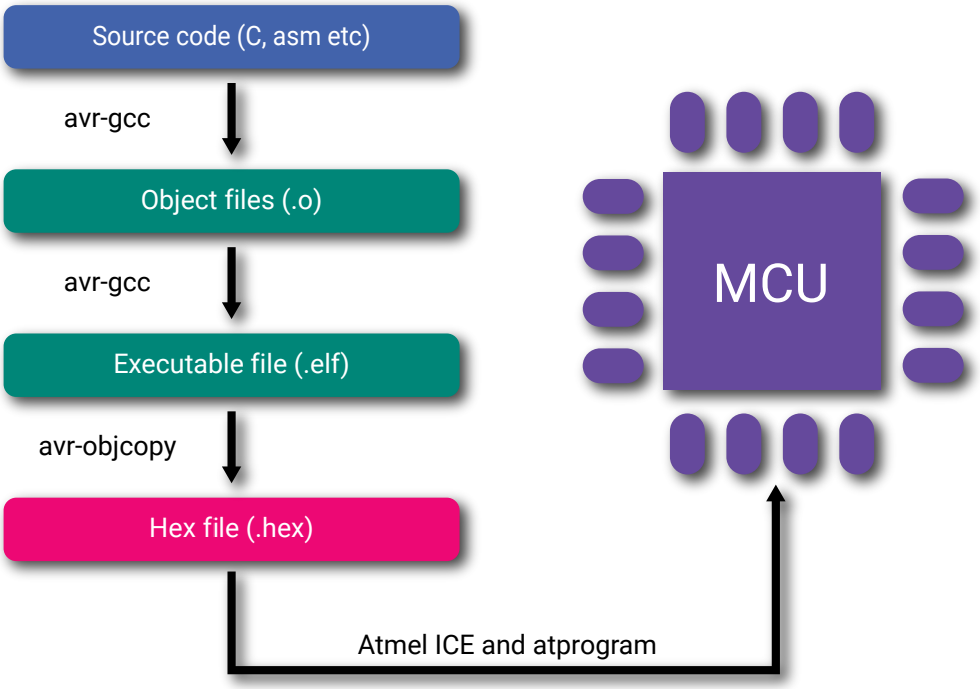


Figure 1.6: Summary of the steps from source code to transfer the binary to the microcontroller.

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.

```
#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 Exercises

Answers to the questions can be found in Appendix C.

CPU Architecture

1.1 What is the difference between a Harvard and a Von Neumann architecture?

1.2 What is the frequency of a modern processor, say Intel or AMD?

1.3 How many CPU cores does an AVR have? What about the latest AMD Ryzen or Intel Core i9?

1.4 What is the size of the RAM in the Atmega1284?

1.5 What is the size of the flash memory in the Atmega1284?

1.6 How much RAM do you have in a modern computer?

1.7 Using an Atmega1284, how many instructions can you execute in 1 second, if each instruction takes one clock cycle each?

1.8 Running at 8 MHz, how many nanoseconds does each instruction have to execute?

1.9 How is the value 0x12FC6701 stored in a memory using little-endian?

1.10 If your compiled code resides in the address range 0x00-0xFF in the processor, will you overwrite the program if you store an array starting at address 0x00? The target processor is an AVR.

C Programming

1.11 What is the size of a `char` for an AVR?

1.12 What is the size of an `int` for an AVR?

1.13 What is the size of an `unsigned int` for an AVR?

1.14 If you store -2 in an `int` variable, what is the hexadecimal representation?

1.15 If you store -2 in an `unsigned int`, what is the hexadecimal representation?

1.16 If you add two 8-bit numbers, how many bits does the result require?

1.17 If you multiply two 8-bit numbers, how many bits does the result require?

1.18 Given the following C code,

```c
int alpha = 1;
char vec[3] = { 1, 2, 3 };
char state;

int main()
{
    static char statham = 666;

