Tano Zori

ASSEMBLER FOR ARDUINO UNO

ver. 0.1 - 2022, August 19th

Dear friend,

hope that this working in progress notes are useful to you in some ways. This work cames with a Creative Commons licence so you can reuse it as you desire but not for commercial purpose and alwats acknowledging the author and this original document. I'm writing this notes for hobby in the spare time, and as you know, writing content takes time and it requires a great effort. If you desire you can support this work for the community via `Ko-fi.com/tanozori`.

Support this work at `Ko-fi.com/tanozori`.

Tano Zori **Assembler for Arduino Uno**
version 0.1 - August 19th, 2022
e-mail: `tanozori[at]gmail[dot]com`

Indice

Capitolo I

TURN ON THE BUILT-IN LED.

The Arduino Uno has a led already soldered on its board. The objective is to turn on the light on this led. It is a very easy task and it is the opportunity to uderstand how a microcontroller works at a very low level. And of course we will do it using assembler.

§ 1. Reference documentation.

The first thing that I suggest to do is to use the documentation since the first time and also for this very simple example. Understanding and learning how to use the documentation will be fruitul for complex projects and can save time for sures.

On the Arduino Uno board there is the *ATmega 328P* microcontroller so most of the content is about the assembler programming of this microcontroller but the same content can be used also for other microcontrollores in the same family. I suggest to download these documents available in PDF:

- *megaAVR Data Sheet* (Bibliography [1]) which contains all the information about the microcontroller and its internal organisation;
- *arduino uno pinout* to identity components, labels, and pins (see [3]) on the Arduino board;
- *AVR Instruction Set Manual* which contains the descriptions of all the assembler instructions [2];
- *Arduino(TM) Uno Rev3 schematic*;

links to some of these documents are in the bibliography but you can easily find them on any search engine.

§ 2. Built-in led.

Opening the document with the Arduino Uno pinout [3] you can observe that there is a built-in led already soldered on the board. The led is referenced as `LED_BUILTIN`. I suggest to identify the led on the physical board and on the Arduino Pinout document [3]. You can also observe that there is a label `PB5` on a yellow background. `PB5` means that the led is attached to the Port B on the bit five.

The Atmega328P has two ports named port A (`PA`) and port B (`PB`). Each port has eight pins, for sake of simplicity here at the moment we can say that this eight pins are digital. Each pin can have a value of 0 or 1 which are respectively the value of 0 [volts] and +5 [Volts]. So with zero value the led is turned off and with a value of one the led is turned on. So what we need is to control the value of a bit placing it to zero or one; in this example we want that the led is turned on so we must place the bit to one. As we will see in order to control the led we will control the bit of a register.

§ 3. ATmega328P architecture, Registers and Direct Addressing.

It is important to understand the microcontroller internal organisation. Turning on the led is very easy it requires two instructions, but the important thing is to learn and understand how things work.

§ 3.1. Microcontroller Architecture

In the architecture block diagram there is the *Flash Program Memory* which stores the program to execute. When the program is compiled and trasferred on the borad the instructions are stored in this memory.

The *Program Counter* (PC) is a counter pointing to the next instruction to be loaded and executed. The program is a sequence of instructions, each instruction has an index staring from zero which is the first instruction. Instructions are executed in order (except when there are jumps we will see in next chapters). So the program counter (PC) contains the index of the next instruction to be executed. Initially, the first time the board is powered on the PC value is zero and it increments every time an instruction is loaded in the Instruction Register.

The current instruction to be executed and pointed by the program counter is loaded in the *Instruction Register*. In this way the instruction can be decoded and properly executed.

The *General Purpose Registers* are 32 registers which are accessed and used by CPU to temporarily store data during computations and executions. Each register has eight bits, and indeed the ATmega328P is an eight bit microcontroller.

The interesting thing is that in the AVR CPU block diagram there is an arrow from the Instruction Register pointing to the General Purpose Registers (GPRs) labelled *direct addressing*. This means that the CPU can directly access to these registers within the instruction cycle. To simplify it means that an instruction can operate directly on these registers.

In brief, there are 32 general purpose registers, each one has 8-bits, and they are directly addressable and they work with a single clock cycle access time. The clock cycle is the internal timer which beats time for all the components in order to work toghether in a proper synchronized way. The clock is a digital signal having squared wave.

Another interesting thing is that the Port B corresponds to one of these 32 registers so we can directly access to the Port B register and set the value zero or one to turn on/off the led. Great!

This section does not explain the whole microcontroller architecture but just the blocks we need to understand in order to turn on the led. The other blocks of the architcture will be exaplained in the next sections.

The important notion to take away in this Section is that there are 32 registers, each one has 8-bits, and that Port B is one of these registers. So controlling the Port B register we can turn on the built-in led.

For further details about the internal organisation of the microcontroller see the Section 7 AVR CPU Core of the ATmega328P Data Sheet [1].

§ 3.2. General Purpose Registers.

