An Introduction to programming an Atmega microcontroller

Benjamin Reh

November 14, 2013

Contents

1. Preface

This document an introduction into the programming of an Atmega microcontroller. It is separated into the first part guiding like a tutorial for beginners and a second part which is a reference book to the functions provided in the `basis`.

The examples and explanations provided are neither exhaustive nor complete. The only aim of this document is to lower the burden of getting started. Only a basic knowledge in `C` is required.

2. Compilation and transfer

2.1. Preparation

Before starting to write own programs, it is advisable to first familiarize with the tool-chain which will be used.

Prerequisites

To compile your programs and transfer them to the microcontroller, a few tools have to be installed. If you are working on a PC in the Lab, then you can skip this section.
First, I assume you are working with Linux, so I also assume you already have your favorite editor. If you do not, I recommend `geany`, which is a light, but powerful editor using gnome.

For the Atmega you need the avr port of the gcc, a program for uploading the code to the device (`avrdude`) and `make` for running Makefiles.
Everything can be installed on Debian/Ubuntu systems with the following command

```
sudo apt-get install make avr-libc avrdude binutils-avr gcc-avr gdb-avr
```

For other distributions please take a look inside your documentation how to install new packages and how to find their names.

2.1.1. Basis

For a clean start you should get a fresh copy of the basic programming environment we call *basis*. You can either check it out from the svn-repository with

```
svn checkout http://ornella.iwr.uni-heidelberg.de/svn/basis
```

or download it from `http://roboter.uni-hd.de`. After downloading it, you will need to extract the files from the archive of course.
Change the directory (`cd basis`) and start with your first steps.

2.1.2. First steps

Just type

```
make
```

to compile and get an output similar to this

```
Compiling main.c ...
Compiling adc.c ...
Compiling timer.c ...
Compiling pwm.c ...
Compiling uart.c ...
Compiling servo.c ...
Linking basis.elf ...
Creating basis.hex ...
Creating basis.eep ...
avr-objcopy: --change-section-lma .eeprom=0x0000000000000000 never used
```

If you encounter any problems or errors with this, you should verify your installation of the toolchain.

The `make` command produces a binary output file called `basis.elf` which now needs to be transferred to the microcontroller. You should supply power to the board and plug in the programming device.
By typing

```
make prog
```

a program called `avrdude` will be started to initiate the transfer.
Since there is more than just one possible programmer, maybe the `Makefile` has to be adjusted to the one you are using.

```
...
#select programmer
prog: prog-usb
#prog: prog-ser
#prog: prog-funk

...
```

In this example the usb-programmer is chosen because it is the only line that does not start with a `#`.

If you encounter problems that look like a problem with permissions you can try executing the command as root with for example `sudo make prog`. For being able to program as a normal unprivileged user, you can modify your udev-rules accordingly. In the lab this has already been done. This problem should therefor not occur.

2.1.3. Fuse-bits

A really essential part of initializing a new controller is setting the so called fuse-bits. You should take a look at the board's documentation how to do that in detail or visit `http://www.engbedded.com/fusecal`.
From this point of view it is sufficient to just call

```
make fuse
```

once every time you put a new controller chip in your board. If your are in doubt whether your controller has already been fused, just it again. Fusing more than once doesn't hurt.

3. The basics

In this chapter you learn what registers and ports are and how to set a pin on a certain port.

3.1. Pins and Ports

Pins

The word *pin* can be referred to as the metal connectors on the package of the controller chip. While some of those pins can be freely used within your programs, some have fixed purposes for example to supply voltage. There are also some pins, which are both: They normally have a distinctive purpose, but in some applications can also be used freely.[1]

The electric connector pins that are not configurable cannot be accessed from within the software.
So whenever the word *pin* is used in the following text, the ones that are accessible from inside the software are meant.
Generally every (software) pin can be used as a digital input or output. This means that it can either "listen" to the signal that is supplied to the pin or provide a signal itself. These two behaviors are called the *Data Direction*.

