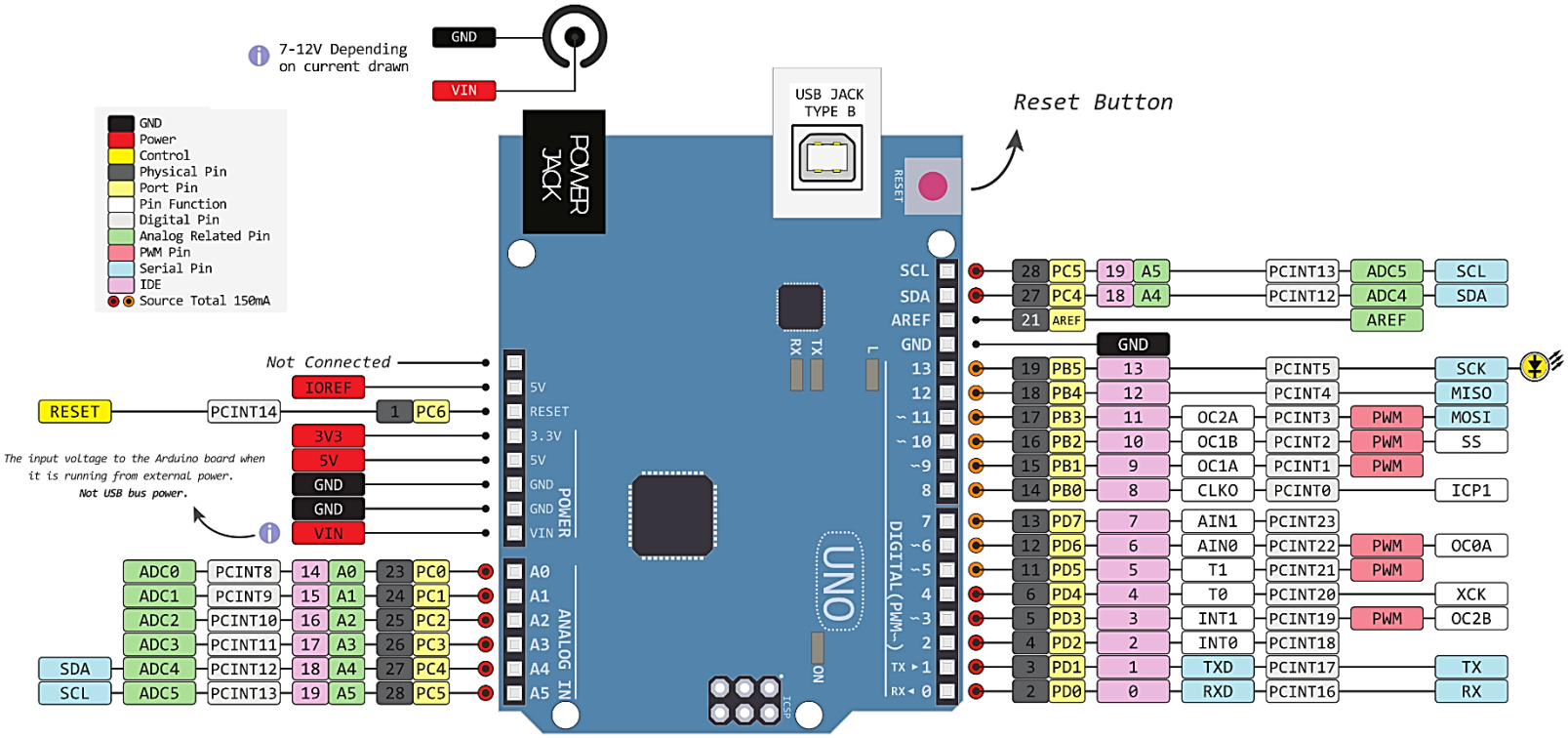
Lab 12 - Introduction to AVR Assembly

As we are going to use ATmega328P MCU (microcontroller), we need a development board having that MCU so that we can easily connect it to our computer for programming and connect different peripherals devices (sensors and actuators) to the microcontroller easily without doing any soldering and wiring stuff.

12.1 Microcontroller Pinout

Pinout of any electronic device means the description and definition of its exposed pins. These pins are used to communicate with the device.

