

First, we need the ATmega328P Datasheet from the Supplier of the ATMEGA328P Micro-controller (since most of our Arduino Uno has this chip)

I have a copy in my github repository,

https://github.com/teaksoon/lmaewapm/blob/main/ATmega328P_Datasheet.pdf

You should download and keep one copy on your Computer

The ATMEGA328P Datasheet file is very big, there is no need to print it or read everything in it. We may only need to use some of it. For now, take a look at these two topic first (dont worry if they do not make sense to you at all now)

1. AVR INSTRUCTION SET: ATMEGA328 Datasheet:

Topic "31. Instruction Set Summary", from page 281

2. ATMEGA328 CPU Memory (REGISTERS): ATMEGA328 Datasheet:

Topic "30. Register Summary", from page 271

Those two topics are the ones that will get the micro-controller CPU do work for us, from those we make our PROGRAM.

There are many REGISTERS in the CPU, **some values in the REGISTERS will cause a task being performed**. Sometimes CPU also updates the REGISTERS with certain values for our PROGRAM to use (The Datasheet will tell us every single one of them). **In order to update or retrieve the REGISTERS**, we need to use the instructions from the **AVR INSTRUCTION SET**

For example:

```
SBI DDRB, 5
```

The code above has an instruction "**SBI**" from **AVR INSTRUCTION SET**, will set the **CPU Memory (REGISTER)** "**DDRB**", 5th bit, to have a **value of '1'**

Once the CPU see this instruction "**SBI DDRB 5**", will set our **Arduino Uno I/O Pin 13 to become an OUTPUT Pin**

If we program in **Assembly Language**, we will have to **code exclusively with the AVR INSTRUCTION SET** and the **CPU Memory (REGISTER)**. Like the code above

However, when we Program using **C-Language with the Arduino Libraries**, we **seldom see the AVR INSTRUCTION SET or the REGISTERS**. This is because they are all hidden from us by our C-Language and the Arduino Libraries (which makes things alot easier).

The **C-Language and Arduino Libraries** has it own set of instructions which will be **converted into AVR INSTRUCTION SET/REGISTERS** for us by the **C-Language Compiler Software**.

Although most of the time we might not need to use the **AVR INSTRUCTION SET** and **REGISTERS** in our C-Language PROGRAM, we need to know that they exist. As we progress, sooner or later we will be digging into them. **Good thing about C-Language is that, although we do not need, we can access into the AVR INSTRUCTION SET and CPU Memory (REGISTER), if we wish to do so.**

Some of you may be curious, why not just use Assembly Language and make full use of the INSTRUCTION SET and REGISTERS

For example Assembly Language code ***"SBI DDRB 5"*** looks so simple. In fact seems easier than our familiar C-Language with Arduino Libraries ***"pinMode(13, 5);"***

For small PROGRAM, yes... Assembly Language is a great option. Assembly Language is small and we can access into every single details of the micro-controller INSTRUCTION SET and the REGISTERS

Lets look at another example, this will probably change your minds about coding in Assembly Language. Lets say we want to 250ms delay, in C-Language with Arduino Libraries we will simply code,

```
delay(250);
```

in Assembly Language, there is no delay(250), we have to make our own. The code below is for a simple fixed 250ms delay, if we want a variable delay time, a lot more coding will be needed (we do not want to go there, we just use the easier C-Language option)

```
delay_250_ms:
    ; One millisecond is 16,000 cycles at 16MHz.
    ; 250ms = ideally 4,000,000 cycles
    ; 601+3998600+800+1 = 4,000,002, closed enough
    ; Total = 1 cycle
    LDI    r20,200          ; 1-cycle

reset_ctr:
    ; Total = 4 x 200 = 800 cycle
    NOP          ; 1-cycle
    NOP          ; 1-cycle
    LDI    31, 4998>>8    ; 1-cycle
    LDI    30, 4998&255    ; 1-cycle

delay_loop:
    ; Total = ((4 x 4998) + 1) x 200 = 3998600 cycle
    SBIW   r30, 1          ; 2-cycle
    BRNE   delay_loop      ; 2-cycle, BRNE=1-cycle when ends

    ; Total = (3 x 200) + 1 = 601 cycle
    SUBI   r20, 1          ; 1-cycle
    BRNE   reset_ctr       ; 2-cycle, BRNE=1-cycle when ends

    RET
```