Ports

As the reader may recall from lectures in computer engineering[2], a processor has a set of registers which contain data or control the peripheral devices. A microcontroller typically is nothing more than a microprocessor with memory, some special hardware (counter, ADCs,...) and direct access to IO.
The Atmega series we are using has a 8 bit architecture. It can calculate values of the size of 1 byte at normally one clock cycle.
For this reason *8 pins* are combined to *1 port*, which simply is represented by one register in the CPU.
Ports are enumerated with capital letters (A,B,C,...), pins with number from $0 \ldots 7$. For example `PIND6` represents the pin no. `6` on port `D`
Each bit in a register for IO represents a (physical) pin. This makes it necessary to know how to manipulate just several bits at a time while leaving the others untouched.
If your are not familiar with bit shifting and bit algebra in expressions like `a |= (1<<5);` you can take a look at A.1 to learn the basics.

4. First steps

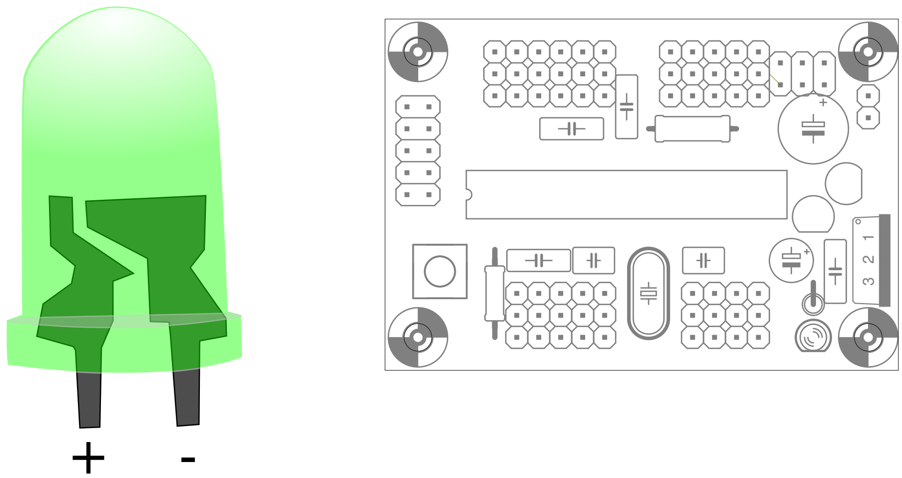
This chapter elaborates about setting an output, reading an input and programming simple time depending tasks. If you are not familiar with the bit expressions used in this section feel free to take a look at A.1.

4.1. Controlling a LED

4.1.1. Plugging in the LED

In the lab we provide simple LEDs with integrated resistors soldered on a connector to be directly plugged onto the board.

[1]An example is the reset pin which can also be used for normal IO. This tutorial keeps on the usage that is recommended by an unchanged board with unchanged software.
[2]Technische Informatik, typically Informatik II



(a) Hold LED against light to see the polarity

(b)

Figure 1: LED and board

To find out the polarity of the LED, it can simply he held against the light, see 1(a). The *smaller* piece inside the plastic housing is the anode (+). It will be plugged on the board on the pin next to the microcontroller. The controller will provide its signal there.
The other pin, the cathode (-), is plugged to the pin which is the most far away from the controller. This pin is on ground. If you do not have a board, take a look at 1(b).

The LED will be placed on `PORTC5`, which is located in the corner of the MC at pin no 28. Take a look at A.2.

4.1.2. Switching it on

In the previous paragraph the LED was placed in `PORTC5`. In the file `main.c` where the main-loop is located the following changes need to be made:

1. Set pin `5` in `PORTC` to output mode with the Data Direction Register for the port `DDRC`.
2. Switch pin `5` in `PORTC` on
3. Stop the controller

A program on a microcontroller should never leave its main loop. This normally causes the controller to start from the beginning of the programs. But not all hardware is reset resulting in an undefined state in not properly initialized hardware. To prevent this, normally the controller is kept in an infinite loop at the end of a program.

