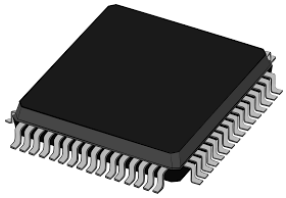
Digitala projekt

EITF12
Project manual



EITF12

Purpose of the document

- To understand what a microcontroller is
- To understand how to use a microcontroller
- To give a very brief introduction to C programming
- To give a summery of commonly used schematic symbols

Christoffer Cederberg
Jonathan Sönnerup
2021

Contents

1 Introduction to the AVR Microcontroller **2**
- 1.1 AVR Architecture . . . 2
- 1.2 Memory Layout of a Program . . . 3
- 1.3 AVR Toolchain . . . 5
  - 1.3.1 GCC - GNU Compiler Collection . . . 6
  - 1.3.2 avr-libc . . . 7
  - 1.3.3 The Atmel-ICE Debugger and atprogram . . . 7
- 1.4 Programming Basics . . . 9
  - 1.4.1 C Data Types in an AVR . . . 9
- 1.5 Exercises . . . 10

2 I/O Ports **12**

3 Timers **14**
- 3.1 PWM with an AVR Timer/Counter . . . 14
- 3.2 Fast PWM Mode . . . 15
- 3.3 Time measurement using the AVR Timer/Counter 1 . . . 18
- 3.4 Exercises . . . 19

4 USART - A Serial Communication Protocol **20**
- 4.1 Data register . . . 21
- 4.2 USART control and status . . . 21
- 4.3 USART Baud Rate Register `UBRR0H` and `UBRR0L` . . . 22
- 4.4 Exercises . . . 24

5 Volts to Bits - Analog-to-Digital Conversion **25**
- 5.1 ADC Configuration . . . 26
- 5.2 Exercises . . . 28

6 Answers to Exercise Questions **29**
- 6.1 Exercise . . . 29
- 6.2 Exercise 1 . . . 30
- 6.3 Exercise 2 . . . 31
- 6.4 Exercise 3 . . . 32
- 6.5 Exercise 4 . . . 33
- 6.6 Exercise 5 . . . 33

Chapter 1

Introduction to the AVR Microcontroller

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

Many of the peripheral units will be used during the this course, 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. The pinout of the ATmega1284 is shown in Figure 1.2.

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 1.3, 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 RC oscillator, by an external clock or with the help an external crystal.

[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.

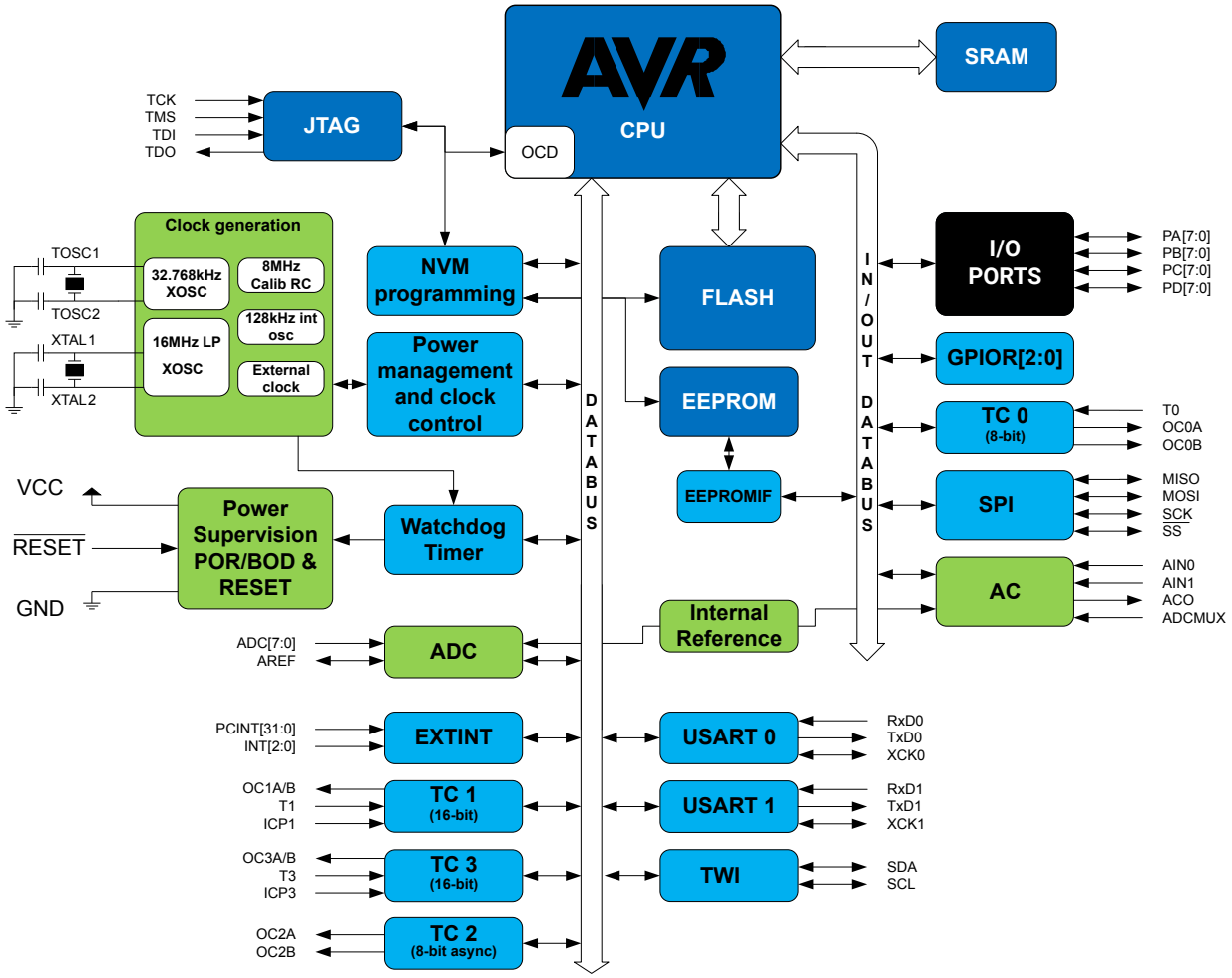


Figure 1.1: Block diagram for an ATmega1284.

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 1.4. 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