Coding in C-Language without the INSTRUCTION SET or the CPU Memory, does not mean C-Language is unable to do things that the Assembly Language can do. The final uploaded code from C-Language PROGRAM is still the same AVR INSTRUCTION SET codes with the same REGISTERS

If we insist must code in AVR INSTRUCTION SET and the REGISTERS, we can still do it from our C-Language PROGRAM. For example: if we wish to run the "NOP" and access the DDRB REGISTER from C-Language in Arduino, we can code in C-Language the following,

```
__asm__("NOP"); // We run the "NOP" from the INSTRUCTION SET, same as "NOP"
DDRB |= (1<<5); // CPU Memory, DDRB Register, same as "SBI DDRB 5"
```

Source Codes in
C Language/Assembly Language
- This is what we enter into
our Arduino IDE Software

AVR Compiler Suite
Software (Compile)

Source Codes converted into
INSTRUCTION SET codes (stored
in .elf file)

AVR Compiler Suite
Software (Cont...)

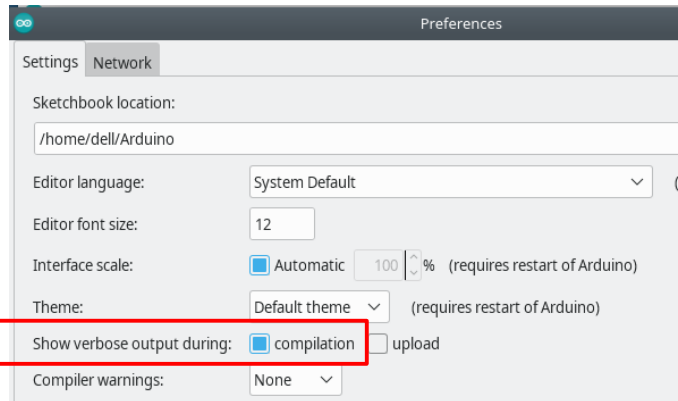
Remove all the all the
unnecessary remarks and codes
and make the final compact
executable hex file that can
be executed directly by the
mirco-controller CPU (stored
in .hex file)

AVR Micro-controller Upload
Software

.hex file is now stored in the
Micro-Controller Program
Memory, to be executed by the
Micro-Controller CPU