...

```
int main(void) {
        //Set the Data Direction Register to output
        DDRC |= (1<<5);
        //Set the signal to high
        PORTC |= (1<<5);
```

```
        //Stop in an infinite loop
        while (1);
}

...
```

You can now type `make` to see if the program compiles.
After connecting the board to power and plugging in the programmer, the binary can be transferred with the command `make prog`.
After the transfer the LED should now emit light. If it does not, check for possible errors during the transfer on the screen. Or maybe the LED is placed with the wrong polarity. Simply turn it around for a try, wrong polarity does not damage the LED.

4.1.3. Making the LED blink

Now that we know how to switch on a LED, we also want to switch it off again after a certain time. We do this by the most simple way: a delay function.
The AVR library provides a very accurate delay function. Make sure your file has the line `#include <util/delay.h>` to include the delay functions.
Now the program looks like this:

```
...

#include <util/delay.h>

...

int main(void) {
        //Set the Data Direction Register to output
        DDRC |= (1<<5);
        while (1) {
                //Set the signal to high
                PORTC |= (1<<5);
                //wait 0.5 sec
                _delay_ms(500);
                //Set the signal to low
                PORTC &= ~(1<<5);
                //wait 0.5 sec
                _delay_ms(500);
        }
}

...
```

After compiling and transferring the program with `make prog` the led should blink in a frequency of 1 sec.
If it is blinking faster or slower then you maybe have forgotten to fuse the controller. Try `make fuse` and see if it changes.
This program does what it is supposed to, but it can be written shorter and with a reduction of redundant code. First the bit is set and later cleared, this can also be done with the toggle function which is the equivalent of the XOR operator (see A.1):

```
...

#include <util/delay.h>

...
```

```
int main(void) {
        //Set the Data Direction Register to output
        DDRC |= (1<<5);
        while (1) {
                //Toggle the signal
                PORTC ^= (1<<5);
                //wait 0.5 sec
                _delay_ms(500);
        }
}

...
```

4.1.4. Making the LED blink (advanced)

The previous program is nice but the delay function blocks the whole main loop from working on other tasks. It seems wise to find a method of doing actions in a certain time but without blocking the controller. Such situations are normally resolved using a state machine.
In the basis-code a function called `getMsTimer()` is provided. It returns the time in milliseconds ($\frac{1}{1000}s$) since resetting the controller. Beware that the return value of this function is a `uint32_t`, when storing it. The variable `ms_timer` is deprecated and cannot be used in newer revisions.

```
...

//Time between switching the led on or off
#define delay 500
int main(void) {
        init();
        //Set the Data Direction Register to output
        DDRC |= (1<<5);
        uint32_t next_ms_timer=getMsTimer()+delay;
        while (1) {
                //Is it time to do smth.?
                if (getMsTimer()>next_ms_timer) {
                        //Toggle the signal
                        PORTC ^= (1<<5);
                        //don't forget to set the next "alarm"
                        next_ms_timer+=delay;
                        }
                //plenty of free time to do other stuff
        }
}

...
```

4.2. Reading an input

If the bit in the `DDR` (Data Direction Register) for the corresponding pin is 0, it is configured as input. A voltage provided at that pin is interpreted as either 0 or 1 and stored in the `PIN`-Register (e.g. `PIN`).

Example

In the following example a switch is placed on pin C3. While the switch is pushed, the LED (still placed at C5) is turned off. In both examples the so called internal pull-up resistor is switched on. Please refer to section 4.2.2 for its meaning.

```
...
int main(void) {
        init();
        //Set the Data Direction Register for the LED to output
        DDRC |= (1<<5);
        //Set the Data Direction Register for the switch to input
        DDRC &= ~(1<<3);
        //Enable the pullup resistor for the switch
        PORTC|=(1<<3);
        while (1) {
                if (PINC & (1<<3))
                        PORTC|=(1<<5);
                else
                        PORTC&=~(1<<5);
        }
}

...
```