    for (int i = 0; i < 10; i++) {
        int looper = 12;
    }

    while (1) {
        // wait for better times...
    }
}
```

How many variables are created, how large are they, and in which memory segments are they stored?

1.6 Lab Exercises

It is now time to get familiar with the lab equipment. This will be done by creating a couple of different applications and analyze them. Do not be afraid to modify the code to try out your own ideas. After all, this is what a lab is all about. Just be sure to show your solutions/answers to a lab assistant before.

1.6.1 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.

Tasks:

- Create a new project folder on the H: drive and create a file using your favourite text editor and save it as <`a_name`>`.c`. If you, by any chance, do not have a favorite text editor, you can use Notepad++.

  Home Assignment 6.1
  Write a program that blinks a LED with the frequency of 1 Hz. Use the functions described in Appendix A.1.

- Implement the program from the home assignment. Do not forget to include the `yoda.h` file. The course library is located at "S:\Courses\eit\EITF70\course_library". Copy all files into your own project folder.

  Home Assignment 6.2
  Write the command for compiling a single c-file into an executable ELF file, see Appendix B.2 for details.

  – You must specify where the .a- and .h-file are located.
  – The `-lyoda` flag needs to be placed last in the command.

- Use your command to compile the program. Make sure no errors (or warnings) are present.

  Home Assignment 6.3
  Write the command for extracting the `.text` and `.data` sections from the ELF file, using `objcopy`, into an ihex file. Refer to Appendix B.3 for details.

- Create the ihex file using the command from the home assignment.
- Last, program the microcontroller with the generated hex file using the `atprogram` command, see Appendix B.6.

Lab Question 6.1
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?

Make sure that the the PCB is powered on and that the JTAG is connected before programming the device.

If the program does not work, check the output in each command for warnings or errors.

If everything works, you have now succeeded in writing the code for the "hello, world"-application, compiled it, and transferred it to the microcontroller.

1.6.2 Code Analysis with `objdump`

In some cases it is necessary to examine what the compiler has generated. A typical example is when the code is not working as expected or if the execution time needs to be improved. Any executable can be dissected by using the `objdump` command, see Appendix B.4. The output is shown in the assembly language for the processor. This will be covered in detail in Lab 3.

Tasks:

- Open the text editor and write the program listed in Listing 1.3.
- Compile the program with the commands previously used, using **no** optimization. Refer to Appendix B.2 for optimization options.

Home Assignment 6.4
How do you disassemble an ELF file using `objdump` showing only instructions?

- Perform an object dump on the ELF file and look for the **main** function. Your variables are stored in registers r_0 to r_{31}. Look in the "instructions" column in the dump, see Listing 1.2.

```
000000a4 <main>:
(addr) (machine code)  (instructions)            (comments)
  a4:   cf 93           push    r28
  a6:   df 93           push    r29
  a8:   00 d0           rcall   .+0             ; 0xaa <main+0x6>
  aa:   00 d0           rcall   .+0             ; 0xac <main+0x8>
  ac:   00 d0           rcall   .+0             ; 0xae <main+0xa>
  ae:   cd b7           in      r28, 0x3d       ; 61
  b0:   de b7           in      r29, 0x3e       ; 62
  ...
```

Listing 1.2: `objdump` example.

```c
#include <stdint.h>

int main()
{
    char a      = 'A';
    int b       = -2;
    uint8_t c   = 1337;
    uint16_t d  = 1337;