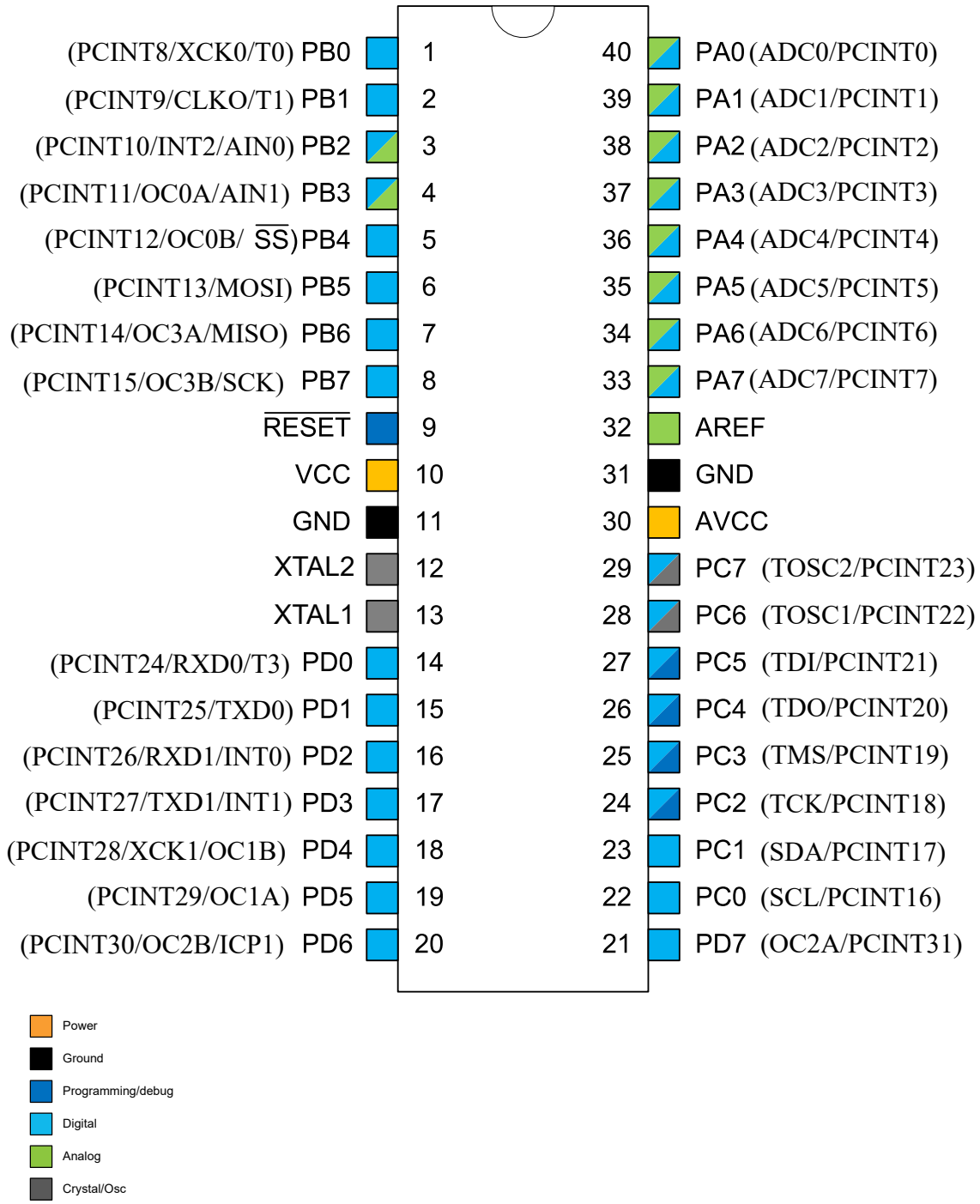


Figure 1.2: Pinout of the ATmega1284.

global or static variable is stored in the flash. The required space for the `.data` section is equal to the sum of

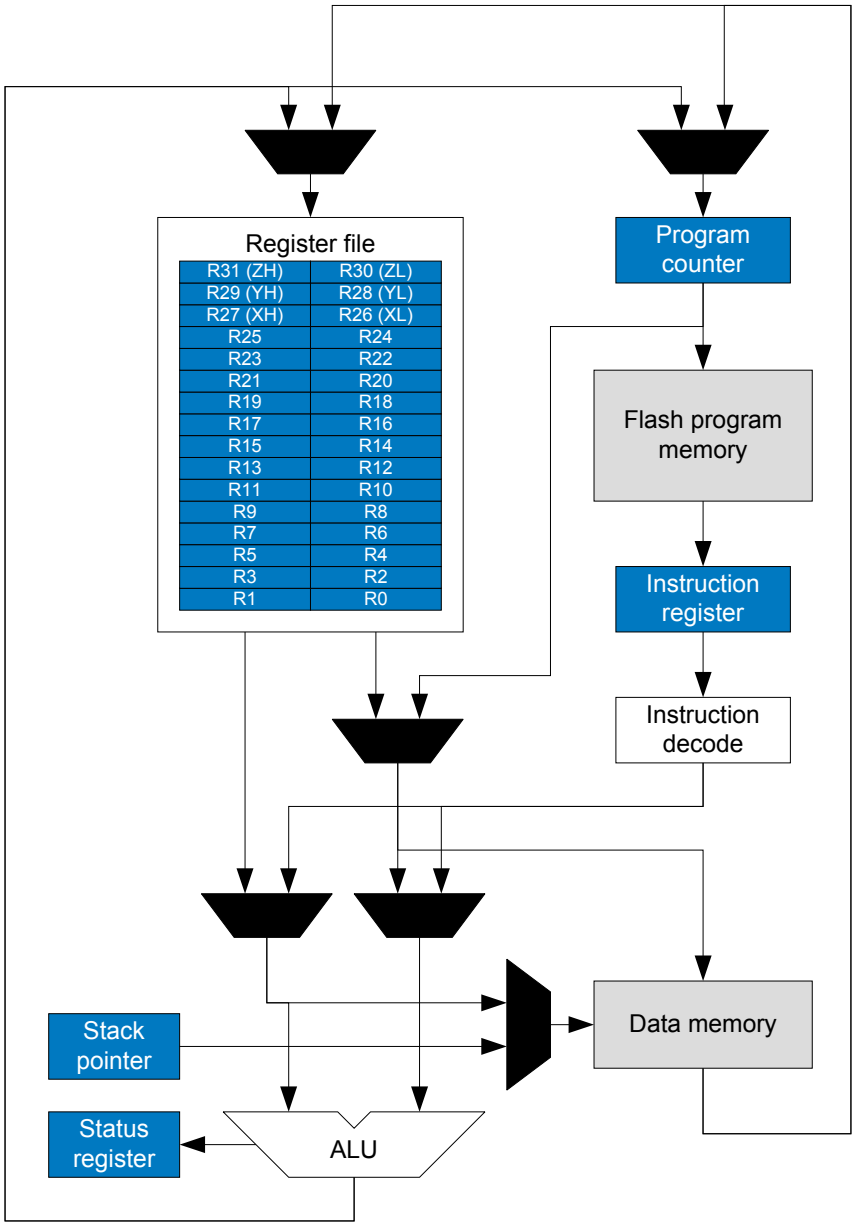


Figure 1.3: Block diagram for a AVR processor.

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 needed to

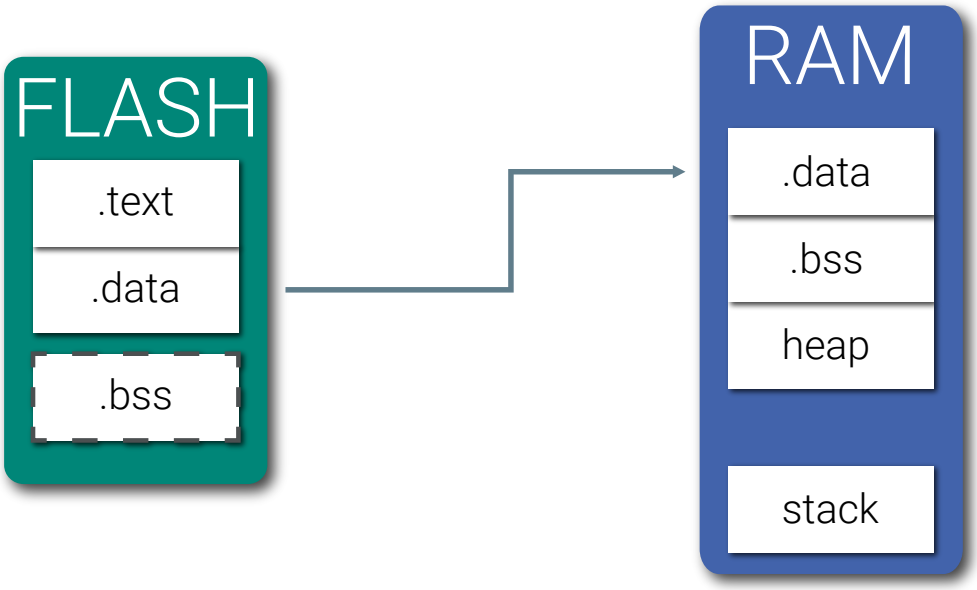


Figure 1.4: Memory layout and the content of the RAM and FLASH memory.