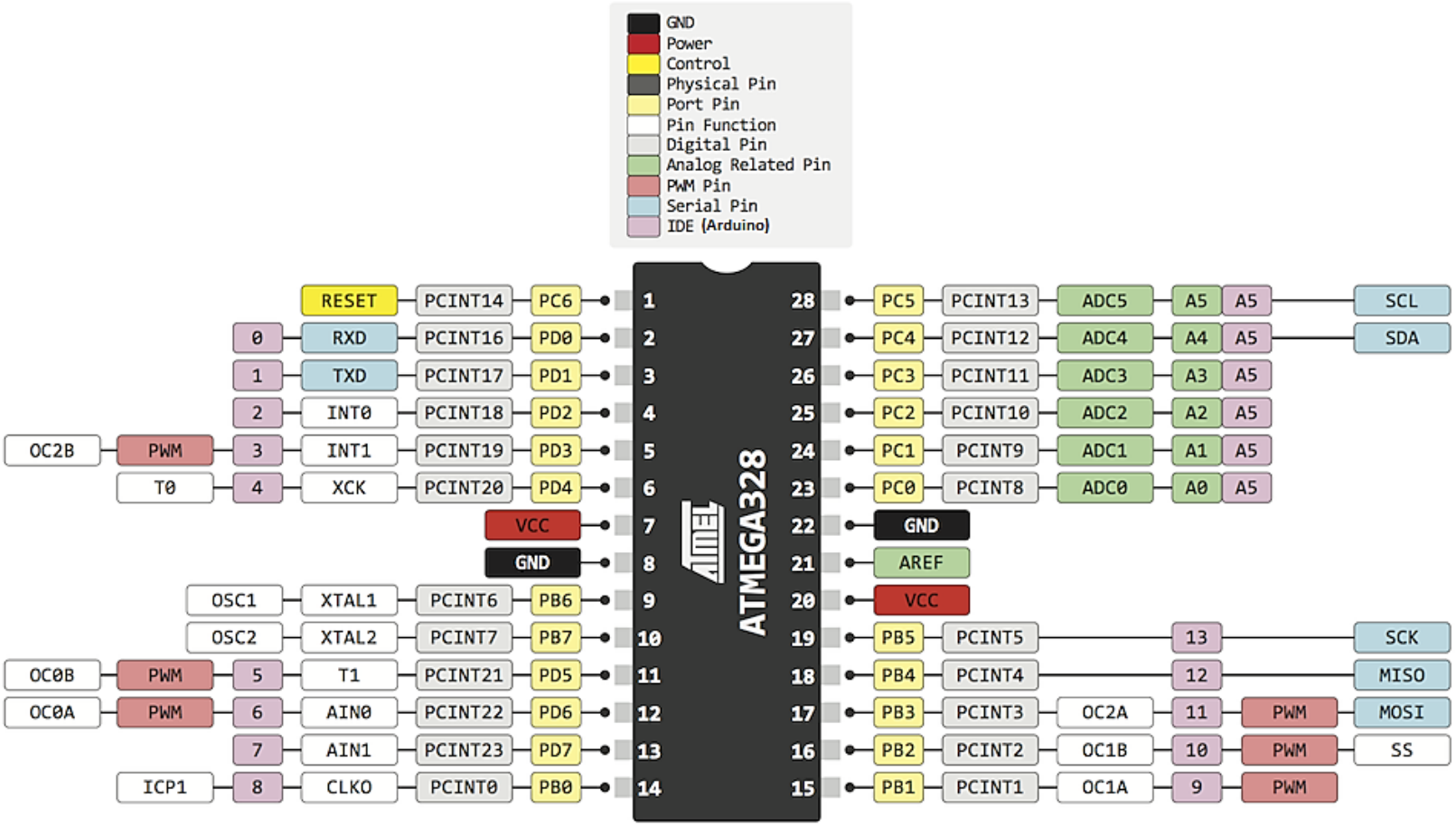
12.1.1 Arduino UNO Pinout



Important Note: In this pinout diagram we will mainly focus on the "*Port Number*" of the I/O pins. For example, if we want to turn ON and OFF a digital pin "*D13*" on the Arduino board, we will perform this function using Assembly language by changing I/O Register value that controls this D13 pin. In this case D13 pin is controlled by port "*PB5*" (Bit number *5* of the "*PORTB*" register of ATmega328P).

12.1.2 ATmega328P Pinout

The microcontroller used in the Arduino UNO development board is “ATmega328P”. We can also use this MCU directly without Arduino development board. If you want to use the standalone MCU then this is the pinout diagram indicating the I/O pins, power pins and programming pins, etc.

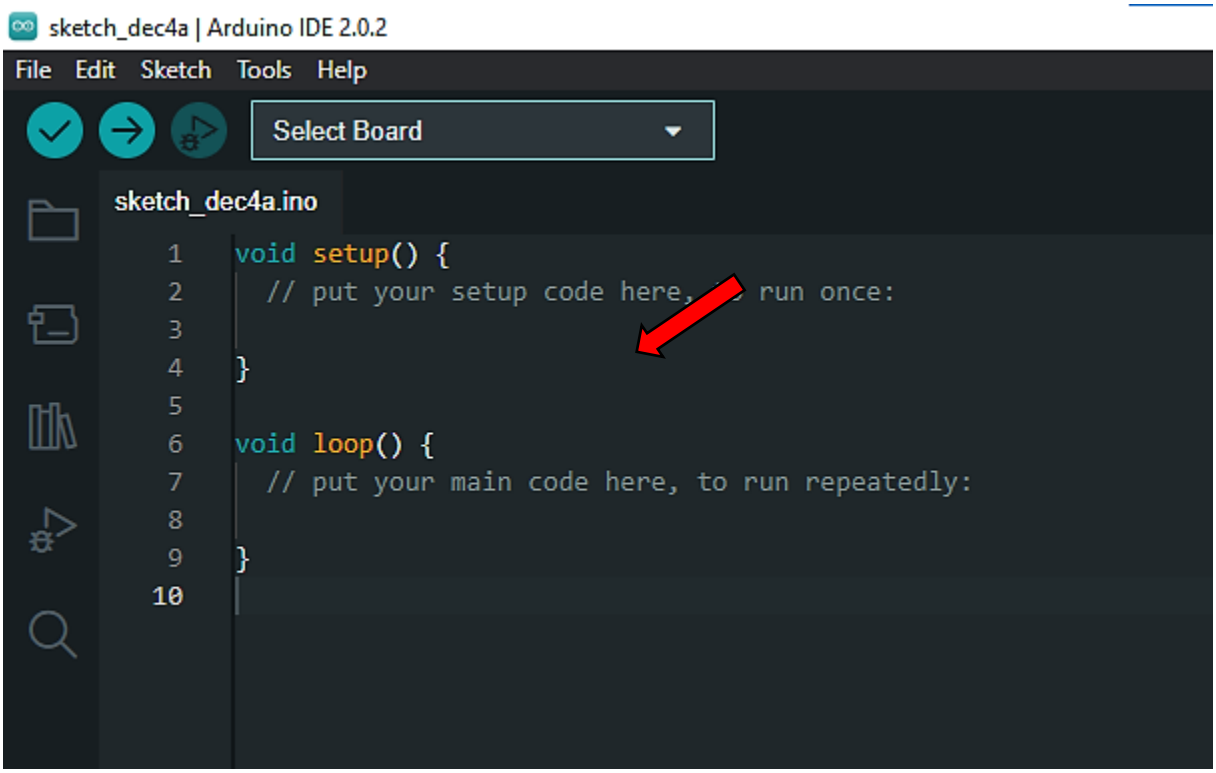


Note: The programming of a standalone MCU is done by a special programming device called “AVR ISP Programmer”. But using this standalone MCU is not ideal for learning and testing purposes. Standalone MCU is used in end-products where customized circuit boards are developed according to the product then the MCU is installed on that circuit board after programming. So, we will only use Arduino UNO development board.

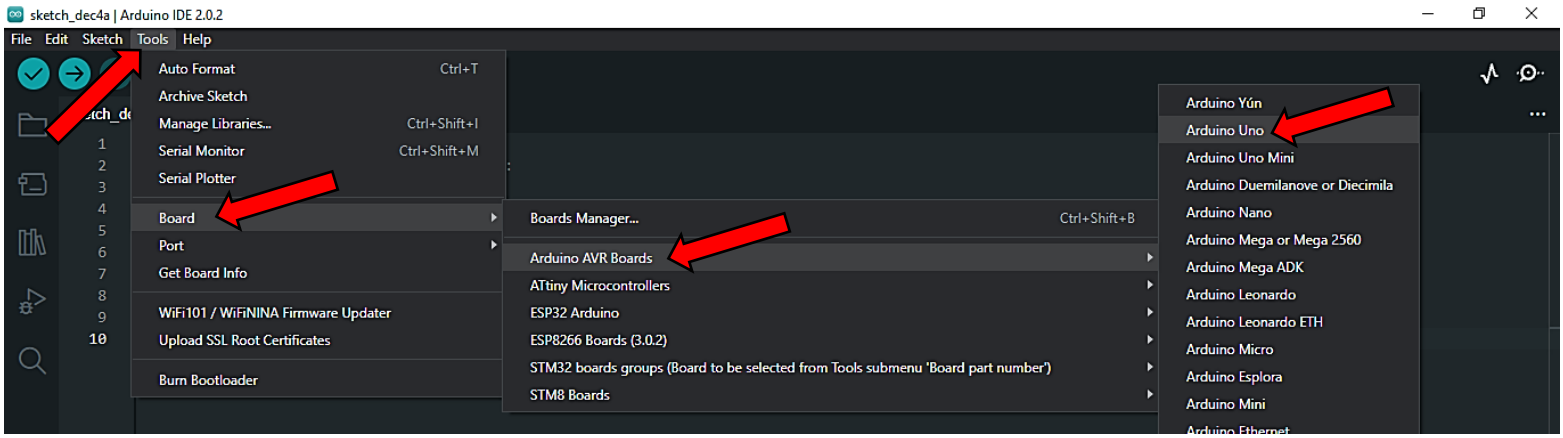
12.2 First AVR Program for ATmega328P Microcontroller

Now after understanding the hardware of the Arduino UNO board, we will write a small code to understand the working of this microcontroller. So, we will start programming in Arduino IDE and write an example program, then we will move forward and write that same program in AVR Assembly language using “Microchip Studio”.

