Christoph Redecker
www.avrbeginners.net

draft

A short introduction to AVR assembler and the embedded "Hello, World!"

This is a tutorial about the very first steps with an AVR. It contains

1. A few words about AVR assembler and a very short example,
2. a more useful example that makes an LED flash.

You don't need to download or buy anything for this tutorial. However, you might want to at least simulate the code presented — in that case you'll have to download an AVR simulator.[1]

1 AVR Studio is an integrated development environmant provided by Atmel, and it includes a simulator. It is available from www.atmel.com.

AVR assembler and a short example

Assembler is a low–level language, and so is the AVR variant of assembler. AVR assembler consists of about 120 instructions[2], which is not much — if you have used a high–level language (like C) before, you will notice that most "single steps" of a program translate to a few more assembler instructions. However, assembler does not have to be complicated — let's look at probably the simplest program possible:

2 The actual instruction set for a particular AVR model depends on its size and family, and is summarized in its datasheet.

```
end:       ; a label, followed by a comment
  rjmp end // relative jump to end, and another comment
```

This program can be assembled for any AVR model, regardless of its size or any feature it might have or not have. It consists of two lines: in the first line, the *label* `end` is placed. Placing a label is equivalent to naming an address in memory. In this case, the name `end` is assigned to the address where the following instruction (which is in the second line of the above example) in located. The label itself is *not* an instruction!

Furthermore, a *comment* has been placed to the right of the label. It starts with a semicolon (`;`), and everything between the semicolon and the next line is ignored by the assembler. This is where things get a bit more complicated: Atmel introduced a new assembler, named AVR assembler 2, which has a built–in C–style preprocessor. It allows for C–style "`//`" comments (and a lot more extensions), which has been used in the second line.[3]

3 AVR Assembler 2 will be used here, if not otherwise specified.

Apart from the comment, the second line contains an instruction. The instruction `rjmp` is a "relative jump", and it takes *one* argument: a constant address — the jump destination. In this case, the jump destination is the address we formerly assigned the label `main` to, so in the end the AVR's core is instructed to jump to `main`. It will then again encounter the `rjmp`. This results in an infinite loop, which (in C) could be written as

```
while(1);
```

It is called a *relative* jump, because the destination address is not stored as an absolute value. Only the difference between the jump destination and the current address is stored.[4]

If not otherwise specified with an assembler directive, the assembler will place the assembled code in program memory (FLASH), starting at address 0.[5] After power–up, the AVR will jump to program memory address 0[6] and start executing instructions[7]. It could even eventually reach the end of the program memory, and in that case, the *program counter* will wrap over to program memory address 0 again. The result of that event can be similar to a reset, depending on the state of the microcontroller.[8] You might have concluded from the massive amount of notes that there are *a lot* of common pitfalls, and that this section might need some "relaxing".

So far, the program presented above only consists of an infinite loop. It can run on a real AVR or be simulated in a simulator. Running it on an AVR would not be very enlightening, because it would just silently execute the infinite loop, without any interaction with the outside world. Running it in a simulator would at least visualize some of what was said above.

Now that you've seen a (hopefully) non–scaring assembler example, it is time to add some interaction with the outside world to the program. The first step in that direction is adding an LED and switching it on, the second is making it flash.

4 Of course, there is a reason for this: The relative distance can quite often be stored more efficiently.

5 This can be changed with the `.org` directive.

6 It is also possible to jump to a so–called *boot reset vector* after power–up, but that is not the out–of–the–box behavior.

7 It will only stop if a `sleep` instruction puts the core into a sleep mode.

8 A great source for bugs.

Adding an LED and making it flash

Although we're talking about hardware now, it is not absolutely necessary to buy or build anything. It is sufficient to understand some very basic electronics. The following assumptions are made:

- the AVR is an ATmega168P with factory settings[9];
- a standard LED with a current limiting resistor is connected between VCC and PB0 see figure 1;
- the circuit is running on V_{CC} 5 V.

