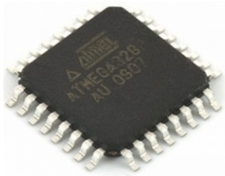
ATMEGA328/Arduino Uno – c_language
https://github.com/teaksoon/lmaewapm



First, we need the ATMEGA328P Datasheet from the Supplier of the ATMEGA328P Micro-controller (Arduino Uno is using this chip)

For everyones convenience, I have a copy of the ATMEGA328P 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. For now, just take a look at these two topic from the ATMEGA328P Datasheet (do not worry if they do not make any sense for 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

The micro-controller Central Process Unit (CPU) only can understand instructions from the **AVR INSTRUCTION SET** to manipulate the BITS inside the **ATMEGA328P CPU Memory (REGISTERS).** The bahavior or the micro-controller depends on what is inside the CPU Memory(REGISTERS)

Example: ***SBI 0x04, 5***

The code above has an instruction ***"SBI"*** from the **AVR INSTRUCTION SET.**
"SBI" is an instruction to **turn a single BIT from a REGISTER into 1**

"0x04" is one of the many **CPU Memory REGISTER**, ***0x04*** also known as **DDRB**

When the CPU gets the instruction, ***"SBI 0x04, 5"***
1. The **BIT** at position **#5** from the **DDRB REGISTER** will be set to **1.**
2. When the **DDRB REGISTER BIT** at position **#5 is 1**, the **Arduino Uno I/O Pin 13(PB5)** will become an **OUTPUT Pin**

Turning the Arduino Uno I/O Pin into an OUTPUT Pin is just one step. After the I/O Pin becomes an OUTPUT Pin, what do we do with the OUTPUT Pin ? Alot more more instruction codes will be needed, before we can get something useful working (a PROGRAM)

Since our PROGRAM will have alot of instruction codes, the instruction codes needs to be arranged based on a set of rules. The code arrangement and rules are Programming Language. Sometimes, the Programming Language have its own "instruction set" in addition to the micro-controllers "INSTRUCTION SET"

For example, from the code above ***"SBI 0x04,5"***,
- we have a **comma ,** between "0x04" and "5"
- we also have **space** between "SBI" and "0x04"
those are not from our "AVR INSTRUCTION SET" but rules from the Programming Language that we use (this coding is from the Assembly Language)

We need to convert all those extras from our Programming Language PROGRAM into a pure AVR INSTRUCTION SET codes. We also need to converted the AVR INSTRUCTION SET into just 1s and 0s since our micro-controller CPU and Memory works only with 1's and 0's at machine level

Some of you may be curious, why we use C-Language instead of Assembly Language

Lets look at this line of Assembly Language code: ***"SBI 0x04, 5"*** looks simple enough. It looks even easier than the other two options,
- C-Language with Avr Libraries we code: ***"DDRB = DDRB | (1<<5);"***
- C-Language with Arduino Libraries we code: ***"pinMode(13, 5);"***

For small PROGRAM, yes... Assembly Language is a great option. We build our PROGRAM by accumulating instructions from the "INSTRUCTION SET" one by one, step by step.

Lets look at another example, this will probably change your minds about coding in Assembly Language. Lets say we want to 250ms delay,

- C-Language with Avr Libraries we code: ***"_delay_ms(250);"***
- C-Language with Arduino Libraries we code: ***"delay(250);"***

in Assemby Language, there is no ***"_delay_ms(250)"*** or ***"delay(250)".*** We have to make our own. The code below is for **delay, fixed at 250ms**, using Assembly Language. If we want a flexible 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
```

We can make PROGRAM in C-Language without using INSTRUCTION SET or the CPU Memory (REGISTERS), does not mean C-Language is unable to do things that the Assembly Language can do.

The final uploaded code from a C-Language PROGRAM or the Assembly Language PROGRAM is the same AVR INSTRUCTION SET codes manipulating 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:

```
__asm__("SBI 0x04, 5"); // Same as coding in Assembly Language

DDRB = DDRB|(1<<5); // No Assembly Code but updating CPU Memory 0x04(DDRB)
```

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...)

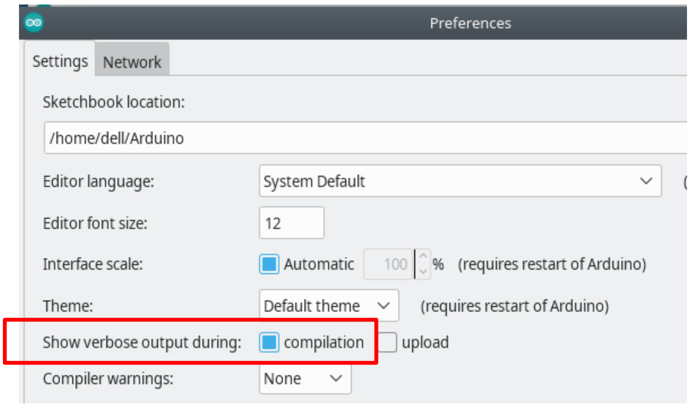
Remove all the all the unecessary remarks and codes and make the final compact executable hex file that can be executed directly by the mirco-controller CPU (stored in .hex file)

AVR Micro-controller Upload Software

.hex file is now stored in the Micro-Controller PROGRAM MEMORY in binary, 1s and 0s, to be executed by the Micro-Controller CPU

To see Compiled C-Language Source Codes

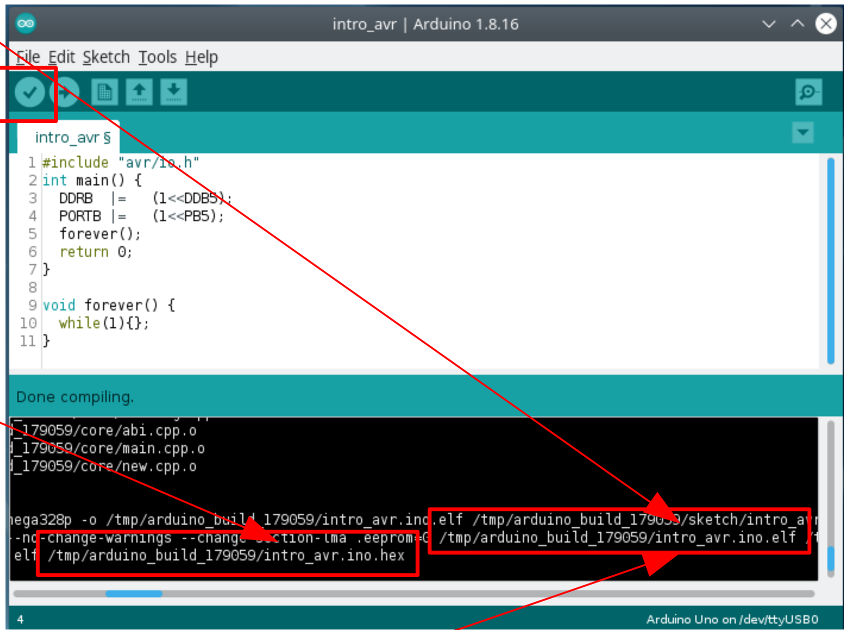
From the Arduino IDE Software, Select “File | Preferences”



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 the Optiplex 780 Desktop Computer, which is more than 10 years old. I got this Computer from the "besi buruk" junk-yard store for just RM60

I installed Linux Operating System and able to do all these tutorials and run the 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 taking place, ***"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

```
:100000000C9434000C943E000C943E000C943E0082^M
:100010000C943E000C943E000C943E000C943E0068^M
:100020000C943E000C943E000C943E000C943E0058^M
:100030000C943E000C943E000C943E000C943E0048^M
:100040000C943E000C943E000C943E000C943E0038^M
:100050000C943E000C943E000C943E000C943E0028^M
:100060000C943E000C943E0011241FBECFEFD8E04C^M
:10007000DEBFCDBF0E9440000C9443000C940000F2^M
:0A008000259A2D9AFFCFF894FFCFC8^M
:00000001FF^M
```

If you want to see the final .hex file(picture above) you can open the intro_avr.ino.hex from the compilation process folder with any text editor

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

C-Language