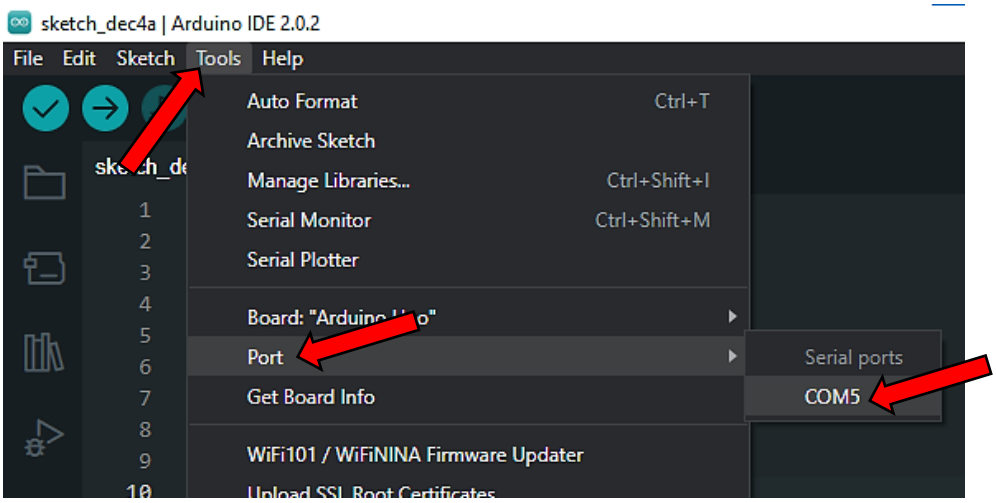
First, install Arduino IDE (latest version) from Arduino’s official website. After installing the Arduino IDE open it. A blank window will open containing some lines of code.



Now connect the Arduino UNO with the USB port of the computer and click on the “Tools” menu, go to “Board” then “Arduino AVR Boards” menu and select the “Arduino Uno” board.



Then again go the “Tools” menu, click on the “Port” menu, and select the port of the Arduino Uno.

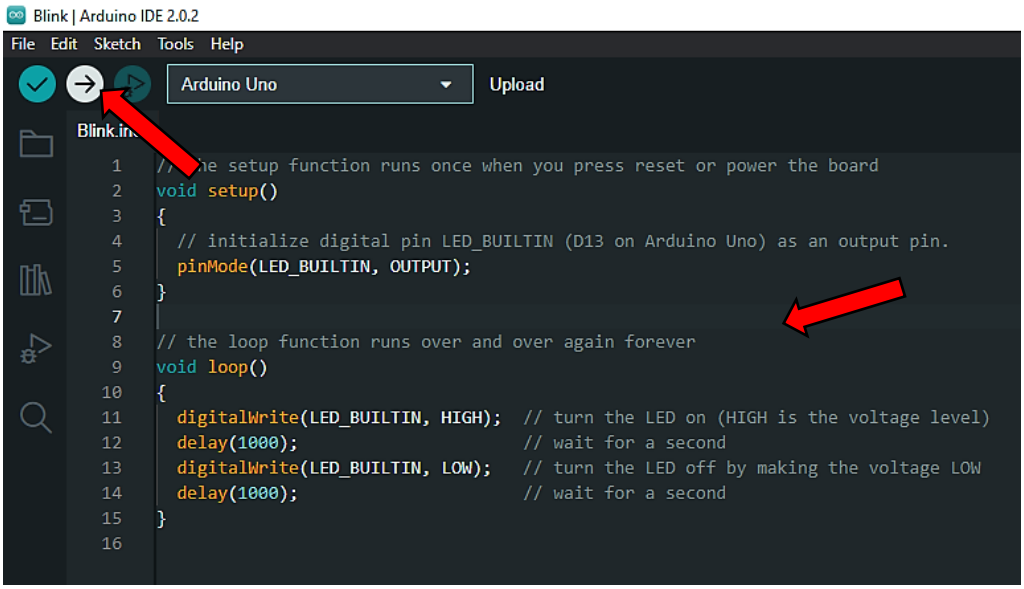


Now write the following example code in the Arduino IDE:

```
// the setup function runs once when you press reset or power the board
void setup()
{
  // initialize digital pin LED_BUILTIN (D13 on Arduino Uno) as an output pin.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop()
{
  digitalWrite(LED_BUILTIN, HIGH);  // turn the LED on (HIGH is the voltage level)
  delay(1000);                      // wait for a second
  digitalWrite(LED_BUILTIN, LOW);   // turn the LED off by making the voltage LOW
  delay(1000);                      // wait for a second
}
```

Then click on “Upload” button to compile and upload this code to Arduino Uno.



After successfully upload the code to Arduino, an LED light on the Arduino Uno board will start blinking. This LED is blinking according to the code we uploaded to the ATmega328P MCU. The LED is attached to D13 pin of the Arduino (PB5 pin of ATmega328P).

12.2.1 Code Breakdown

Now we understand the above code line by line. The main structure of the program consists of an infinite loop where the whole logic of our program is implemented. Then there is a function named "Setup()" it called once after the MCU power on then the MCU will start executing the code in "Loop()" function.

Note: *Loop()* function is just an ordinary *while()* function to implement the infinite loop. (e.g., *while(1)*).

The *PinMode()* function is called once. It is used for configurations and other initializations. It is also used to declare a pin as either input or output. It accepts two parameters: pin number and the required pin mode INPUT or OUTPUT.

```
pinMode(5, INPUT);           //setting Arduino Uno's pin D5 as input pin
pinMode(13, OUTPUT);         //setting Arduino Uno's pin D13 as output pin
```

Important: An important feature of any MCU is that any I/O pin of the MCU can be configured/reconfigured as input or output pin at any time in the program.