4.2.1. Detecting an edge

Most of the time it is requiered to detect a change of a pin. This is done by storing the old value and comparing it to the current one.

```
  ...
int main(void) {
        init();
        //Set the Data Direction Register for the LED to output
        DDRC |= (1<<5);
        //Set the Data Direction Register for the switch to input
        DDRC &= ~(1<<3);
        //Enable the pullup resistor for the switch
        PORTC|=(1<<3);
        //save old state
        uint8_t oldc=PINC;

        while (1) {
                if ((oldc & (1<<3)) != (PINC & (1<<3)) ) {
                        //Switch LED
                        PORTC^=(1<<5);
                        //save new state
                        oldc=PINC;
                }
        }
}

...
```

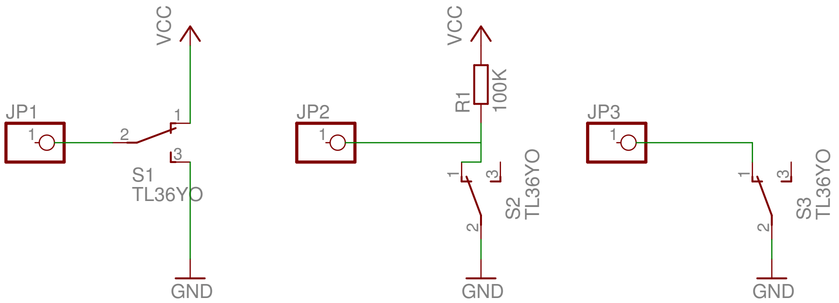


Figure 2: Differently connected switches: (a) between 5V and GND, (b) between pin and GND with an external pull-up resistor, (c) as (b) but with an internal pull-up

4.2.2. Providing a clean signal

Some words on a well defined signal for the input should be said. The input stage of the controller has an intrinsic capacitor that keeps its electric charge for a certain time. If you connect a switch, it should always switch the pin between 5V and GND (0V) to allow it to get properly (dis-)charged. This is shown on 2 (a).

If a switch can not change between two connectors but just opens and closes a single line, a so called pull-up resistor must be used. This can be an external resistor with a value between $100K\Omega$ and $1M\Omega$ (2 (b)).

The most elegant way is to use the internal pull-up already installed inside the Atmega. To activate them the PORT is set to 1, while the DDR remains set to 0.

...

```
DDRC &= ~(1<<3);
PORTC = (1<<3);
if (PINC & (1<<3)) {
        //do smth.
}

...
```

Be aware of the fact that by using a pull-up the logic of the switch is inverted!

5. Timer/Counter

This is the end of the basic tutorial section of this document. The following sections describe the usage of functions provided in the basis-code.

5.1. Hardware timer

Figure 3: Interior of a servo. Picture from wikipedia[2]

The Atmega controllers provide hardware counters. Those counters are registers that are incremented normally by a signal from the oscillator which also drives the Atmega. The oscillator is not connected directly but via a variable *prescaler* to run slower.
To every counter there is at least one *compare register*. When the counter value equals the value of the compare register, a certain action can be triggered. This can be a reset of the timer, throwing an interrupt (like in 5.3) or toggling a pin as used in PWM or for servo control (5.2). Actions can also be triggered when the timer value reaches an overflow. Please take a look at the data sheet [1] for more information, this is just a brief introduction.

Example calculation

As an example we calculate the frequency in which Timer1 overflows. Timer1 is a 16-bit timer.

$$f_{oscillator} = 16MHz = 16 \cdot 10^6 Hz = 16000000Hz$$

$$prescaler = 8$$

$$f_{overflow} = \frac{f_{oscillator}}{prescaler \cdot 2^{16}} = \frac{16000000}{8 \cdot 65536} Hz \approx 30.5Hz$$

$$t_{overflow} = \frac{1}{f_{overflow}} \approx 0.0327s \approx 33ms$$

As we see the timer would overflow about every 33ms that equals a frequency of about 31Hz.