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. The assembler translates the assembly code to machine code, and the linker links all of the object files to a executable file.

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.

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 1.5, and a software tool called `atprogram` is used.

Figure 1.5: The Atmel-ICE programmer and debugger.

The Atmel-ICE programmer and debugger needs to be connected to the microcontroller, see Figure 1.6.

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 4.3.

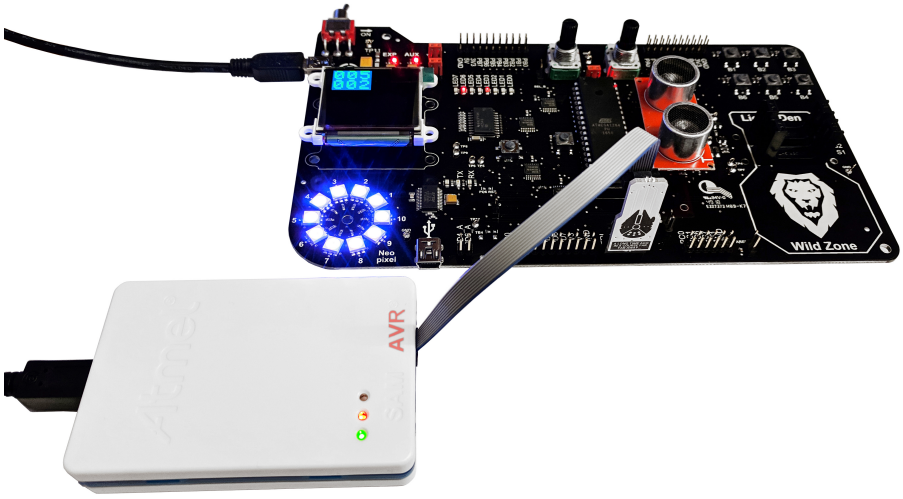


Figure 1.6: Connecting the debugger.

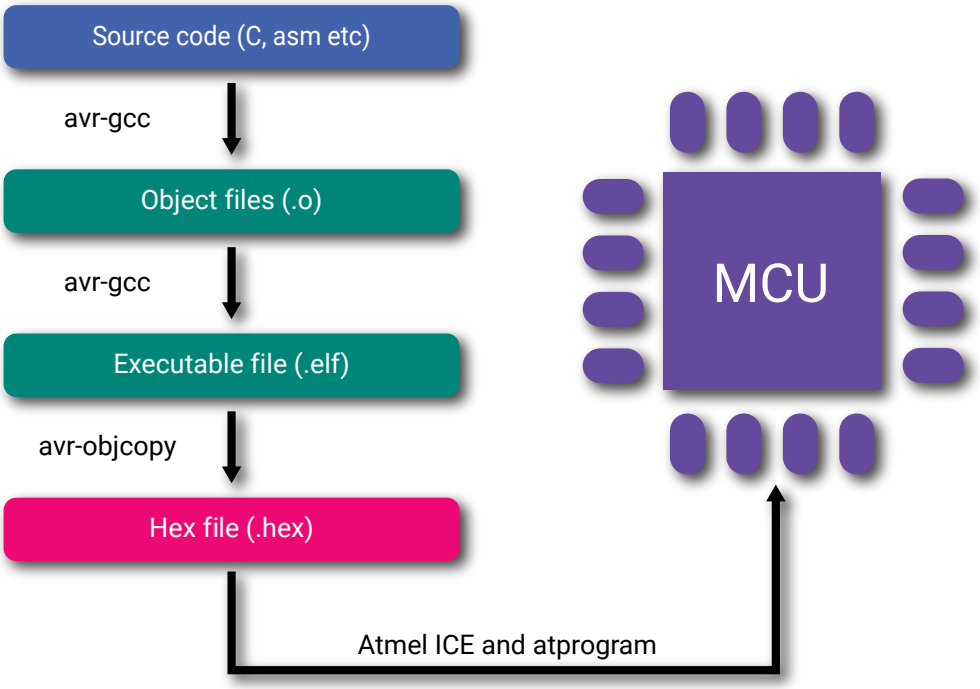


Figure 1.7: Summary of the steps from source code to transfer the binary to the microcontroller.

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 6.

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?

Chapter 2

I/O Ports

The microcontroller has 32 general purpose I/O pins. They are divided into four groups called ports. The name of the ports are Port A, B, C, and D. In Figure 1.2 the pin name is shortened PB0-PB7 for port B (the same applies to the other ports). With this piece of hardware it is possible to change the state (high or low, i.e. 5V or 0V) of an individual I/O pin. The I/O ports can also be configured to be an input.

Each port has three different registers. For Port A the name of these registers are `DDRA`, `PINA` and `PORTA`. Each port contains 8 pins and thus the corresponding registers have the same length. In Table 2.1 below, their functionalities are described.

Table 2.1: I/O port registers and their corresponding functionality.

Register	**Description**
DDRx	Specifies the data direction (input or output)
PINx	Read the state of the port, if it is an **input**
PORTx	Read and write to the port, if it is an **output**

To set the first pin on Port B, `PB0`, 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 `PB0` 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 important to use the logic operator, OR (|), instead of directly assigning a value. By using the "|" operator, a bitwise OR operation is performed with the content of the register and the value `0b00000001`. 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. To clear a bit the bitwise AND operator, "&", can be used in combination with the bitwise complement operator, "~".

In Listing 2.1 a simple program that toggles the state of I/O pin 3 on port B with a frequency of 0.5 Hz can be seen. The comment in the listing explains the steps to clear a bit.

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


int main()
{
    DDRA |= 0b00001000;

    while (1) {
        PORTA |= 0b00001000;
        _delay_ms(1000);
        PORTA &= ~(0b00001000);
        _delay_ms(1000);
    }
}


