BIRZEIT UNIVERSITY

Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

Microprocessor and Embedded Systems Lab (ENCS5101)

Experiment No. 1 Introduction to ARM Assembly Language and Keil uVision5

Objectives

1. Introduce some of the ARM architecture to students.
2. Begin to use the lab tool - Keil uVision 5.
3. The students will create a project and write an ARM assembly language program based on a simulated target.

The ARM (Advanced RISC Machine) architecture is introduced in the class (also see http://www.arm.com.) Keil MDKARM is a complete software development toolkit for ARM processor-based microcontrollers. Keil uVision5 will be used in the lab. The ARM Cortex-M3 processor will be examined with the STM32VLDISCOVERY board. The following is some important information for you.

Important Information

1. In the lab room Masri207, computers are running the operating system Windows 10 Pro, and ARM Software Microcontroller Development Kit Version 5.21a (Keil uVision5) is installed.
2. To install it in your own computer, you can download the following files:
   ~ftp/pub/class/301/ftp/uVision5/MDK521a.EXE
   ~ftp/pub/class/301/ftp/uVision5/Keil.STM32F1xx_DFP.2.1.0.pack
   Here is the Link to Keil Tools.
3. To know more about Keil, visit http://www.keil.com/
4. To see STM32VLDISCOVERY board, visit STM32VLDISCOVERY Board.
5. To see The Cortex-M Instruction Set, visit Cortex-M Devices Generic User Guide.
6. To see more references of Cortex-M, visit the following:
   Cortex-M Technical Reference Manual - ARM Information Center
   Cortex-M programming manual

Create an ARM Assembly Language Program

To create an assembly language program, you need to use a text editor such as **NotePad** in Microsoft Windows environment. There is a text edit in the Keil uVision5 for you to use too. The file name must have a**.s** at the end. Let's look at the following program called **FirstArm.s** on a PC. The file **FirstArm.s** contains the source code of the program to load registers and demonstrate a few other operations. We will use Keil uVision5 to create a project and execute this program so that you can get a feel of how Keil uVision5 works.

```
;The semicolon is used to lead an inline documentation.
;This is the first ARM Assembly language program you see in the lab.
;This program skeleton was from Dave Duguid and Trevor Douglas in summer 2013.
;When you write your program, you could have your info at the top document block.
;For Example:  Your Name, Student Number, what the program is for, and what it does
etc.
;;; Directives
          PRESERVE8
          THUMB
; Vector Table Mapped to Address 0 at Reset
; Linker requires __Vectors to be exported

          AREA    RESET, DATA, READONLY
          EXPORT  __Vectors

__Vectors
       DCD  0x20001000     ; stack pointer value when stack is empty
                    ;The processor uses a full descending stack.
                    ;This means the stack pointer holds the address of the last
                    ;stacked item in memory. When the processor pushes a new item
                    ;onto the stack, it decrements the stack pointer and then
                    ;writes the item to the new memory location.

          DCD  Reset_Handler  ; reset vector

          ALIGN

; The program
; Linker requires Reset_Handler

          AREA    MYCODE, CODE, READONLY

       ENTRY
       EXPORT Reset_Handler

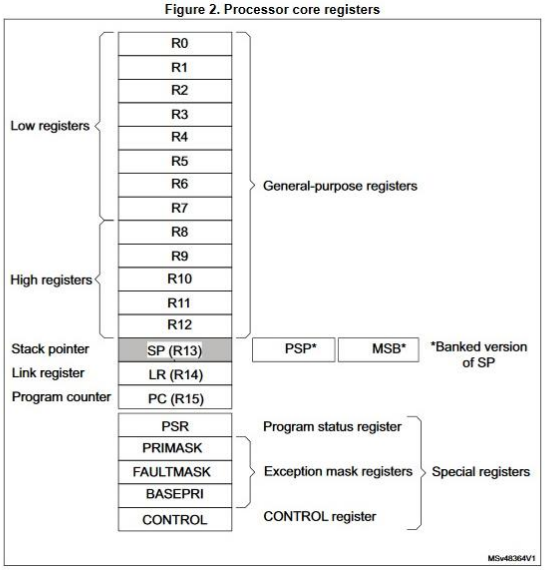

Reset_Handler
;;;;;;;;;;User Code Starts from the next line;;;;;;;;;;;;

       MOV R0, #12
STOP
       ADD R0, R0, #4
       B  STOP

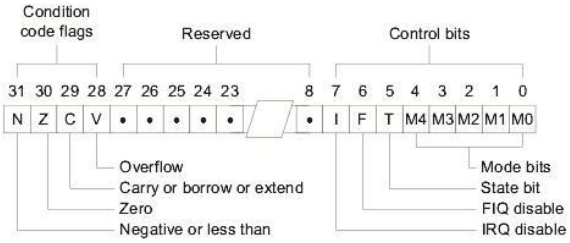
          END;End of the program
```

ARM Cortex-M Core Registers

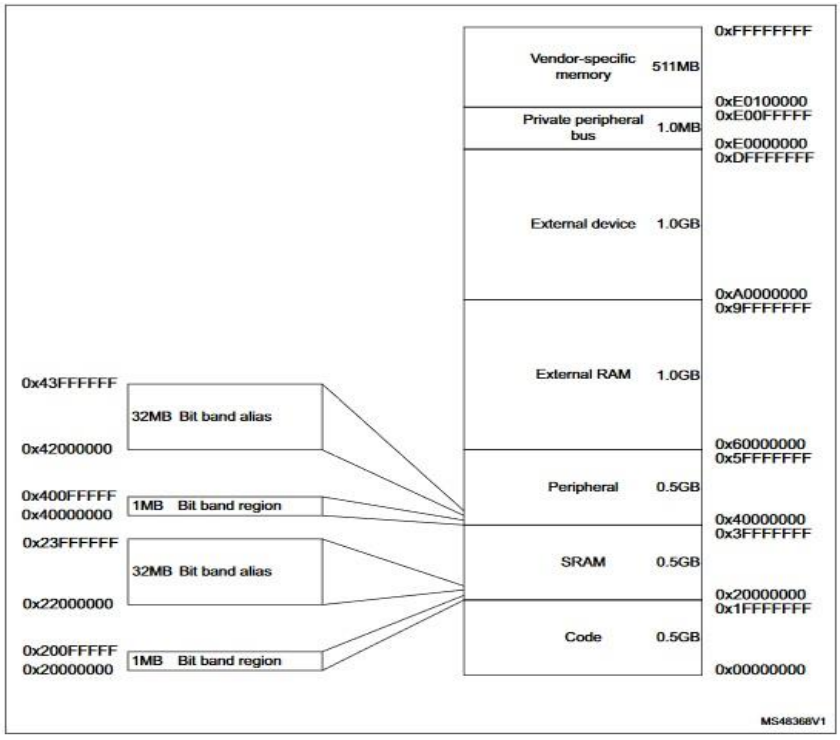
Figure 2. Processor core registers



Here is the Program Status Register Format:



ARM Cortex-M Memory Map

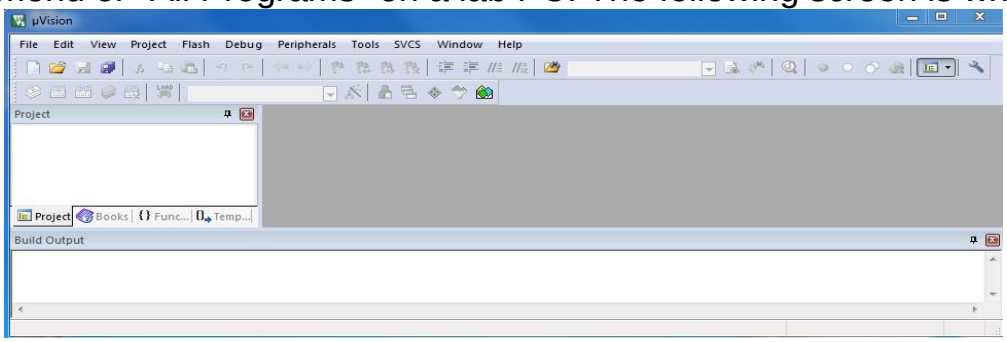


Start up Keil uVision5

Before you start up, you are recommended that you create a folder to hold all your project files. For example: you can have a folder "FirstARM-Project" ready before hand.

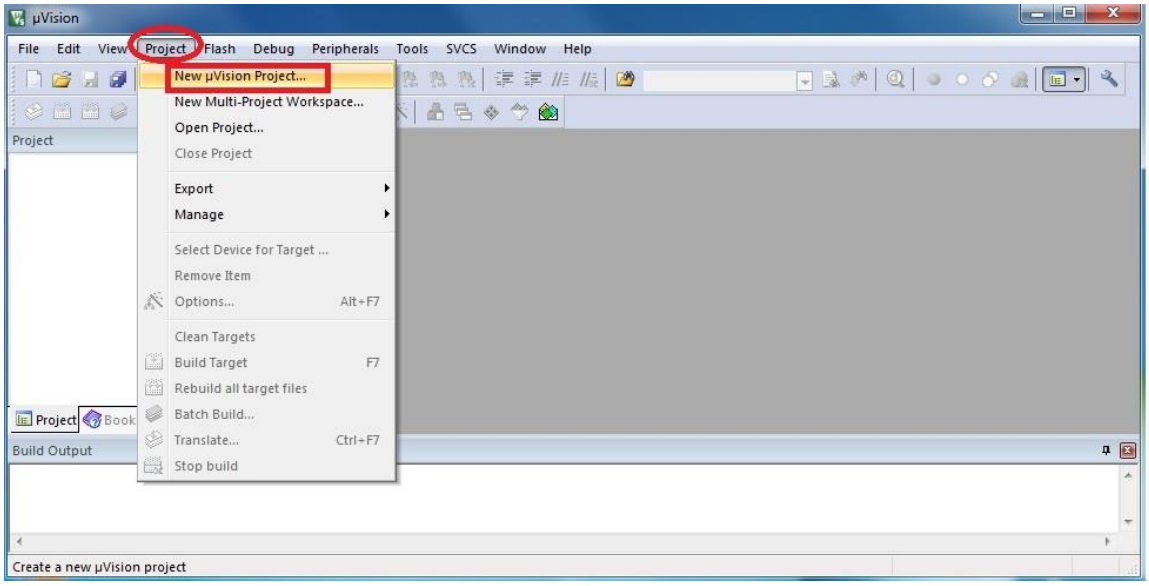
You can start up **uVision5** by clicking on the icon 

from the desktop or from the "Start" menu or "All Programs" on a lab PC. The following screen is what you will see.

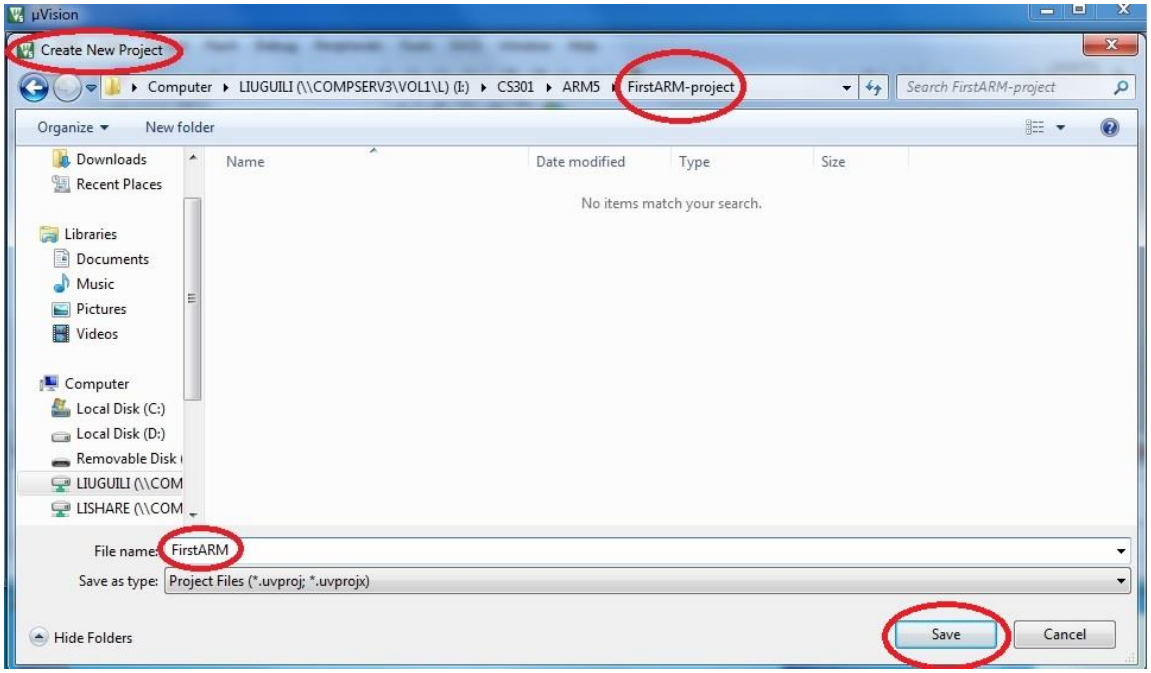


Create a project

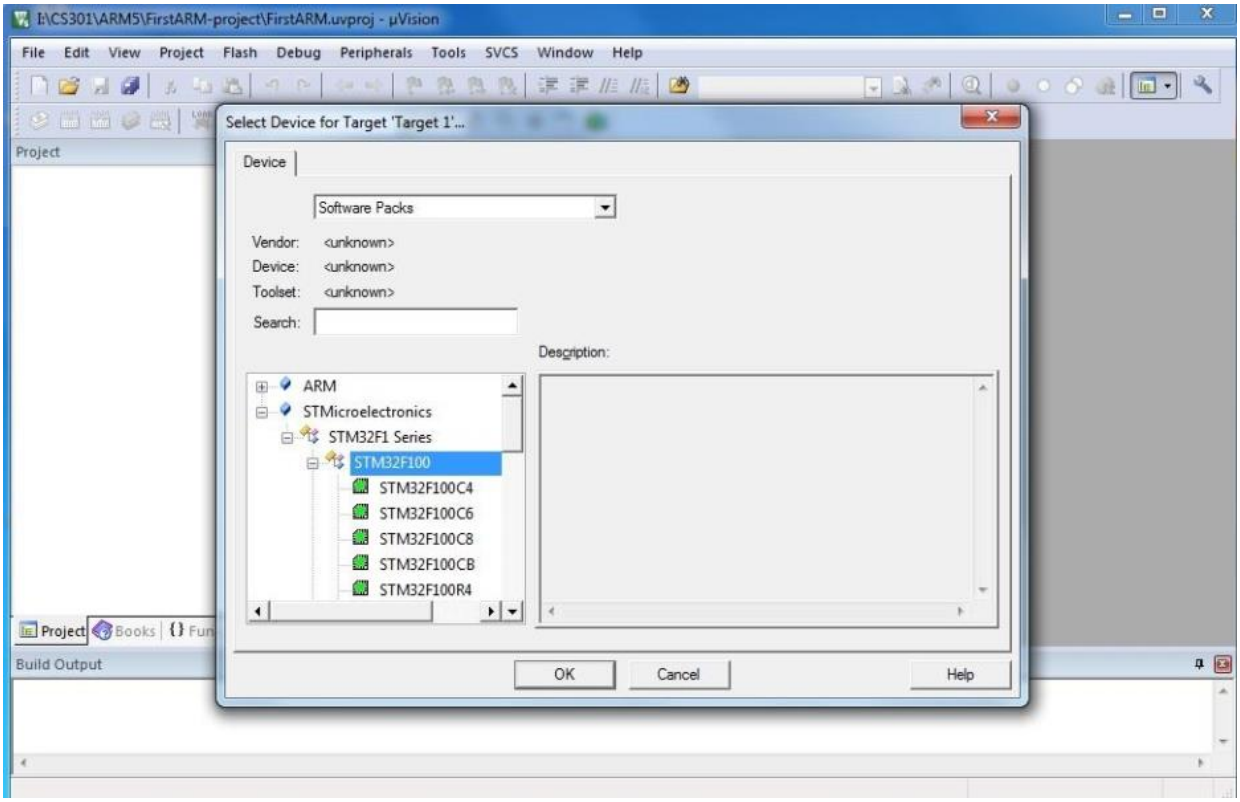
Let's create our first ARM **uVision5** project now. To create a project, click on the "Project" menu from the **uVision5** screen and select "New uVision Project...".



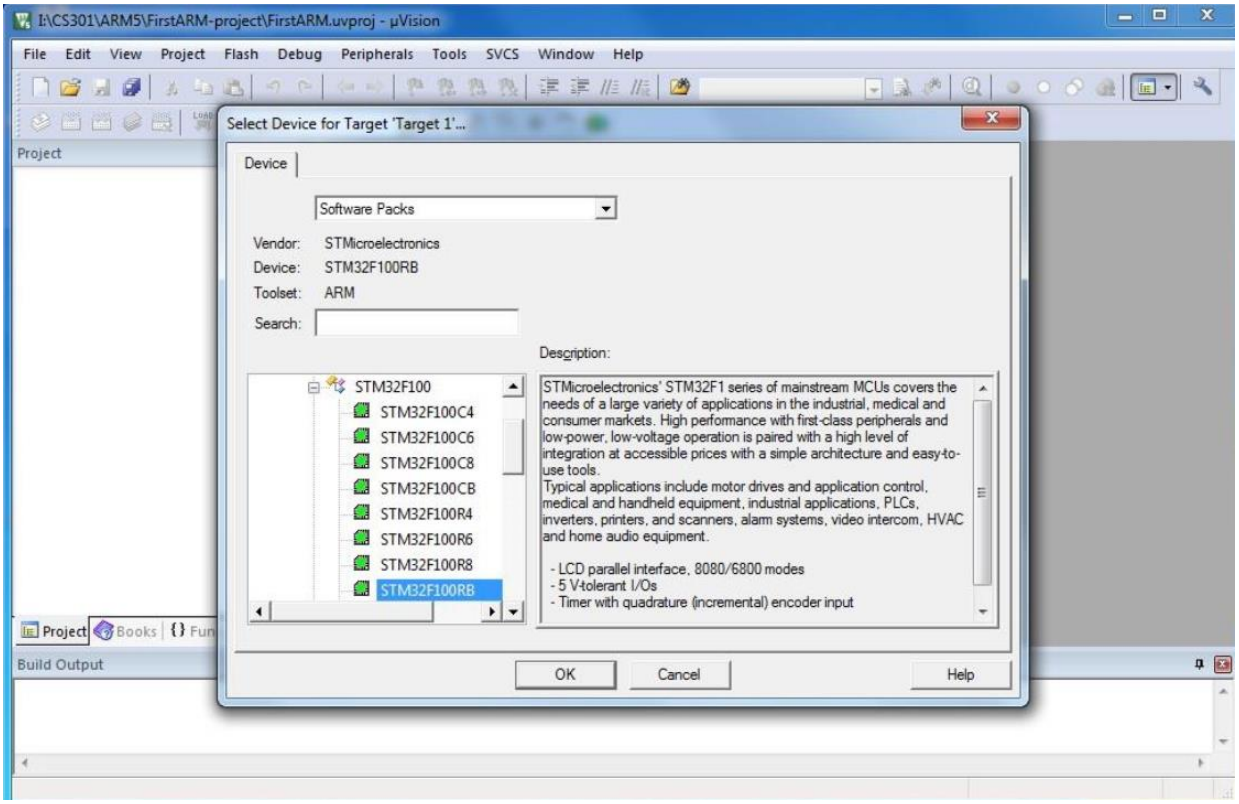
Then, select the folder that you prepared for, give project a name and save it.



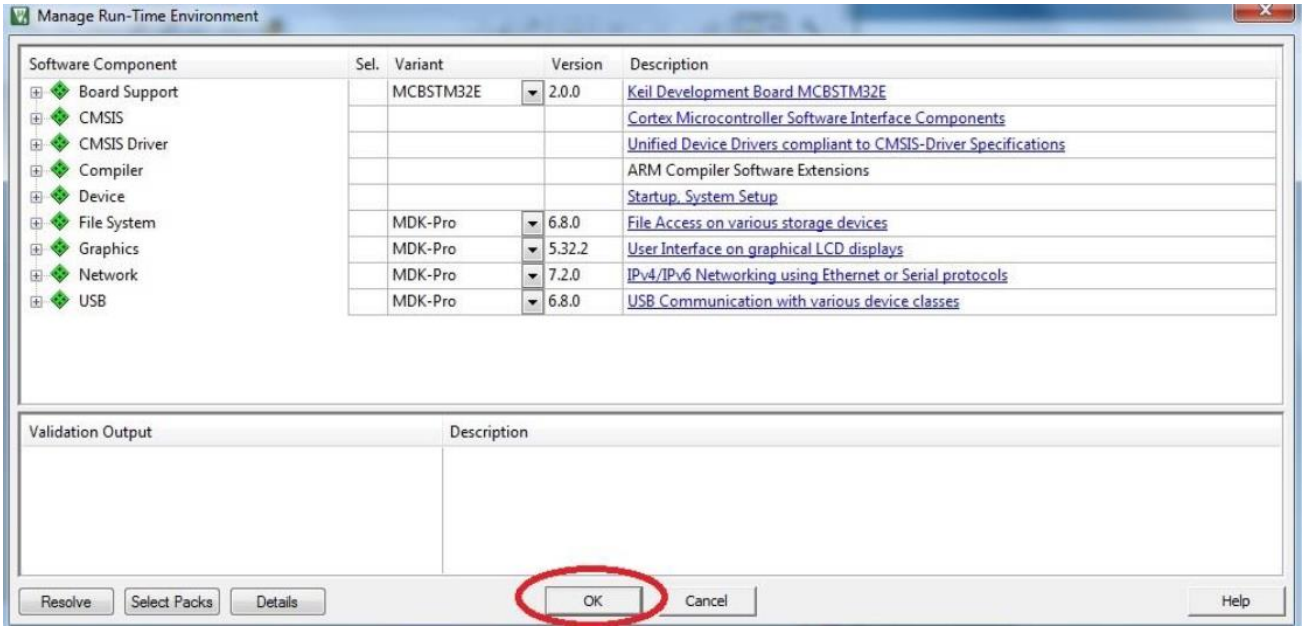
From the "Select Device for Target" window, select "STMicroelectronics" and then "STM32F1 Series".



click on "+" beside "STM32F100" and then select "STM32F100RB" and click on "OK".

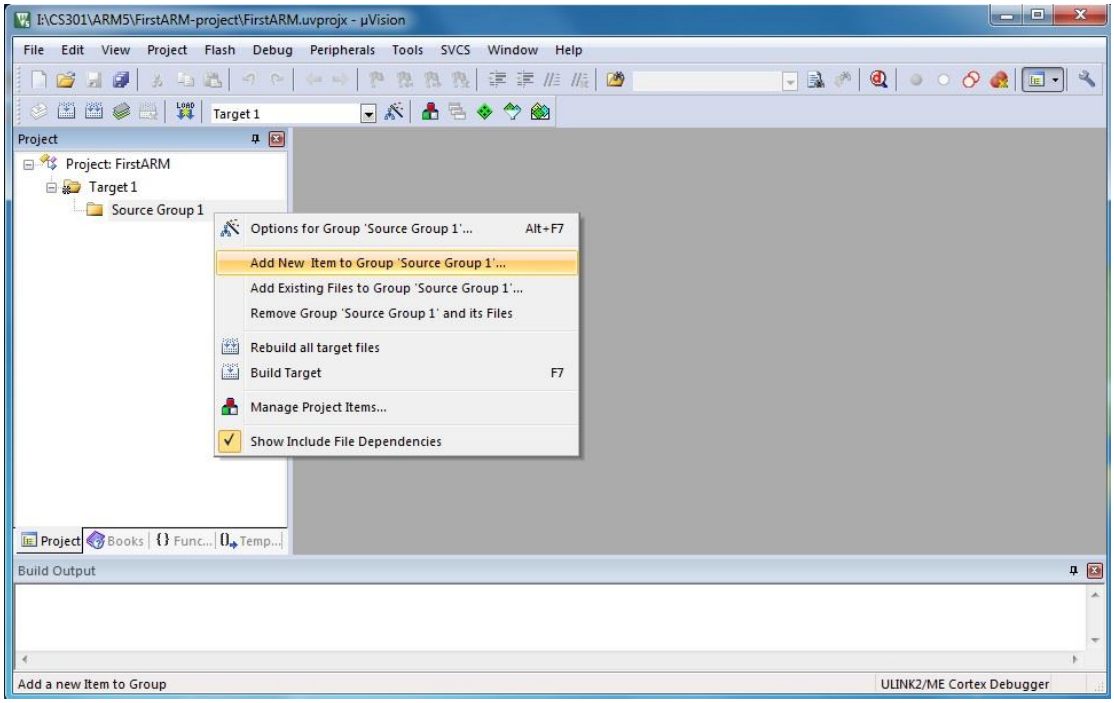


Make sure you click on "OK" for the following pop up window.