9 These will be discussed later

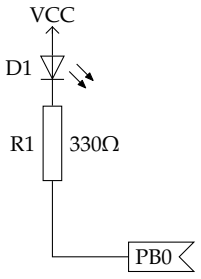

Figure 1: How a standard LED is connected to the AVR in our example.

"PB0" means "bit 0 of Port B". Port B is one of the AVR's I/O ports, each of which is usually 8 bits wide. Each of these bits of a port is connected to a pin of the AVR's package. If one or the other bit is missing, this will be documented in the datasheet.

After power–up, the port pins — including PB0 — are configured as inputs. This means that they do not source or sink current, and the LED as shown in figure 1 will not light up. What we need to do is to tell the AVR to pull the port pin low. The situation then changes for the LED, because the voltage at the LED's cathode is now close to 0 V. Current can flow, limited by R1, the the LED lights up. Pulling a port pin low is a simple task, yet requires some knowledge about the basic features of AVR I/O ports.

AVR I/O ports consist of three registers each:

The Data Direction Register (DDR) is used to switch between input (default state after power–up) and output mode. Writing a DDR bit to one configures the port bit as an output.

The PORT register is used to set the output value of a port bit when it is in output mode. If it is in input mode, an internal pull-up resistor can be activated by writing the bit to one.

The PIN register is used to read the actual value of the pin. When a PIN bit is written, the corresponding PORT bit is *toggled*.[10]

10 Old AVRs did not have this feature.

For Port B, the registers described above are named DDRB, PORTB, and PINB. All registers can be manipulated bit–wise. A single bit (bit 0 for example) of Port B is named DDB0, PORTB0 and PINB0, respectively. These bit names are listed in the AVR's datasheet in the section about I/O ports and also in the register summary which can be found at the end of each datasheet.

Now we know almost enough to light up the LED. There are two things to set: PB0 must be configured as an output and the output value must be zero, thus DDB0 = 1, PORTB0 = 0. As PORTB0 is set to zero after power–up anyway, there is nothing more to do than to configure DDRB0. The LED will then light up, and we can enter an endless loop:

```
sbi DDRB,0 // set DDB0 or more verbose: set bit 0 in DDRB
end:
  rjmp end
```

Again, there are a few things to explain here. A new assembler instruction, `sbi`, is used. It means "set bit in I/O register" and does just that. It takes two arguments: The register address (hidden behind the label DDRB) and the bit number that should be set. Also, we have quietly assumed that DDRB has been defined somewhere earlier. If you are using AVR Studio, an include file is available for every AVR. This include file can be used just like an include file in C, because AVR assembler 2 has a C–style preprocessor. Adding the appropriate directive results in

```
#include <m168pdef.inc>

sbi DDRB,0 // set DDB0
end:
  rjmp end
```

That's it: DDRB is now known to the assembler. This piece of code can be put into an asm file, assembled, and the resulting binary file can be used to program an AVR, and it would work. However, we have not yet reached our goal — the LED is supposed to *flash*. In addition to just switching it on, we need to

- wait for a specified time until we can
- switch it off again,

- wait again,
- and start over by switching the LED on again.

The last point, *starting over*, indicates that we need a loop. We can re–use the endless loop we already had before and label it `main`, because we add functionality to it. Switching the LED off is equally simple and if you have read carefully, you will have noticed that there are at least two ways of doing that: by setting PORTB0, or by writing one to PINB0 (utilizing the toggle feature explained above). Both variants can be used, but need to be treated in slightly different ways. Figure 2 shows an algorithm that switches the LED on and off, figure 3 shows one that toggles the LED. The second variant will be implemented in this tutorial.

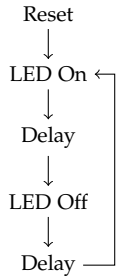

Figure 2: An algorithm that makes the LED flash by switching it on and off, using delays in between.

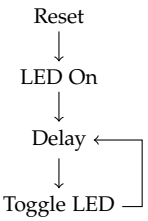

Figure 3: An algorithm that makes the LED flash by toggling it. Again, a delay is used.