Then in the loop() function we used *digitalWrite()* function, this function is used to turn a pin high or low. High means the voltage level on that pin becomes 5v and low means the voltage level on that pin becomes 0v. The digitalWrite() function only works if you have already configured that pin as an OUTPUT pin using pinMode() function.

```
digitalWrite(5, HIGH); //setting D5 pin to High
digitalWrite(13, LOW); //setting D5 pin to Low
```

Delay() function is used to pause the execution of the program for specific number of milliseconds.

```
delay(1000);     // pause 1 second
delay(300);      // pause 300 milliseconds
```

Similarly, we can configure a pin as input to read the data coming from a sensor or detect the pin state using a push button on that pin. For that purpose, we will first configure that pin as input then call *digitalRead()* function to get the state of the pin. Or we can use *analogRead()* function to read the analog data reading coming from some sensor attached on that pin.

```
void setup()
{
  pinMode(5, INPUT);      // setting D5 pin as INPUT pin
  pinMode(A0, INPUT);     // setting A0 pin as INPUT pin
}
void loop()
{
  bool x = digitalRead(5);   // get the pin status (HIGH or LOW --> 5v or 0v)
  float y = analogRead(A0);  // read the analog value on the analog pin A0
                             // (in the range of 0-255)
}
```

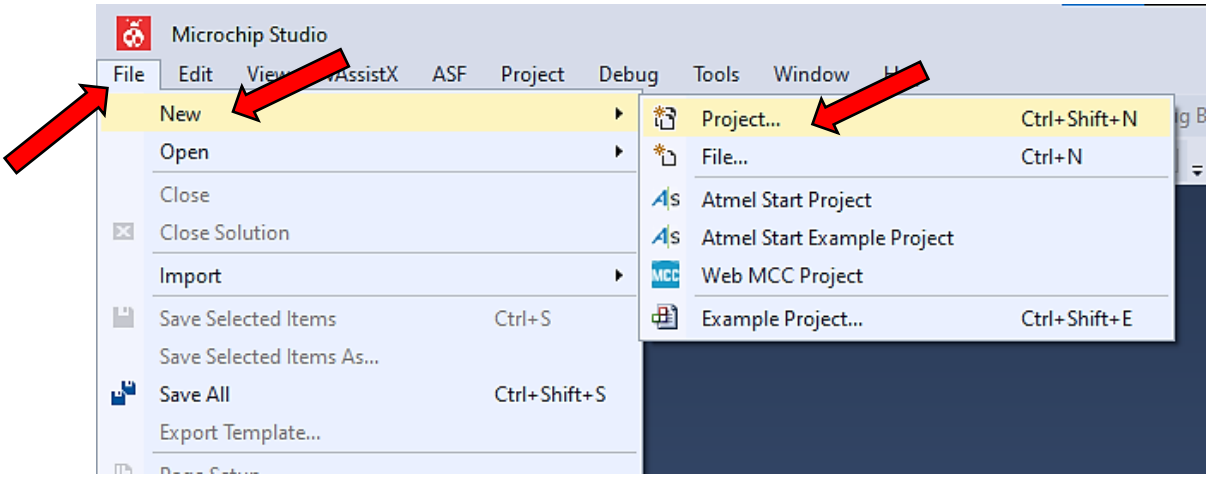
Student Task: Perform the analog reading and get the value from a sensor (e.g., LDR sensor) and show the results on the Console window in Arduino IDE.

12.3 AVR Assembly

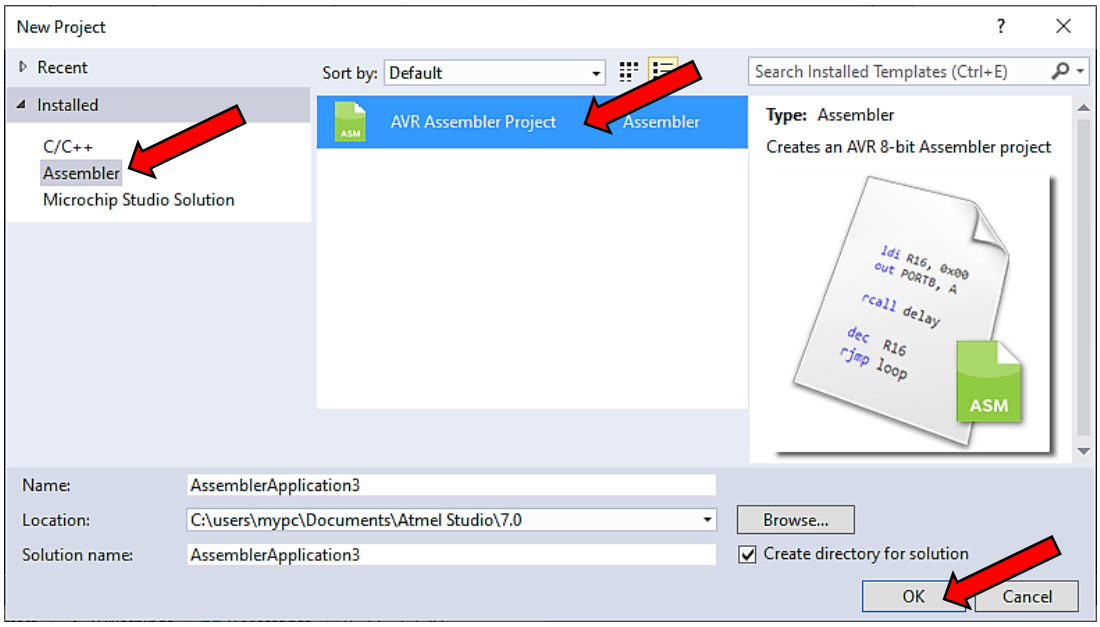
Now we write the same code of LED controlling in AVR Assembly language. One of the main differences between x86 assembly and AVR assembly is that in AVR assembly the chip executes anything you tell it to do and does not ask you if you are sure overwriting this and that. All protections must be programmed by you, the chip does anything like it is told. No window warns you unless you programmed it before.

However, the testing and debugging programs on MCU chips is very easy. If it does not do what you expect it to do, you can easily add some diagnostic lines to the code, reprogram the chip and test it.

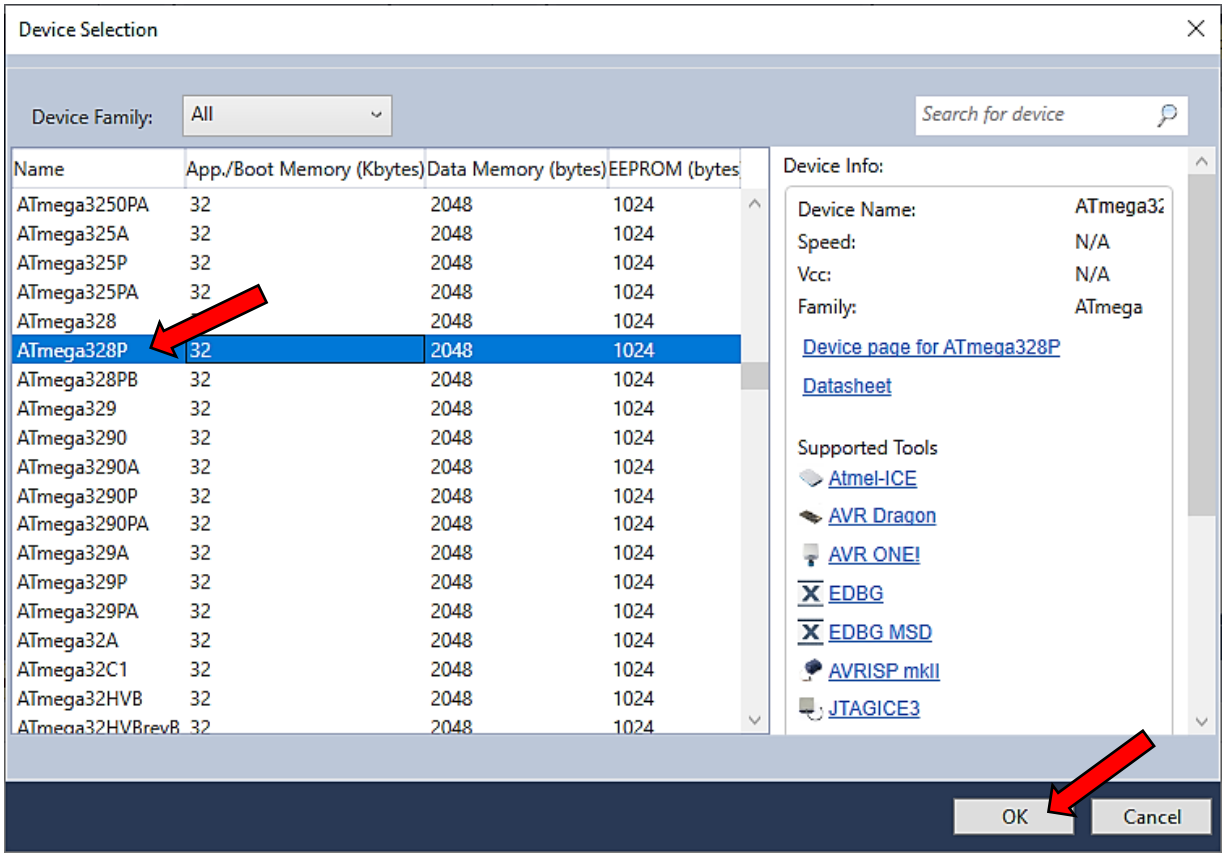
First, open Microchip Studio Then go to “File” menu. Go to “New” and press “Project” button.