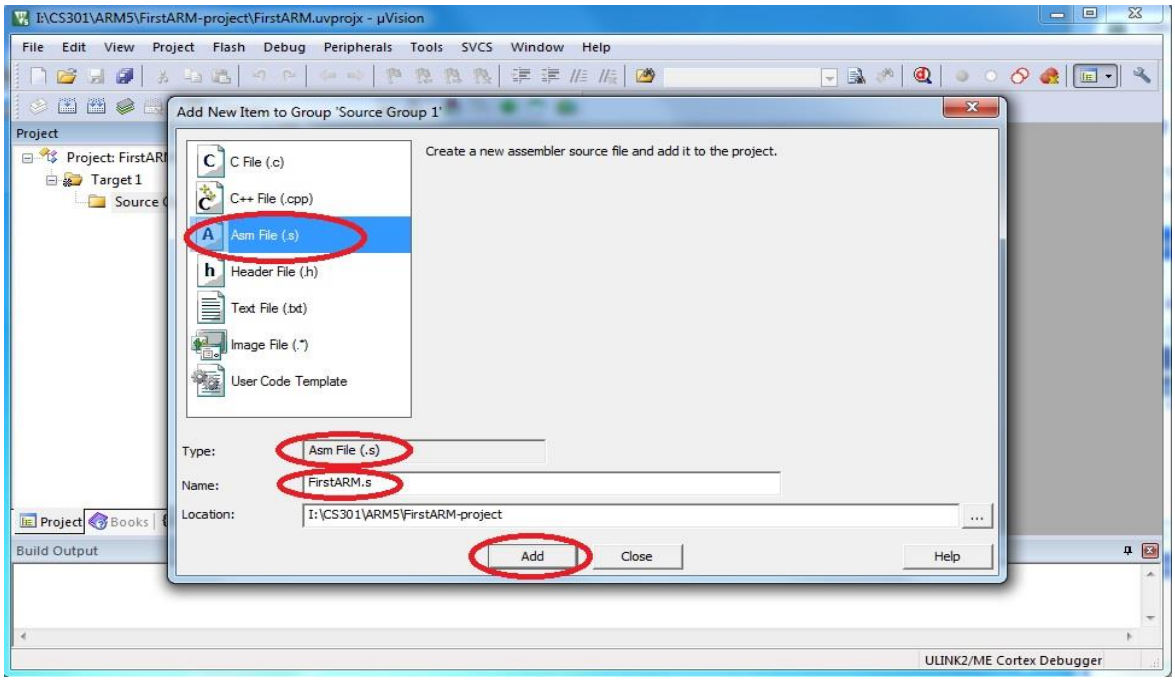


Create Source File and Add Source File to the Project

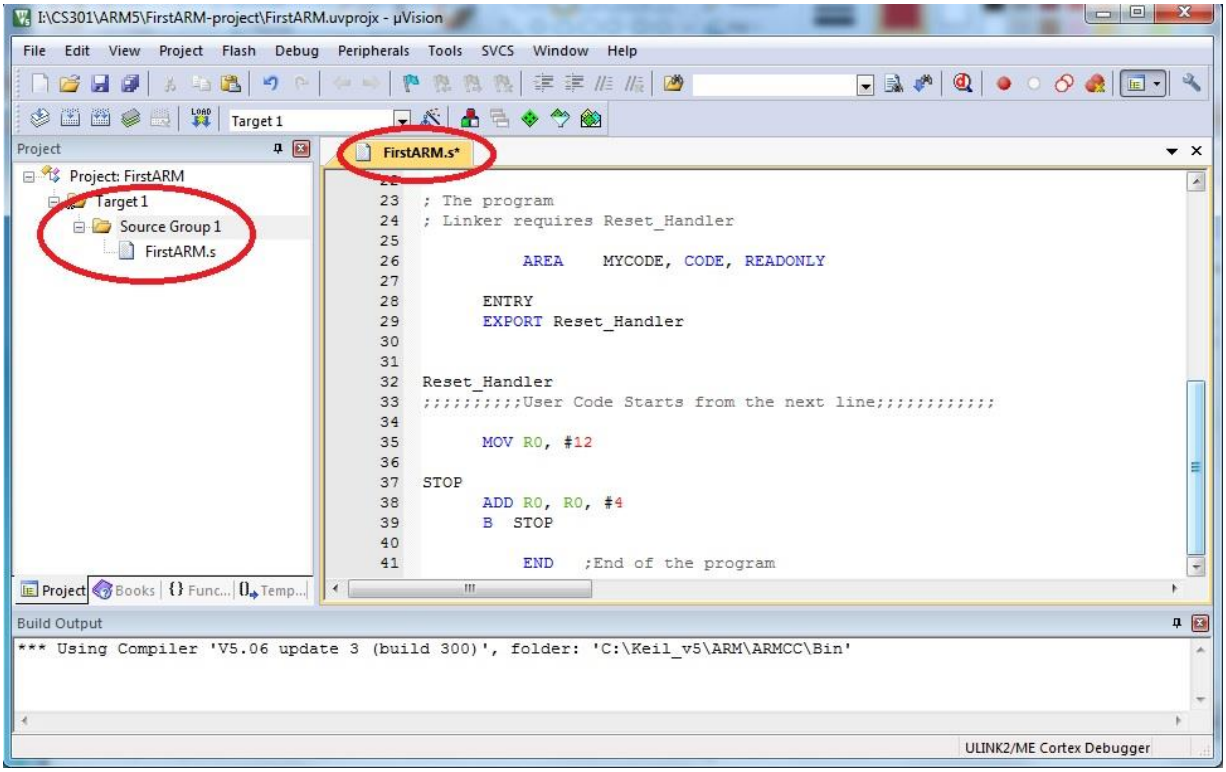
Right click on "Source Group 1" and then select "Add New Item to Group 'Source Group 1'...".



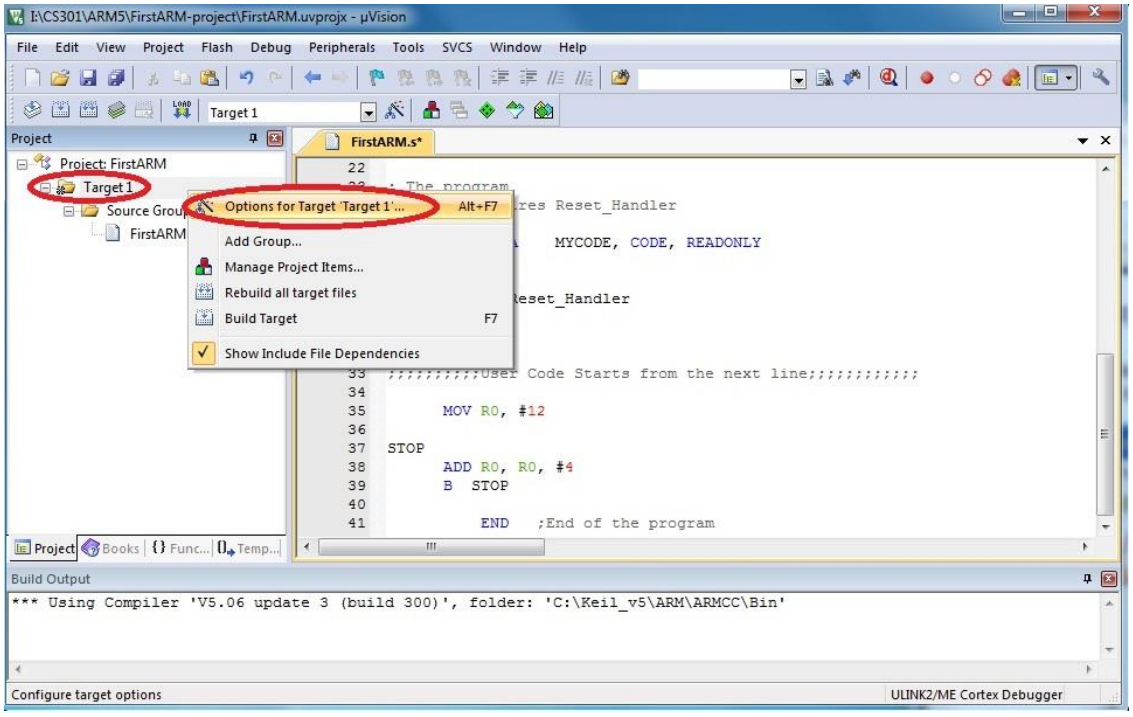
You will see the following window and make the suggested selections to proceed.



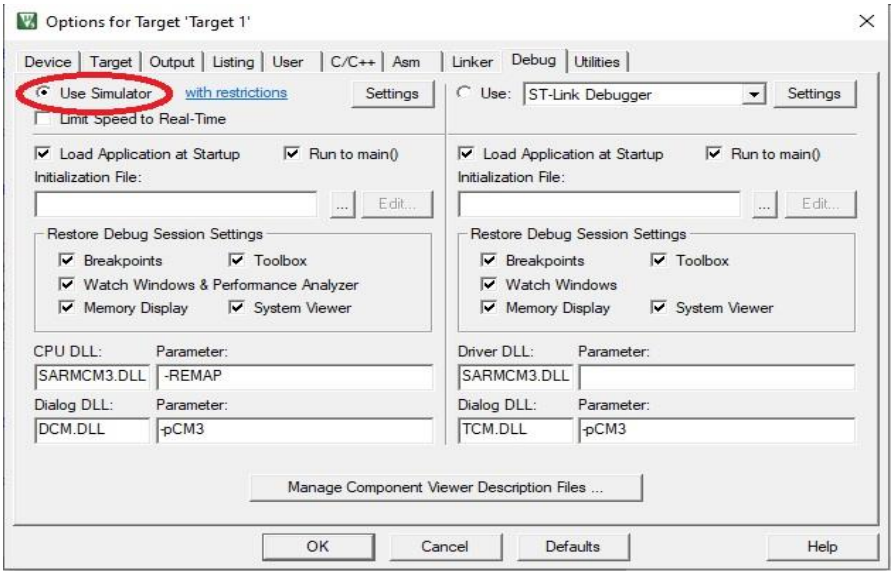
You will see the "FirstARM.s*" text edit window. That is the place you will write your ARM Assembly language program. For a test, you can copy and paste the example program into this window. You can click on the "save" buttom to save your project.



You can right click on "Target 1" and then select "options for Target 'Target 1'..." the same as the following screen.

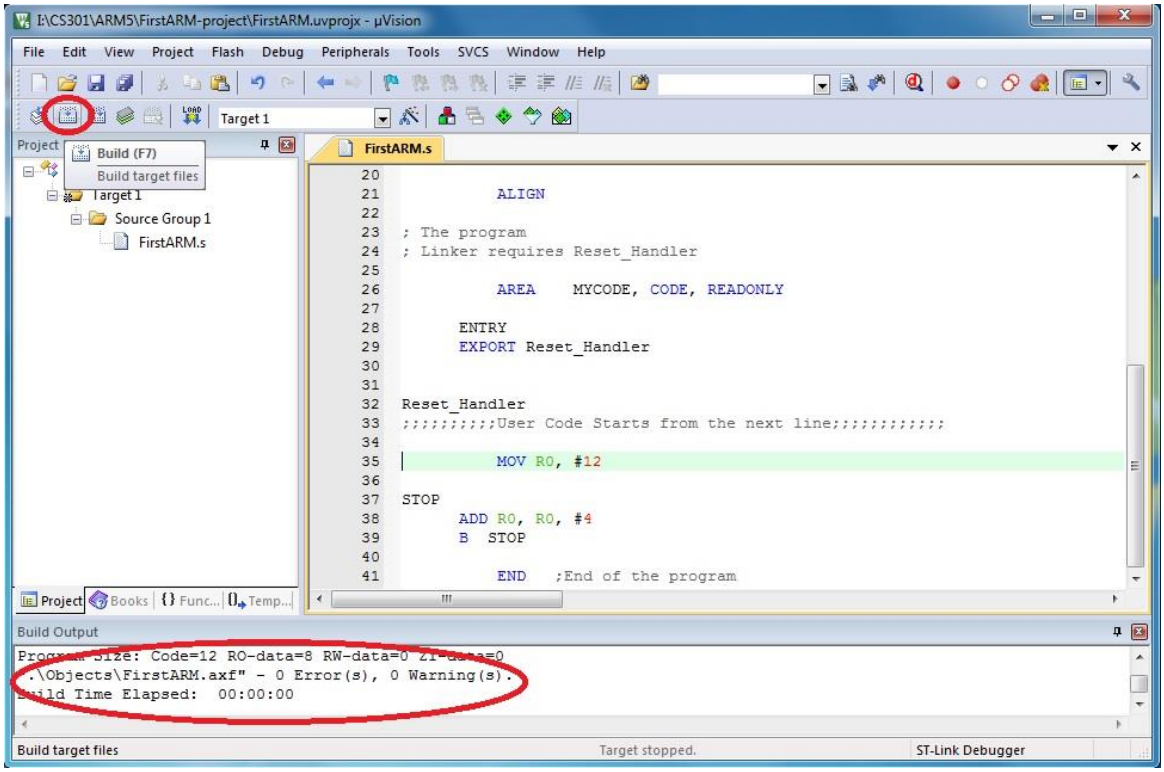


Please click on "Debug" and then select "Use Simulator".



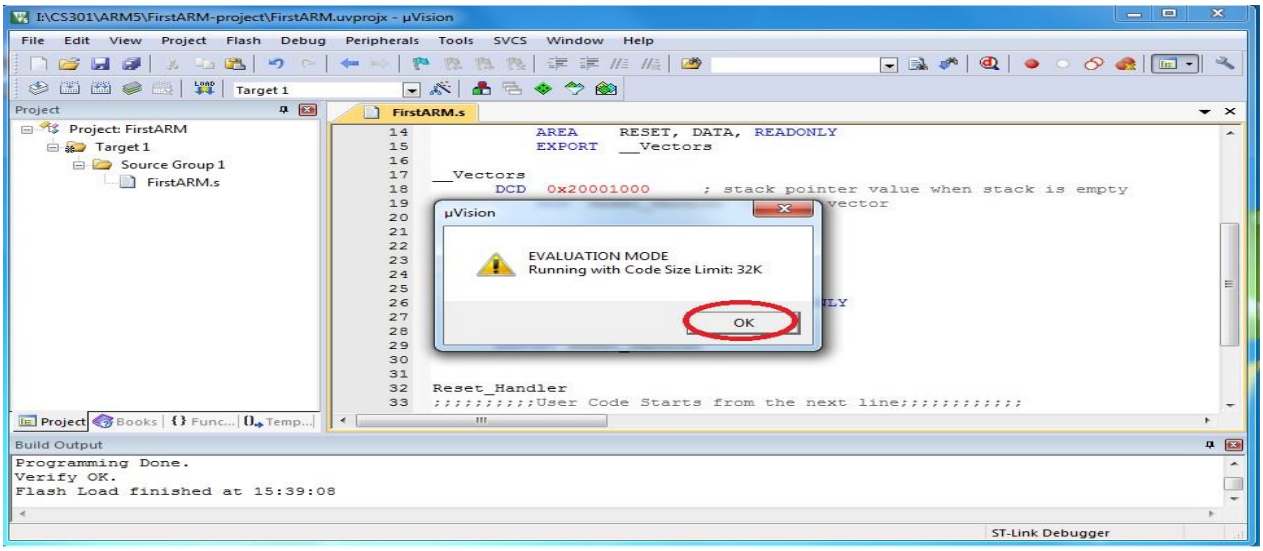
Build your project

Click on the "Build" button or from the "Project" menu, you will see the following screen.



Run the program in your project

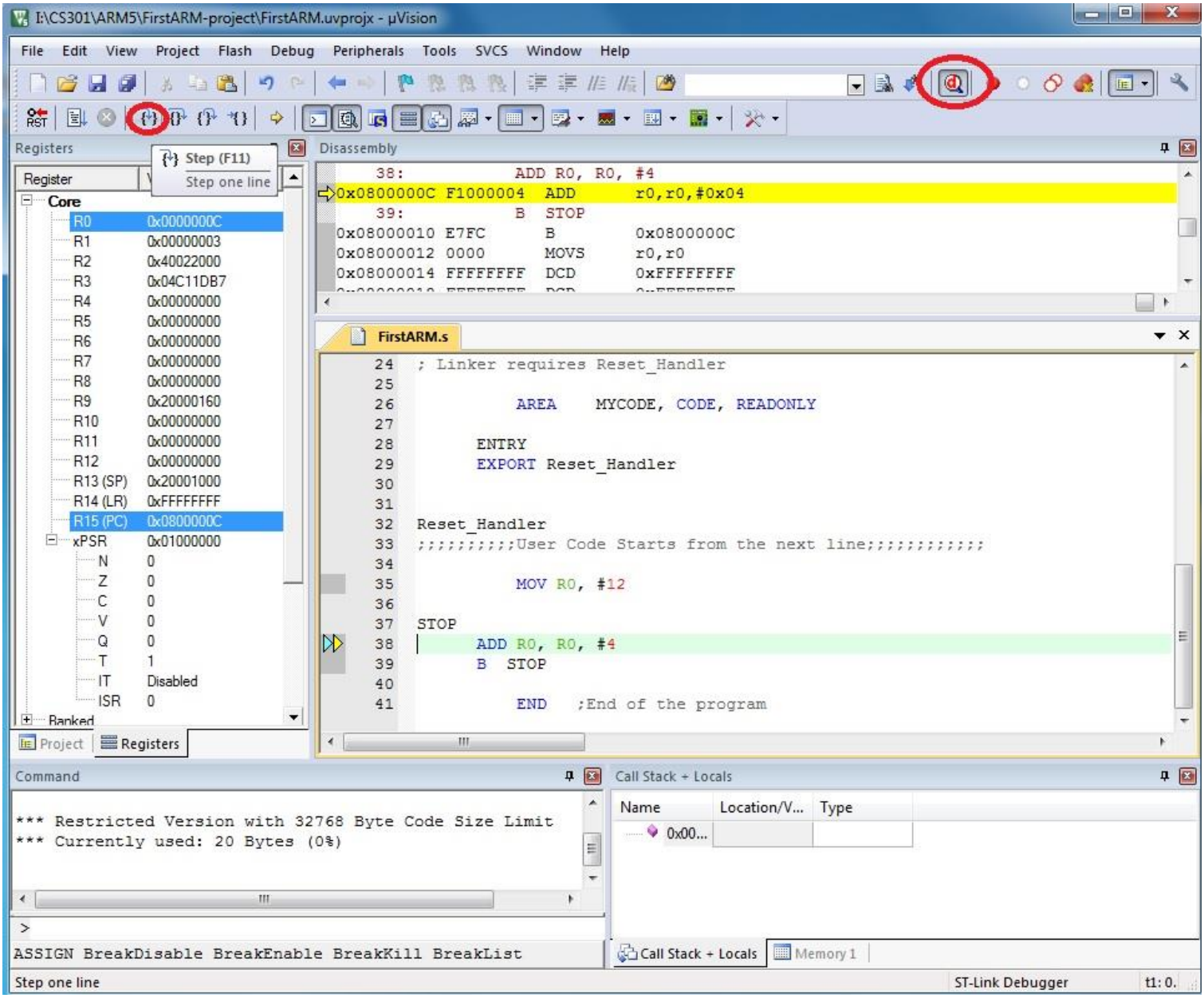
When the assembler is happy with the program, we can run the program by selecting "Start/Stop Debug Session" from the "Debug" menu or clicking on the debug button.



Click on "OK" for the pop up window showing "EVALUATION MODE, Running with Code Size Limit: 32K".

Open your uVision5 to full screen to have a better and complete view. The left hand side window shows you the registers and the right side window shows the program code. There are some other windows open. You may adjust the size of them to see better.

Run the program step by step, you can observe the change of the values in the registers.



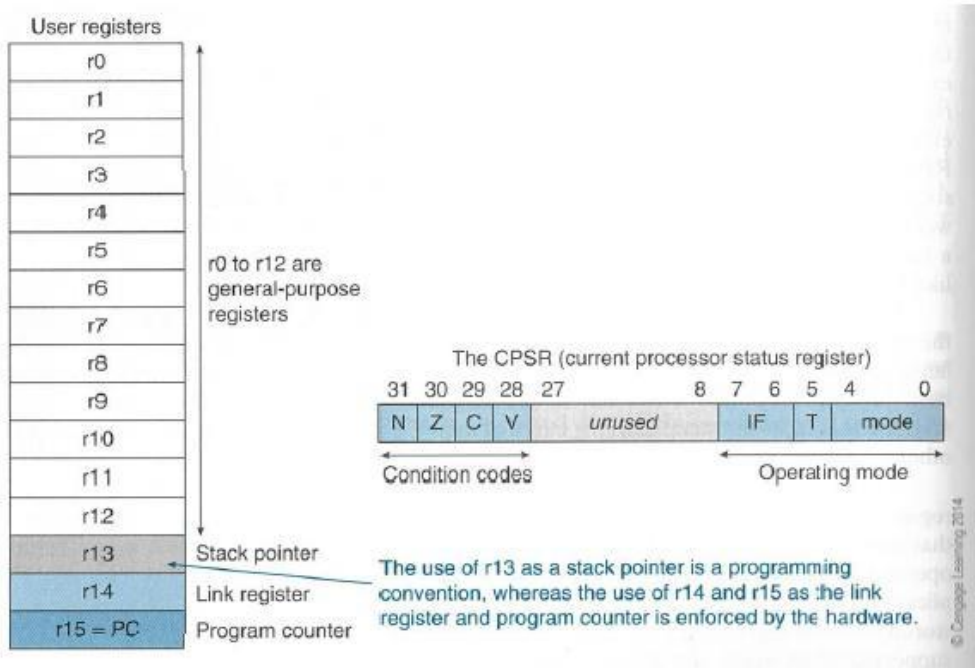
Click on the "Start/Stop Debug Session" from the "Debug" menu or click on the debug button to stop executing the program.

Please analyze the program and see how it works.

It works with both the simulated target and the real circuit board STM32VLDISCOVERY Board.