// How does PORTA &= ~(0b00001000); clear a the bit???
// Content of PORTB register:          0b00001000
// Bitwise complement of 0b00001000:   0b11110111
// Bitwise AND:                        0b00000000
```

Listing 2.1: A basic C program.

A common use-case for an I/O pin is to control a LED. See Figure 2.1.

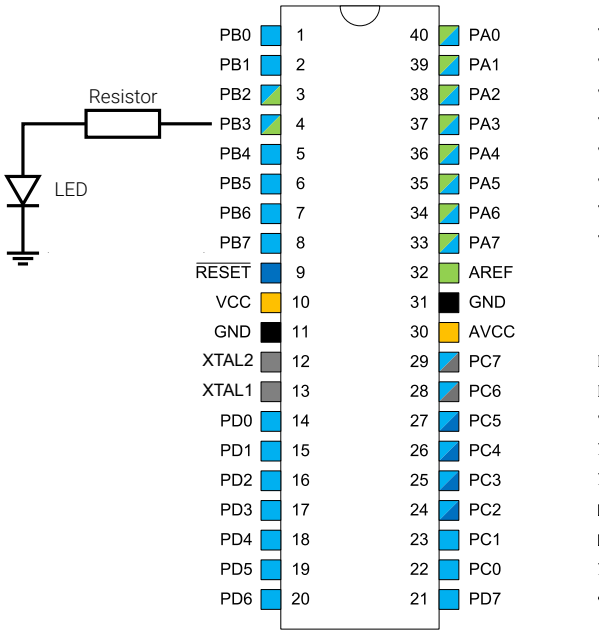


Figure 2.1: A LED and a resistor connected to PB3 on the microcontroller.

Chapter 3

Timers

A timer/counter is a hardware peripheral that essentially counts pulses. The most common used input to a timer is the CPU clock signal. The counter value will be incremented once every clock cycle. The timer/counter unit that is used in the AVR family contains a prescaler (among other things). With the prescaler the input to the timer, that is, the clock signal, can be divided with a selectable factor ranging from 1 to 1024. If the CPU clock frequency is 16 MHz results in that the time between a counter increment can range from 62.5 ns to 64 μs. The count limit of the timer is given by its word length. The Atmega1284 has two 8-bit and two 16-bit timers and thus, it can count from 0 to 255 or from 0 to 65535.

3.1 PWM with an AVR Timer/Counter

A pulse width-modulated (PWM) signal is a pulse train where the pulse width is modulated but the period is, in most cases constant. See Figure 3.1. The relationship between the active time and the period is referred to as the duty cycle, see Equation 3.1 below.

$$D = \frac{PW}{T}, \quad \text{where } D \text{ is the duty cycle, } PW \text{ is the pulse width and } T \text{ is the period} \tag{3.1}$$

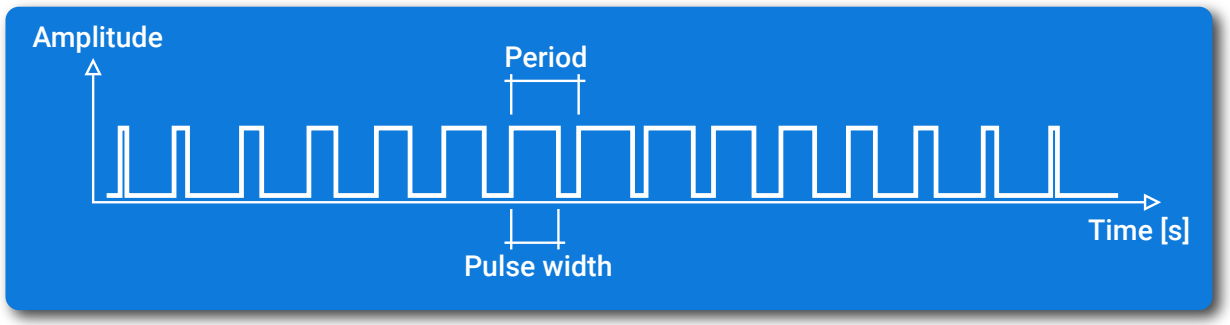


Figure 3.1: Pulse width modulation.

Such a signal can be used for a variety of things. For example it can be used to produce audio, controlling the rpm of a DC motor, or to adjust the intensity of an LED.

The timer/counter units can, as the section title implies, be used to produce a pulse width modulated (PWM) signal that will be available on certain I/O pins. Each timer/counter unit can control at least two I/O pins

individually. Each pin is called a channel. The channels are labeled `A`, `B`, and so on. The pins associated with the timer/counter PWM functionality are named `OC` (Output Compare) followed by a suffix that denotes which timer/counter unit and channel it is connected to. For timer/counter 3 Channel `A` is named `OC3A` and thus `OC3B` for Channel `B`.

The registers associated with timer/counter 3 can be seen in Table 3.1. Not all of them are need during the laboratory exercises.

Table 3.1: Timer 3 registers.

Register	Description
`TCCR3A`	Control Register A
`TCCR3B`	Control Register B
`TCCR3C`	Control Register C
`TCNT3`	Counter Value
`ICR3`	Input Capture Register 3
`OCR3A`	Output Compare Register A
`OCR3B`	Output Compare Register B
`TIMSK3`	Interrupt Mask Register
`TIFR3`	Interrupt Flag Register

For detailed information, please refer to page 187-198 in the data sheet.

3.2 Fast PWM Mode

There are several varieties of a PWM signal that can be generated with an AVR timer/counter. Here is a description on how to initialize timer/counter 3 to generate "fast PWM" with `ICR3` as top. For that purpose an output compare register, `OCR3A` or `OCR3B`, is used together with the counter value register. The desired behaviour is as follows. When the counter value register is zero, the selected `OCR3` pin is set to high. As the timer increments the value, it is constantly compared with the value of the output compare register. When they match, the `OCR3` pin is cleared (set to low). When the counter value is the same as the value in register ICR3 the timer/counter overflows and the output pin is set to high. After this the procedure starts again. See Figure 3.2 and 3.3.

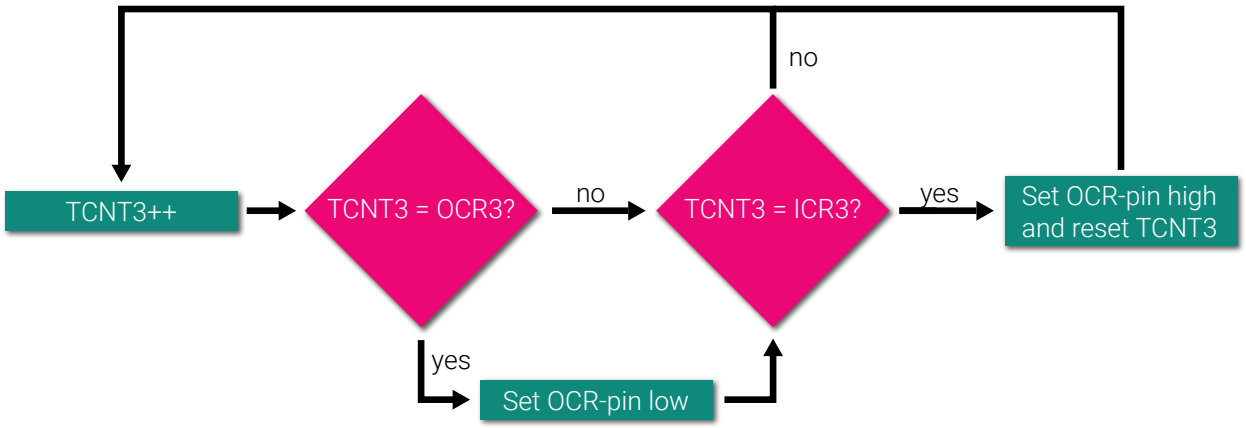


Figure 3.3: Flowchart describing how the timer/counter creates a "fast PWM" signal.

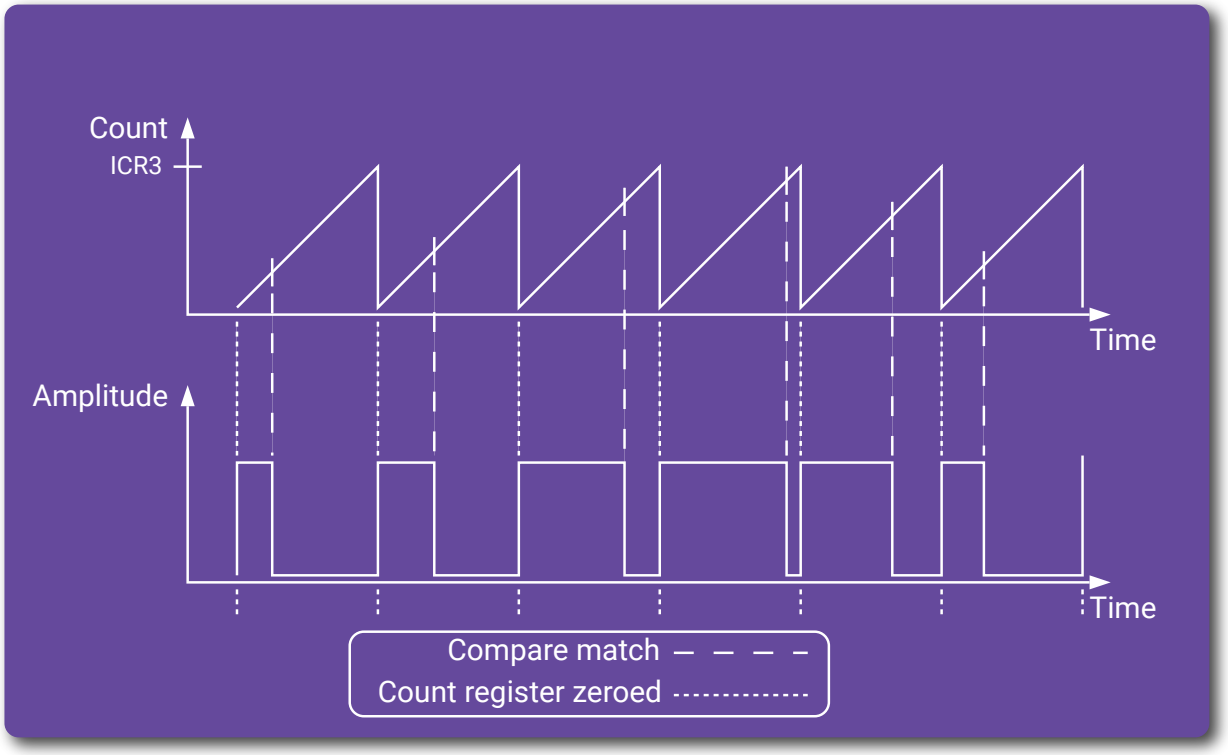


Figure 3.2: Pulse width modulation with an AVR timer/counter.