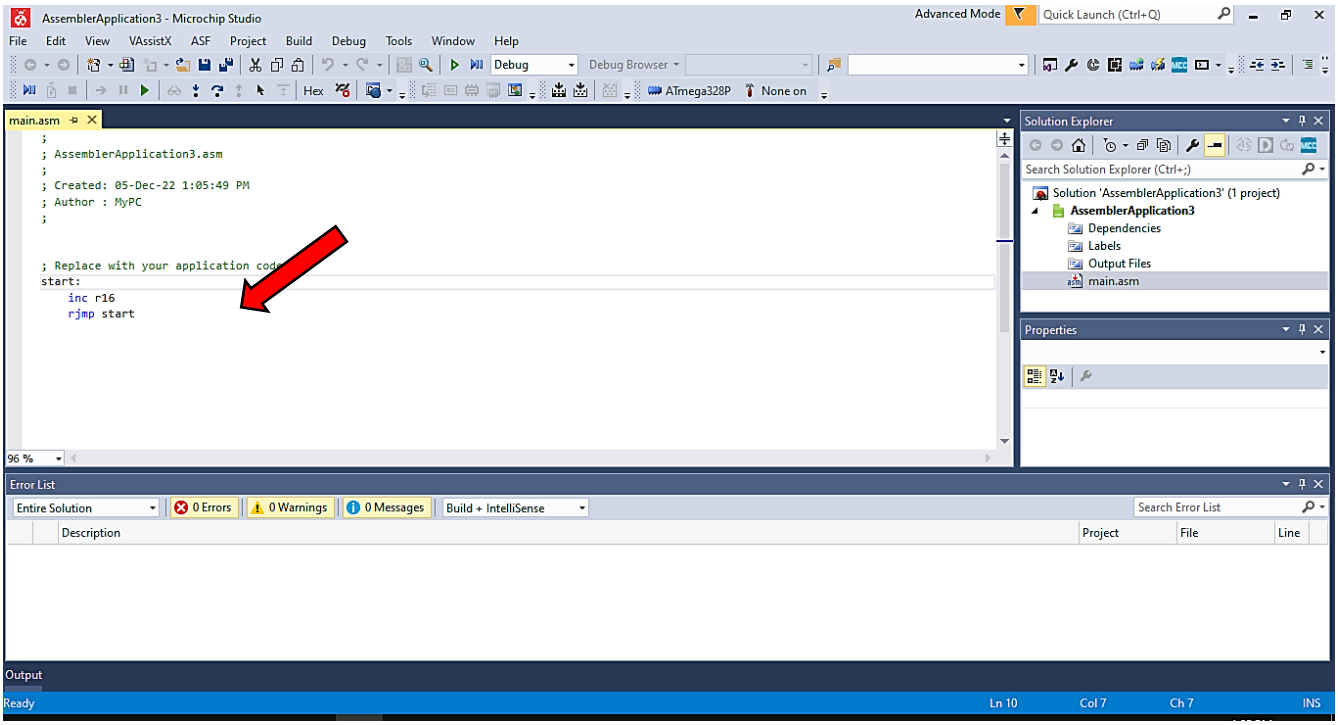
A dialog box will appear. Select “Assembler” from the left panel then select “AVR Assembler Project” and press “OK” button.



Then on the next screen, find the “ATmega328P” from the list and select it then press “OK” button.



An empty AVR Assembly project will be created having an example file.



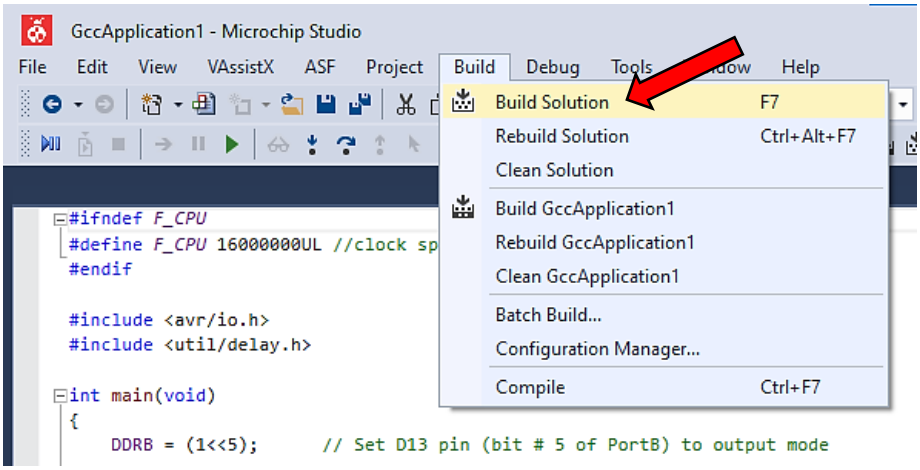
Erase the existing code and write the following assembly code in the file then build the project by pressing F7 key then upload the code to Arduino Uno from “Tool” menu then pressing the “UNO - Program via Bootloader” button.

```
.include "m328pdef.inc"
.cseg
.org 0x00
      LDI r16, (1<<PINB5)       ; load 00100000 into register R16
      OUT DDRB, r16             ; write register R16 value to DDRB register
      LDI r17, (1<<PINB5)       ; load 00100000 into register R16
      OUT PORTB, r17            ; write register R16 value to PORTB register

loop:
rjmp  loop                      ; stay in infinite loop
```