ARM Registers and the Conventions of Use

As mentioned in the previous lab, ARM has 16 programmer-visible registers and a *Current Program Status Register*, CPSR.
Here is a picture to show the **ARM register set.**




```
R0 to R12 are the general-purpose registers.
R13 is reserved for the programmer to use it as the stack pointer.
R14 is the link register which stores a subroutine return address.
R15 contains the program counter and is accessible by the programmer.

CPSR: current program status register (32 bit)
Stores the status of the previous ALU operation.
4 flags
N = 1, if result was negative
Z = 1, if result was zero
C = 1, if result had a carry-out
V = 1, if result was an overflow
These can be used to make decisions later on.
```

ARM Instructions

ARM instructions have the following general format:

```
Label Op-code operand1, operand2, operand3   ; comment
```

Arithmetic Instructions

Arithmetic instructions are very basic and frequently used in your ARM programming. Here is a table that demonstrates the usage of the ARM processor's arithmetic instructions with examples.

```
  Instruction        Mnemonic               Meaning
=============================================================================
  Addition           ADD   R0, R1, R2       ; R0 = R1 + R2
-----------------------------------------------------------------------------
  Addition           ADDS  R0, R1, R2       ; R0 = R1 + R2,
                                            ; and FLAGs are updated
-----------------------------------------------------------------------------
  Subtraction        SUB   R1, R2, R3       ; R1 = R2 - R3
-----------------------------------------------------------------------------
  Subtraction        SUBS  R1, R2, R3       ; R1 = R2 - R3,
                                            ; and FLAGs are updated
-----------------------------------------------------------------------------
                     SUBS  R7, R6, #20       ; R7 = R6 - 20
                                            ; Sets the flags on the result
-----------------------------------------------------------------------------
  Reverse Subtraction       RSB R4, R4, #120        ; R4 = 120 - R4
-----------------------------------------------------------------------------
  Multiply           MUL   R0, R1, R2        ; R0 = R1 * R2
-----------------------------------------------------------------------------
                     UMULL R0, R4, R5, R6   ; Unsigned (R4,R0) = R5 * R6
-----------------------------------------------------------------------------
                     SMLAL R4, R5, R3, R8   ; Signed (R5,R4) = (R5,R4) + R3 * R8
-----------------------------------------------------------------------------
  Division           SDIV  R0, R2, R4        ; Signed divide, R0 = R2/R4
-----------------------------------------------------------------------------
                     UDIV  R8, R8, R1       ; Unsigned divide, R8 = R8/R1.
-----------------------------------------------------------------------------
```

Examples of Move Instructions

```
     Mnemonic           Meaning
-----------------------------------------------------------------------------
     MOV   R1, #0xFA05 ; Write value of 0xFA05 to R1, flags are not updated
-----------------------------------------------------------------------------
     MOVS R11, #0x000B ; Write value of 0x000B to R11, flags get updated
-----------------------------------------------------------------------------
     MOVS R10, R12     ; Write value in R12 to R10, flags get updated
-----------------------------------------------------------------------------
     MOV R3, #23       ; Write value of 23 to R3
-----------------------------------------------------------------------------
     MOV R8, SP        ; Write value of stack pointer to R8
-----------------------------------------------------------------------------
     MVNS R2, #0xF     ; Write value of 0xFFFFFFF0 (bitwise inverse of 0xF)
                         ; to the R2 and update flags.
      MOVT   R2, #0x7654; Move the 16-bit Hex number 7654 to the high portion of R2

      MOV32   R2, #0x76543210        ; Move the 32-bit Hex number 76543210 to the R2
```

Logical Operation Instructions

```
-------------------------------------------------------------------------
     AND R9, R2, R1                ; R9 = R2 AND R1
-------------------------------------------------------------------------
     AND R9, R2, #0xFF00           ; R9 = R2 AND #0xFF00
-------------------------------------------------------------------------
     ORR R9, R2, R1                ; R9 = R2 OR R1
-------------------------------------------------------------------------
     ORR R9, R2, #0xFF00
-------------------------------------------------------------------------
     ORREQ R2, R0, R5
-------------------------------------------------------------------------
     ANDS R9, R8, #0x19
-------------------------------------------------------------------------
     EOR  R7, R11, R10             ; R7 = R11 XOR R10
-------------------------------------------------------------------------
     EORS R7, R11, #0x18181818
-------------------------------------------------------------------------
     BIC R0, R1, #0xab             ; R0 = R1 AND (NOT(#0xab))
-------------------------------------------------------------------------
     ORN R7, R11, R14, ROR #4      ; R7 = R11 OR (NOT(R14 ROR #4))
-------------------------------------------------------------------------
     ORNS R7, R11, R14, ROR #2     ; update the flags
-------------------------------------------------------------------------
```

Examples of Shift Instructions

```
-------------------------------------------------------------------------
     LSL R4, R5, #2  ; Logical shift left by 2 bits
-------------------------------------------------------------------------
     LSR R4, R5, #6  ; Logical shift right by 6 bits
-------------------------------------------------------------------------
     LSLS R1, R2, #3 ; Logical shift left by 3 bits with flag update
-------------------------------------------------------------------------
     ROR R4, R5, R6  ; Rotate right by the value in the bottom byte of R6
-------------------------------------------------------------------------
     RRX R4, R5      ; Rotate right with extend (one bit only).
-------------------------------------------------------------------------
```

Load and Store Instructions

To access memory, we can only use Load and Store instructions.

```
     LDR dest, expression
     LDR R6, [R4]    ; load R6 with the value in the memory whose address is in R4
      LDR     R2, = 0x76543210       ; Load R2 with the 32-bit Hex number 76543210


     STR STR{cond} srce,[base],offset
```

```
     STR R0,[R1]            ; store R0 in the byte address R1
     STR R0,[R1,#20]        ; store R0 in the byte address R1+20
     STR R0,[R1,R2,LSL#2]   ; store R0 in the address given by R1+R2*4

     Examples:
     LDR     R0, NUM                ; load R0 with the value of NUM in memory
     LDR     R6, = NUM              ; Load the address of NUM to R6
     MOV     R0, #0x001C            ; Load the value to the R0
     STR     R0, [R6]               ; Store the value in R0 to NUM


    B       LABEL                   ; Branch to LABEL
```

Example (1) of Using Arithmetic Instructions:

```
;The semicolon is used to lead an inline documentation
;
;
;     This program will catculate the value of the following function:
;     f(x) = 5x^2 - 6x + 8  when x = 7.
;



;;; Directives
          PRESERVE8
          THUMB

; Vector Table Mapped to Address 0 at Reset
; Linker requires __Vectors to be exported

          AREA    RESET, DATA, READONLY
          EXPORT  __Vectors

__Vectors
       DCD  0x20001000     ; stack pointer value when stack is empty
          DCD  Reset_Handler  ; reset vector

          ALIGN

; The program
; Linker requires Reset_Handler

             AREA    MYCODE, CODE, READONLY

             ENTRY
             EXPORT Reset Handler


Reset_Handler

;;;;;;;;;;User Code Start from the next line;;;;;;;;;;;;

             MOV R0, #7      ; x = 7

             MUL R1, R0, R0  ; R1 = x^2
```

```
            MOV R4, #5
            MUL R1, R1, R4

            MOV R5, #6
            MUL R2, R0, R5  ; R2 = 6x

            SUB R3, R1, R2  ; R3 = 5x^2 - 6x
            ADD R3, R3, #8  ; R3 = 5x^2 - 6x + 8

            ALIGN
STOP
            B  STOP

            END             ; End of the program
```

Example (2):

```
;The semicolon is used to lead an inline documentation
;When you write your program, you could have your info at the top document block
;For Example:  Your Name, Student Number, what the program is for, and what it does
etc.
;
;       See if you can figure out what this program does
;

;;; Directives
          PRESERVE8
          THUMB

; Vector Table Mapped to Address 0 at Reset
; Linker requires __Vectors to be exported

          AREA    RESET, DATA, READONLY
          EXPORT  __Vectors

__Vectors
       DCD  0x20001000     ; stack pointer value when stack is empty
          DCD  Reset_Handler  ; reset vector

          ALIGN

;Your Data section
;AREA DATA

SUMP    DCD SUM
NUM1  DCD 5
NUM2  DCD 7
; The DCD directive allocates one or more words of memory,
; aligned on four-byte boundaries,
; and defines the initial runtime contents of the memory.
;
; For example,   data1   DCD     1,5,20
; Defines 3 words containing decimal values 1, 5, and 20

       AREA    MYRAM, DATA, READWRITE
SUM   DCD 0
```

```
; The program
; Linker requires Reset_Handler

          AREA    MYCODE, CODE, READONLY

        ENTRY
        EXPORT Reset_Handler


Reset_Handler
;;;;;;;;;;User Code Start from the next line;;;;;;;;;;;;

      LDR R1, NUM1
      LDR R2, NUM2


      MOV R0, #0

      ADD R0, R1, R2
      SUBS R0, R0, #1

      LSLS R3, R0, #2        ; Logical shift left by 2 bits with flag update

      LDR R4, SUMP
      STR R3, [R4]

      LDR R6, [R4]

      ALIGN
STOP
        B  STOP


      END
```

Lab work (1):

Write your first ARM assembly language program **MyFirstARM.s**.
The program will execute the following instructions. You will run the program step by step, observe and answer the question after each statement.

```
      MOV     R2,     #0x01           ; R2 = ?
      MOV     R3,     #0x02           ; R3 = ?

      ;Other examples to move immediate values
      MOV     R5,     #0x3210         ; R5 = ?

      MOVT    R5,     #0x7654         ; R5 = ?

      MOV32   R6,     #0x87654321     ; R6 = ?

      LDR     R7,     = 0x87654321     ; R7 = ?


      ADD     R1,R2,R3                ; R1 = ?
```

```
    MOV32   R3,     #0xFFFFFFFF     ; R3 = ?
    ADDS    R1,R2,R3                ; R1 = ?
                                    ; specify Condition Code updates

    SUBS    R1,R2,R3                ; R1 = ?
                                    ; specify Condition Code updates

    MOV     R4,     #0xFFFFFFFF     ; R4 = ?
    ADD     R1,R2,R4                ; R1 = ?
                          ; How did that operation affect the flags in CPSR?

    ADDS    R1,R2,R4        ; R1 = ?
                            ; Please specify Condition Code updates
                            ; and now what happened to the flags in the CPSR?


    MOV     R2,     #0x00000002     ; R2 = ?
    ADDS    R1,R2,R4                ; R1 = ?
                                    ; again, what happened to the flags?

    MOV     R2,     #0x00000001     ; R2 = ?
    MOV     R3,     #0x00000002     ; R3 = ?
    ADDS    R1,R2,R3                ; R1 = ?
                                    ; Add some small numbers again
                                    ; and check the flags again......

    ; Add numbers that will create an overflow
    MOV     R2,     #0x7FFFFFFF     ; R2 = ?
    MOV     R3,     #0x7FFFFFFF     ; R3 = ?

    ADDS    R1,R2,R3                ; R1 = ?
                                    ; Check the flags in the CPSR?
```

Lab work (2):

You can convert temperatures from Celsius to Fahrenheit or from Fahrenheit to Celsius. Here are the two formulas for your reference.

```
    C = 5 * (F - 32) / 9

    F = (9 * C / 5) + 32
```

Write an ARM assembly language program **convertF2CandC2F.s**.
You will do the following:

1. You can put the Fahrenheit temperature, say 70, in the register R0;
2. and have the converted temperature in Celsius in the register R1.
3. You can put a Celsius temperature, say 22 in register R2;
4. and have the converted temperature in Fahrenheit in the register R3.
5. Build the program if there are any bugs, fix them.
6. Run the program step by step and see how values are changing in the registers.

BIRZEIT UNIVERSITY

Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

Microprocessor and Embedded Systems Lab (ENCS5101)

Experiment No. 2 ARM Flow Control Instructions

Objectives:

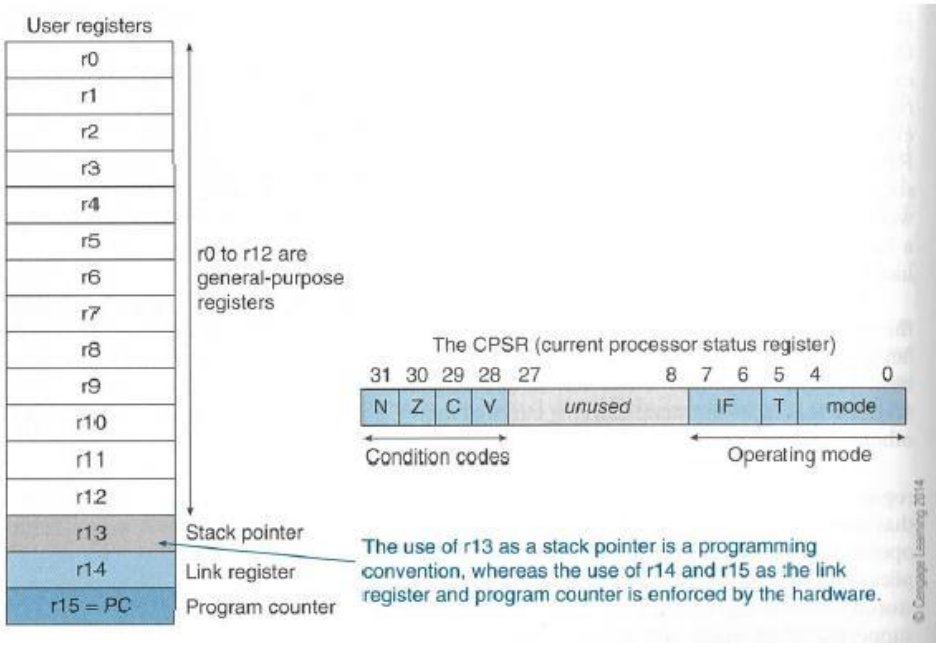
1. To explore ARM branch instructions and implement them in Keil uVision5

2. To investigate how to use strings in Keil uVision5.

ARM's Flow Control Instructions modify the default sequential execution. They control the operation of the processor and sequencing of instructions.

Review of ARM Register Set

As mentioned in the previous lab, ARM has 16 programmer-visible registers and a *Current Program Status Register*, CPSR.
Here is a picture to show the **ARM register set**.



```
R0 to R12 are the general-purpose registers.
R13 is reserved for the programmer to use it as the stack pointer.
R14 is the link register which stores a subroutine return address.
R15 contains the program counter and is accessible by the programmer.

Conditonion code flags in CPSR:
N - Negative or less than flag
Z - Zero flag
C - Carry or bowrrow or extendedflag
V - Overflow flag
The least-significant 8-bit of the CPSR are the control bits of the system.
The other bits are reserved.
```

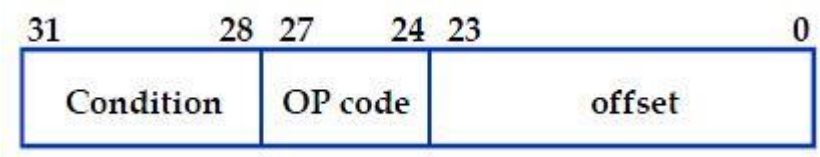
Setting Condition Code Flags

Some instructions, such as Compare, given by **CMP R1, R2** which performs the operation R1-R2 have the sole purpose of setting the condition code flags based on the result of the subtraction operation.

The arithmetic and logic instructions affect the condition code flags only if explicitly specified to do so by a bit in the OP-code field. This is indicated by appending the suffix S to the OP-code. For example, the instruction **ADDS R0, R1, R2** sets the condition code flags. But **ADD R0, R1, R2** does not.

The Encoding Format for Branch Instructions

Conditional branch instructions contain a signed 24-bit offset that is added to the updated contents of the Program Counter to generate the branch target address. Here is the encoding format for the branch instructions:



Offset is a signed 24-bit number. It is shifted left two-bit positions (all branch targets are aligned word addresses), signed extended to 32 bits, and added to the updated PC to generate the branch target address. The updated PC points to the instruction that is two words (8 bytes) forward from the branch instruction.

ARM instructions are conditionally executed depending on a condition specified in the instruction. The instruction is executed only if the current state of the processor condition code flag satisfies the condition specified in bits b31-b28 of the instruction. Thus the instructions whose condition does not meet the processor condition code flag are not executed. One of the conditions is used to indicate that the instruction is always executed. Here is a more detailed description.

31-28	27	26	25	24-21	20	19-16	15-12	11-0
cond	0	0	I	opcode	S	Rn	Rd	Operand 2

- Rn = source register operand 1
- Rd = destination register

} 4 bits = 1 of 16 registers

- 31-28: condition code
  - ALL arm instructions can be conditionally executed
  - eg: ADDEQ
    - add, but only if the previous operation produced a result of zero
    - checks CPSR stored from previous operation

All the ARM instructions are conditionally executed depending on a condition specified in the instruction (bits 31-28).

CONDITION		Flags	Note
0000	EQ	Z==1	Equal
0001	NE	Z==0	Not Equal
0010	HS/CS	C==1	>= (U) / C=1
0011	LO/CC	C==0	< (U) / C=1
0100	MI	N==1	minus(neg)
0101	PL	N==0	plus(pos)
0110	VS	V==1	V set(ovfl)
0111	VC	V==0	V clr
1000	HI	C==1&&Z==0	> (U)
1001	LS	C==0||Z==1	<= (U)
1010	GE	N==V	>=
1011	LT	N!=V	<
1100	GT	Z==0&&N==V	>
1101	LE	Z==1||N!=V	<=
1110	AL	always	
1111	NE	never	
			(U) = unsigned

- The instruction is executed only if the current state of the processor condition code flag satisfies the condition specified in bits b31-b28 of the instruction.

```
For example:
CMP R0, #'A'          ; flags are updated according to (R0 - #'A')
BEQ  VowelCount
```

- The instructions whose condition does not meet the processor condition code flag are not executed.
- One of the conditions is used to indicate that the instruction is always executed.

Branch and Control Instructions

Branch instructions are very useful for selection control and looping control.

Here is a list of the ARM processor's Branch and Control instructions.

```
-------------------------------------------------------------------
     B loopA         ; Branch to label loopA unconditioally
-------------------------------------------------------------------
     BEQ target      ; Conditionally branch to target, when Z = 1
-------------------------------------------------------------------

     BNE AAA         ; branch to AAA when Z = 0
-------------------------------------------------------------------
     BMI BBB         ; branch to BBB when N = 1
-------------------------------------------------------------------
     BPL  CCC        ; branch to CCC when N = 0
-------------------------------------------------------------------
     BLT  labelAA    ; Conditionally branch to label labelAA,
                     ; N set and V clear  or  N clear and V set
                     ; i.e.  N != V
-------------------------------------------------------------------
     BLE labelA      ; Conditionally branch to label labelA,
                     ; when less than or equal, Z set or N set and V clear
                     ; or N clear and V set
                     ; i.e.  Z = 1 or N != V
-------------------------------------------------------------------
     BGT  labelAA    ; Conditionally branch to label labelAA,
                     ; Z clear and either N set and V set
                     ;             or  N clear and V clear
                     ; i.e.   Z = 0 and N = V
-------------------------------------------------------------------
     BGE labelA      ; Conditionally branch to label labelA,
                     ; when Greater than or equal to zero,
                     ; Z set or N set and V clear
                     ; or N clear and V set
                     ; i.e.  Z = 1 or N !=V
-------------------------------------------------------------------
     BL funC         ; Branch with link (Call) to function funC,
                     ; return address stored in LR, the register R14
-------------------------------------------------------------------
     BX LR           ; Return from function call
-------------------------------------------------------------------
     BXNE R0         ; Conditionally branch to address stored in R0
-------------------------------------------------------------------
     BLX R0          ; Branch with link and exchange (Call)
                     ; to a address stored in R0.
-------------------------------------------------------------------
```

Examples of Compare Instructions

```
     Mnemonic                  Meaning
-------------------------------------------------------------------------
     CBZ R5, target          ; Forward branch if R5 is zero
-------------------------------------------------------------------------
     CBNZ R0, target         ; Forward branch if R0 is not zero
-------------------------------------------------------------------------
     CMP R2, R9              ; R2 - R9, update the N, Z, C and V flags
-------------------------------------------------------------------------
     CMN R0, #6400           ; R0 + #6400, update the N, Z, C and V flags
-------------------------------------------------------------------------
     CMPGT SP, R7, LSL #2    ; update the N, Z, C and V flags
-------------------------------------------------------------------------
```

Here are two links for your references.

1. ARM branch instructions from ARM Information Center.

2. Cortex-M Devices Generic User Guide Section 3.9.

Example (1) of Using Branch Instructions:

```
;The semicolon is used to lead an inline documentation
;
;When you write your program, you could have your info at the top document block
;For Example: Your Name, Student Number, what the program is for, and what it does
etc.
;
;     This program will count the length of a string.
;

;;; Directives
          PRESERVE8
          THUMB

; Vector Table Mapped to Address 0 at Reset
; Linker requires __Vectors to be exported

          AREA    RESET, DATA, READONLY
          EXPORT  __Vectors

__Vectors
       DCD  0x20001000     ; stack pointer value when stack is empty
          DCD  Reset_Handler  ; reset vector

          ALIGN

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Byte array/character string
; DCB type declares that memory will be reserved for consecutive bytes
; You can list comma separated byte values, or use "quoted" characters.
; The ,0 at the end null terminates the character string. You could also use "\0".
; The zero value of the null allows you to tell when the string ends.
;
```

```
; The DCB directive allocates one or more bytes of memory, and defines the initial
; runtime contents of the memory.
;
; Example
; Unlike C strings, ARM assembler strings are not null-terminated.
; You can construct a null-terminated C string using DCB as follows:
; C_string   DCB  "C_string",0
;
;*******************************************************************************
string1
      DCB     "Hello world!",0

; The program
; Linker requires Reset_Handler

      AREA    MYCODE, CODE, READONLY

      ENTRY
      EXPORT Reset_Handler


Reset_Handler

;;;;;;;;;;User Code Start from the next line;;;;;;;;;;;;


      LDR     R0, = string1   ; Load the address of string1 into the register R0

      MOV     R1, #0          ; Initialize the counter counting the length of string1

loopCount

      LDRB    R2, [R0]        ; Load the character from the address R0 contains

      CMP     R2, #0
      BEQ     countDone
                              ; If it is zero...remember null terminated...
                              ; You are done with the string. The length is in R1.

      ADD     R0, #1          ; Otherwise, increment index to the next character

      ADD     R1, #1          ; increment the counter for length

      B       loopCount



countDone

STOP
        B  STOP

      END                     ; End of the program
```

Example (2):

```
;The semicolon is used to lead an inline documentation
;When you write your program, you could have your info at the top document block
;For Example:  Your Name, Student Number, what the program is for, and what it does
etc.
;
;       See if you can figure out what this program does
;

;;; Directives
          PRESERVE8
          THUMB

; Vector Table Mapped to Address 0 at Reset
; Linker requires __Vectors to be exported

          AREA    RESET, DATA, READONLY
          EXPORT  __Vectors

__Vectors
        DCD  0x20001000     ; stack pointer value when stack is empty
          DCD  Reset_Handler  ; reset vector

          ALIGN

;Your Data section
;AREA DATA

      ; AREA    MYRAM, DATA, READWRITE

SUMP    DCD SUM
N     DCD 5

       AREA    MYRAM, DATA, READWRITE
SUM   DCD 0



; The program
; Linker requires Reset_Handler

          AREA    MYCODE, CODE, READONLY

        ENTRY
        EXPORT Reset_Handler


Reset_Handler

;;;;;;;;;;User Code Start from the next line;;;;;;;;;;;;


      LDR R1, N              ;Load count into R1

      MOV R0, #0             ;Clear accumulator R0

LOOP
      ADD R0, R0, R1         ;Add number into R0
      SUBS R1, R1, #1        ;Decrement loop counter R1
```

```
      BGT LOOP                ;Branch back if not done

      LDR R3, SUMP            ;Load address of SUM to R3
      STR R0, [R3]            ;Store SUM

      LDR R4, [R3]

STOP
        B  STOP

      END
```

Lab work (1):

Write an ARM assembly language program **CountVowelsOne.s** to count how many vowels and how many non-vowels are in the following string.

```
      "ARM assembly language is important to learn!",0
```

Recommendations for writing the program:

- Put the string in the memory by using DCB.
- Use R0 to hold the address of a character in the string.
- Use R1 to be the counter for vowels.
- Use R2 to be the counter for non-vowels.
- Build the program, debug if needed.
- Run the program step by step and see how values are changing in the registers. OR just run the program and see the final result in the register R1 and R2.

Lab work (2):

Write an ARM assembly program to count the number of values in a 10-dimensional integer vector that is greater than the average of the vector.

```
E.g.  12, -3, 54, 81, 23, -15, 0, 40, -73, 66
```

Recommendations for writing the program:

- Use R1 to hold the address of the vector.
- Use R0 to be the average of the vector.
- Use R2 to be the counter of values greater than the average.
- Build the program, debug if needed.
- Run the program step by step and see how values are changing in the registers. OR just run the program and see the final result in the register R1 and R2.

BIRZEIT UNIVERSITY

Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

Microprocessor and Embedded Systems Lab (ENCS5101)

Experiment No. 3 - Introduction to Microcontrollers & Arduino

3.1 Objectives

1. Learn a programming tool that can simulate and manipulate the controller actions.
2. Learn microcontroller programming steps (edit, build, debug, upload).
3. Learn the structure of a digital microcontroller (Arduino).
4. Learn to program microcontroller ports and access control, status and data registers and ports.
5. Learn to program and test microcontroller internal units (PWM, and Serial communication).
6. Learn to communicate with Arduino microcontroller board, read data from switches and control the lighting of LEDs.

3.2 Equipment's & Tools

- Proteus v8.13 (for simulation purposes)
- Arduino Mega 2560
- Arduino IDE v2.1.1
- Switches
- LEDs

3.3 Prelab

Read the theory section to get the required knowledge and be ready for doing the experiment.

3.4 Theory

3.4.1 Introduction to microcontroller:

A microcontroller is a small computer built onto a single integrated circuit and designed to manage operations in electronic and embedded systems. On a single chip, a typical microcontroller has a CPU, memory, and input/output (I/O) peripherals.

3.4.2 What are the elements of a microcontroller?

As mentioned before, the microcontroller includes certain components, which we'll go through below:

1) **The processor (CPU):** The microcontroller's operation is guided by the processor's (CPU) ability to process and react to different instructions. This involves performing basic logic, I/O, and arithmetic operations. Additionally, it carries out data transmission operations that send commands to other embedded system parts.
2) **Memory:** It is used to store data that the processor receives and makes use of to carry out the instructions that it has been programmed to execute. The two primary memory types in a microcontroller are:
    a) **Program memory: s**tore long-term information regarding the instructions that the CPU executes. Since program memory is non-volatile, it can store data over time without a power source.
    b) **Data memory:** which is necessary for storing temporary data while the instructions are being carried out. Data memory is volatile, which means the information it stores is only kept current if the device is plugged into a power source.
3) **I/O peripherals:** the processor's connection to the outside world is made through its input and output ports. Information is received by the input ports and sent as binary data to the CPU. After receiving the data, the processor transmits the appropriate instructions to output devices that carry out operations external to the microcontroller. Peripherals categories are listed following:
    a) **Data converters:** analog-to-digital converter, digital-to-analog converter, reference-voltage generator
    b) **Clock generation:** internal oscillator, crystal-drive circuitry, phase-locked loop.
    c) **Timing:** general-purpose timer, real-time clock, external-event counter, pulse-with modulation.
    d) **Analog signal processing:** operational amplifier, analog comparator.
    e) **Input/output:** general-purpose digital input and output circuitry, parallel memory interface.
    f) **Serial communication:** UART, SPI, I2C, USB.

4) **System bus:** is the connecting wire that links the microcontroller's parts together.

Fig. 1 shows the main components of the microcontroller.