    while (1);
}
```

Listing 1.3: Simple C program with local variables.

Lab Question 6.2
For each variable, what value is stored in the corresponding register? Notice that each register is only 1 byte, and for some variables the processor needs 2 registers.

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

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 Creating a Project

To create a project targeting the microcontroller used in this course 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.

2.2 Lab Exercises

2.2.1 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 2.1
  Write a program that toggles a LED when a button is clicked, see the state diagram in Figure 2.1. Use the functions provided in the course library, see Appendix A.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.
  - See this video, `https://youtu.be/e00Ef4CDI9U`, on how to add the library.
  - This video will show you how to program the microcontroller: `https://youtu.be/GlCbV565vNA`

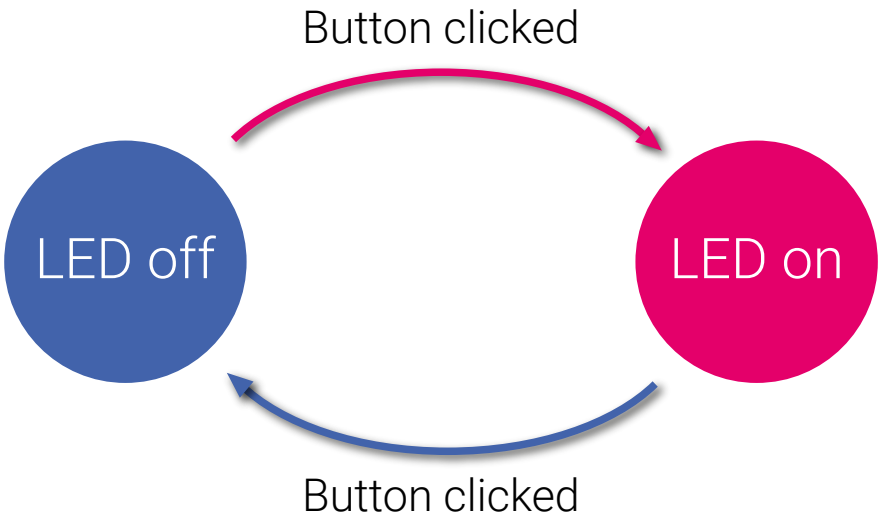


Figure 2.1: State diagram of the program.

- 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 2.1**
> 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.

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.

Once again, it is a good idea to disable optimizations.

> **Lab Question 2.2**
> What is the problem with the code? How do you solve it?

2.2.2 Investigating Endianness

The task at hand is to determine the endianness of the microcontroller. This can be done by analyzing how variables are stored in the memory. Using Atmel Studio, we can view the content of the memory of the microcontroller. Start looking around address `0x40FF`. See the video on the following link, `https://youtu.be/LKDaf7qKiIU`, for how to analyze the memory in Atmel Studio.

Tasks:

- Create a new "GCC C Executable" project.
- Declare at least three different *local* variables of any type.
- Initialize the variables to "convenient" values.
- Enter debug mode and analyze your code.

It is a very good idea to disable the compiler optimization before trying to analyze the code. See the following video, `https://youtu.be/aElLa6ujruk`, for how to do that.

Below is a snippet of the data section of the RAM shown.

section	address	content	
data	0x40DC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
data	0x40EC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
data	0x40FC	00 00 00 4d ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??	@ÿ.M......

Lab Question 2.3
How many bytes are required to determine the endianness? What about the values of the variables?

Lab Question 2.4
What is the endianness of the microcontroller?

- In your project, add an array of type `uint8_t` with at least two elements.
- Initialize the array to values of your liking.

Lab Question 2.5
How is the array stored in memory? Does it conform to the endianness?

2.2.3 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 2.6
What would be required in order to get the fading effect seen in KITT? You will see this effect in later labs.

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

Appendix A

Libraries

A.1 Course Library

Functions provided in the course library, `yoda.h`, are described here.

A.1.1 LEDs

`void led_init()` - Initializes the I/O port where the LEDs are connected, making them an output.

Note: This function must be called before the other LED functions.

`void led_on(uint8_t num)` - Sets the I/O-pin connected to LED number `num` to high. Legal numbers are 0 to 7.

`void led_off(uint8_t num)` - Sets the I/O-pin connected to LED number `num` to low. Legal numbers are 0 to 7.

`void led_toggle(uint8_t num)` - Toggles the value of LED number `num`. Legal numbers are 0 to 7.

A.1.2 Buttons

`void button_init()` - Initializes the I/O pins where the buttons are connected, making them an input.

Note: This function must be called before the other button functions.

`uint8_t button_read(uint8_t num)` - Reads button number `num`. Legal numbers are 1 to 6. The function returns 1 if the specified button is pressed, otherwise 0.

A.2 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. See below for an example on how to do that.

```
#define F_CPU 16000000UL
```

Appendix B

AVR Toolchain and Commands

B.1 The GNU Binutils

The GNU Binutils contains more than just the previously mentioned assembler and linker. See the description below.

avr-as The Assembler.

avr-ld The Linker.

avr-ar Create, modify, and extract from libraries (archives).

avr-ranlib Generate index to library (archive) contents.

avr-objcopy Copy and translate object files to different formats.

avr-objdump Display information from object files including disassembly.

avr-size List section sizes and total size.

avr-nm List symbols from object files.

avr-strings List printable strings from files.

avr-strip Discard symbols from files.

avr-readelf Display the contents of ELF format files.

avr-addr2line Convert addresses to file and line.

avr-c++filt Filter to demangle encoded C++ symbols.

B.2 AVR GCC

First off, open a command prompt. Then navigate to the folder where the source code is placed. To compile the code, use the command line below. But first, replace (including the brackets) [*target*], [*Optimization flag*], [*myfile.c*] and [*myfile.elf*] with the target name, the desired optimization level, the name of the your C file, and the name of the output file. Click here to fast travel back to Lab Exercise 1.6.1 , price: 1 bit.