5.2. Servo control

5.2.1. About servos

Servos are motors with an integrated controller unit. They normally have a range of motion of about 180°, most servos also a little more. A signal is applied to tell the servo the desired position and the controller inside then tries to actuate the motor to reach it. For that reason servos are very handy because of their easy usage and very compact design.

Unfortunately there is no way of asking the servo about its position. If for example the counter momentum is too large and the motor is too weak to reach the desired position, there is no way of knowing it on the microcontroller. So there is no guarantee that a servo actually reaches a position.

A servo normally has three connectors. One for Ground, the middle for supply voltage (typically $\approx 4.8V \ldots 6V$), and the last for the signal. The signal is a high-pulse of 1 . . . 2ms every 20ms. The duration of the signal is proportional to the angle of the position.

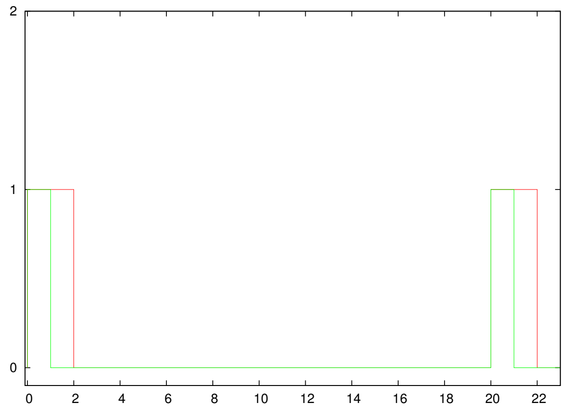


Figure 4: Signal form for a servo: 1 . . . 2ms high every 20ms

5.2.2. Servos in `basis`

The basis contains simple functions to provide this signal for two servos. Those servos should be connected to `PORTB1` and `PORTB2` where is output pins for Timer1 are located. They are called `OC1A` and `OC1B` (see A.2).

After initializing the timer with function `servoInit()` the function `setServo(uint8_t nr, uint16_t us)` is the only interface needed. `nr` is the servo number, i.e. 0 or 1. The second parameter is the signal length in $1\mu s = \frac{1}{1000}ms$. The range is therefore 1000 . . . 2000.
Some servos allow a wider range which can be used safely. For safety reasons they are not allowed per default in the basis. If you want to use them you need to make changes in the `servo.h` file.
But be careful not to stall the servo over a longer period of time! It can overheat and destroy the electronic inside!

Example

This example sets the first servo to one boundary position.

```
...
int main(void) {
        servoInit();
        setServo(0, 1000);
        while (1) {     }
}

...
```

5.3. Timer for software interrupt

The hardware Timer2 is used to provide a software interrupt at 1kHz. Every time the interrupt is triggered, the global variable `ms_timer` is incremented.
Therefor in every part of the code the variable has the actual time since in ms since the last reset of the controller.

To get the actual time the function `uint32_t getMsTimer()` must be used. Be aware of the fact that the return value is a `uint32_t`. Storing it in a container of smaller size will probably not work. Even if a 32-bit variable seems quite large it will overflow every 49.7 days. For long-time applications another way of storing the time must be developed.

6. Analogue-Digital-Converter (ADC)

An input pin in default mode is only able to distinguish between a high and a low signal. Some pins can be configured as ADCs. They can be read in a resolution of up to 10 bit ($2^{10} = 1024$). The controller has only one ADC but uses a multiplexer or *mux* to switch between the pins. On an Atmega168 those are `C0` to `C5`.