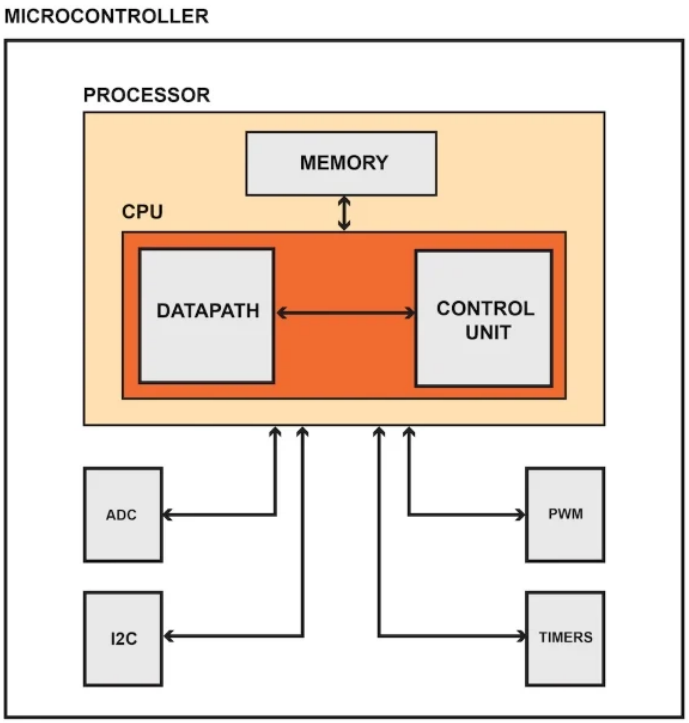


Fig. 1: The components of the microcontroller

Support Circuitry:

Because their main job is not to operate, monitor, or connect with external components, microcontrollers include a range of functional blocks that cannot be categorized as peripherals. Despite this, they are crucial because they support the device's internal functionality, make implementation simpler, and improve the development process.

1) **Debug circuitry:** enables the programmer to carefully monitor the microcontroller as it executes instructions. This is a crucial—and occasionally required—method of finding bugs and improving firmware performance.
2) **Interrupts:** a very useful feature of microcontroller functionality. The CPU responds to interruptions by quickly executing a specified set of instructions in response to external or internal hardware-based events.
3) **Clock-generation module:** the primary function of a microcontroller's internal oscillator is frequently to generate a clock signal for the CPU and peripherals. A clock-generation module might be regarded as a peripheral if it is designed to provide signals that will be used outside the device. Internal oscillators are a practical and efficient technique to simplify the design and reduce board space even though they frequently have low precision in the applications that can accept this low precision.
4) **Power-supply circuitry:** Integrated voltage regulators allow for on-chip generation of required supply voltages, power-management modules can be used to significantly reduce the device's current consumption during inactive states, and supervisor modules can place the processor in a stable reset state when the supply voltage is not high enough to ensure reliable operation.

3.4.3 Microcontrollers Characteristics:

a. **Ports:** is an array of usually 8 pins (GPIOs) which can be configured either to act as digital (on/off) inputs, or outputs, or be used by a special peripheral such as an Analogue to Digital Converter, Timer for things like Output Compare (PWM), and sometimes these can be connected to special internal interrupt generation for example to be woken up by an external source if the microcontroller is sleeping.

Microcontrollers can have any number of ports, it's up to the architecture, pin count of the package, and overall design which dictates how many pins are in a port, how many ports/pins are in a microcontroller, and what they might do.

b. **GPIO (General Purpose Input/Output):** GPIO refers to pins on a microcontroller that can be used for both input and output operations. These pins can be configured as digital inputs to read external signals or as digital outputs to drive signals to external components. GPIO pins are versatile and can be used for various purposes, making them a fundamental component of microcontroller programming.

c. **Port-Bit Configuration Control Register:** A port-bit configuration control register (or simply a control register) is a special register associated with each GPIO pin in a port. It contains settings and configuration options that determine how the pin behaves. Each bit in the control register might correspond to a specific setting for a pin, such as input/output mode, pull-up/pull-down configuration, and more.

d. **Pin Type Settings:** Pin type settings define how a GPIO pin should behave in terms of its electrical characteristics and behaviour. Common pin types/settings include:
   - **Input:** The pin is configured to receive external signals.
   - **Output:** The pin is configured to send signals to external devices.
   - **Bidirectional:** The pin can be used for both input and output operations, allowing signals to flow in both directions.
   - **Pull-Up Resistor:** A pull-up resistor is a resistor connected between a pin and a positive voltage source (usually Vcc or 5V). It ensures that the pin reads as a logical high (1) when it's not actively being driven low. This is useful for preventing floating inputs.
   - **Pull-Down Resistor:** A pull-down resistor is a resistor connected between a pin and ground. It ensures that the pin reads as a logical low (0) when it's not actively being driven high.
   - **Push-Up Resistor (Internal Pull-Up):** Some microcontrollers offer internal pull-up resistors that can be enabled through software. These resistors act as pull-ups without the need for an external component. They are useful when connecting a button or switch to a pin, where the resistor keeps the pin high until the button is pressed.
   - **Three-State Logic:** Three-state logic is a mechanism that allows a pin to be disconnected or in a high-impedance state (Z-state) in addition to being driven high or low. This is useful for bus systems where multiple devices share a communication line.
   - **Analog Input:** The pin is used to measure analog voltages.
   - **Interrupt:** The pin is configured to generate an interrupt when a specific condition is met.

   General Pin Configuration Steps: To configure a GPIO pin's behaviour, you typically follow these steps:
   - Access the control register associated with the pin.
   - Set or clear specific bits in the control register to configure the pin's mode, pull-up/pull-down settings, and other options.
   - Depending on the microcontroller, you might also need to configure other registers, such as data registers for reading or writing to the pin.

e. Pulse Width Modulation (PWM):

A typical method for changing the width of the pulses in a pulse-train is pulse width modulation, or PWM. Many devices can be controlled via PWM, including servos, speed controllers, LEDs and motors with

limited operating power. Pulse width modulation is basically, a square wave with a varying high (1) and low (0) time. A basic PWM signal is shown in the following figure.

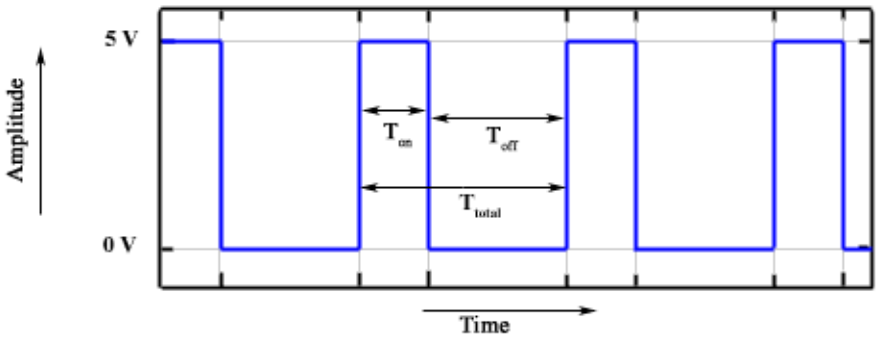


Fig. 2: PWM

There are various terms associated with PWM:

- **On-Time:** Duration of time signal is high.
- **Off-Time:** Duration of time signal is low.
- **Period:** It is represented as the sum of on-time and off-time of PWM signal.
- **Duty Cycle:** It is represented as the percentage of time signal that remains on during the period of the PWM signal.

Period: As shown in Fig. 2, T_{on} denotes the on-time and T_{off} denotes the off-time of signal. Period is the sum of both on and off times and is calculated as shown in *Eq (1)*:

$$T_{total} = T_{on} + T_{off} \qquad \textbf{\textit{Eq (1)}}$$

Duty Cycle: Duty cycle is calculated as the on-time of the period of time. Using the period calculated above, duty cycle is calculated as shown in *Eq (2)*:

$$D = \left(\frac{T_{on}}{T_{total}}\right) \times 100\% \qquad \textbf{\textit{Eq (2)}}$$

f. Serial Communication

The serial communication is a simple scheme that uses the **UART** (Universal Asynchronous Receiver/Transmitter) on the Microcontroller. It uses,

- *5V for logic 1 (high)*
- *0V for logic 0 (low)*

For a 3.3V board, it uses:

- *3V for logic 1 (high)*
- *0V for logic 0 (low)*

Every message sent on the UART is in the form of *8 bits* or *1 byte*, where 1 byte = 8 bits. The messages sent to the computer from Arduino are sent from *PIN1, PIN18, PIN16,* or *PIN14* of the Arduino Mega board, called *TX0*, *TX1, TX2,* and *TX3* (Transmitter) respectively. The messages being sent to the Arduino from the computer are received on *PIN0, PIN19, PIN17, or PIN15* called *RX0, RX1, RX2, and RX3* (Receiver) respectively. These eight pins on the Arduino Mega 2560 board look like the below Figure:

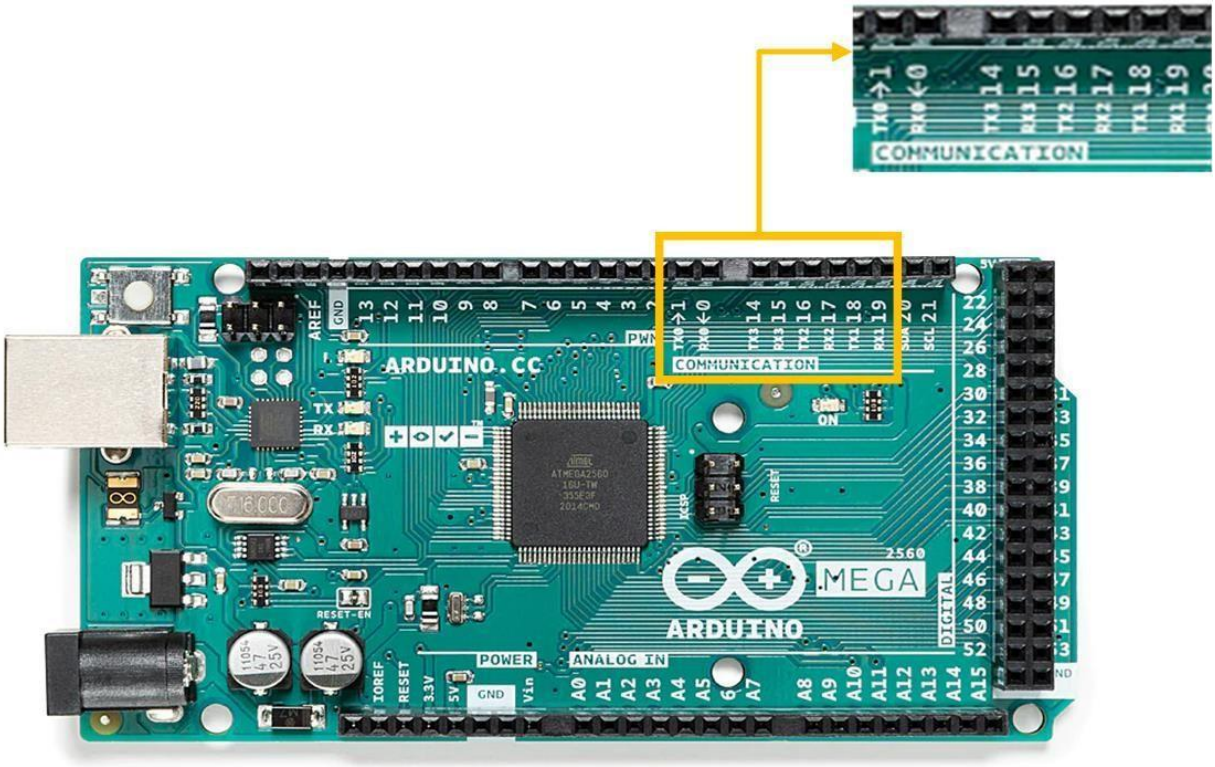


Fig. 3: Tx and Rx for Arduino Mega 2560

Serial: 0 (RX0) and 1 (TX0); Serial 1: 19 (RX1) and 18 (TX1); Serial 2: 17 (RX2) and 16 (TX2); Serial 3: 15 (RX3) and 14 (TX3). Used to receive (RX) and transmit (TX) TTL serial data.

g. **Timer:** A timer is a hardware component in a microcontroller that generates time-based events or intervals. It is essentially a specialized counter that increments at a known rate. Timers are often used to measure time intervals, generate precise delays, and produce periodic events. Timers can trigger actions when a specific time has elapsed or when a specific count value is reached. Fig. 4 shows a periodic timer:

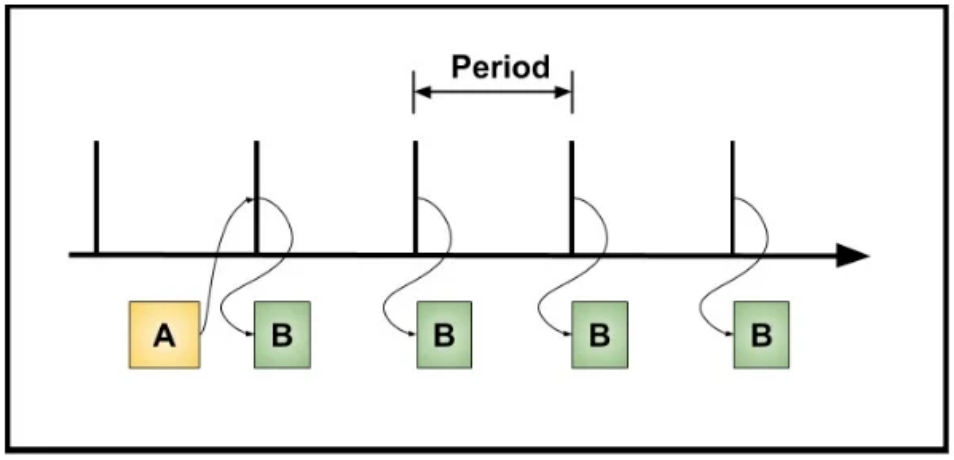


Fig. 4: Periodic Timer

h. **Counter:** A counter is a hardware component that counts events or pulses. It increases or decreases value in response to external events. Counters are used to measure external events such as input pulses, encoder

signals, or any other event that requires counting. Counters can also trigger actions when a specific count value is reached.

In many microcontrollers, timers and counters can be closely related. Some timers have a counter mode where they count external events, and some counters can be used as timers if they are clocked by a known frequency. The terms "timer" and "counter" are often used interchangeably in many contexts, but the key difference lies in their primary purpose: generating time-based events (timer) or counting external events (counter).

i. **Interrupt:** is a mechanism in microcontrollers that allows the processor to temporarily suspend its current execution and handle a higher-priority task by running a special interrupt service routine (ISR). After the ISR has been finished, the running program is continued with the next instruction. Instruction means a single machine instruction, not a line of C or C++ code. Interrupts are often used to respond to external events, such as a button press, a change in signal level, or a timer reaching a specific value. Interrupts help in managing multiple tasks concurrently without the need for continuous polling. Fig. 5 shows a periodic interrupt:

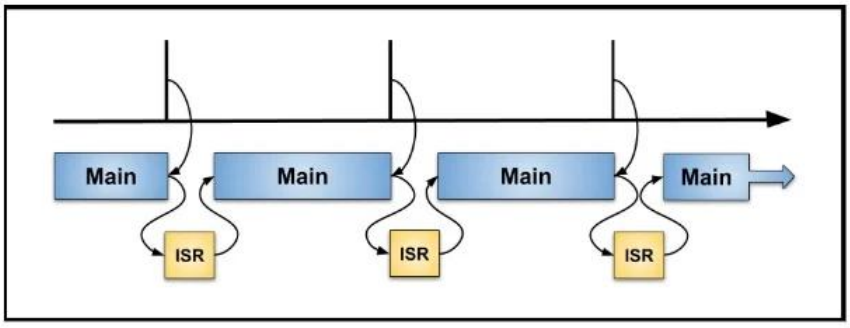


Fig. 5: Periodic Interrupt

Interrupts can be associated with both timers and counters. For example, a timer can generate an interrupt when it reaches a specific value, and a counter can trigger an interrupt when a certain event count is reached. Interrupts allow microcontrollers to handle tasks in a more efficient and timely manner, freeing up the processor from constant polling and enabling more responsive and multitasking applications.

3.4.4 Microcontrollers Programming:

3.4.4.1 Program Components:

In general, a program for microcontrollers consists of various components that work together to define the behavior of the microcontroller. These components can include directives, instructions, variables, functions, and more. Here's an overview of the typical components found in microcontroller programs:

Table 1: Microcontroller Program Components

Component	Definition	Examples
Directives	Instructions to the compiler or assembler that provide guidance on how to process the code	**#include** for including header files, **#define** for creating macros, **#pragma** for compiler-specific settings
Global Variables	They are declared at the top of the program or in a specific section	int sum char myChar
Functions	Blocks of code that perform specific tasks	int sum(int x, int y) { //your code

		}
Built-in Functions	Predefined functions provided by the programming language or the microcontroller's development environment to perform common tasks, interact with hardware, and implement specific functionalities without having to write everything from scratch	PinMode(), digitalWrite() in Arduino IDE
Instructions	The fundamental operations that the microcontroller performs	Arithmetic operations, memory access, conditional branching
Control Flow Statements	Dictate the order in which instructions are executed	Conditionals (if, else, switch) and loops (for, while, do-while)
Data Types	Define the kind of data a variable can hold and the operations that can be performed on it	Integers, floating-point numbers, characters, and arrays
Constants and Literals	Fixed values used in the program, like numerical values, character codes, or symbolic values	#define SUM 0
Macros	Symbolic names representing a sequence of code or a value. They are usually defined using the **#define** directive and are replaced by their actual values during preprocessing.	#define DELAY 200000
Interrupt Service Routines (ISRs)	Special functions that are executed in response to specific hardware events or interrupts	Interrupts handlers

Initialization Code	Initialization code configures the microcontroller's peripherals, memory, and system settings to prepare the microcontroller for proper operation	Setup function in Arduino IDE:
		void setup () {
		pinMode(LED_BUILTIN, OUTPUT);
		Serial.begin(9600);
		}

Comments and Documentation	Explanatory text within the code that provide information about the code's purpose, behavior, and usage	; used before comments in assembly //and /* */ used to comment in C/C++

These components come together to define the program's structure, logic, and behavior in a microcontroller-based system. The specifics of each component depend on the microcontroller's architecture, the programming language used, and the requirements of the application.

3.4.4.2 Program Structure:

While the specific details and syntax may vary depending on the microcontroller architecture and the programming language being used, the general structure of programs for microcontrollers follows a common pattern. Here's a generalized program structure that applies to many microcontrollers:

Table 2: Microcontroller Program Structure

Section	Usage
Header Comments	Provide information about the program, its purpose, creation date, and any relevant details
Include Directives	Include any necessary header files or libraries
Global Variable Declarations	Declare any global variables that will be used throughout the program
Function Declarations	Declare any functions that will be defined later in the program
Setup Function	It Called once when the microcontroller starts, it initializes system settings, peripherals, and data structures
Main Loop	It's an infinite loop that repeats the main tasks of the program
Other Functions	Functions that support the main loop or setup function
Peripheral Control Functions	These functions help manage communication and interaction with external devices

Remember that the specific order and details may vary based on the microcontroller's architecture, programming language, and development environment. However, the general structure outlined above provides a solid foundation for organizing code on various microcontroller platforms. Fig. 6, shows the structure of the source code of the Arduino in the proteus VSM:

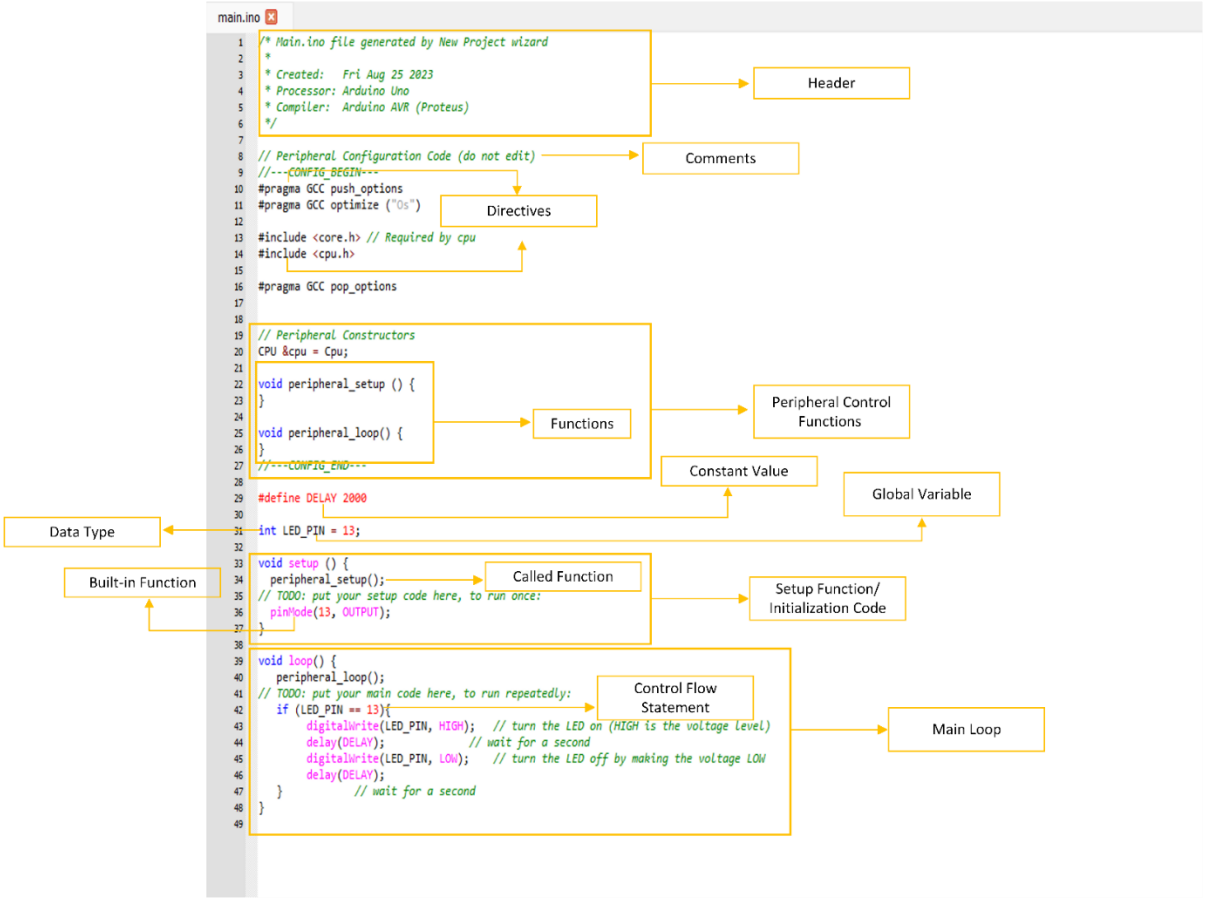


Fig. 6: Arduino Program Structure in Proteus VSM

3.4.5 Microcontrollers Simulation:

3.4.5.1 Proteus Software:

Proteus is a computer-aided design (CAD) software suite used for electronic circuit simulation, schematic capture, and PCB (printed circuit board) design. It is commonly used by engineers, students, and hobbyists for designing and testing electronic circuits before they are physically implemented.

Proteus software offers a virtual environment where users can design and simulate electronic circuits using a wide range of components, such as microcontrollers, sensors, transistors, and displays. The software provides tools for creating schematics, designing PCB layouts, and simulating the behaviour of circuits.