Registers can be depicted as a pile of rectangles as shown in Fig. 1.

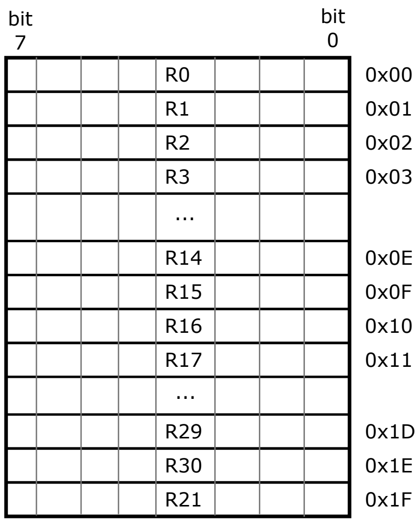

Figura 1: Representation of the General Purpose Registers.

Each register has an index, the first index is 0 and the last one is 31. Each register has a name that is the letter R followed by its index. So we names range from `R0` to

`R31`. Each register has eight bits (8-bit register), so each register can store a numeric value from zero up to 255 ($2^8 - 1$).

Each register has an address. The address is very important because instructions and the CPU identifies the register via its address and not via its mnemonic name. The address of the first register is 0x00, the address of the last register is 0x1F. For instance in order to set a numeric value in the register named `R0`, the CPU of the microcontroller needs the address, so with in the istruction to set the value we need to specify the address 0x00.

The addresses use the exadecimal notation because it has the prefix `0x`. The second part after the `x` is the address represented with a pair of numbers/letters. The numbers from zero to nine are represented with the classical symbols from 0 to 9. The values 10, 11, 12, 13, 14, and 15 are represented with letters, respectively A, B, C, D, E, and F. Each number or letter represents 4 bits. So since we have a pair, each pair represents a value of 8 bits.

Section 14.4 of the data sheet [1] describes in details all the register and their usage.

The important notion of this section is that each register has an address represented in exadecimal notation with the prefix `0x`.

§ 4. Port B registers.

In order to turn on the built-in led we need to (a) configure the pin direction as output and (b) set the logical value one for the bit 5.

Among the 32 General Purpose register there are two registers named: DDRB and PORTB (Fig. 2).

DDRB 0x04	DDR7	DDR6	DDR5	DDR4	DDR3	DDR2	DDR1	DDR0
initial value	0	0	0	0	0	0	0	0

PORTB 0x05	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
initial value	0	0	0	0	0	0	0	0

Figura 2: The PORTB and DDRB registers.

§ 4.1. Direction Input/Output.

Each pin of the microcontroller can be configured to be an output or an input. Configuring a pin in output means that we set a specific bit of a register with a zero or one and the corresponding pin on the microcontroller has respectively a zero voltage or +5v. Output means that we write in the register and the value is outputed on the pins. Input is the viceversa, the logical value on the physical pin of the microcontroller becomes a logical value in the corresponding register so we can read it. So when we talk about direction we mean to configure a specific pin as input or output.

The register `DDRB` of Fig. 2 is the register to configure the direction of the PORTB. DDR stands for Data Direction Register. Each pin can be configured as input or output independently from the others, so we can have some pins configured as input and others as output.

DDRB has address 0x04, it has 8 bits and we are interested in the bit with index 5. Everything has a mnemonic lable so we can refer to the bit 5 as `DDRB5`. When the bit has a high logical value (one) the pin is configured as output, viceversa when the bit has a low logical value (zero) the pin is configured as input.

We desire to confire the bit DDRB to one in order to configure the corresponding pin as an output pin.

§ 4.2. Port B Data Register.

`PORTB` is the data register of 8-bits corresponding to the pins in output of the microcontroller. Each bit of the register corresponds to a specific pin of the microcontroller. We can set the content of the PORTB register to set the corresponding output on the pins. PORTB is the register `R5` which has address 0x05. `PORTB` is an alias to say that we are address the register with address 0x05.

Fig. 2 shows the PORTB register. Observe that each bit has a label. The bit with index 5 and named `PORTB5` corresponds to the pin connected to the built-in led. It is possible to read or write each bit of the PORTB register. We are interested in writing the value one in the position PORTB5 that is we desire to write one in the bit of index 5 of the register named PORTB with address 0x05, so the corresponding pin will be +5V and the led will turn on.

So we desire to set the bit `PORTB5` to one to obtain a +5V on the corresponding pin turning on the led. We are not interested in configuring the other bits of the PORTB, indeed by default they have a value of zero which is a zero volt in output.

§ 5. Instructions to set registers' bits.

Until now we talked about the DDRB register to decide the direction of the pin and the PORTB register to decide the value of the pins. We need an instruction to effectively manipulate the bits of those registers. In particular we need an instruction to set the bits `DDRB5` and `PORTB5` to one in order to configure the pin number five as an output pin and set the high logical value in output to turn on the led.