By following the steps below the timer/counter unit will be initialized as described. The timer/counter registers can be seen in Table 3.2.

- In control register A, `TCCR3A`, the Compare Output Mode for Channel A is selected by bit `COM3A1` and `COM3A0`. For Channel B the corresponding bits are `COM3B1` and `COM3B0`. The bits should be set so that the `OC3A` or `OC3B` pin gets cleared on a compare match. For more details, please refer to Table 17-9 for B on page 187-189 in the data sheet. Both channels can be found on Port B. For this reason it does not matter which one you choose, since all I/O pins are connected to LEDs. The two I/O pins that are connected to Channel A and B are highlighted with red in Figure **??**.
- Select Fast PWM with ICR3 as top by setting the Waveform Generation Mode Bits `WGM30`, `WGM31`, `WGM32` and `WGM33` accordingly. See Table 17-11 on page 188-189 in the data sheet. Having ICR3 as top means that at this value the timer will restart, that is, clearing the counter register and start from zero again. See Figure 3.2. The Waveform Generation Mode Bits can be found in control register A and B.
- Configure the data direction register so that the `OC3A` or `OC3B` becomes an output.
- Set the prescaler bits, `CS30, CS31` and `CS32` in the timer/counter control register `TCCR3B` so that an appropriate division factor is used. See Table 17-12 on page 190-191 in the datasheet.

Table 3.2: Timer 3 registers.

Register	Description
TCCR3A	Control Register A
TCCR3B	Control Register B
TCCR3C	Control Register C
TCNT3	Counter Value
ICR3	Input Capture Register 3
OCR3A	Output Compare Register A
OCR3B	Output Compare Register B
TIMSK3	Interrupt Mask Register
TIFR3	Interrupt Flag Register

3.3 Time measurement using the AVR Timer/Counter 1

In order to measure time with the timer/counter, the prescaler bits (`CS10, CS11` and `CS12`) need to be set. They are found in the timer/counter control register `TCCR1B`. These bits control how fast the timer/counter will increment the counter value register, that is, `TCNT1`. For time measurement, this is all that has to be done to start the timer, that is, the counter value register will be incremented at every pulse from the prescaler. Before a time measurement is done, the counter value register needs to be set to a known value (preferably zero). To stop the timer from incrementing the counter value register, the prescaler bits should be set to zero. This action disconnects the clock signal from the timer prescaler.

3.4 Exercises

Answers to the questions can be found in Appendix 6.4.

Signals

3.1 You have a continuous stream of pulses. The pulses are high (5V) for 20 ms, and low (0V) for 60 ms.

(a) What is the period time of the signal?

(b) From above, what is the frequency of the signal?

(c) From above, what is the duty cycle?

3.2 The RMS value is defined as the amount of AC (alternating current) power that produces the same effect as DC (direct current) power. For an AC signal, $u(t)$, it can be calculated as,

$$u_{RMS} = \sqrt{\frac{1}{T}\int_0^T u(t)^2 dt},$$

where T is the period. Use the above formula to calculate the RMS value for the pulse signal from the above exercises.

3.3 What is the RMS value for a general square wave with amplitude V_p and duty cycle D? Use the integral formula above.

3.4 Suppose you have a heater at home that you want to control using PWM. You measured the voltage to the heater when it has a comfortable temperature. The measured value is 10V DC. You have a relay capable of handling 25V, which you can control using PWM. What do you need to set the duty cycle to in order to get the same power to the heater, thus the same temperature?

Chapter 4

USART - A Serial Communication Protocol

A microcontroller is often used to gather some sort of information and then send it to a computer for analysis. Another typical application is that a computer send commands to a microcontroller, which then performs an action that corresponds to the received command. This can be done with the USART peripheral device inside the microcontroller. The abbreviation USART stands for Universal Synchronous Asynchronous Receiver and Transmitter. With this device, the data is sent bit by bit. The transfer rate, that is, bits/s, is referred to as the baud rate. Each data package consists of one start bit, a number of data bits, a parity bit (which is optional), and one or two stop bits. See Figure 4.1.