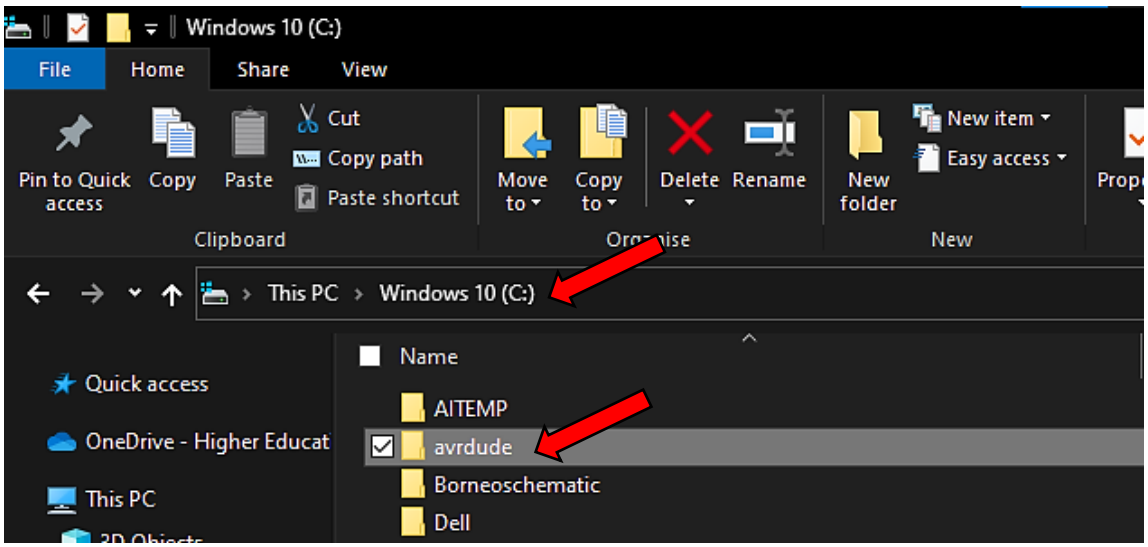
This will turn on the LED attached to D13 pin of Arduino Uno (built-in LED on Arduino Uno development board).

Then go to “Build” menu and press “Build Solution”. Or press “F7” key on keyboard. It will compile the program.

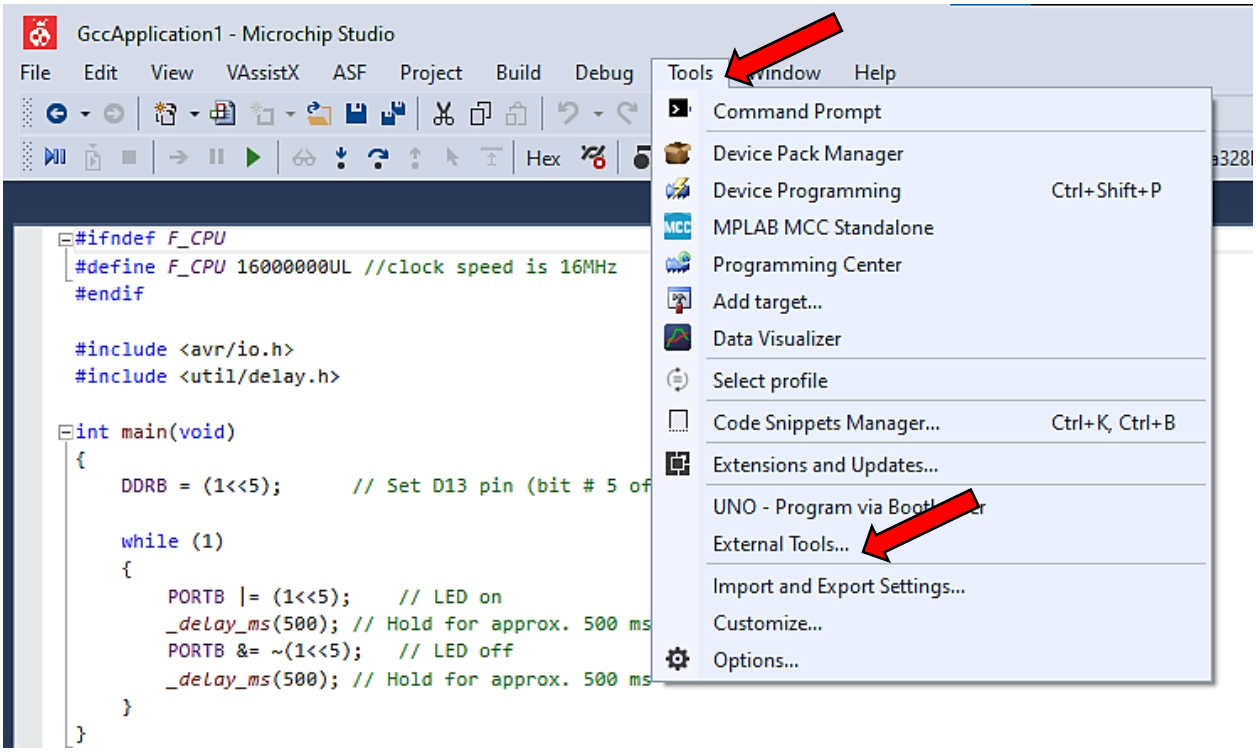


To upload the compiled program to the MCU, we need to configure the target device (Arduino Uno in our case).

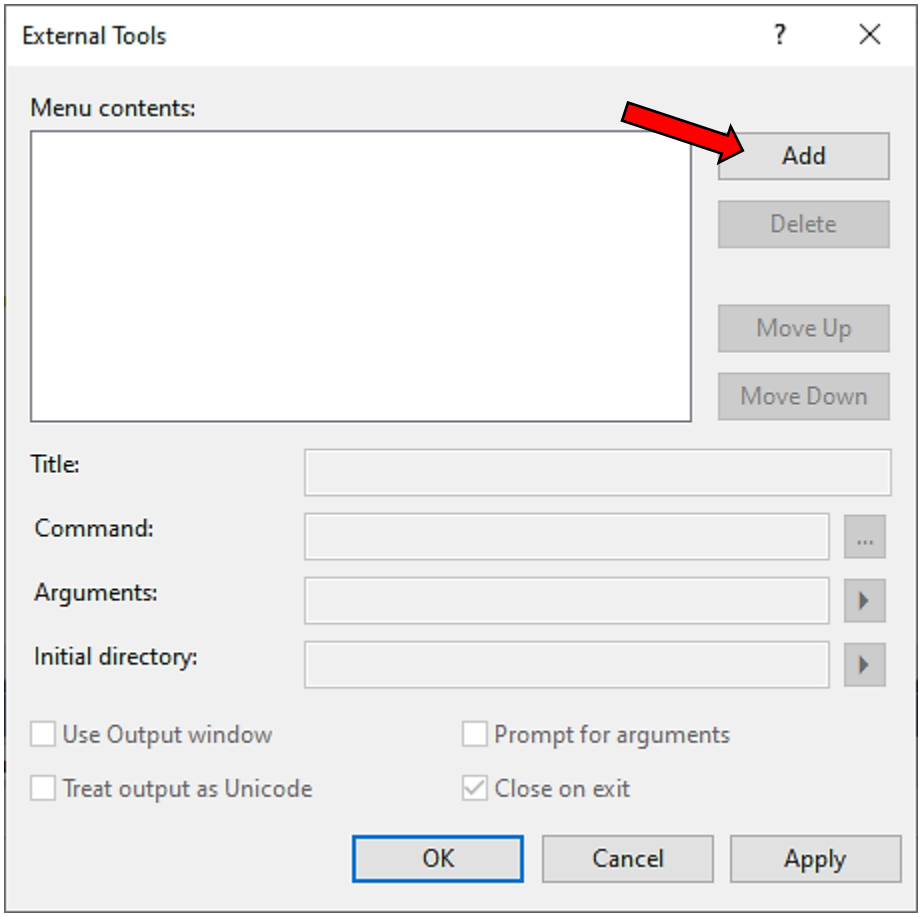
First of all, go to this **Link** and download the zip file named “avrdude.zip” and extract it in the C drive.