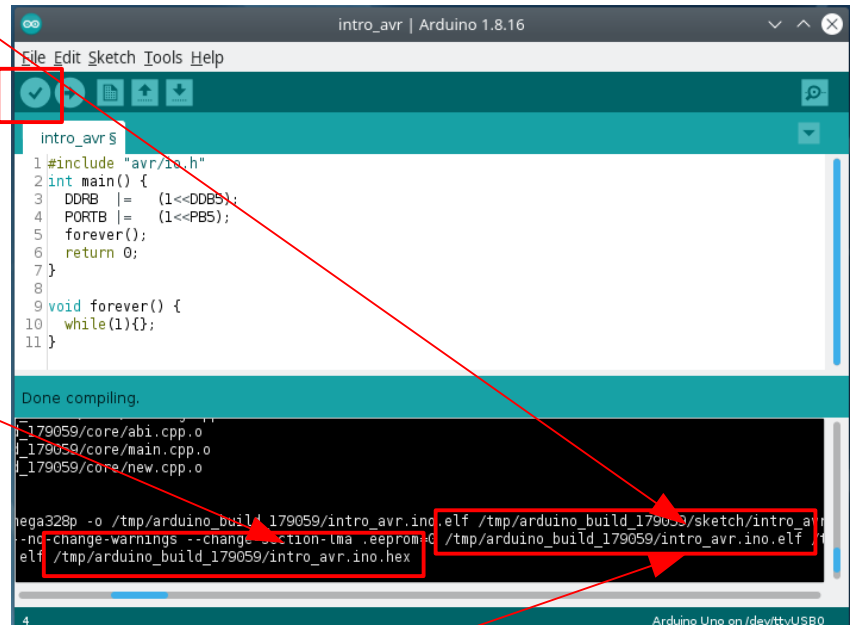
To see Compiled C-Language Source Codes

From the Arduino IDE Software,
Select "File | Preferences"



As you can see in the C-Language Source
Codes below, There are no INSTRUCTION SET
codes in it. Just See what happens after
we compiled it

Enter our Program as below and click the
Compile Button

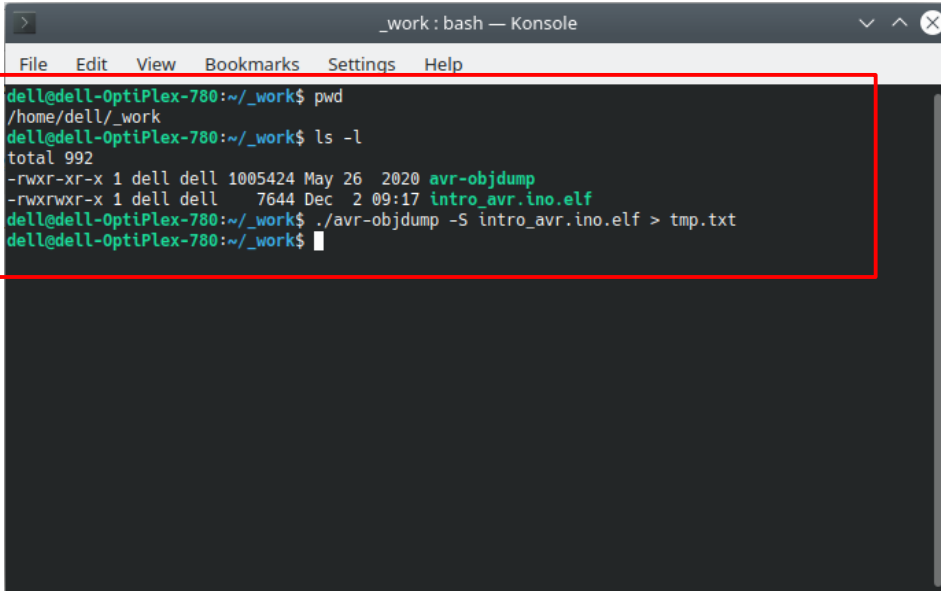


We want to see the content of the .elf file. To see if our C-Language Source
Code has been converted into INSTRUCTION SET code.

Please take note, I am using a Linux Operating System, so the "Screenshot"
may look a little bit different compared to Windows Operating System.

1. Create a folder that is easiest to access via "command line". Copy the
file "intro_avr.ino.elf" from the folder listed by the Compilation process
into this folder

2. Use file explorer, go to <arduino install folder>/hardware/tools/avr/bin
Copy "avr-objdump" excutable file into the same folder where you put your
"intro_avr.ino.elf" file



```
_work: bash — Konsole
File Edit View Bookmarks Settings Help
dell@dell-OptiPlex-780:~/_work$ pwd
/home/dell/_work
dell@dell-OptiPlex-780:~/_work$ ls -l
total 992
-rwxr-xr-x 1 dell dell 1005424 May 26 2020 avr-objdump
-rwxrwxr-x 1 dell dell 7644 Dec 2 09:17 intro_avr.ino.elf
dell@dell-OptiPlex-780:~/_work$ ./avr-objdump -S intro_avr.ino.elf > tmp.txt
dell@dell-OptiPlex-780:~/_work$
```

Some of you might have noticed, I am using Optiplex 780 desktop computer, which is more than 10 years old. I get this computer from the “besi buruk” junk-yard store for RM60.