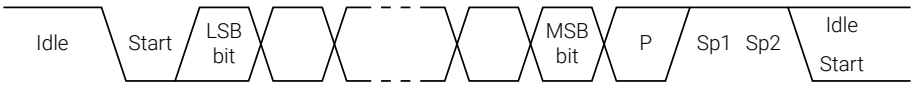


Figure 4.1: UART data package.

There are two versions of this type of communication, one is asynchronous and one is synchronous. The synchronous version requires that a clock signal is connected between the two devices. The more common is the asynchronous version, which is usually referred to as UART (Universal Asynchronous Receiver and Transmitter). This is the version that is going to be used during the laboratory exercises. Each UART device consists of a transmitter and a receiver. The receiver is often labeled Rx and the transmitter Tx. See Figure 4.2.

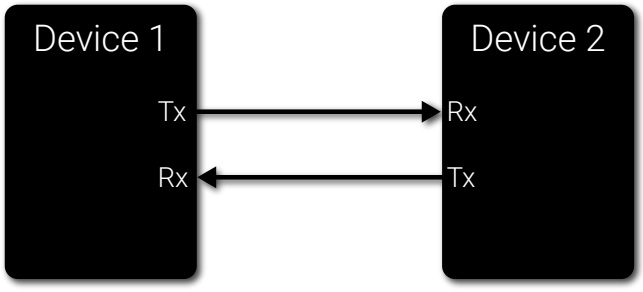


Figure 4.2: UART communication between two devices.

Before the communication starts, the devices needs to be configured, that is, the mode of operation, baud rate, the number of data bits, stop bits, and parity[1] mode has to be selected. This is done by setting bits in the status and control registers of the USART device that corresponds to these options. The registers of the USART device for the used microcontroller can be seen in Table 4.1. For a detailed version, please refer to page 257-265 in the data sheet.

Table 4.1: Registers of the USART 0 unit.

Register	Description
UDR0	USART I/O Data Register 0
UCSR0A	USART Control and Status Register 0 A
UCSR0B	USART Control and Status Register 0 B
UCSR0C	USART Control and Status Register 0 C
UBRR0L	USART Baud Rate 0 Register Low byte
UBRR0H	USART Baud Rate 0 Register High byte

4.1 Data register

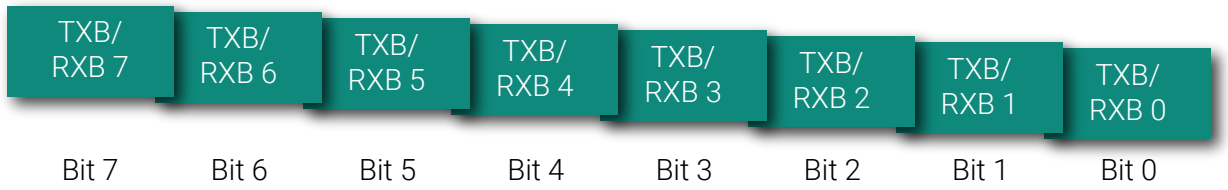


Figure 4.3: UART data Register.

The USART Transmit Data Buffer Register and USART Receive Data Buffer Registers share the same I/O address referred to as USART Data Register or UDRn[2]. The Transmit Data Buffer Register (TXB) will be the destination for data written to the UDR1 Register location. Reading the UDRn Register location will return the contents of the Receive Data Buffer Register (RXB).

4.2 USART control and status

Control and status register A - UCSR0A:
Only two bits in this register is important for the laboratory exercises, UDRE0 and RXC0. They can be used while transmitting and receiving data.

Control and status register B - UCSR0B:
The only important bits in this register are the bits that enable the USART transmitter and receiver.

Control and status register C - UCSR0C:
The default value of UMSEL0[1:0], UPM0[1:0], USBS0 and UCSZ0[2:0] initializes the USART unit as following:

- USART mode is asynchronous,

[1] The parity is used as an error detection feature, but it will not be used during the lab exercise.
[2] When coding, the n in UDRn should be replaced with 0 since it is USART0 that will be used. This goes for all names containing an n.

- parity mode is disabled,
- one stop bit and
- the frame size is 8-bits.

During the laboratory exercises there is no need to change this configuration.

4.3 USART Baud Rate Register `UBRR0H` and `UBRR0L`

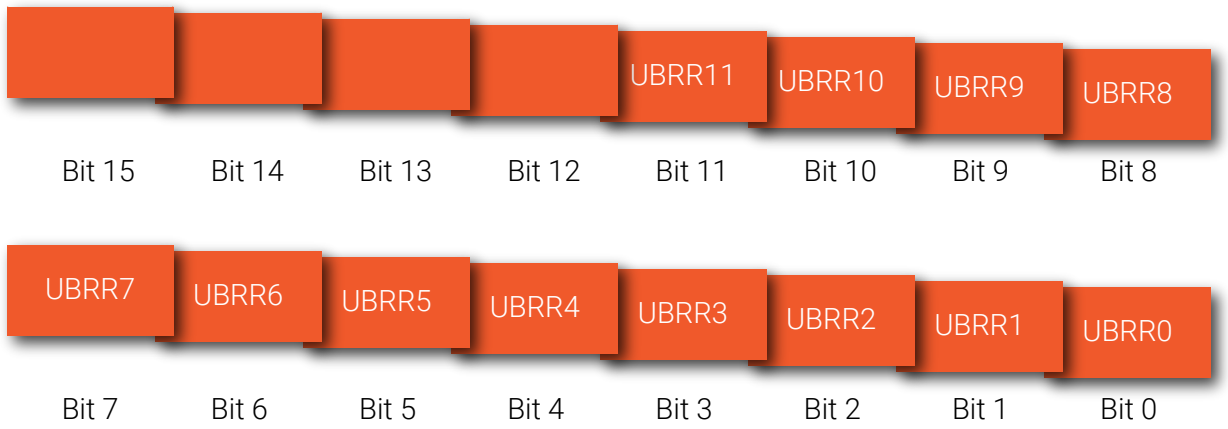


Figure 4.4: UART data Register.

Bits 11:0 – `UBRR011:0`: USART Baud Rate This is a 12-bit register which contains the USART baud rate. The `UBRR0H` contains the four most significant bits and the `UBRR0L` contains the eight least significant bits of the USART n baud rate. Ongoing transmissions by the Transmitter and Receiver will be corrupted if the baud rate is changed. Writing `UBRR0L` will trigger an immediate update of the baud rate prescaler.

Baud Rate [bps]	**fosc = 16.0000MHz and U2X = 0**	**Error rate**
2400	416	-0.1%
4800	207	0.2%
9600	103	0.2%
14.4k	68	0.6%
19.2k	51	0.2%
28.8k	34	-0.8%
38.4k	25	0.2%
57.6k	16	2.1%

To write a value to both `UBRR0L` and `UBRR0H` "simultanously" use the macro in the listing below.