Then go to the “Tools” menu and press the “External Tools…” button.



A dialog box will appear. Click on the “Add” button to add the Arduino Uno as a target device.



Then enter the following data in the text boxes:

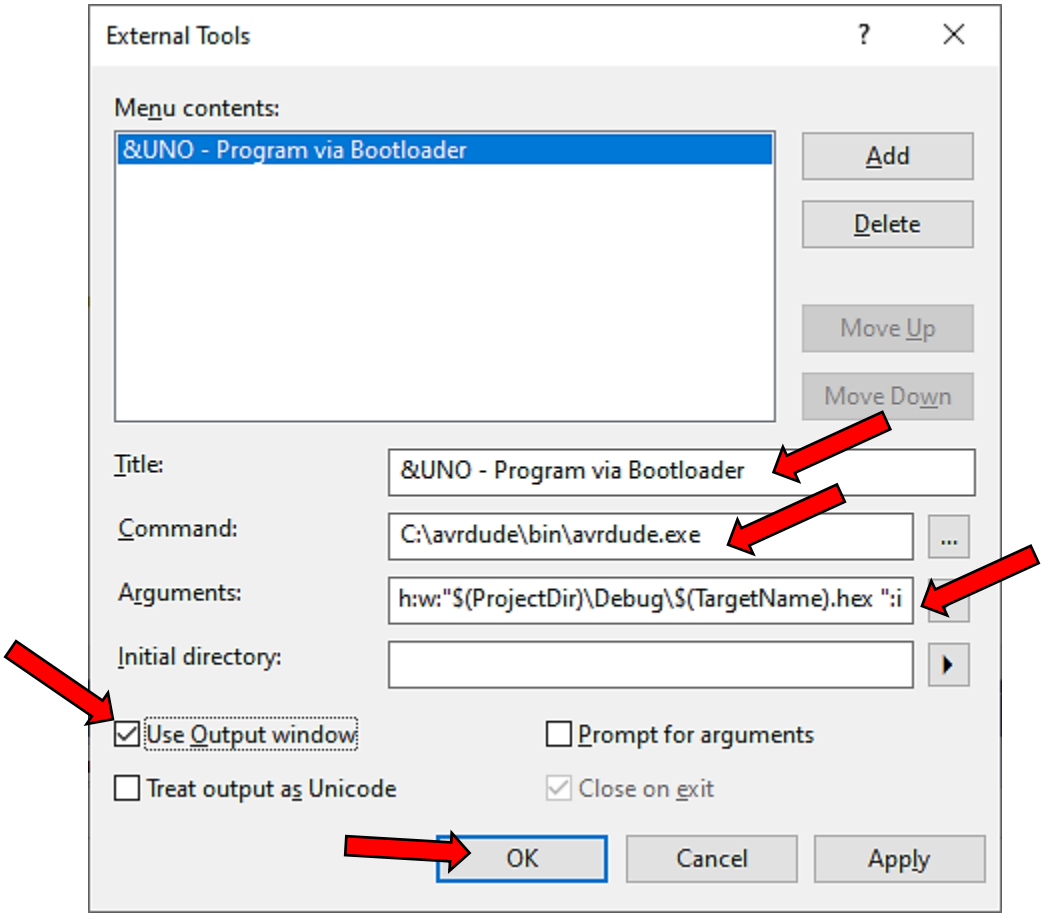
Title: *&UNO - Program via Bootloader*

Command: *C:\avrdude\bin\avrdude.exe*

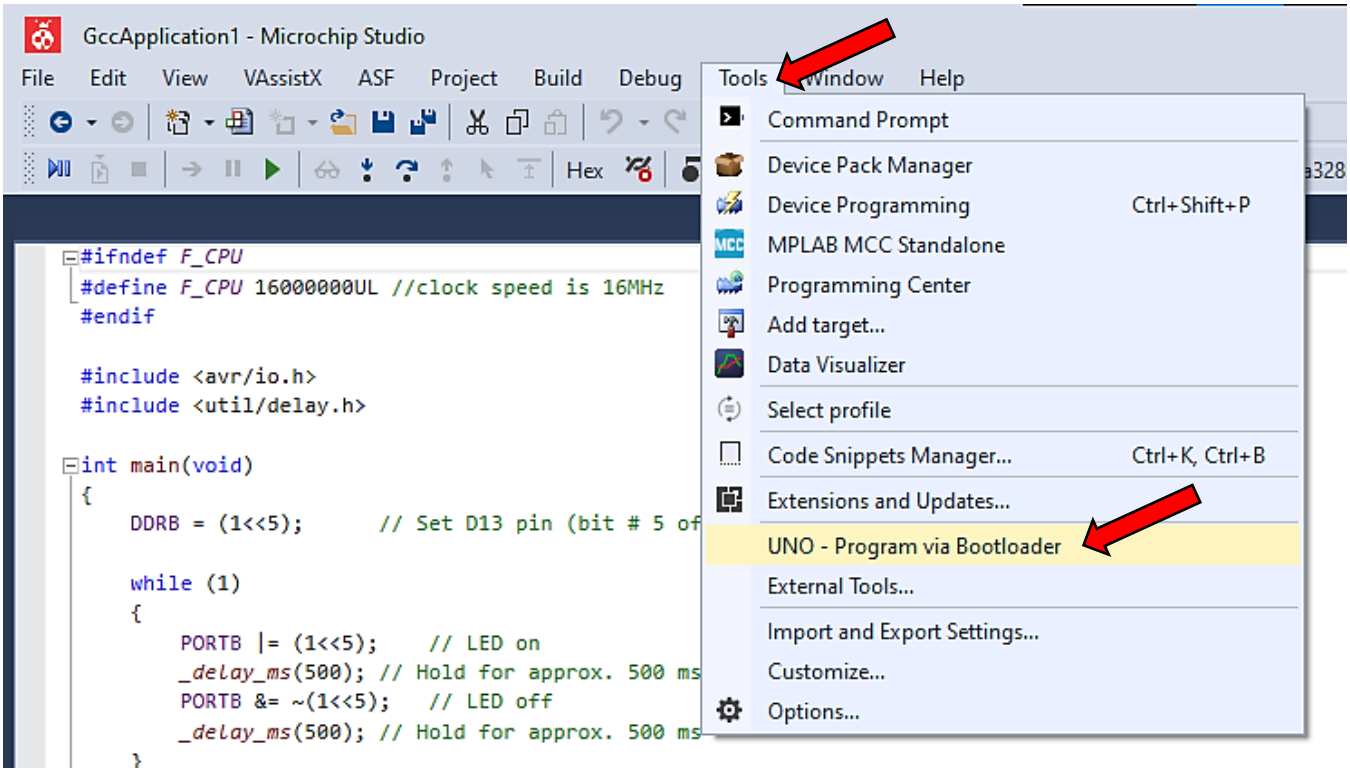
Arguments: *-CC:\avrdude\etc\avrdude.conf -v -V -patmega328p -carduino -PCOM5 -b115200 -D -Uflash:w:"$(ProjectDir)\Debug\$(TargetName).hex ":i*

Important: in the above entry, change the COM port number in ***-PCOM5*** parameter according to your Arduino Uno's COM port number.