I installed Linux and able to do all these tutorials and run Arduino IDE comfortably

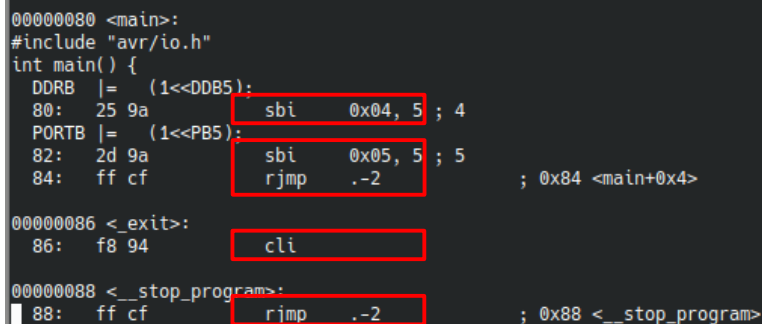
As you can see from the screenshot, we have two files in the work folder.

1. **intro_avr.ino.elf** (from the compilation process)
2. **avr-objdump** (an utility software from the Arduino installation)

The **avr-objdump** utility allows us to see what is inside the **.elf file**. You can also use this utility to see how efficient is your code. To run this utility, simply use the command line and key in the following,

```
avr-objdump -S intro_avr.ino.elf > tmp.txt
```

We will get a text file name tmp.txt. Use any text editor and open the tmp.txt file



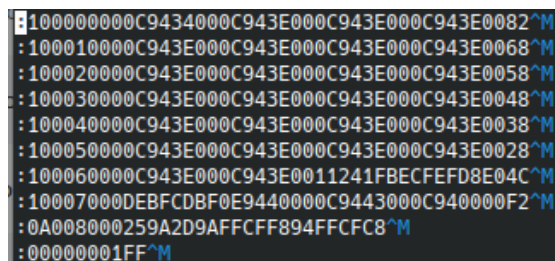
```
00000080 <main>:
#include "avr/io.h"
int main() {
  DDRB |= (1<<DDB5);
80: 25 9a      sbi    0x04, 5 ; 4
  PORTB |= (1<<PB5);
82: 2d 9a      sbi    0x05, 5 ; 5
84: ff cf      rjmp   .-2      ; 0x84 <main+0x4>

00000086 <_exit>:
86: f8 94      cli

00000088 <__stop_program>:
88: ff cf      rjmp   .-2      ; 0x88 <__stop_program>
```

You will see alot of things in there, just ignore them and look for our code. Did you see the conversion from “**DDRB |= (1<<DDB5);**” to “**sbi 0x04,5**” ? We also have other conversion, CLI and RJMP also from the “AVR INSTRUCTION SET”

This is not the final code yet, .elf file will be converted into a pure hex file before uploading into the micro controller, intro_avr_ino.hex below



```
100000000C9434000C943E000C943E000C943E000C943E0082^M
:100010000C943E000C943E000C943E000C943E0068^M
:100020000C943E000C943E000C943E000C943E0058^M
:100030000C943E000C943E000C943E000C943E0048^M
:100040000C943E000C943E000C943E000C943E0038^M
:100050000C943E000C943E000C943E000C943E0028^M
:100060000C943E000C943E0011241FBECFEFD8E04C^M
:100070000DEBFCDBF0E9440000C9443000C940000F2^M
:0A0080000259A2D9AFFCFF894FFCFC8^M
:000000001FF^M
```

IF you want to see the final .hex file you can open the intro_avr_ino.hex from the compilation process folder with any text file editor

So now we make up our minds, it is going to be

C-Language