3.4.5.2 Create new project in Proteus:

To create new proteus project, follow the nest steps:

1. Search about proteus program inside the search bar, and click on it to open it:

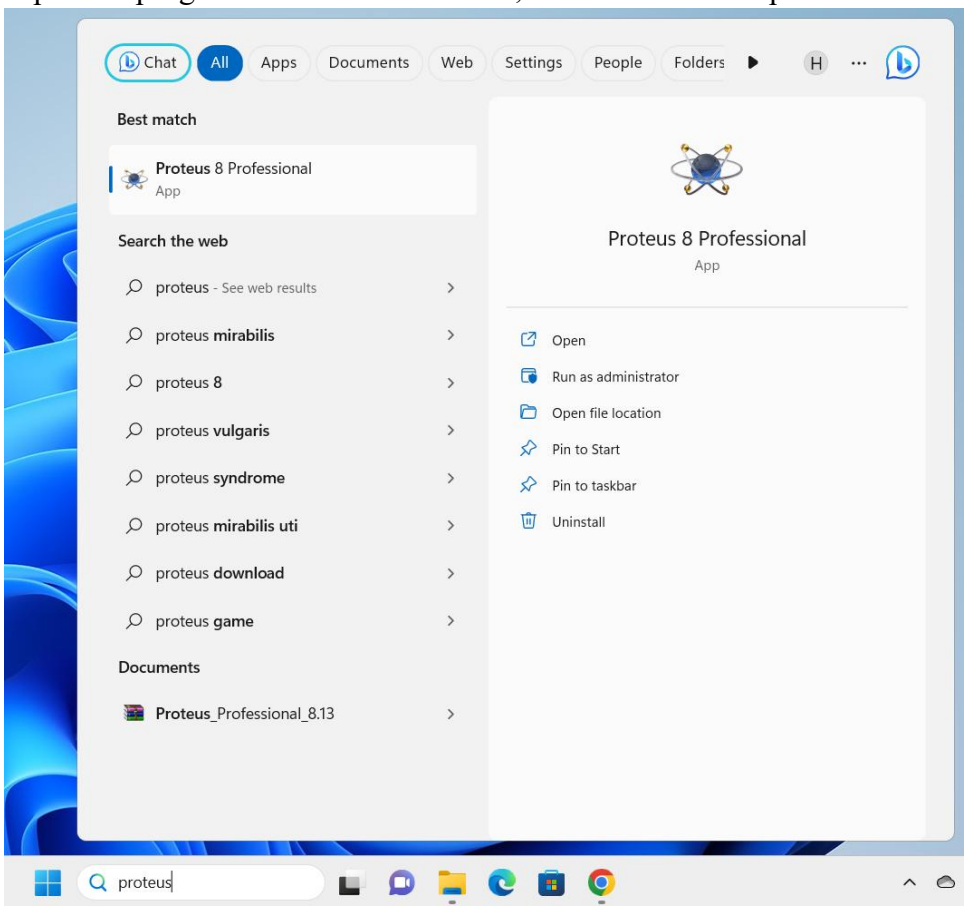


Fig. 7: Create New Project – Step1

The following logo will appear on your desktop, wait until the main window appear:

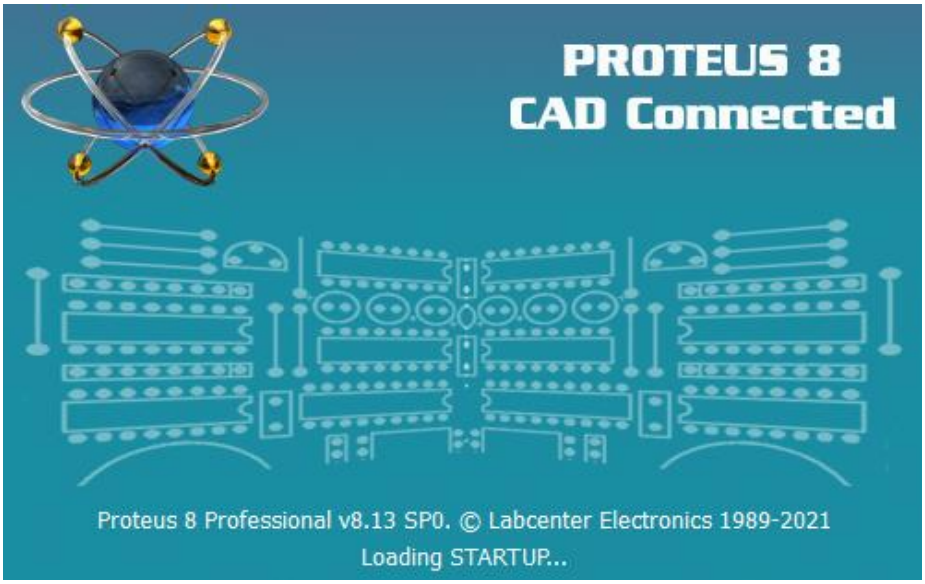


Fig. 8: Proteus Loading window

2. The home page of the Proteus will be shown, as you can see in Fig. 9:

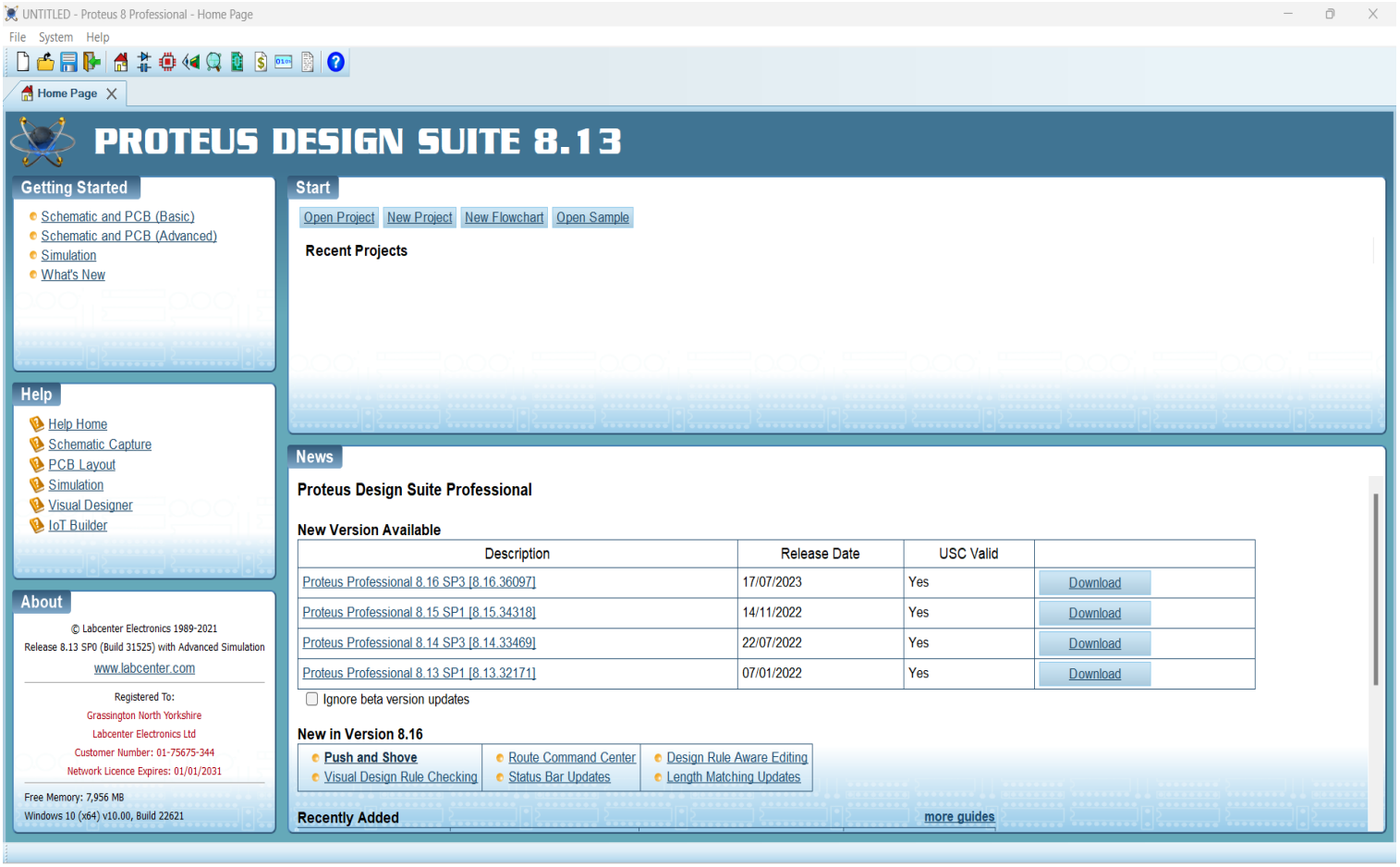


Fig. 9: Proteus Main Window

3. From the ***Start*** subsection, click on ***New Project*** to create new project, as shown in Fig. 10:



Fig. 10: Create New Project – Step2

4. Give your project a name and choose the directory to save it in, then click on ***Next,*** as shown in Fig. 11:

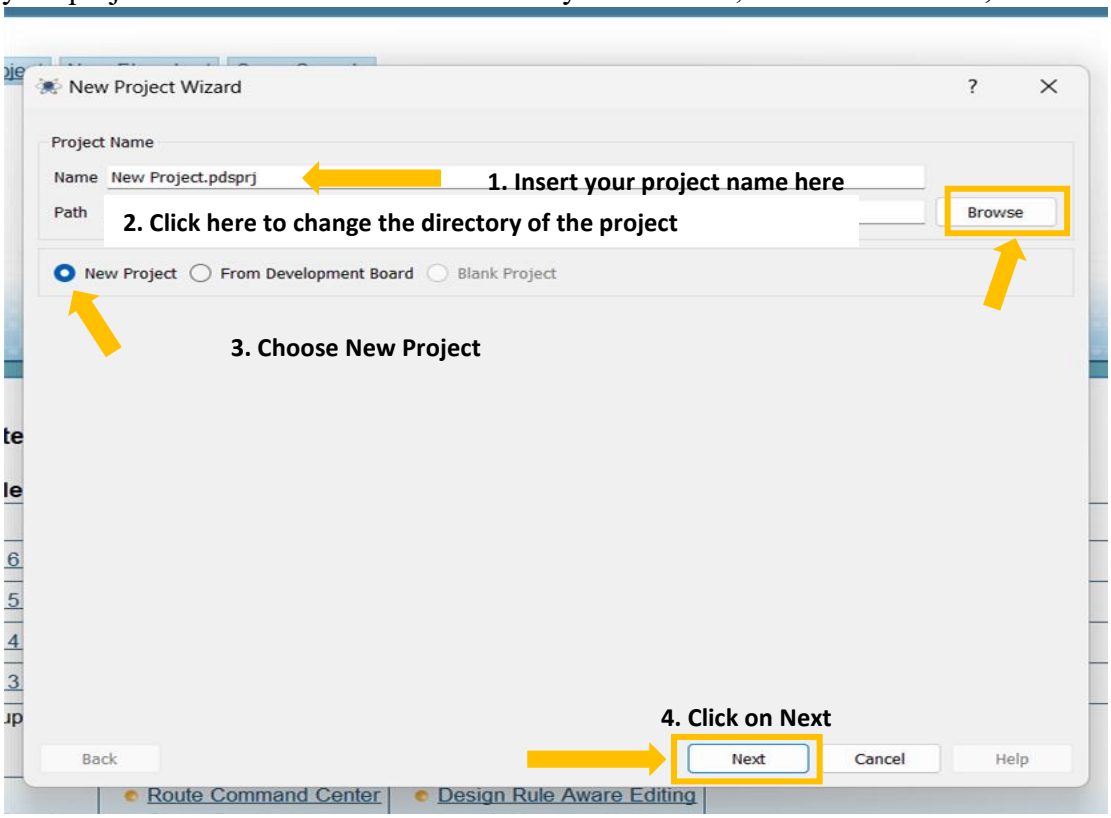


Fig. 11: Create New Project – Step3

5. The following popup will appear, keep it as it is and click on ***Next***:

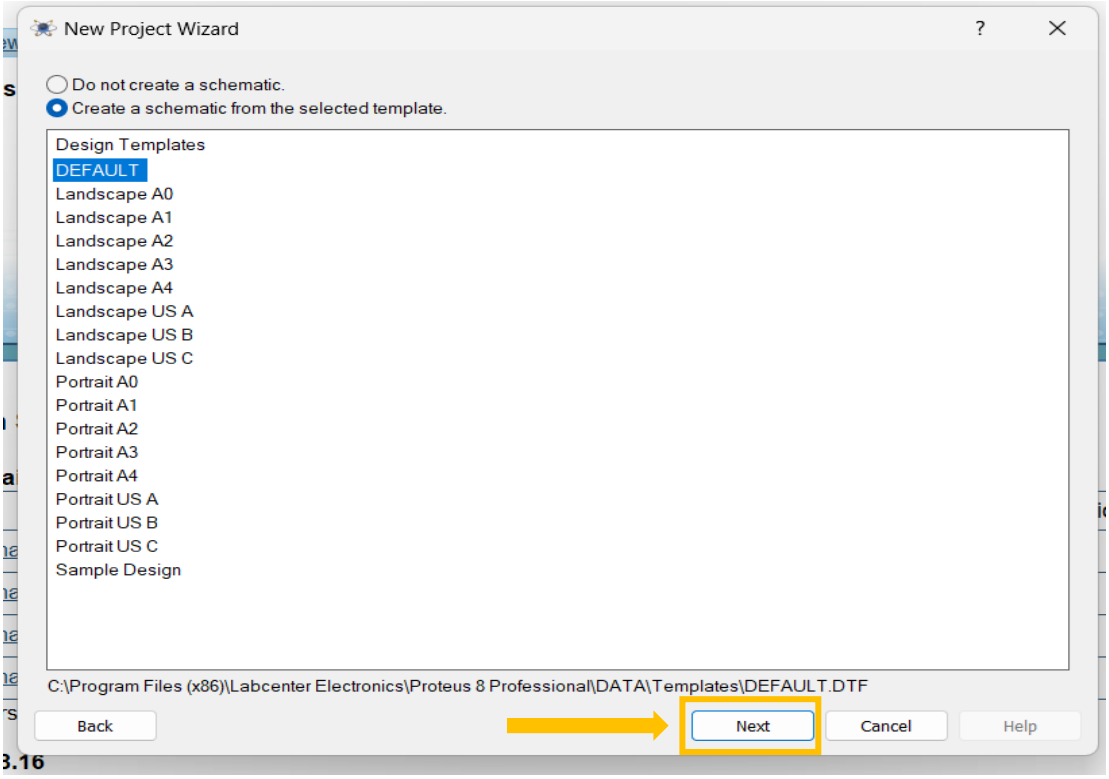


Fig. 12: Create New Project – Step4

6. As shown on the following Figures, click on ***Next*** only:

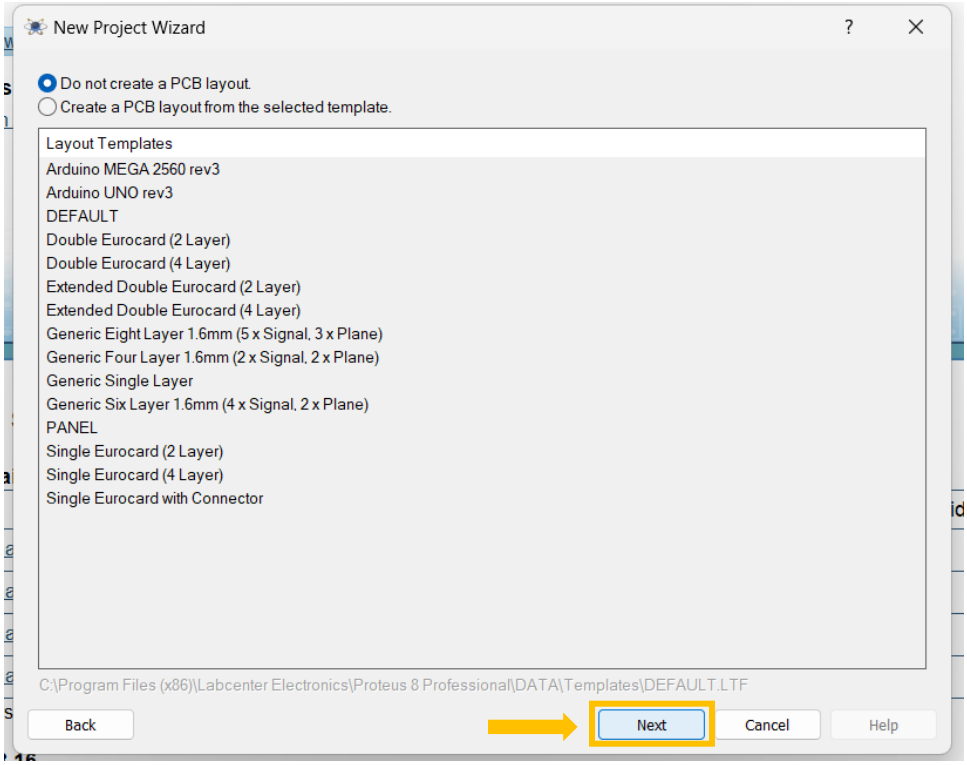


Fig. 13: Create New Project – Step5

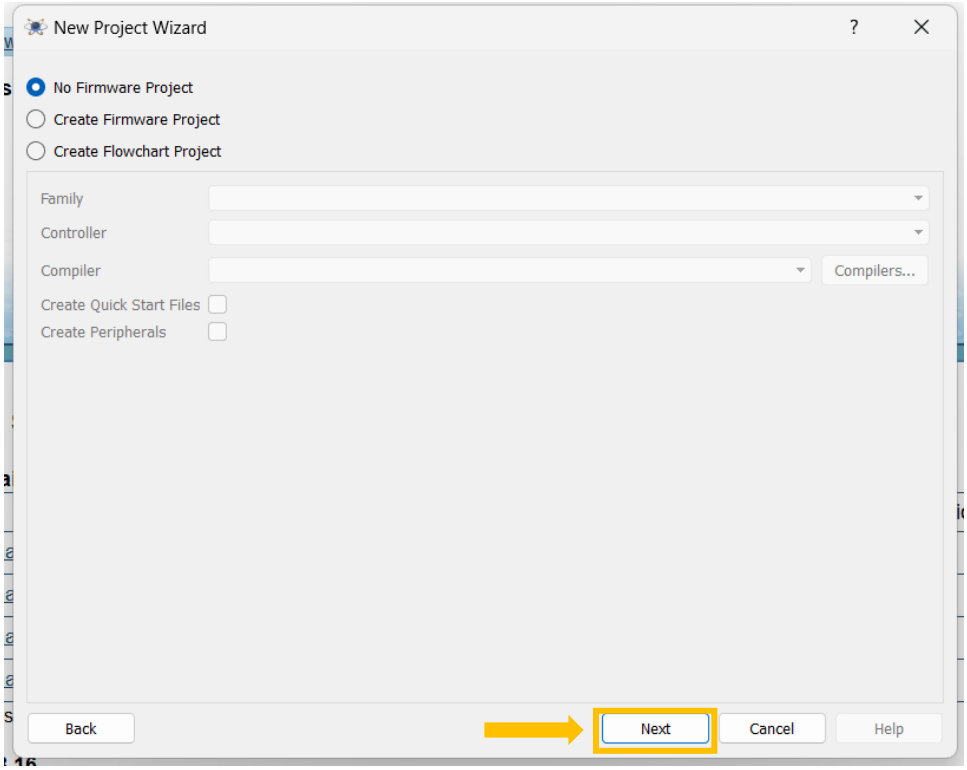


Fig. 14: Create New Project – Step6

7. Click on ***Finish***:

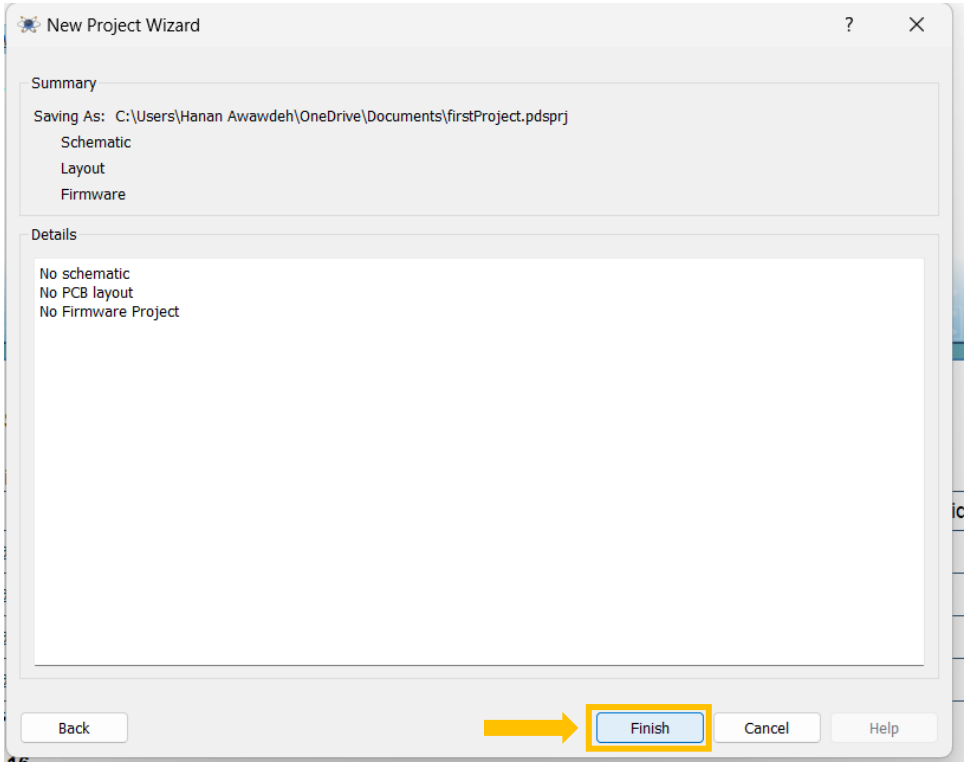


Fig. 15: Create New Project – Step7

3.4.5.3 How to add components on Proteus?

Proteus working sheet will open after you create a new project, as shown in Fig. 16:

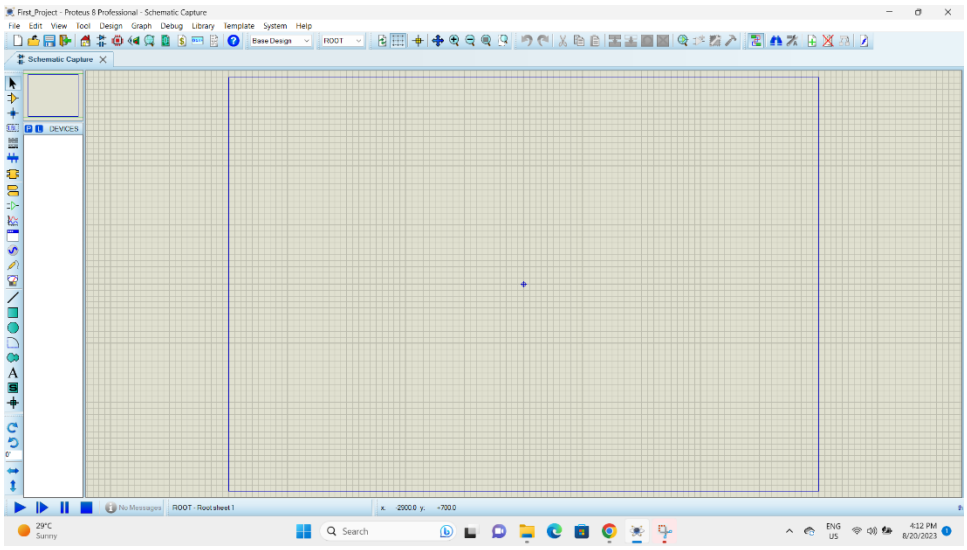


Fig. 16: Project Working Sheet

The working sheet has three windows as shown in Fig. 17:

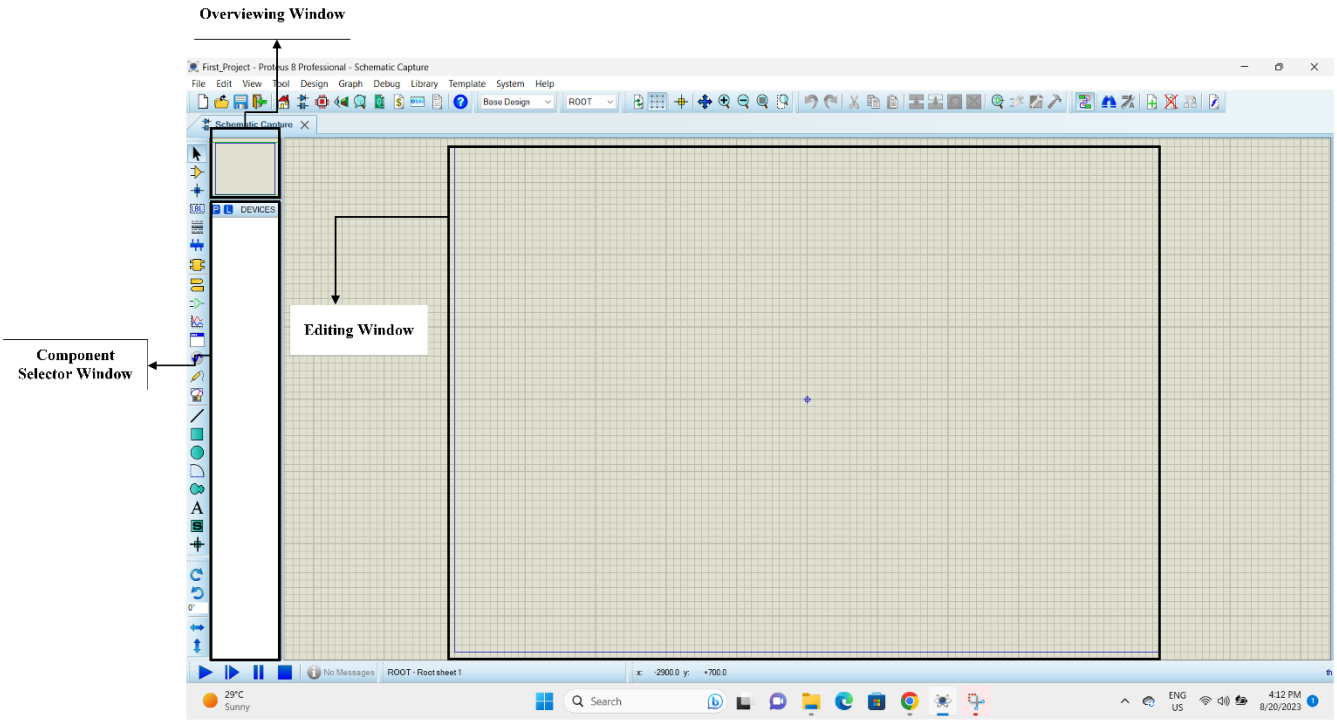


Fig. 17: Working Sheet windows

To add new component to the working sheet, you can do that in two ways as described below:

1. Right click by the mouse on any area on the sheet, then select ***Place*** then ***Component,*** and then ***From Libraries*** as shown in Fig. 18:

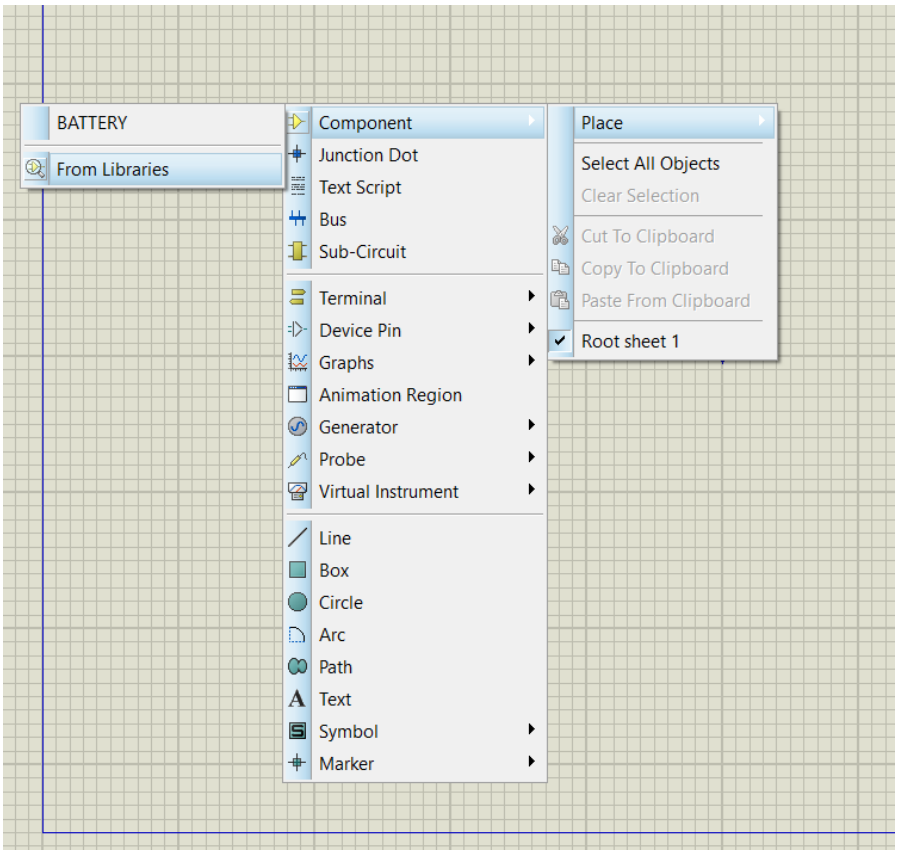


Fig. 18: Add component to the sheet - First Way

Or for speed, go to Component Selector Window and click on ***P*** as shown in Fig. 19:

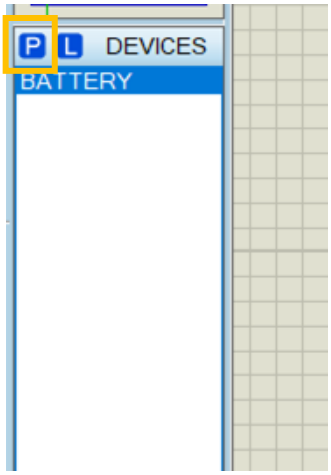


Fig. 19: Add component to the sheet – Second Way

In both ways the following popup window will appear:

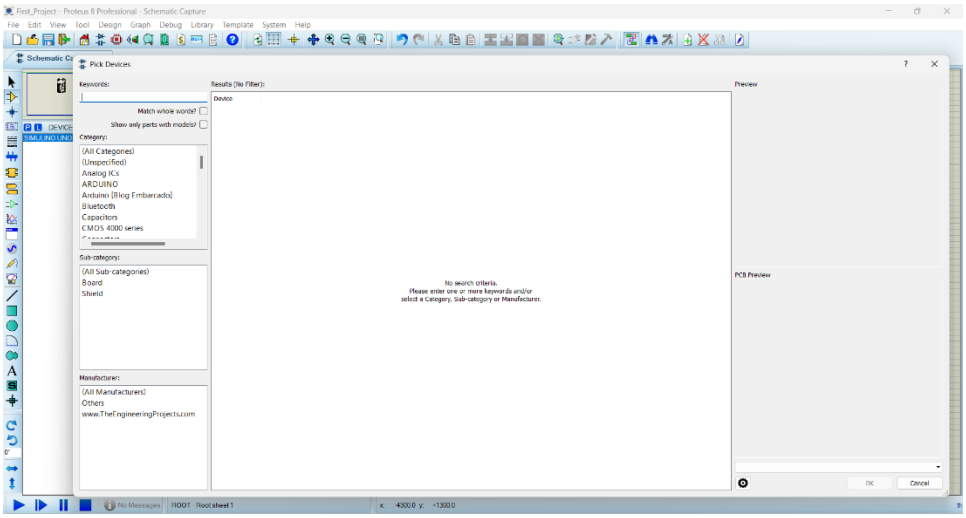


Fig. 20: Choose Component Window

For example, to add Arduino Mega, you should search about it on the search box then select it from the Devices result Box as shown in Fig. 21:

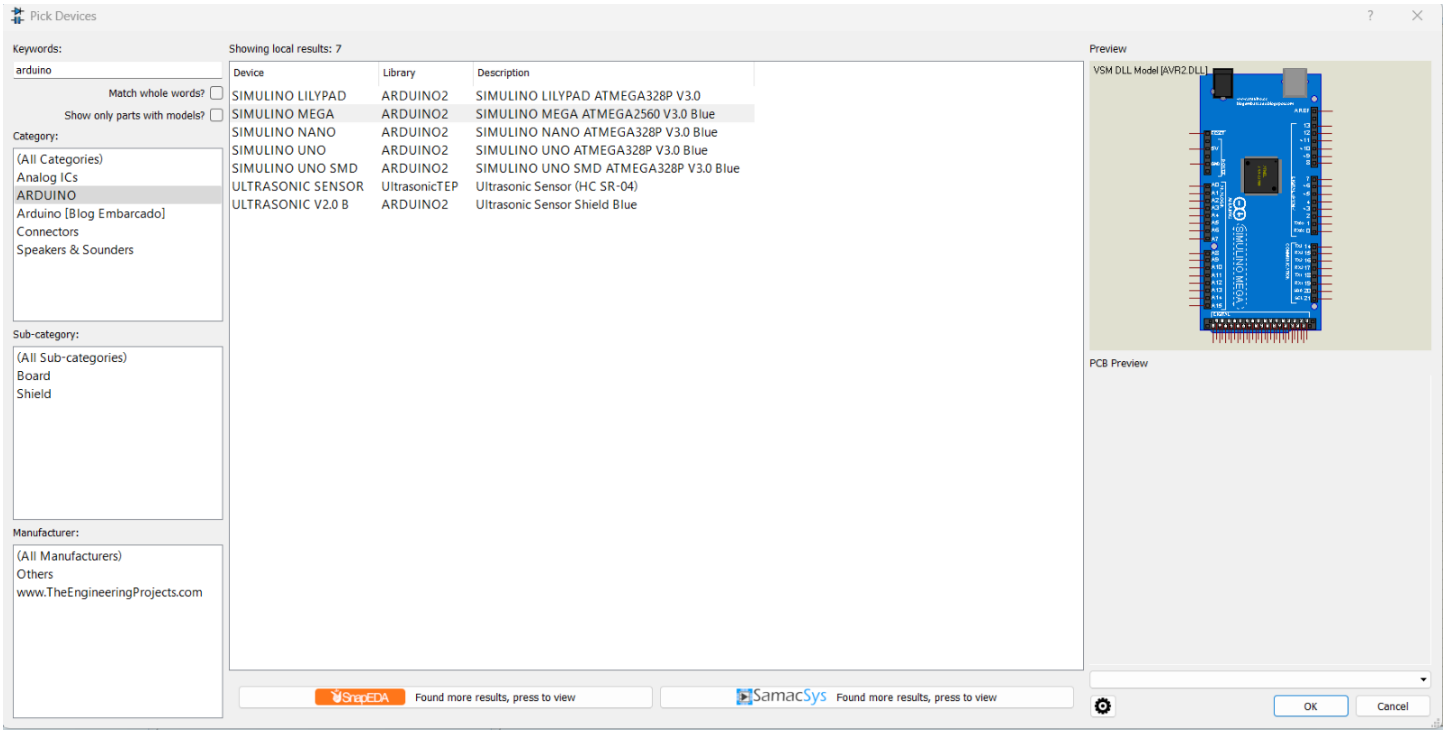


Fig. 21: Add Arduino Mega

Then click on any place of the working sheet to insert your component to it. Repeat the process to add any other component.

3.4.5.4 How to program your circuit on Proteus?

1. If you want to add code to your project, you should choose ***Create Firmware Project*** as shown in Fig. 22:

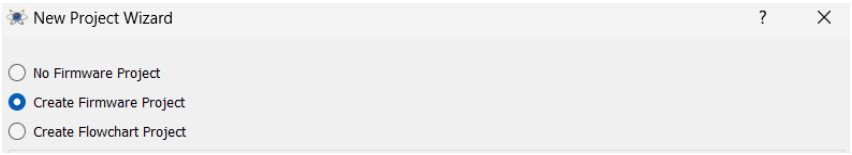


Fig. 22: Create Firmware Project

2. Then choose your microcontroller Family, e.g., Arduino as shown in Fig. 23:

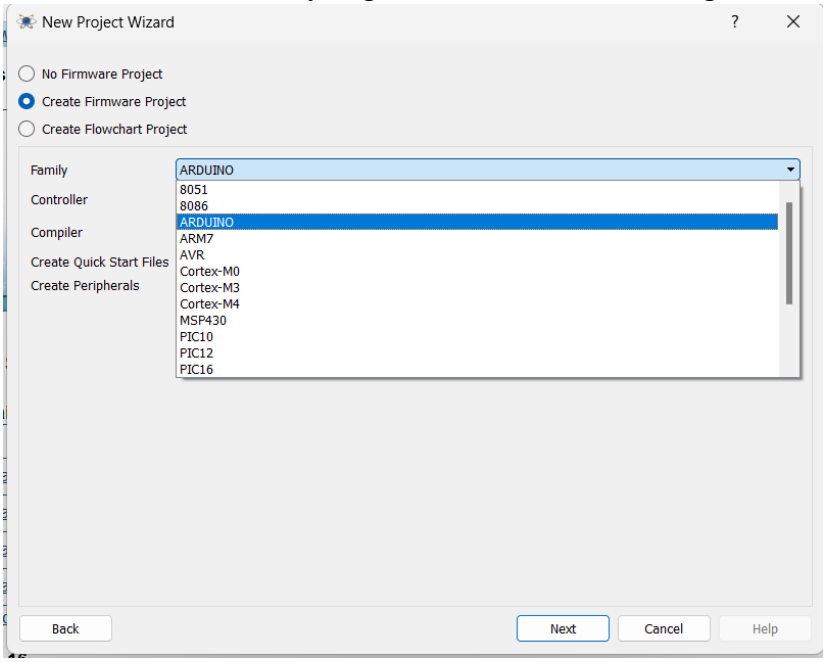


Fig. 23: Choose Controller Family

3. Then choose controller that you used in your circuit e.g., Arduino Mega as shown in Fig. 24:

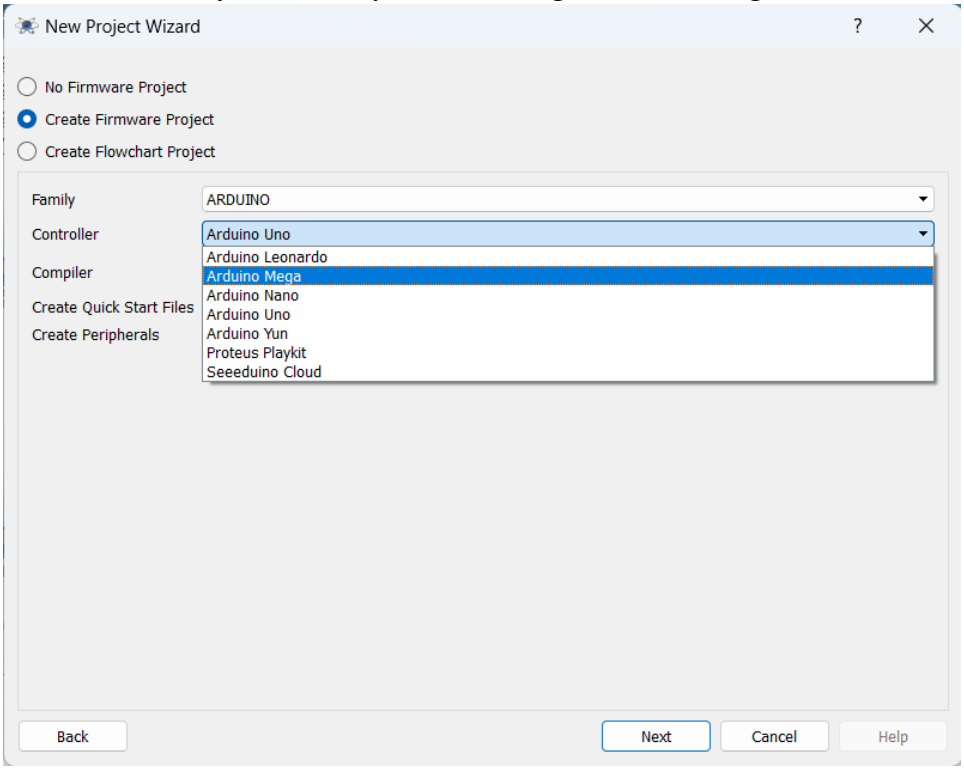


Fig. 24: Choose Controller Type

4. Then choose the compiler e.g., Arduino AVR (Proteus) as shown in Fig. 25:

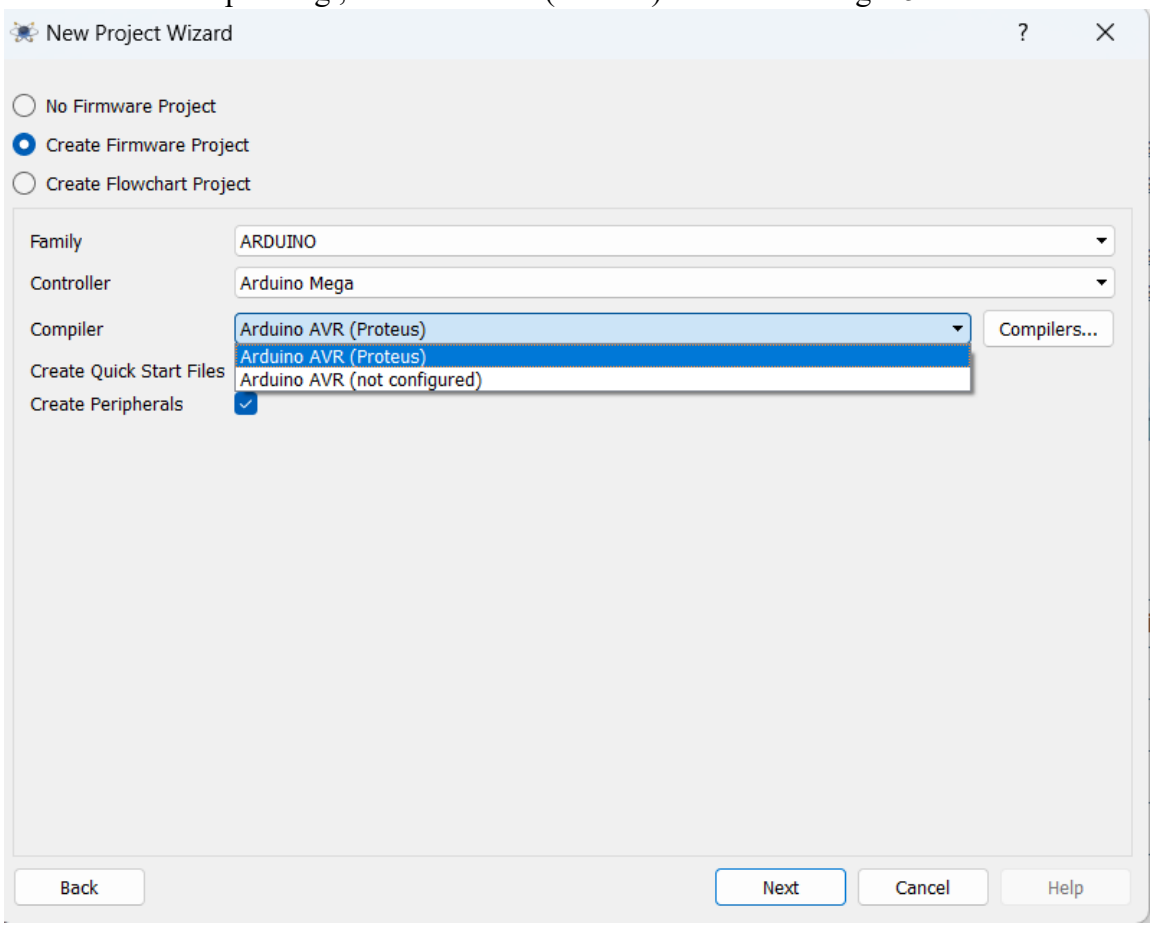


Fig. 25: Choose Compiler

5. After you click on finish, you will see the ***Source Code*** file where you can add your code to control your microcontroller in addition to the Schematic Capture file, as shown in Fig. 26:

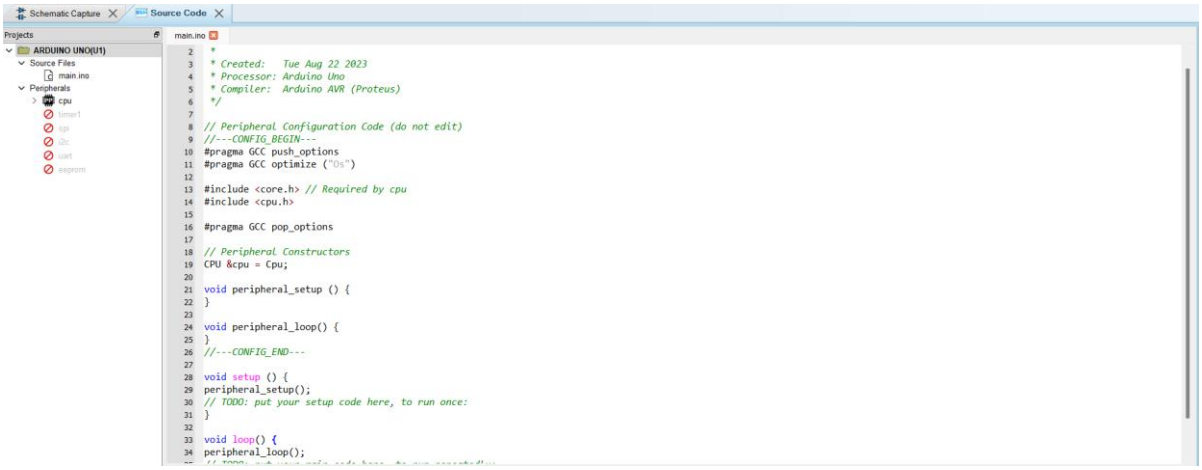


Fig. 26: Source Code File

<u>In our experiment we will use the Arduino Mega microcontroller, so let's discuss some aspects about Arduino in general, and Arduino Mega in special.</u>

3.4.6 Arduino Microcontroller:

3.4.6.1 Introduction to the Arduino

Is an open-source programmable circuit board that has been introduced in 2005 in Italy by Massimo Banzi as a way for non-engineers to have access to a cheap, straightforward tool for building hardware projects. Various sensors and inputs can send signals to Arduino for processing, Arduino may alter the surrounding environment by managing light sources, motors, or other actuators, and it may be incorporated into a wide range of makerspace projects, both basic and complicated. Due to its adaptability and low cost, Arduino has grown to be a very popular option for manufacturers and makerspaces wishing to construct interactive hardware projects.

3.4.6.2 Arduino IDE

The Arduino IDE (Integrated Development Environment) software is required to program the Arduino microcontroller. It was created by Arduino.cc and is primarily used for authoring, compiling, and uploading code to practically all Arduino Modules. The primary code, often referred to as a sketch, written on the IDE platform will eventually produce a Hex File, which is transported to and uploaded into the controller on board.

The IDE environment primarily consists of two fundamental components: Editor and Compiler. The Editor is used to create the necessary code, and the Compiler is used to compile and upload the code into the specified Arduino Module. Fig. 27 shows some features of the Arduino IDE:

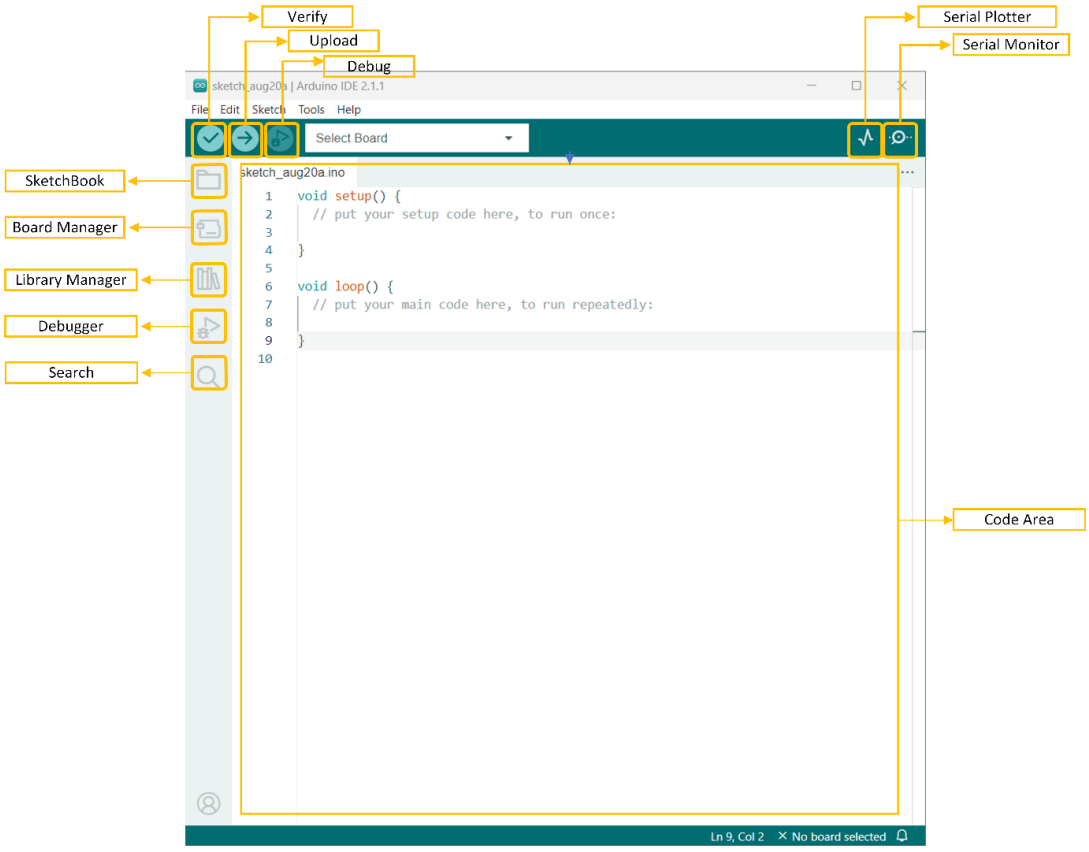


Fig. 27: Arduin IDE Features

3.4.6.3 Create new Sketch in Arduino IDE:

<u>To create new proteus project, follow the nest steps:</u>

1. Search about Arduino IDE program inside the search bar, and click on it to open it:

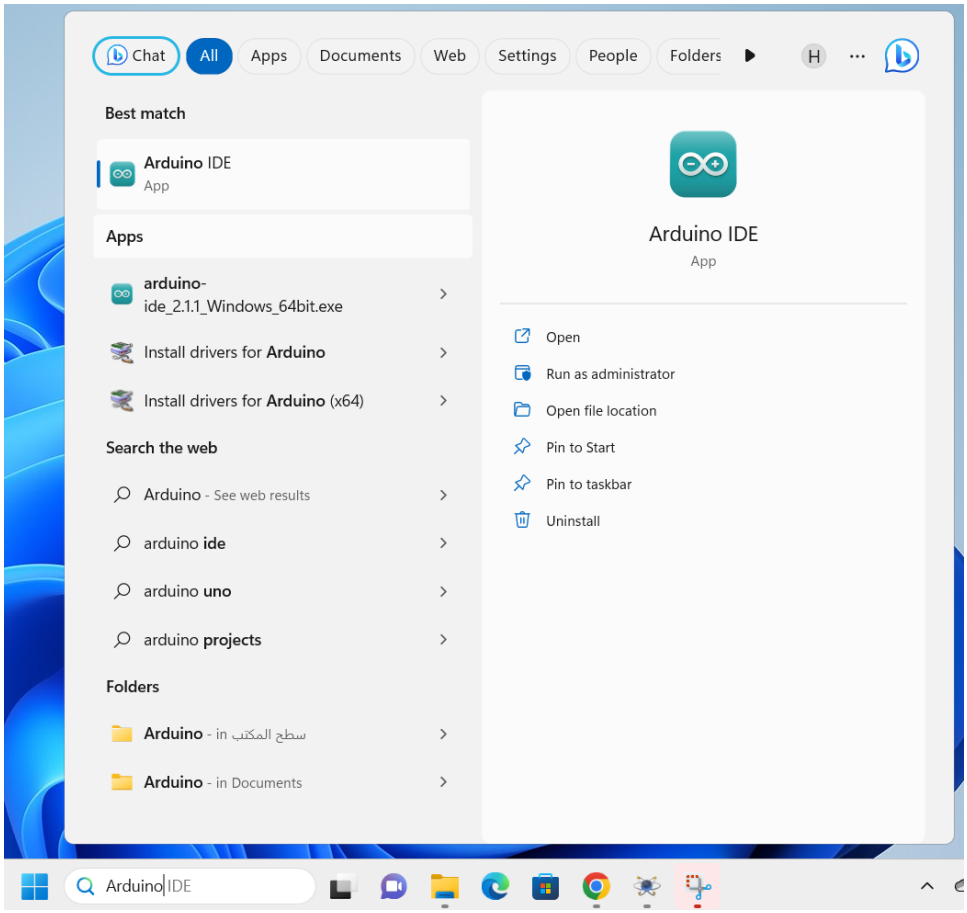


Fig. 28: Create New Sketch– Step1

2. The following window will appear, it already has a sketch as shown in Fig. 29:

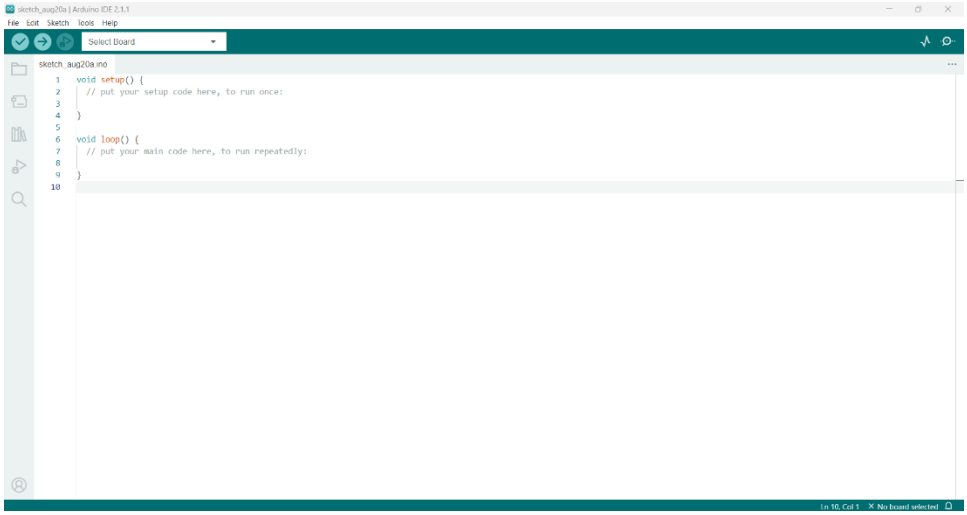


Fig. 29: Create New Sketch– Step2

3. If you want to create new Sketch, click on the File menu item in the menu bar, then click on ***New Sketch***, or press ***Ctrl + N*** as shown in Fig. 30:

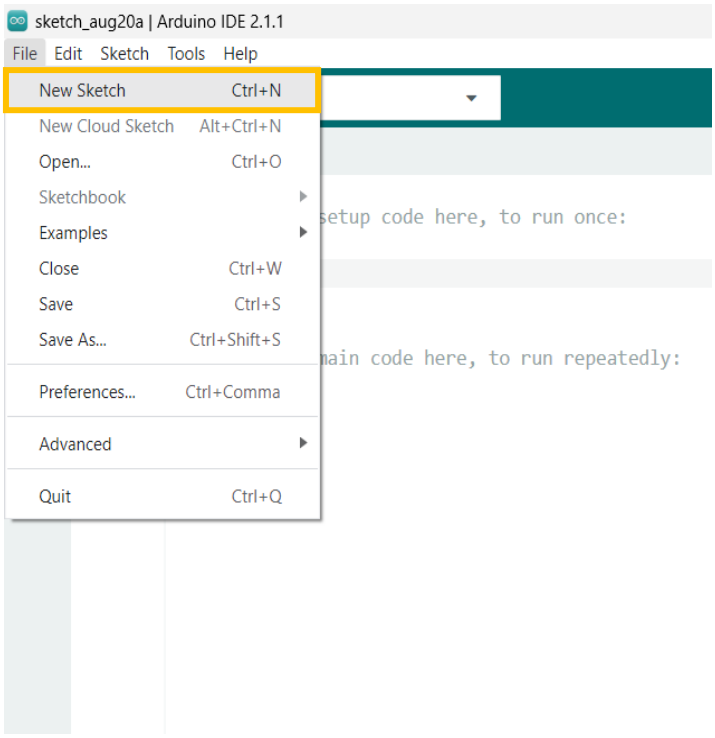


Fig. 30: Create New Sketch– Step3

3.4.6.3 Arduino IDE Main Blocks:

As Fig. 31 shows, the program contains 2 main blocks which are *setup()* and *loop()* functions.

- The *setup()* function is also known as *preparation block* includes the first segment of the code that will be run. In the setup part, the pin modes, libraries, variables, etc., are all initialized. It only runs once when the software is uploaded and once when the Arduino board has been reset or powered on. Each sketch starts with a zero *setup()*.
- The *loop()* which is also known as *execution block* contains repeated statements that are executed. Depending on the value of the variables, the block of code enclosed in curly brackets is repeated.

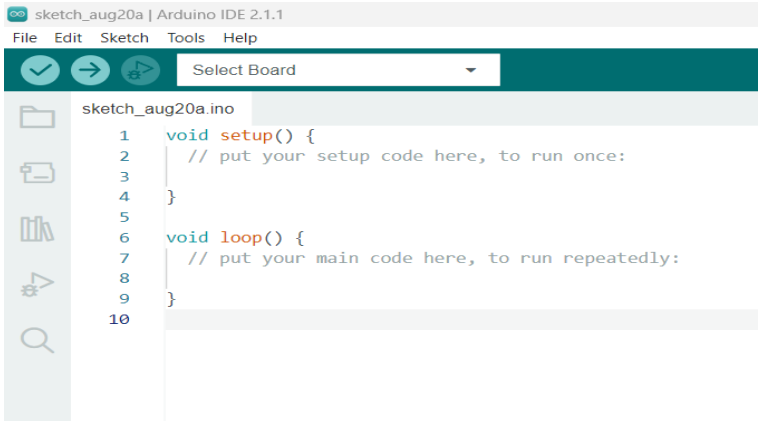


Fig. 31: The Program Structure of Arduino IDE

*Only in function declarations is the **void** keyword utilized. It means that the function being called should not expect any information to be returned to it.*

3.4.6.4 Common Arduino Functions:

C language is the programming language that is used to program the microcontrollers in the Arduino IDE. The table below shows the most common Arduino functions:

Table 3: Most Common Functions in Arduino

Type	Instruction	Usage	Syntax	Example
I/O Functions	pinMode()	set up a specific pin to operate either as an *input* or as an *output*, or *input_pullup*	pinMode(pin, mode)	pinMode(3, INPUT), pinMode(5, OUTPUT), pinMode(7, INPUT_PULLUP),
	digitalWrite()	Write a HIGH or a LOW value to a digital pin	digitalWrite(pin, value)	digitalWrite(5, HIGH), digitalWrite(5, LOW),
	digitalRead()	Returns values in the form 1 and 0, which correspond to 1 - HIGH and 0 - LOW values	digitalRead(pin)	digitalRead(3)
	analogRead()	Read the voltage applied to one of the pins	analogRead(pin)	analogRead(9)
	analogWrite()	Writes an analog value (PWM wave) to a pin	AnalogWrite(pin, value)	AnalogWrite(9, 127)
Control Flow Functions	delay()	Pauses the program or task for a specified duration of time	Delay(durartion)	Delay(500)
Serial Communication Functions	Serial.begin()	Set the baud rate for serial data communication	Serial.begin(baud_rate)	Serial.begin(9600)
	Serial.print()	Prints the data to the serial port	Serial.print(message_to_print)	Serial.print("Hello Arduino")
	Serial.println()	print line, which sends the string followed by the carriage return ('\r' or ASCII 13) and newline ('\n' or ASCII 10) characters	Serial.println(message_to_print)	Serial.println("Hello Arduino")
	Serial.available()	Gets the stored bytes from the serial port that are	Serial.available()	Serial.available()

		available for reading		
	Serial.read()	Reads the incoming serial data in the Arduino	Serial.read()	Serial.read()
	Serial.readString()	Reads the incoming serial data from the serial buffer in the string	Serial.read()	String = Serial.read()
	Serial.write()	Sends the binary data to the serial port in Arduino	Serial.write()	Serial.write(45) Serial.write("Hello Arduino")

3.4.6.5 Arduino IDE Data Types:

The following table provides all the data types that you will use during Arduino programming:

Table 4: Arduino Data Types

void	Boolean	char	Unsigned char	byte	int	Unsigned int	word
long	Unsigned long	short	float	double	array	String-char array	String-object

3.4.6.6 Arduino Mega:

The **Arduino Mega 2560** is a microcontroller board based on the ATmega2560 (datasheet). It has 54 digital input/output pins (of which 15 can be used as PWM outputs), 16 analog inputs, 4 UARTs (hardware serial ports), a 16 MHz crystal oscillator, a USB connection, a power jack, an ICSP header, and a reset button. It contains everything needed to support the microcontroller; simply connect it to a computer with a USB cable or power it with a AC-to-DC adapter or battery to get started. The Mega 2560 board is compatible with most shields designed for the Uno and the former boards Duemilanove or Diecimila. Fig 32. and Fig 33 shows the components that make up the Mega board, and the Pinout diagram of it respectively:

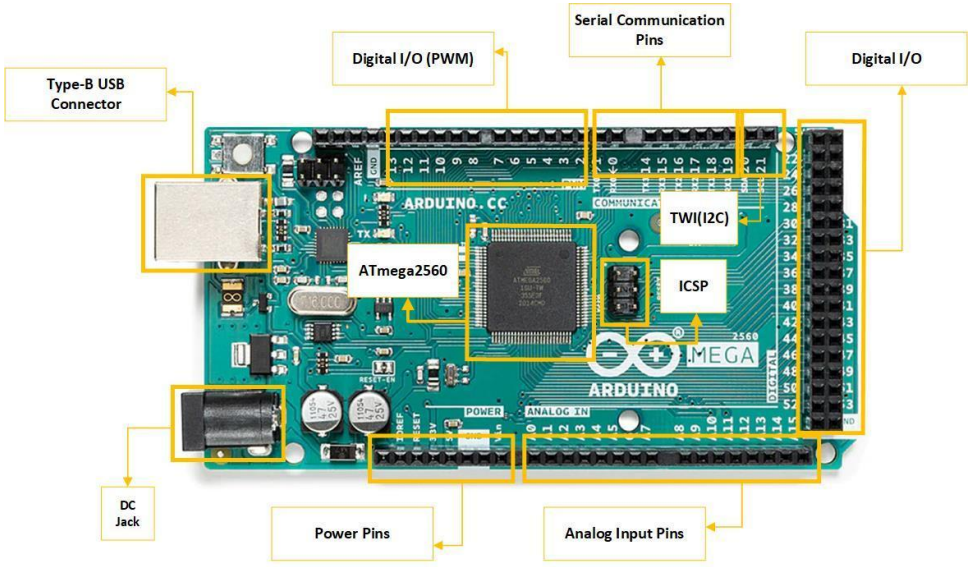


Fig. 32: The components of the Arduino Mega Board

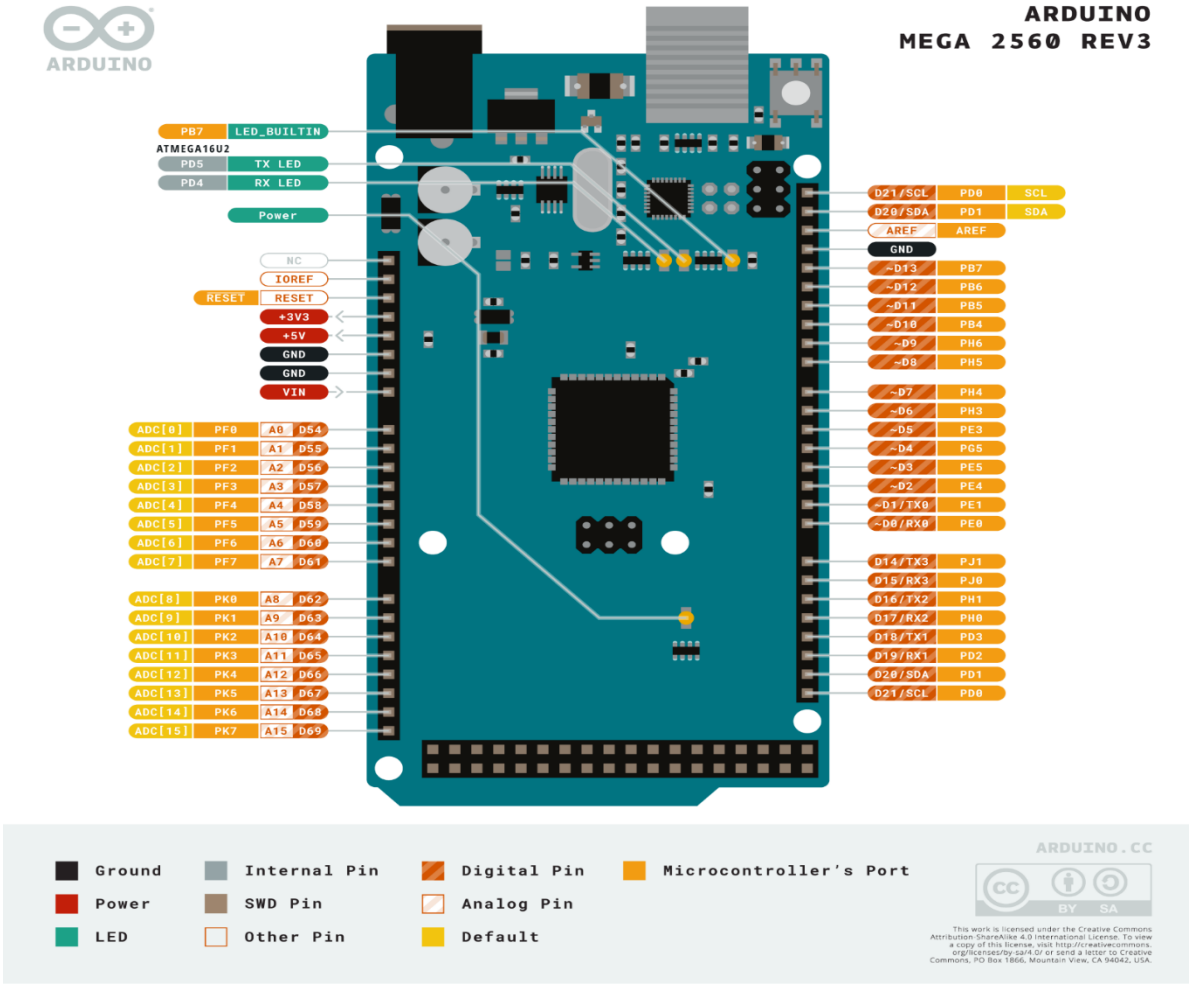


Fig. 33: The Pinout Diagram of Arduino Mega

3.5 Procedure

Part 1: Control LEDs Programmatically

Task1: Blinking LED

Let's begin our exploration of Arduino programming with this very basic program that will make an LED blink. The LED will function as a basic light that can be turned ON and OFF for a set period of time.

Structure of the LED: An LED is a two-terminal device. Cathode and Anode are the names of the two terminals. Anode and Cathode are the names of the long and short terminals, respectively. Anode is the positive terminal in this instance, and cathode is the negative terminal, as shown in Fig. 35:

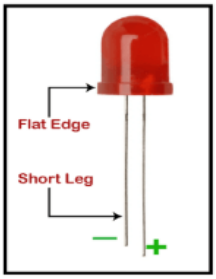


Fig. 35: Structure of the LED

Components of the project:

The components used in the blinking of an LED are listed below:

a. 1 x Arduino Mega 2560 board.
b. 1 x Breadboard
c. 2 x Jump wires
d. 1 x LED
e. 1 x Resistor of 220 Ohm

Procedure:

We will simulate this example on the proteus, so follow the following steps:

- As you learned before, add the Arduino Mega to the working sheet, as shown in Fig. 36:

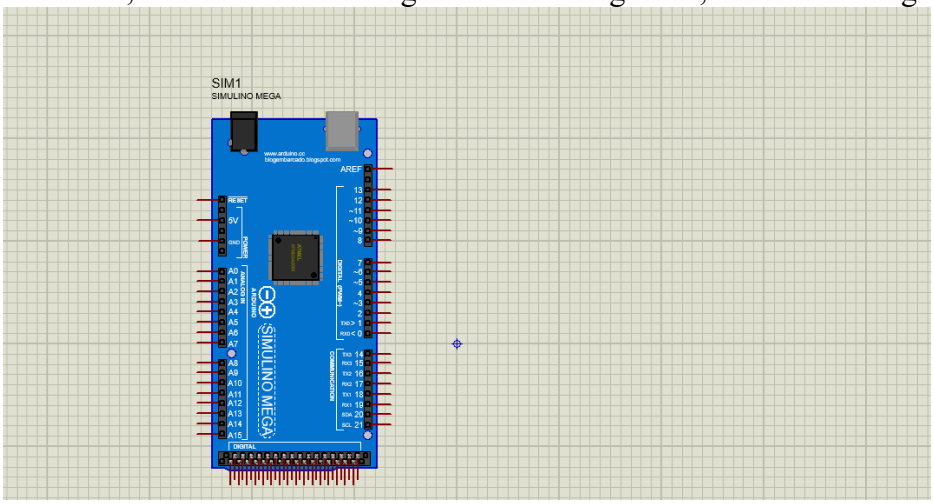


Fig. 36: Circuit Connection I

- Add a Yellow_LED to the working sheet:

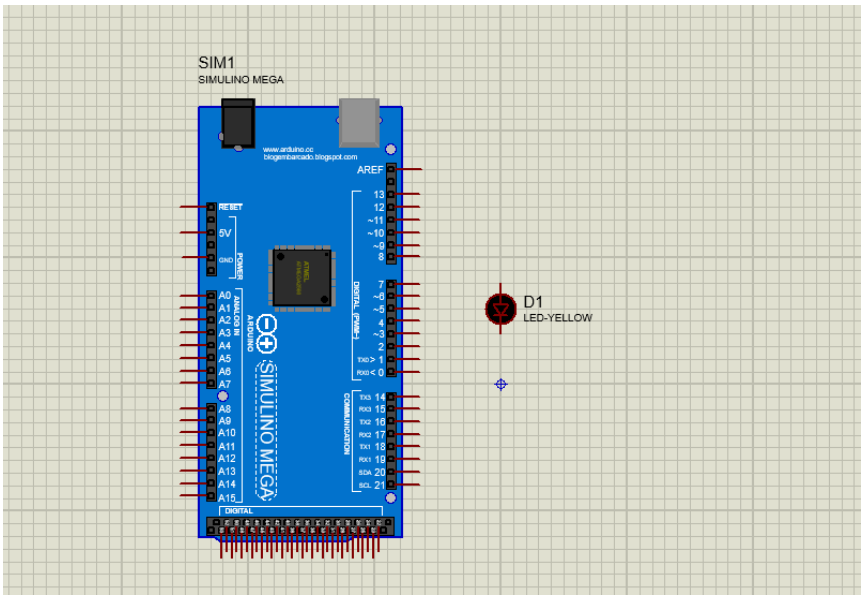


Fig. 37: Circuit Connection II

- Add a 220 ohm resistor and connect it in series with the LED, as shown below:

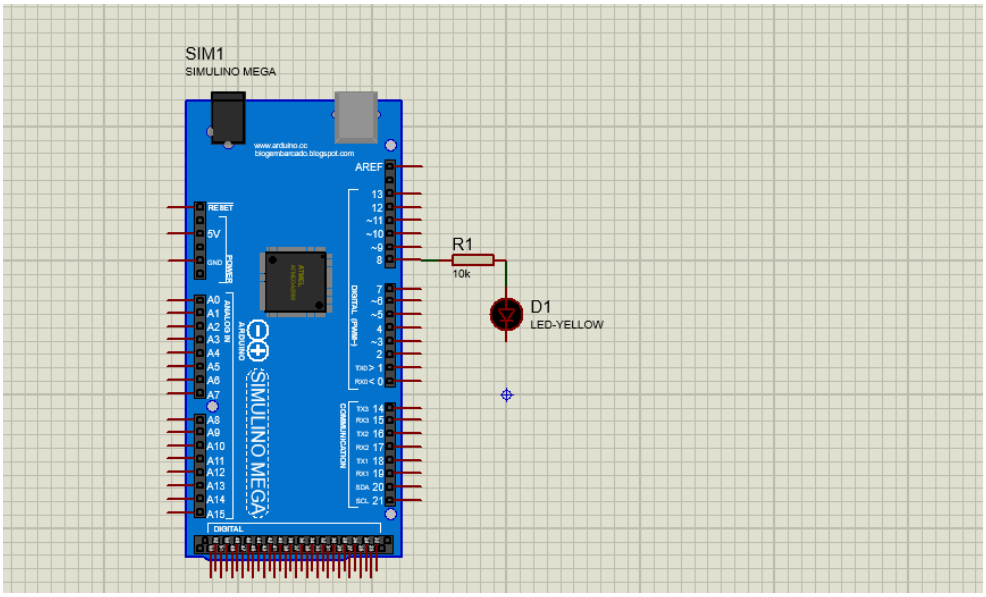


Fig. 38: Circuit Connection III

- Connect the negative/shorter terminal (Cathode) of the LED to the GND pin of the Mega board using the wire.

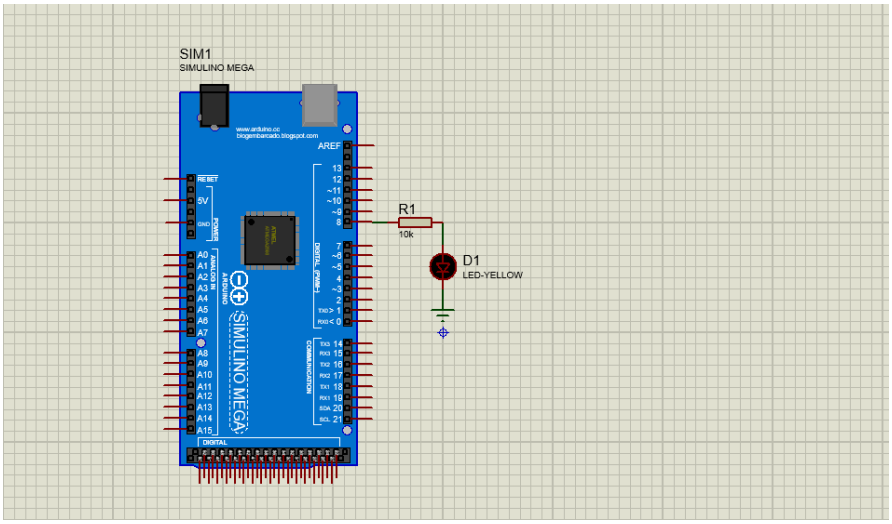


Fig. 39: Complete Circuit Connection

Important Note: The resistor must be connected in series with the LED to prevent the excess current from reaching the LED. The excess current in the connection can burn the LED. Hence, a resistor in series with the LED is used in the connection.

Now, after we connected our circuit, we need to write the program to make the LED turn on for 1 second and then turn off for 1 second too, and so on. We will write the code using Arduino IDE, then we will compile this code and add the generated ***.hex*** file to the proteus project so we can simulate our circuit using this code.