```
avr-gcc -mmcu=[target] [optim. flag] -g [myfile.c] -o [myfile.elf] -I [path] -L [path] -l[library]
```

The following list summarizes a selection of flags used when compiling.

General options

- `-g` Produce debugging information in the operating system's native format. This is optional and not needed to run the application on the AVR.
- `-o` Write output to file [*myfile.elf*].
- -I Search path for the .h-file. The path for the current folder is the period sign (.).
- -L Search path for the .a-file. The path for the current folder is the period sign (.).
- `-l` Search for the specified library to be linked. Note that there should be no space between the flag and library name.

Machine-specific options In order to perform compiling and linking, the target needs to be specified.

- `-mmcu=` This specifies the microcontroller as target.

Optimization There are a couple of options for optimizing the code.

- `-O0` Optimization is turned off. This is the default optimization flag passed to the compiler if the user has not chosen any optimization.
- `-O1` When this flag is selected, the size of the output binary code is larger and the execution speed is increased. This is done without considerably increasing the compilation time. `-O` is equivalent to `-O1`.
- `-O2` This option is selected when the compiler should optimize the code but not compromise space for speed, thus it increases the execution time (performance = execution time?) and potentially reduces the size of the output binary code. The compilation time is increased as well.
- `-O3` In contrast to the `-O2` flag, the compiler now optimizes the code for execution time even if the output binary code size is increased.
- `-Os` When this flag is selected, the compiler perform optimizations in order to reduce the size of the binary output code and does not take execution speed in consideration.

Click here to fast travel back to Lab Exercise 1.6.2, price: 1 bit.

Table B.1: Examples of target processors

atmega603	at90can128
at43usb355	at90usb1286
atmega103	at90usb1287
at43usb320	atmega128
at90usb82	atmega128a
at90usb162	atmega1280
ata5505	atmega1281
ata6617c	atmega1284
ata664251	atmega1284p
atmega8u2	atmega128rfr2
atmega16u2	atmega1284rfr2
atmega64hve	atmega2560
atmega64hve2	atmega2561
atmega64m1	atmega256rfr2
m3000	atmega2564rfr2

B.3 AVR `objcopy`

To extract the needed parts from the `.elf`-file and create a `.hex`-file, use the command below and a command prompt (make sure that the command prompt is set to the correct location). Click here to fast travel back to Lab exercise 1.6.1, price: 1 bit.

`avr-objcopy -j [`*section*`] -O [`*format*`] [`*myfile.elf*`] [`*myfile.hex*`]`

The following list summarizes a selection of flags used when compiling.

General options

- `-j` Copy only the named [*section*] from the input file to the output file. This option may be given more than once if more than one section is needed. Valid sections are `.text`, `.data`, `.bss`.
- `-O` Write the output file using the object format. For these labs, the format is `ihex`.

B.4 AVR `objdump`

To inspect your code with objdump, use the command below (make sure that the command prompt is set to the correct location). Click here to fast travel back to Lab Exercise 1.6.2, price: 1 bit.

`avr-objdump [`*option*`] [`*myfile.elf*`]`

The following list summarizes a selection of flags used when compiling.

General options

- `-d` Display assembler contents of executable sections.
- `-D` Display assembler contents of all sections.
- `-S` Intermix source code with disassembly. The source file needs to have been compiled with the `-g` flag.

B.5 AVR `size`

To view the size of the data segments use the command below.

`avr-size` [*myfile.elf*]

B.6 AVR `atprogram`

To program the microcontroller use the command below. Make sure that the command prompt is set to the correct location. Click here to fast travel back Lab exercise 1.6.1 , price: 1 bit.

`atprogram.exe -t [`*tool*`] -i [`*interface*`] -d [`*device*`] program -c -fl --verify -f [`*myfile.hex*`]`

The following list summarizes a selection of flags used when compiling.

General options

- `-t` Tool name, atmelice.

`-i` Physical interface, jtag.

`-d` Device name, atmega1284.

Program options

`-c` Chip erase.

`-fl` Program flash.

`--verify` Verify memory after programming.

`-f` File to be programmed.

Appendix C

Answers to Exercise Questions

C.1 Exercise 1

Click here to fast travel back to Section 1.5, price: 1 bit.

1.1 In a Von Neumann architecture data and code shares the same memory, whereas in a Harvard architecture they are separated.

1.2 From 2-5 GHz roughly

1.3 The AVR has 1 core. The latest Intel and AMD have around 8 cores, while the AMD Threadripper has 32 cores.

1.4 16 kiB RAM.

1.5 128 kiB of flash.

1.6 around 8-32 GiB

1.7 Running at 16MHz, it can execute 16 million instructions in 1 second.

1.8 125 ns.

1.9 The least significant byte at the lowest address, see Table C.1.

Table C.1: A number stored in little endian.

Addr	x	x+1	x+2	x+3
Value	0x01	0x67	0xFC	0x12

1.10 On an Intel or AMD, yes, but for AVR, no, due to the Harvard architecture.

1.11 1 byte.

1.12 2 bytes.

1.13 2 bytes.

1.14 0xFFFE

1.15 0xFFFE

1.16 9 bits. One extra for the carry, else overflow.

1.17 $\log_2\left((2^8-1)\cdot(2^8-1)\right) \approx 16$ bits.

1.18 6 variables are created, see Table C.2.

Table C.2: Variables with size and location.

Variable	**Size**	**Memory segment**
alpha	2	`.data`
vec	$3 \cdot 1$	`.data`
state	1	`.bss`
statham	1 (truncated)	`.data`
i	2	`stack`
looper	2 (not 20)	`stack`