Tick the "Use Output window" checkbox and then press "OK" button.



It will create the target device entry in the “Tools” menu. Now press this “UNO - Program via Bootloader” button to upload the compiled program to the Arduino Uno (ATmega328P MCU).



12.3.1 Code Breakdown

Now we will explain the above assembly code in detail. Just like C language, assemblers have a directive to include the contents of another file in your code, written as *.include*. In this case the file we include is "m328pdef.inc" which contains definitions specific to the microcontroller that we are using.

```
.include "m328pdef.inc"
```

The next few lines introduce the directives *.cseg* and *.org*.

```
.cseg
.org   0x00
```

The *.cseg* directive indicates that what follows is part of the code segment and should be placed in flash memory (as opposed to data memory or EEPROM).

Next, we use the *.org* directive to specify the origin of our code segment. In this case we specify the address 0x00 to put our code at the beginning of flash memory. This ensures it is executed immediately when the microcontroller starts.

Note: By default, *.cseg* will start at the beginning of flash so the line *.org 0x00* is not strictly necessary. However, it is good practice to always specify the origin to make it obvious where the code is being placed.

Now we get to our first instruction.

```
LDI r16, (1<<PINB5)       ; load 00100000 into register R16
```

The instruction *LDI* - load immediate, lets us load an 8-bit constant value into one of the general-purpose working registers between r16 and r31. Here we use the LDI instruction to place the 8-bit value 00100000 into register R16.

Next, we have the instruction *OUT*.

```
OUT DDRB, r16              ; write register R16 value to DDRB register
```

The *OUT* instruction writes the value of a general-purpose registers to one of the registers mapped to I/O memory. Here we write the contents of r16 (which is loaded with 00100000) to *DDRB* which will set PINB5 to output.

We then write the value 00100000 to *r17* register and then write it to *PORTB* register which will set the output of *PINB5* high.

```
LDI r17, (1<<PINB5)        ; load 00100000 into register R16
OUT PORTB, r17             ; write register R16 value to PORTB register
```

Note: It is important to note that we cannot directly write a constant value to an I/O register using the *OUT* instruction. For example, the following will not work.

```
OUT DDRB, 0b00100000       ; incorrect
```

Only certain instructions work with immediate constants which will usually be indicated by the word *immediate* in the name. *OUT* instruction can only take a general-purpose register as an operand which is why we had to first load the value to general-purpose register then move to I/O register using *OUT* instruction.

Now we get to the final lines of our program:

```
loop:
rjmp   loop                ; stay in infinite loop
```

Here we define a label “loop”. The loop label is followed by the instruction *RJMP* - relative jump.

12.4 Adding 16-bit Numbers

Here is another example program to give us some context before looking at the instructions that operate on general-purpose registers. In the following example we will add two 16-bit hexadecimal numbers 0x1234 and 0xABCD.

```
.include "m328pdef.inc"
       .def num1L = r16          ; define lower byte of number 1 as r16
       .def num1H = r17          ; define upper byte of number 1 as r17
       .def num2L = r18          ; define lower byte of number 2 as r18
       .def num2H = r19          ; define upper byte of number 2 as r19
.cseg
.org 0x00
       ldi    num1L, 0x34        ; load 0x34 into r16
       ldi    num1H, 0x12        ; load 0x12 into r17
       ldi    num2L, 0xCD        ; load 0xCD into r18
       ldi    num2H, 0xAB        ; load 0xAB into r19
       add    num1L, num2L       ; add lower bytes of number
       adc    num2H, num2H       ; add upper bytes of number
loop:
rjmp   loop                      ; stay in infinite loop
```

Student Task: Debug this program in the Microchip Studio by adding breakpoints on different lines and watch the values of registers and flags.

12.4.1 Code Breakdown

The *.def* directive allows us to rename a register with a more descriptive mnemonic. This mnemonic can then be used later in the code and the assembler will automatically replace the mnemonic with the register.

```
.def num1L = r16          ; define lower byte of number 1 as r16
```

Because we cannot add two 16-bit numbers directly on an 8-bit MCU, so we split the values in to two 8-bit registers to contain a single value. The we used *ADC* - add with carry instruction.

```
add    num1L, num2L       ; add lower bytes of number
adc    num2H, num2H       ; add upper bytes of number
```

When the addition operations are complete, num1H and num1L will contain the value 0xBE01, which is the sum of 0x1234 and 0xABCD.

12.5 What does digital mean?

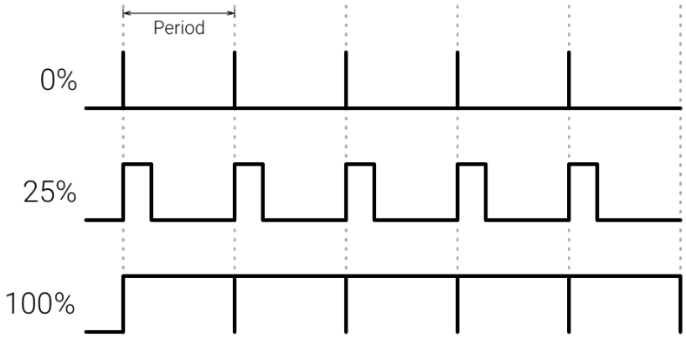
When the digital pins are configured as input, the voltage is supplied from an external device (e.g., sensor). This voltage can vary between 0-5 volts which is converted into digital representation (0 or 1). To determine this, there are 2 thresholds:

- Below 0.8v - considered as 0.
- Above 2v - considered as 1.

12.6 What is PWM?

In general, Pulse Width Modulation (PWM) is a modulation technique used to encode a message into a pulsing signal. A PWM is comprised of two key components: **frequency** and **duty cycle**. The PWM frequency determines how long it takes to complete a single cycle (period) and how quickly the signal fluctuates from high to low. The duty cycle determines how long a signal stays high out of the total period. Duty cycle is represented in percentage.

In Arduino, the PWM enabled pins produce a constant frequency of ~ 500Hz, while the duty cycle changes according to the parameters set by the user. See the following illustration:



PWN is mainly used to control the speed of motors, brightness of LEDs, controlling heaters, etc.