```
UBRR0 = 103;
// alternatively
UBRR0H = (103 >> 8);
UBRR0L = 103;
```

Listing 4.1: Test.

4.4 Exercises

Answers to the questions can be found in Appendix 6.3.

Communication

4.1 The maximum transfer speed in the Atmega is 2.5 Mbit/s. What is the fastest transfer speed of the SATA protocol used for SSD drives in a modern computer? What about USB 3.0?

4.2 Using UART with 1 start bit and 1 stop bit, sending data 8 bits, how much overhead do you get?

4.3 Using the UART above, transmitting bits at 1 Mbit/s (including start and stop bits), what is the effective transmission rate (excluding control bits)?

4.4 You recently bought a remote switch for controlling the lights and the TV in your apartment. Being an engineer, you find it a tad boring controlling it manually, hence you decide to use a microcontroller (MCU) to do it for you. You attach a WiFi circuit both to the MCU, and the switches. The WiFi circuits use a serial protocol (UART) for transmitting data. You have the following functions at your disposal. `set_baud(char val)` takes a `char` argument, `val`. The value of `val` is calculated as

$$\text{val} = \left\lfloor \frac{f_{osc}}{16 \cdot \text{baud rate}} - 1 \right\rfloor .$$

The function `set_control(char val)` takes an argument, `val`. Bit 0 determines the number of parity bits (0 - 1), bit 1 determines the number of stop bits (1 - 2), and bits 2 to 4 determine how many data bits (5 - 9) are used. 5 data bits are encoded as "000" and so on. Bits 5 - 7 are not used. The last function is `send_char(char c)` that sends a single character over WiFi using UART.

To control the lights and the TV, you send an identifier along with a value in the following format:

`L1:<on/off>`, for light 1
`L2:<on/off>`, for light 2
`TV:<channel>`, for the TV

For example, sending

"TV:5"

turns on the TV and changes it to channel 5. Send a zero (0) to turn the TV off.

There are two global variables, `hours` and `minutes` that you may use. You want the following functionality:

- At 06:30, turn on light 1 and the TV on channel 4
- At 07:15, turn off the TV, and turn on light 2
- At 07:45, turn off all lights
- At 12:00, turn on light 2
- At 17:30, turn on the TV on channel 6, and both lights
- At 22:00, turn off everything

The UART communicates at a baud rate of 9600 using 1 parity bit, 8 data bits, and 1 stop bit. The MCU is running at 16 MHz.

Chapter 5

Volts to Bits - Analog-to-Digital Conversion

An analog-to-digital converter (or ADC) is, as the name implies, used to convert a analog signal[1] to a digital value that represents its magnitude. A key property of an ADC is the resolution. The resolution determines how many quantization levels the magnitude of the analog signal can be encoded in. A digital value from ADC with the resolution of 10-bits can range from 0 to 1023 ($2^{10} - 1$). The resolution can also be directly translated to a voltage. With a 10-bit resolution and a reference voltage[2] at 3.3 V each bit equals 3.22 mV ($\frac{3.3V}{2^{10}}$). A conversion is often repeated, at equidistant time steps, to form a discrete-time and discrete-amplitude digital signal, see Figure 5.1. The rate at which the signal is converted, or sampled at, is referred to as the sampling rate or sampling frequency.

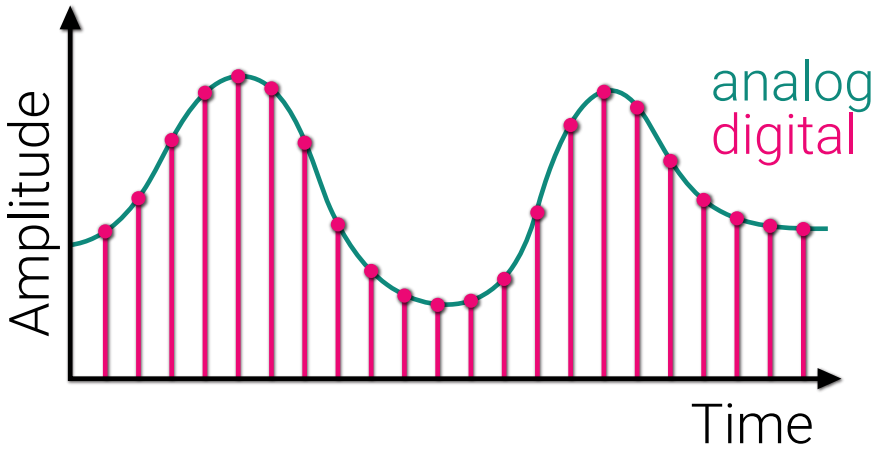


Figure 5.1: An illustration of an analog signal and its digital counterpart.

The reference voltage of the ADC inside the AVR can be set to a variety of things. A common source is AV_{CC}. This is the power supply to the analog parts of the microcontroller. In this case, it is the same as V_{CC}, which is the power supply to the digital part and is equal to 3.3 V. The separation between the two

[1] A continuous-time and continuous-amplitude signal. An example of an analog signal could be audio signal picked up by a microphone or the output voltage from a potentiometer.

[2] The highest allowed voltage that can be converted.

power supplies is done to reduce noise on the ADC value. Almost all digital circuits are considered to be noisy.

See Table 5.1 for the ADC status, control and data registers.

Table 5.1: ADC registers.

Register	Description
ADMUX	ADC Multiplexer Selection Register
ADCSRA	ADC Control and Status Register A
ADCL	ADC Data Register Low Byte
ADCH	ADC Data Register High Byte
ADCSRB	ADC Control and Status Register B
DIDR0	Digital Input Disable Register 0

For more information regarding the associated registers, please refer to page 330-339 in the data sheet.

5.1 ADC Configuration

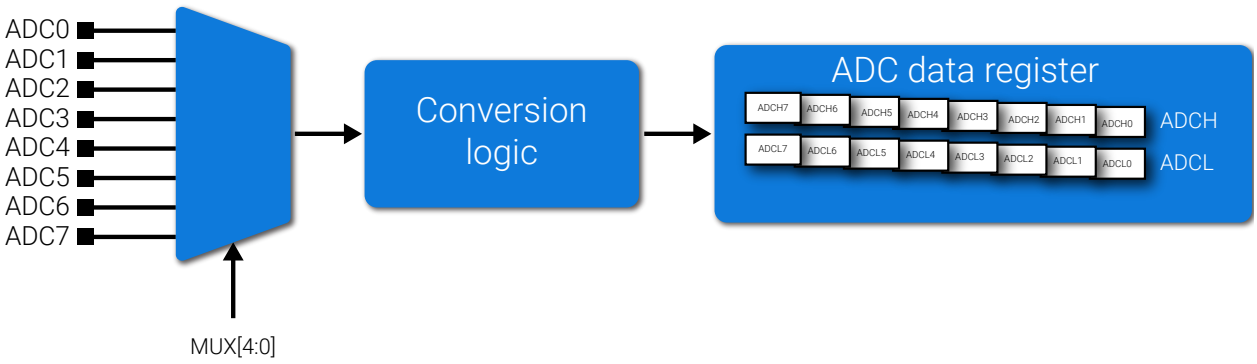


Figure 5.2: Pulse width modulation with an AVR timer/counter.

By completing the bullet points below the ADC will be configured.

- Enable the ADC by setting the enable bit in the ADC Control and Status Register A, `ADCSRA`.
- Furthermore the ADC prescaler should be set to produce a signal (which drives the ADC unit) that has a frequency between 50kHz - 200KHz. This will ensure that the ADC unit performs the conversion with good reliability. See Table 25-5 on page 334 and 335
- In the ADC Multiplexer Selection Register, `ADMUX`, set the `MUX`-bits to select the desired channel. See Table 25-4 on page 332. The ADC should be used in the Single Ended Input configuration.
- In the same register, `ADMUX`, choose the AV_{CC} as the reference voltage. This is done with the `REFS1` and `REFS0` bits.

- With the `ADLAR` bit in `ADMUX` register it is possible to choose how the converted ADC value will be stored in the result register, `ADCH` and `ADCL`. For details see page 336-337. By setting this bit to zero, the most significant bit is placed at index 1 and the next most significant at 0, in the high byte of ADC result register `ADCH`. The remaining part is placed in the low byte of the ADC result register. The way the data is aligned is called right adjusted. See Figure 5.3.

Figure 5.3: The ADC value is stored in the result registers.

- To start a conversion, set the `ADSC` bit to one. This bit is found in the ADC Control and Status Register A, `ADCSRA`. When this is done, the ADC unit performs a conversion. This will take several CPU clock cycles. The result should not be read before the conversion has been completed. The `ADSC` bit (the same bit that starts a conversion) can be used to check if the conversion has completed. Use a loop that breaks if the `ADSC` bit is set to zero. When the loop breaks, it is safe to read the data from the ADC Data Register, `ADC`, and another conversion can be started.

Table 5.2: ADC registers.

Register	**Description**
ADMUX	ADC Multiplexer Selection Register
ADCSRA	ADC Control and Status Register A
ADCL	ADC Data Register Low Byte
ADCH	ADC Data Register High Byte
ADCSRB	ADC Control and Status Register B
DIDR0	Digital Input Disable Register 0

5.2 Exercises

Answers to the questions can be found in Appendix 6.5.

5.1 Assume you have signal with 5V amplitude. Assume you AD convert the signal, using a 5V reference and 10 bit resolution.

(a) How many millivolts (mV) per bit do we get?

(b) What is the largest conversion error (in mV)?

5.2 Given the following signal,

$$v(t) = 5\sin(\frac{15}{11}\pi t),$$

at time $t = 2$, what is the AD converted value (0 - 1023) with 10 bit resolution?

5.3 The Amazon drone from before seems to be crashing form time to time, destroying expensive piece of equipment. To mitigate the damages, you installed an accelerometer, ADXL335, along with a parachute in order to detect when the drone is about to crash to save it from breaking.

The accelerometer has an x, y and a z channel to measure movement in 3D. You have a function to get the value from an 8-bit ADC (3 V reference), `uint8_t read_adc(uint8_t channel)`, where the channel value is 0 to 2 for the x to z channels. The accelerometer is powered with 3 V, with a "zero g bias level" (no acceleration) of 1.5 V on all channels. Then, for every additional g force (up to 3g), the voltage increases with ±300 mV. That is, for every channel, the voltage ranges from $[1.5 - 0.9, 1.5 + 0.9] = [0.6, 2.4]$.

(a) How many distinct values can we measure with the ADC?

(b) What voltage does the maximum ADC value correspond to?

(c) What is the voltage range, per channel, from the accelerometer if we remove the constant offset?

(d) Experiments show that a resultant vector with a magnitude of 0.346 (no offset) results in a crash. Recall the Pythagorean theorem in 3 dimensions,

$$\sqrt{x^2 + y^2 + z^2} = d,$$

implement a program that calls the function `release_parachute()`, if magnitude exceeds 0.346. Note that the square-root function is expensive in a microcontroller and should be avoided if possible.

Chapter 6

Answers to Exercise Questions

6.1 Exercise

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 6.1.

Table 6.1: 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 6.2.

Table 6.2: 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`

6.2 Exercise 1

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

2.1 13

2.2 0

2.3 8

2.4 6

2.5 0

2.6 15

2.7 251

2.8 223

2.9 `char val = *((char *) 0x1337)`

2.10 `char val = *((volatile char *) 0x1337)`

2.11 `*((char *) 0x0666) = 42`

2.12 `int *addr = &a`

2.13 All outputs, `DDRG |= 0xFF`

2.14 Read **and** write from/to PORTG, since it is an output.

2.15 `char val = (PINH & 0xF0) >> 4`

2.16 `char val = ((PINH & 0xF0) >> 4) | (PINH & 0xF0)`

2.17 See code in Listing 6.1.

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

uint8_t release;

int main()
{
    DDRM |= (1 << 3); // pin 3 as output

    while (1) {
        if (release) {
            // send pulse to keep hatch open
            PORTM |= (1 << 3);  // pin high
            _delay_ms(2);
            PORTM &= ~(1 << 3); // pin low
            _delay_ms(18);
        } else {
            // send pulse to keep hatch closed
            PORTM |= (1 << 3);
            _delay_ms(1);
            PORTM &= ~(1 << 3);
            _delay_ms(19);
        }
    }
}
```