6.1. ADC in `basis`

There are numerous ways to configure the ADC. For the basis a compromise between speed and accuracy was chosen that should fit to most purposes. The ADC is initialized with `ADCInit()`. A value can be read with `getADCValue(uint8_t channel)`, where `channel` specifies the pin to read from.

example

```
...

int main(void) {
        ADCInit(0);
        uint16_t value;
        value = getADCValue(2);
        while (1) {     }
}

...
```

7. Serial connection(UART)

The Atmega controller has a so called *Universal Asynchronous Receiver Transmitter (UART)* which can be used to communicate with any other `RS-232` device like a PC. The only hardware required is a signal converter that connects the $0 \dots 5V$ signal of the microcontroller to the $-12 \dots 12V$ of the PC. We provide such converters in the lab.
When programming uart one should always be aware of two pitfalls:

1. *Buffersize:* The hardware buffer for input and output on the Controller only hold 1 byte. That means
   a) If the program does not read fast enough, it loses bytes.
   b) It is only possible to send one byte at a time.

   To prevent data loss and blocking calls for sending it is possible to use interrupts. Those functions are currently not implemented in the basis.

2. *Speed:* The *baud* (bandwidth) has to be generated by the controller's oscillator. An internal prescaler is applied to generate the baud. That limits the baud usable with a 16MHz oscillator to 38.400kbit/s.

7.1. UART in `basis`

First the baud has to be specified in `uart.h`. After initializing with `uartInit()` the following functions can be used:

- `uart_putc(unsigned char c)` sends the char stored in `c`. It blocks if the send buffer is not empty.
- `uart_puts(char *s)` sends a string (terminated with `'\0'`) stored in `s`.
- `uart_puts_pgm (const prog_char * str)` sends a string like `uart_puts()`. The only difference is, that this string is stored in flash memory and therefor does not waste valueable SRAM.
- `uart_puti (int16_t i)` sends an integer. Be careful that cannot be longer than 16 bit.
- `unsigned char uart_getc()` receives a char. It blocks until one is in the in buffer.
- `uart_data_waiting()` determines if the input buffer stores a valid byte. This can be used as a test before calling `uart_getc()` because `uart_data_waiting()` is non-blocking.

7.1.1. Example

```
...

int main(void) {
        uartInit();
        //This string wastes RAM, because it's static and could be stored in flash
        uart_puts("Hello World\n\r");
        //This string is stored in program memory
        uart_puts_pgm(PSTR("Hello World!\r\n");
        if (uart_data_waiting())
                uart_puts_pgm(PSTR("There is already data in the input buffer!\r\n");
        while (1) {
                //get a character (blocking)
                uint8_t c = uart_getc();
                //send it back, aka echo
                uart_putc(c);
        }
}

...
```

7.2. The terminal

On the side of the PC a terminal program is needed to directly communicate with the controller. There is for example `minicom` which is old and text based or `gtkterm`.
Settings:

- Baud as you selected in `uart.h`, i.e. 1200, 9600, 38400 ...
- 8 bit, no parity, 1 stop bit, a.k.a. 8N1
- No hardware flow control (RTS/CTS)
- No software flow control (Xon/Xoff)

A. Appendix

A.1. Algebra with bits

This section shortly introduces how to set, unset and toggle[3] a single or multiple bits in a 8bit register or variable in your program memory.
There are of course more things that can be done in bit-manipulation. If you are interested, feel free to search the web for it. The following part is focusing on the basics.

There are multiple ways to define an unsigned integer value in C:

```
// decimal
a = 123;
// hexadecimal
b = 0x7b;
// octal
c = 0173;
```

But there is no binary representation in standard C.
It becomes necessary to find a way to circumvent this problem. One solution is bit-shifting.

A.1.1. Shifting

In C there are the shifting operators `<<` and `>>`.
They must not be confused with the stream operators in C++!
Therefore it is strongly recommended to use brackets around them.
An expression like `(a << b)` will be evaluated as the binary representation of `a` will be shifted `b` times to the left, the value is getting larger. `(a >> b)` shifts the to the right and therefor decreases the value. Empty spaces are filled with 0.
The example demonstrates:

C code	decimal value	bit representation
`1`	1	00000001
`3`	3	00000011
`4`	4	00000100
`(1 << 2)`	4	00000100
`(1 << 3)`	8	00001000
`(3 << 2)`	12	00001100
`(4 >> 1)`	2	00000010
`(3 >> 1)`	1	00000001
`(4 >> 4)`	0	00000000

[3] change a value to its inverse ($1 \rightarrow 0, 0 \rightarrow 1$)

The obvious way to define bit no. `n` is using the expression. `(1 << n)`.