So we need a delay. A delay is a period of time, and the microcontroller "measures" time in terms of CPU cycles. How long one CPU clock cycle is depends on the CPU clock frequency f_{CPU}. This frequency can have different sources, and an appropriate source can be selected for each application. Common choices are the microcontroller's internal RC oscillator (which is not very precise) or an external crystal.[11] When you buy an ATmega168P, it is shipped with convenient default settings:

- CKDIV8 is programmed. This means that the clock source is divided by eight before being fed to the CPU;
- the internal RC oscillator, running at 8 MHz, is selected as clock source.

The result of these settings is that the CPU runs at 1 MHz and does not require any external clock source. This is absolutely suitable for what we are about to do.

Each AVR instruction takes a certain number of cycles to execute. We can create a certain delay by executing a certain number of instructions in a loop. Short delays can be done without a loop, but ours will be long. Let's make it $\Delta t_d = 0.5\,\mathrm{s}$. Speaking in clock cycles, we need to create a loop that takes $n_d = \Delta t_d / f_{CPU} = 500\,000$ cycles to execute.

AVR registers are 8 bits wide, which means that the highest value they can store is 255. We can also use two of them to store a maximum value of 65 535, or use even more registers. Using registers as a counter, a delay loop can be implemented. In each iteration the counter is incremented or decremented (the direction is not really important unless the counter value serves other purposes as well) and when a certain value is reached, the loop is ended. A simple version of such a delay loop looks as follows:

```
clr r16     // clear register 16, as in r16 = 0
loop:       // we need a label to jump back to
  inc r16   // increment r16, as in r16 = r16 + 1
  brne loop // if r16 is not zero, jump back
finished:
  // this is done after the loop ended (nothing)
```

11 There are, of course, a lot more clock choices and settings. They are documented in the datasheet!

The first line makes sure that r16 is zero before the loop is entered. In a bigger application, r16 might have been in use before, so we need to create known conditions. In each loop iteration, r16 is incremented and will eventually roll over from 255 to 0. As long as this does not happen, the loop will be executed again and again — 256 times to be exact.

Now we need to figure out how long each loop iteration takes. The instruction set *summary* is not really informative when it comes to branch instructions; we need the AVR Instruction Set Manual[12] instead. It lists all AVR instructions (even those the ATmega168P doesn't support, so watch out), explains their operands and a lot more. Important for us is that it also tells us how many cycles a branch instruction needs if it branches and if it doesn't branch. Here is what we get for each instruction used above:

12 Available from `www.atmel.com`

- `clr`: 1 cycle. A different instruction, `eor` (exclusive or), is actually used. Also note that some bits in the *Status Register (SREG)* are set or cleared — we will address that later.
- `inc`: 1 cycle. Again, the SREG is touched.
- `brne`: 1 cycle if the instruction does not branch, 2 cycles if it does. The decision if the instruction branches depends on the *Zero Flag* (Z) in SREG.

You can see that arithmetic instructions like `clr` or `inc` leave a certain value in SREG, and that conditional instructions depend on SREG. This concept is the basis for branches and loops in assembler.

The result of the first 255 executions of `inc r16` is *not zero*, and thus the zero flag in SREG will not be set. This causes `brne` (branch if *not* equal) to branch, effectively jumping to `loop` again. After that, the result is zero, because r16 overflows from 255 to 0, and `brne` will not branch. This is the end of the loop. Let's see how many cycles the whole thing needs for execution:

$$n_{loop} = 1 + 255 \cdot (1 + 2) + (1 + 1) = 768\,\text{cycles}. \tag{1}$$

That is not a lot compared to the $n_d = 500\,000$ cycles we need for a half–second delay. We can either use more registers in the inner loop or construct an outer loop around the loop we created above[13]. Here is a delay that consists of an outer loop and an inner loop, and it uses two registers for the outer loop:

13 Again, there are a lot more ways to generate a delay, using timers for example. Read a tutorial on timers!

```
ldi r24, 0    // load r24 with 0
ldi r25, 0    // load r25 with 0
outer_loop:
  clr r16     // clear r16
  loop:       // this is the inner loop label
    inc r16   // increment r16
    brne loop // if r16 is not zero, jump back
  // the inner loop ends here
  adiw r24, 1 // add 1 to register pair r24:r25
```

```
  brne outer_loop
// end of outer loop
```

Again, new instructions are used:

- `ldi` (*Load Immediate*): loads a constant to a register. The destination register can only be one of the registers r16 to r31. This is a common pitfall, and usage with one of the "lower" register would result in an assembler error.
- `adiw` (*Add Immediate to Word*): This instruction operates on a *register pair* and adds a constant value to that register pair. It can be used on r25:r24 (this is used here), r27:r26, r29:r28 and r31:r30. Only the lower register has to be specified as operand.

The logic that applies for the outer loop is the same as for the inner loop. The inner loop is executed in each iteration of the outer loop and it can be counted like one huge instruction that needs 768 cycles for execution. As it is here, the whole delay now needs

$$\begin{aligned} n_{outer\,loop,0} &= 1 + 65535 \cdot (768 + 1 + 2) + (768 + 1 + 1) \\ &= 50\,593\,792 \text{ cycles.} \end{aligned} \tag{2}$$

That is too much. Fortunately, we used `ldi` for initialisation of the outer loop counter, and it is easy to change the initialization value to something else than zero. Calculating that new value might look a bit awkward at first, but here are the main thoughts:

- The outer loop is using an up–counter;
- the outer loop counter overflows from 65 535 to 0;
- when initialized with n_i, it overflows after $65\,536 - n_i$ iterations;
- $n_{outerloop,0}/n_d$ is the factor (roughly 10 by which the loop takes too long as it is now.

The closest we can now get to a half second delay is

$$n_i = 65\,536 \cdot (1 - n_d/n_{outer\,loop,0}) \approx 64\,888. \tag{3}$$

Inserting this value in the equation above yields:

$$\begin{aligned} n_{outer\,loop,i} &= 2 + (65536 - 64888 - 1) \cdot (772) + (771) \\ &= 500\,257 \text{ cycles.} \end{aligned} \tag{4}$$

Note that the actual delay is not exactly 500 000 cycles, but for a flashing LED, this is close enough. The LED is still missing in the last code example. Putting it all together:

```
#include <m168pdef.inc>

sbi DDRB,0 // set DDB0

main:
  ldi r24, low(64888)  // load r24 with 0
```

```
ldi r25, high(64888) // load r25 with 0
outer_loop:
  clr r16     // clear r16
  loop:       // this is the inner loop label
    inc r16   // increment r16
    brne loop // if r16 is not zero, jump back
    // the inner loop ends here
      adiw r24, 1 // add 1 to register pair r24:r25
      brne outer_loop
  // the outer loop ends here

sbi PINB,0    // toggle LED

rjmp main     // repeat
```

The built–in functions `low()` and `high()` have been used here. They return the first and the second least significant byte, respectively, of the number given as an argument.[14] You can copy this into an asm file, assemble it, and program an AVR with the binary output. With a proper power supply and factory fuse settings, it should make an LED attached to PB0 flash at 1 Hz. You can also simulate the code in the AVR Studio Simulator.

14 In this case, with a 16–bit number, these are what we might call the "low" and "high" byte. While this expression might be correct for the low byte, the "high" byte of a 24– or 32–bit number is not the second least significant byte.

There are many things you could look into now:

The Stack is used for storing return adresses to subroutines. You can use it to *call* subroutines and *return* from them, much like in higher–level programming languages;

An External Clock can replace the internal RC oscillator;

Timers can be used to generate delays in the background and let the CPU do useful things;

Interrupts can make your program more event–driven (and they can be used with timers);

or just read a bit in the ATmega168P datasheet and try to understand the examples there.