Listing 6.1: Program for controlling the drone hatch.

6.3 Exercise 2

Click here to fast travel back to Section 4.4, price: 1 bit.

3.1 SATA 3.2 has a maximum speed of 16 Gbit/s, while USB 3.1 reaches 10 Gbit/s.

3.2 20% overhead

3.3 800 kbit/s or 100 kB/s

3.4 See code in Listing 6.2.

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

void send_message(char *);

uint8_t hours;
uint8_t minutes;

int main()
{
    set_baud(103);
    set_control(0x0D);

    while (1) {
        if ((hours == 6) && (minutes == 30)) {
            send_message("L1:on");
            send_message("TV:4");
        } else if (hours == 7) {
            if (minutes == 15) {
                send_message("TV:off");
                send_message("L2:on");
            } else if (minutes == 45) {
                send_message("L1:off");
                send_message("L2:off");
            }
        } else if ((hours == 12) && (minutes == 00)) {
            send_message("L2:on");
        } else if ((hours == 17) && (minutes == 30)) {
            send_message("TV:6");
            send_message("L1:on");
            // light 2 is already on
        } else if ((hours == 22) && (minutes == 0)) {
            send_message("TV:off");
            send_message("L1:off");
            send_message("L2:off");
        }
    }
}

void send_message(char *msg)
{
    // loop until end of string
    while (*msg != '\0') {
        // send a char and move on to the next
        send_char(*msg++);
    }
}
```

Listing 6.2: Program for controlling lights and the TV.

6.4 Exercise 3

Click here to fast travel back to Section 3.4, price: 1 bit.

4.1 (a) T = 80 ms

(b) f = 12.5 Hz

(c) $D = \frac{20}{80} = 0.25$

4.2 The function is not continuous, hence we must separate the integral as

$$u_{RMS} = \sqrt{\frac{1}{0.08}(\int_0^{0.02} 5^2 dt + \int_{0.02}^{0.08} 0^2 dt)} = \sqrt{\frac{1}{0.08}\int_0^{0.02} 25 dt}$$
$$= \sqrt{\frac{1}{0.08}\left[25t\right]_0^{0.02}} = \sqrt{\frac{1}{0.08}\left[25 \cdot 0.02 - 25 \cdot 0\right]}$$
$$= \sqrt{\frac{25 \cdot 0.02}{0.08}} = \sqrt{6.25} = 2.5V$$

4.3 In the general case, the RMS value for a square wave is calculated to be

$$u_{RMS} = V_p\sqrt{D}.$$

4.4 Using the general formula, we get

$$u_{RMS} = V_p\sqrt{D},$$

where $V_p = 25\text{V}$ and $u_{RMS} = 10\text{V}$. Solve for D, and we get $D = (\frac{10}{25})^2 = 0.16$. The duty cycle should be around 16%.

6.5 Exercise 4

Click here to fast travel back to Section 5.2, price: 1 bit.

5.1 (a) $\frac{5\text{V}}{2^{10}} \approx 4.88\text{mV}$

(b) The largest error is half the value per bit, 2.44 mV.

5.2 The voltage at time $t = 2$ is

$$v(2) = 5\sin(\frac{15}{11}\pi \cdot 2) = 3.778\text{V},$$

resulting in a value of

$$\frac{3.778}{0.00488} \approx 774.$$

5.3 Answer

(a) $2^8 = 256$

(b) $\frac{3\text{V}}{256} \cdot 255 = 2.988\text{V}$

(c) Removing the 1.5 V offset, we get $[-0.9, 0.9]$

6.6 Exercise 5

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

6.1 (a) Each clock cycle takes 1 μs. 64 ms is 64000 clock cycles.

(b) We are counting mod 2^{16}, hence the timer value is $67000 \mod 2^{16} = 1464$. We solve this by another counter increasing every timer overflow.