A.1.2. Bit-wise logic

Assuming, the logic operators a well known. But for the sake of completeness here is another overview:

A	B	AND	OR	XOR	NOT(A)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

The bit-wise logic needs to be distinguished from the Boolean logic.
In *Boolean Logic* the truth value of a whole expression is evaluated. If an expression equals `0` it is `false`, in any other case it is `true`. The representation are double symbols.
In *bit-wise logic* the those are single symbols.

	Boolean	bit-wise
AND	`&&`	`&`
OR	`||`	`|`
XOR	`!=`	`^`
NOT	`!`	`~`

A.1.3. Setting a bit

Finally with bit-wise logic and bit shifting it is possible to set a certain bit while leaving the other bits untouched.
For example bit no. `5` in variable `a` should be set to have the value `1`:

```
// elaborately
a = a | (1 << 5);

// ...or shorter
a |= (1 << 5);
```

The OR-operator is used because `x OR 1 = 1` and `x OR 0 = x` where `x` can be `0` or `1`. Therefor the previous value of this bit is irrelevant and is now set to `1`.
For example:

$$\begin{array}{cl} & 10010010_2 \\ | & 00100000_2 \\ \hline = & 10110010_2 \end{array}$$

A.1.4. Clearing a bit

Clearing a bit back to `0` works similar to setting it to `1`. The only difference is that two operators are needed, namely AND and NOT. The fact that `x AND 0 = 0` and `x AND 1 = x` is used. To achieve a `0` at the specific bit position and `1`s at the others the NOT operator is very helpful.
In this example bit no. `5` in variable `a` should be cleared.

```
// elaborately
a = a & ~(1 << 5);

// ...or shorter
a &= ~(1 << 5);
```

For example:

		10110010_2
&	~	00100000_2
		10110010_2
&		11011111_2
=		10010010_2

A.2. Pin out of the Atmega 168

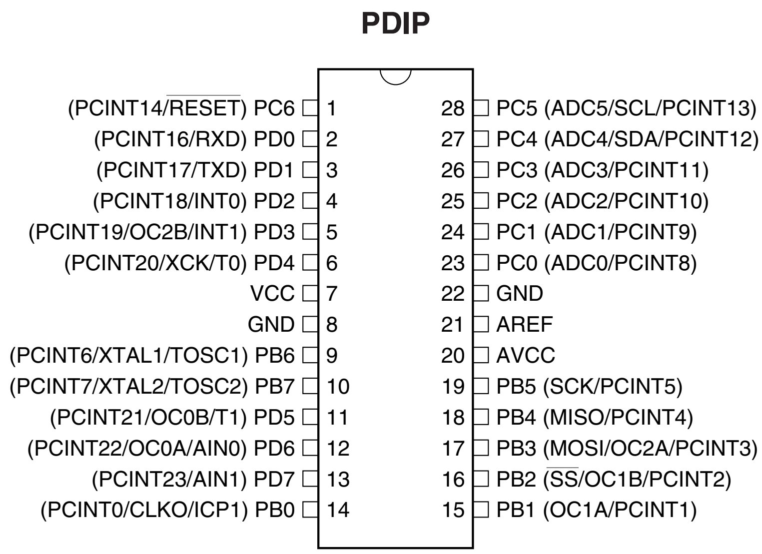


Figure 5: Pin out of the controller excerpt from the data sheet[1]. For orientation use the marking on top

B. References

References

[1] Atmel Corporation. Datasheet to atmega48/88/168, 2005.

[2] Wikipedia. Servo — wikipedia, die freie enzyklopädie, 2011. [Online; Stand 23. November 2011].

Version

This document was created from git `commit 2874ba28f297718d16661fa9a4aee6fd6ce3efb0`