In order to identify the instruction to be used I use the *Instruction Set Manual* [2] which lists and describes all the available instructions. We are interested in these two instructions:

- **sbi** (Set Bit in I/O Register) which sets a specific bit of a register to one;
- **cbi** (Clear Bit in I/O Register) which clears a specific bit of a register to zero.

In this example we will use only the instruction `sbi`, but it is important to undestand also the complementar instruction `cbi`.

§ 5.1. I/O direct addressing and SBI.

Both instruction can directly access to the registers content. In particular an instruction that supports the I/O direct addressing operates on one of the 32 General Purpose Register and has the following syntax:

```
OP A, b
```

`OP` is the operator (in our case `sbi` or `cbi`) which is followed by two operands `A` and `b`.

`A` is the address of the destination register on which the operator operates and can be one of the 32 registers so it can be `R0`, `R1`, ..., `R31`. We will see that we can use the address 0x00 etc. or its mnemonic symbol which will be converted in the corresponding address.

`b` is the bit to manipulate of the register with address `A` which is a number between zero and seven ($0 \leq b \leq 7$).

With the direct access the operator can directly change the register's bit.

The two registers DDRB and PORTB have addresses 0x04 and 0x05 so they are general purpose registers and can be accessed with the I/O direct addressing.

The Set Bit in I/O register (SBI) instruction has the following syntax `SBI A, b`. SBI is the operator, `A` is the address of the register to change and `b` the bit to manipulate.

For example, the instruction:

```
SBI 0x04, 5
```

sets to one the bit 5 of the register with address 0x04. As shortcut in the comment we will use the notation:

$$I/O(A, b) < -1$$

to say that the bit b of the address A is set to one.

§ 6. Complete example.

The following is the complete source code for the program that turns on the built-in led. In assembler everything starts with ; is a comment useful for humans to undestand the code and is ignored by the compiler.

```
; DDRB[5] is an output pin.
SBI 0x04, 5

; PORTB[5] has value one so led on.
SBI 0x05, 5

; main loop
mainloop:

    rjmp mainloop
```

The first instruction sets the bit five of the register DDRB to one, so that the corresponding pin is an output pin.

The second instruction sets the bit five of the register PORTB to one so that the pin that is connected to the built-in led has a high logical value (+5V) and the led turns on.

When a microcontroller is powered on it runs until it goes in standby or someone turns it off. So the program is a program that runs forever. It is common to place a main loop. So we introduced a label `mainloop` (actually you can name it as you desire), so that the next instruction does a jump to this label. In this way the program jumps forever to the previous address.

Instead to use the addresses we can use the mnemonic lables. The following source code is equivalent to the previous one.

```
; DDRB[5] is an output pin.
SBI DDRB, 5

; PORTB[5] has value one so led on.
SBI PORTB, 5

; main loop
mainloop:

    rjmp mainloop
```

Appendice A

COMPILE AND RUN AN ASSEMBLER PROGAM.

§ 1. IDE for Arduino Assembler.

In order to edit the example in this notes I used the Microchip Studio IDE. The first thing to do is to create a new project by clicking on File and then New Project. Select as Assembler as project type.

In this way, the IDE creates a new solution with an example file named `main.asm`. It is possible to clean all the code in the file and paste one of the examples, for instance the first example which turns on the led.

In order to build the solution click on the menu Build and then Build Solution. In this way the we have another file with extension `.hex` which contains the binary code.

§ 2. Load hex file on the board.

The assembler program becomes a binary file with extension `.hex`. The microcontroller runs the binary code, so the hex file is loaded on the Arduino Board.

On windows in the project directory we can create a `.bat` file which contains the command line instructions to program the board with the hex file. This is one long command (without new lines) to past in the bat file.

```
D:\path\arduino\hardware\tools\avr\bin\avrdude
-CD:\path\arduino\hardware\tools\avr\etc\avrdude.conf
-v -patmega328p -carduino -PCOM5 -b115200 -D
-Uflash:w:D:\path\project\file.hex:i
```

We use the program `avrdude.exe` installed toghether with the Arduino Studio in the arduino directory `tools/avr/bin/avrdude.exe`. Arduino IDE uses avrdude for load the program on the board. So instead to call this program from Arduino IDE we call directly it from the command line.

The last parameter is the path of the hex file to transfer on the board.

Now you can connect the Arduino board to the USB port. Then open a new command line (cmd.exe) and run the file `upload.bat`.

BIBLIOGRAPHY

[1] Microchip, *ATmega48A/PA/88A/PA/168A/PA/328/P megaAVR Data Sheet*

[2] Microchip, *AVR Instruction Set Manual*

[3] Arduino, *Arduino Uno Pinout*, available on line at `https://content.arduino.cc/assets/Pinout-UNOrev3_latest.pdf` (last access on 2022-08-19)

[4] *Arduino Trademark and Copyright*, available on line at `https://www.arduino.cc/en/trademark`