Follow the following steps to write your code:

1. Declare delay as a constant using `#define` directive, and declare a global variable for the 13th pin, using lines shown below:

```
#define DELAY 1000
int LED_PIN = 13;
```

2. Set the `LED_PIN` as `OUTPUT` pin inside the setup function as shown below:

```
void setup () {
  pinMode(LED_PIN, OUTPUT); //set the 13th digital pin to the OUTPUT mode
}
```

3. To turn the LED on for 1 second, and then turn it off for another second in a repeatedly manner, we need to write `HIGH` value to the `LED_PIN` and wait for 1 second using `delay` function with 1 second then write `LOW` value to it and wait for 1 second using `delay` function too. This code should be inside the `loop()` function to execute it repeatedly, as shown below:

```
void loop() {
```

```
  digitalWrite(LED_PIN, HIGH);    // turn the LED on (HIGH is the voltage level)
  delay(DELAY);                   // wait for a second
  digitalWrite(LED_PIN, LOW);     // turn the LED off by making the voltage LOW
  delay(DELAY);    // wait for a second
}
```

Now, to compile your code and simulate your circuit follow the following steps:

1. In your Arduino IDE sketch, go to *File* > *Preferences* and tick compilation in '*Show verbose output*', this will enable us to find a path to hex file, as shown below:

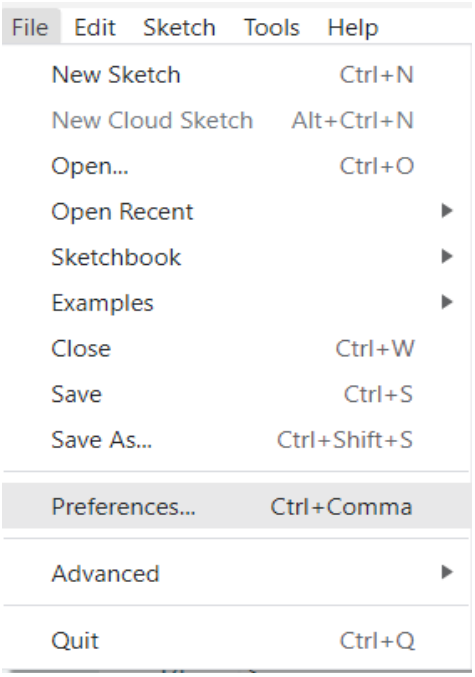


Fig. 40 (a): Edit Preferences

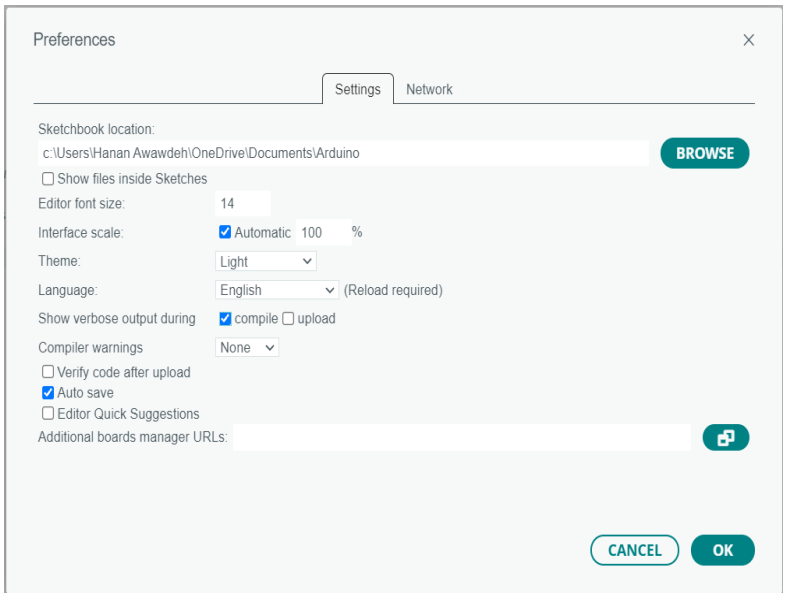


Fig. 40 (b): Edit Preferences

2. To choose Arduino Mega as your board, go to the *Tools* > *Boards* > *Arduino AVR Boards* and choose *Arduino Mega or Mega 2560,* as shown below:

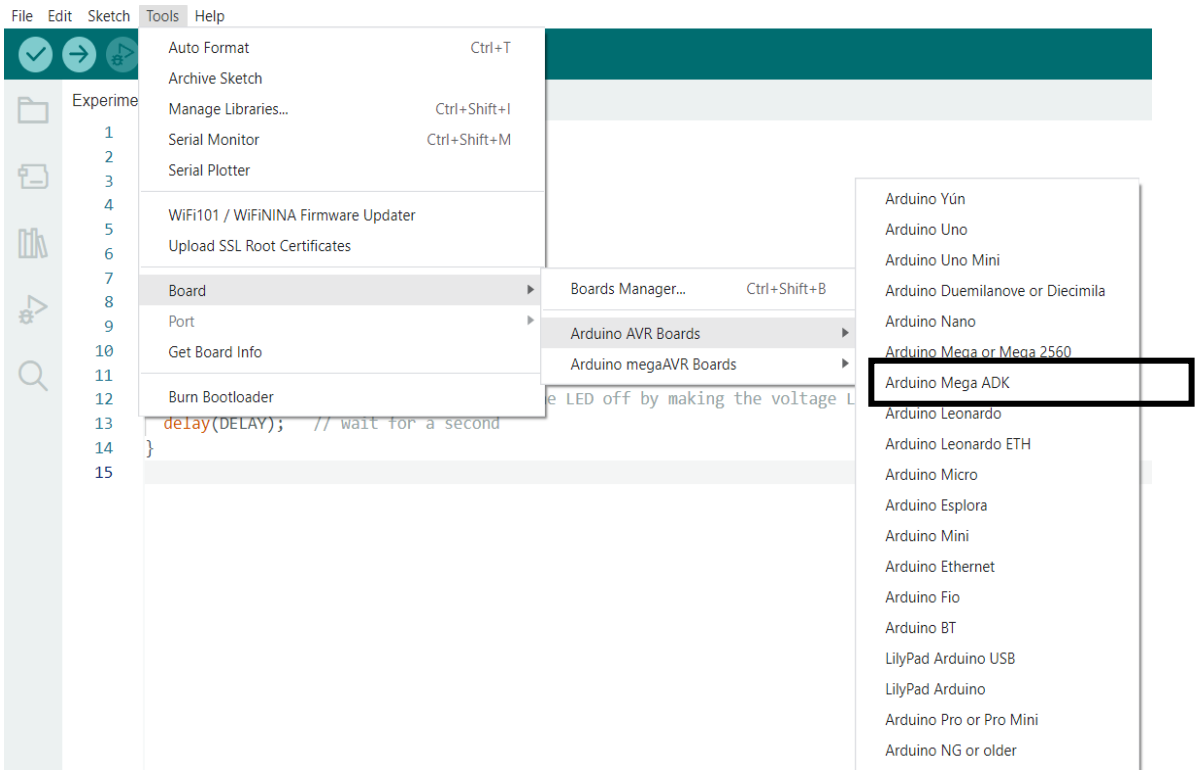


Fig. 41: Edit Preferences

To compile your code, click on  icon or go to the *Sketch >Verify/Compile* or press *Ctrl+R,* as shown below:

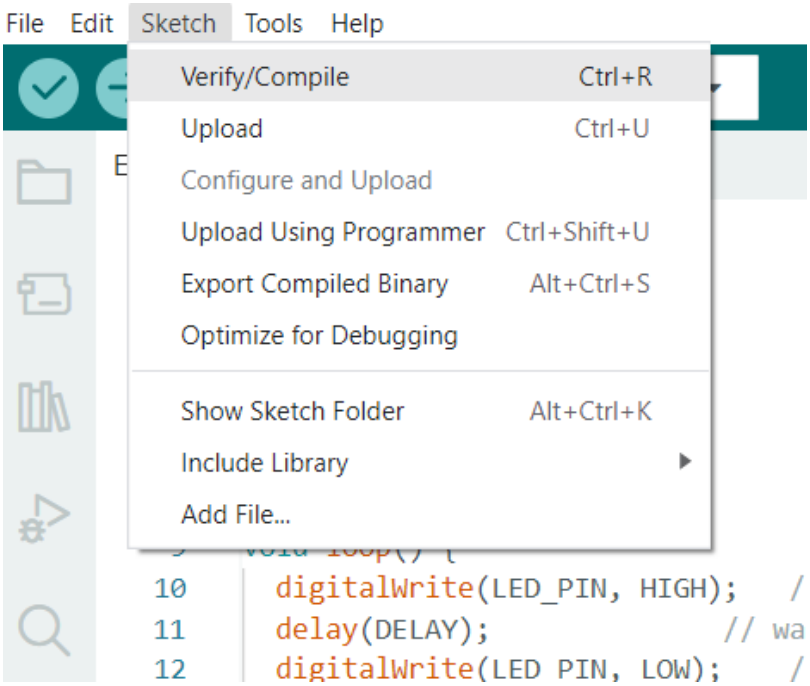


Fig. 42: Compile Code

If the compilation is successful, move on to the following step. If there are any issues, fix them or ask your TA before recompiling.

3. Copy the path to the hex file as given below from the compilation box:

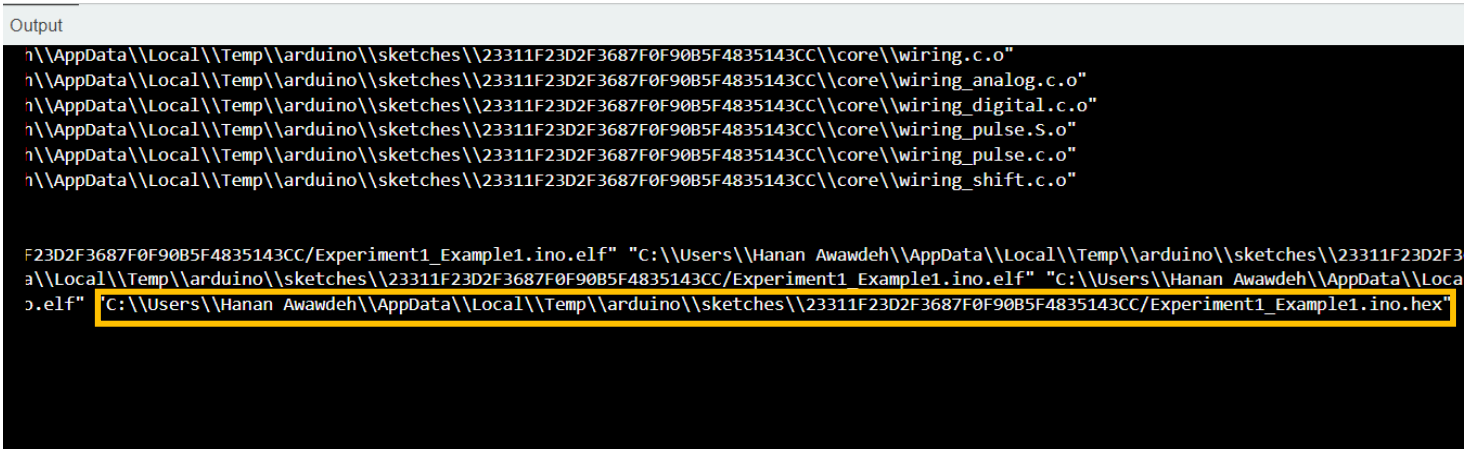


Fig. 43: Hex File Paths

4. In your proteus project, right click by mouse on the Arduino Mega and click on *Edit Properties* then paste the path to the hex file into the *Program File* then click *OK*, as shown below:

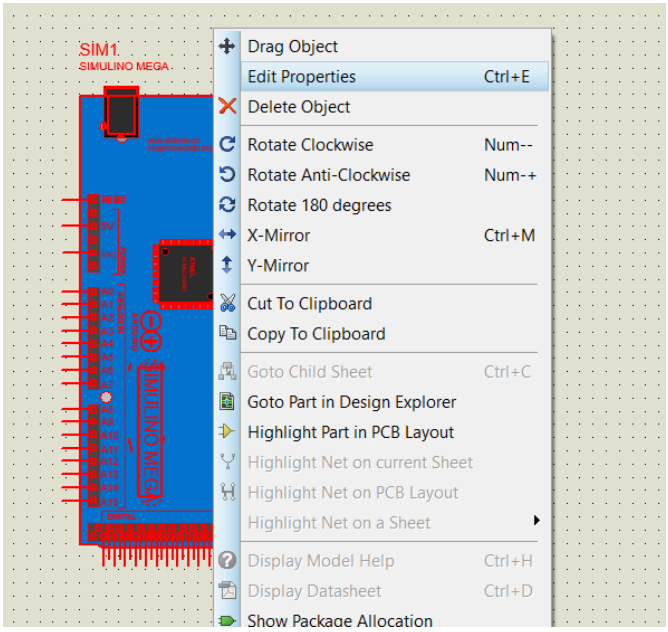


Fig. 44 (a): Add hex file to the File Program

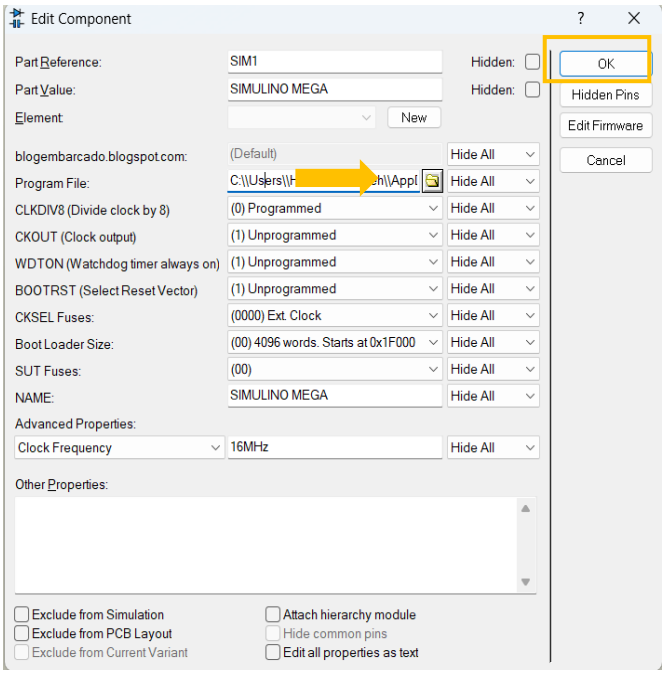


Fig. 44 (b): Add hex file to the File Program

5. Now, to start simulation click at  icon or go to *Debug* > *Run Simulation* or press F12.

Now, to upload your code on the physical Arduino Mega, follow the following steps:

1. Connect the Arduino Serial USB cable to your computer as shown below:

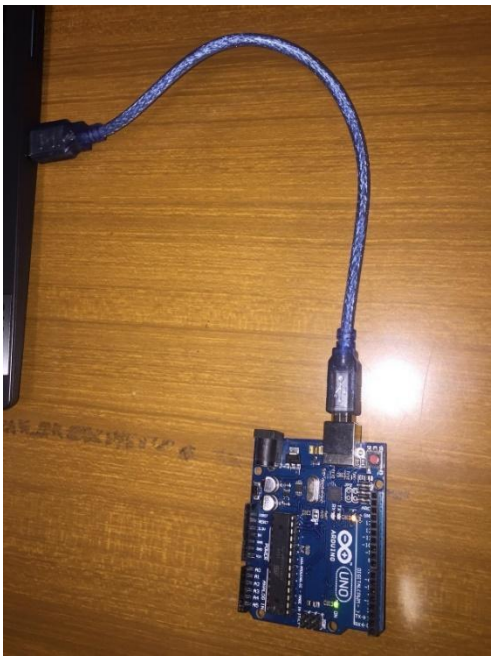


Fig. 45: Arduino Serial Connection with Computer

2. Select the board as mentioned earlier.
3. Go to *Tools* > *Port* and select the serial port in the Arduino IDE as shown below:

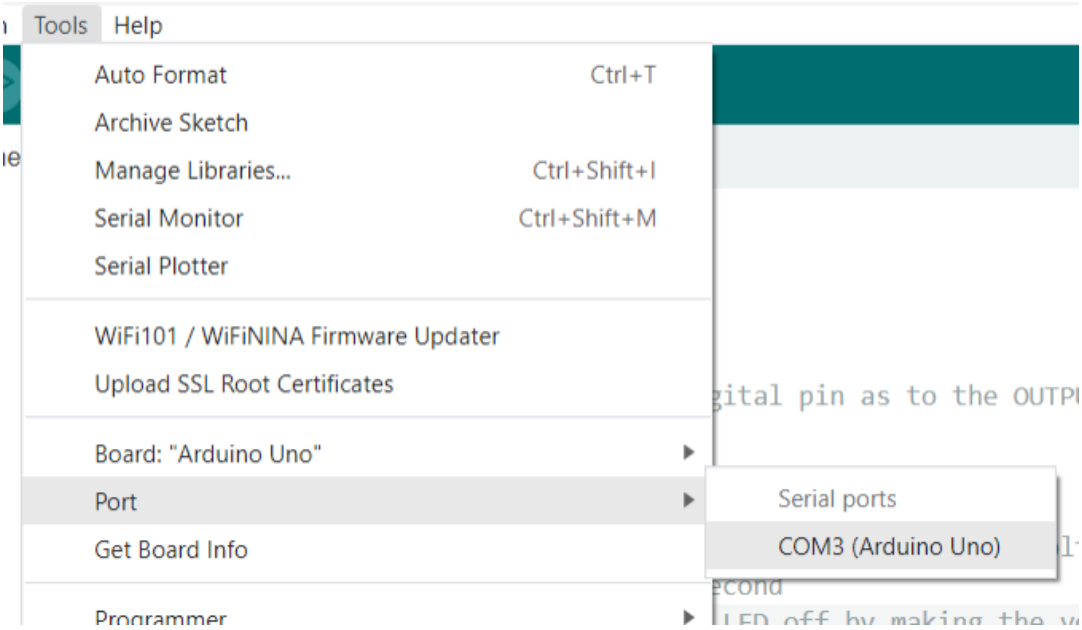


Fig. 45: Select Serial Port

4. Upload the sketch or code on the board by clicking on  icon or go to the *Sketch* > *Upload* or press *Ctrl* + *U* as shown below:

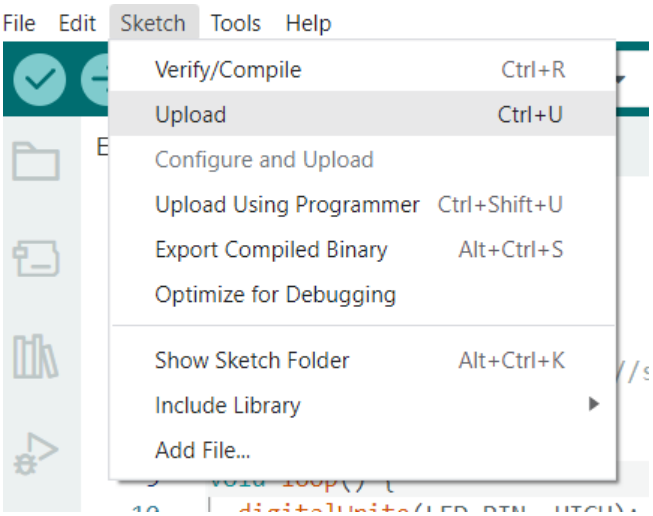


Fig. 46: Upload the code to the Arduino

The on-board LED connected with pin 13 will blink.

TODO 1: Modify your work by using another 3 LEDs, then write a program to make two LEDs are illuminated for 1 second while the other two remain off, followed by a reversal where the previously off LEDs turn on and the initially illuminated ones turn off for another 1-second cycle, and so on. Follow the steps we explained and mentioned previously to do the task.

Part 2: Control LEDs Using Push Button:

Pushbutton: The buttons are like switches that create and break electrical connections in the circuits. The button plays a transition between ON and OFF state. A single press turns the state ON, while another press turns the state OFF. It means that the button connects the two points in a circuit when we press them.

Structure of the slide pushbutton: The pushbutton is a square shaped button with four terminals, as shown in Fig. 47. The two pins are next to each other on one side and the other two pins on the other side. The pins across to each other are connected. The pins next to each other can only be connected when we press the button.

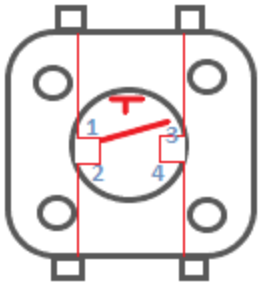


Fig. 47: Push Button Structure

Task 1: Light a LED using a switch

Components of the project:

The components used in the blinking of an LED are listed below:

a. Arduino Mega 2560 board.
b. 1 x 220-ohm resistor

c. 1 x 10 Kohm resistor
d. Jump wires
e. 1 x pushbutton
f. 1 x red LED (we can take LED of any color)
g. 1 x Breadboard

Procedure:

- Connect the negative pin of the LED to the Ground, and the positive one to the resistor to PIN 13.
- Connect a 10 Kohm resistor to one end of the pushbutton to the ground.
- Connect PIN 4 of the Arduino Mega to the 10k resistor to the ground.
- Connect the other end of the pushbutton to 5V.

Your connection should be as shown in Fig. 48:

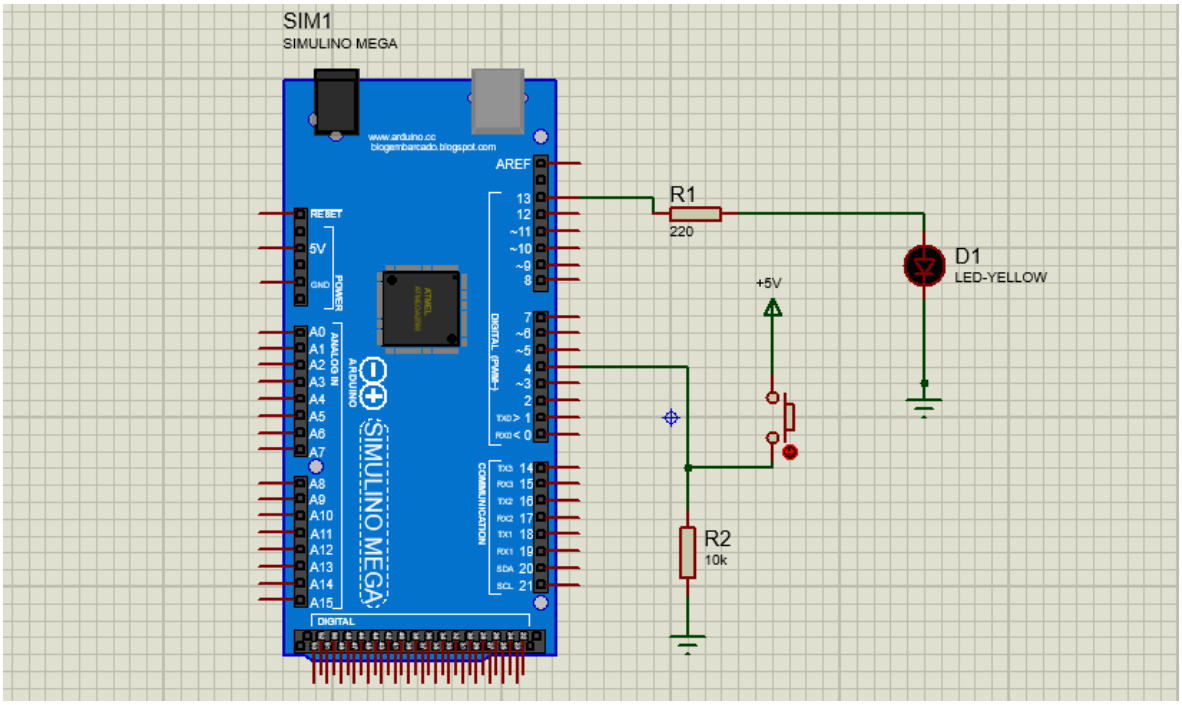


Fig. 48: Complete Circuit Connection

What is the type of 10k resistor above? What the effect of its type on the overall performance of the circuit?
Now, after we connect our circuit, we need to write the program to make the LED turn on when the pushbutton is pressed and turn off when it is released.

Follow the following steps to write your code:

1. We will add a global variable for the 4th pin and another one to save the pushbutton state to the previous code, as shown below:

```
const int Button_PIN = 4;
```

```
const int LED_PIN = 13;
int currentState;
```

2. Set the `LED_PIN` as `OUTPUT` pin, and the `Button_PIN` as `INPUT` inside the setup function as shown below:

```
void setup() {
  pinMode(LED_PIN, OUTPUT);
  pinMode(Button_PIN, INPUT);
}
```

3. To turn the LED on when press on the pushbutton and turn it off when it is released, we need to read the state of the `Button_PIN` using the `digtalRead()` function and store it in the `currentState` variable so we can use it later. If the state of it is `LOW`, then the LED will be off else it will turn on. This code should be inside the `loop()` function to execute it repeatedly, as shown below:

```
void loop() {
  currentState = digitalRead(Button_PIN);
  if (currentState == LOW) {
    digitalWrite(LED_PIN, LOW);
  }
  else if (currentState == HIGH) {
    digitalWrite(LED_PIN, HIGH);
  }
}
```

TODO 2: Connect three LEDs and a switch button to an Arduino and create a binary incremental counter that increments with each button press (from 000-111). **Use Pullup resistor with the push button in this task**.

TODO 3: Use two buttons to control the frequency of a blinking LED. The first button will increase the blinking frequency, and the second one will decrease the frequency of the blinking. **You should use PWM pins for LEDs, and an internal PULLUP resistor with the push buttons**.

The following notes will help you to do the task:

- The Arduino Mega has 15 PWM pins that are 2 to 13 and 44 to 46. These pins are marked with the negation sign "~" on the Arduino. These pins can generate a pulse as per the given inputs. Arduino supports an 8-bit wide pulse that can have 256 possible levels (0 to 255).
- To write on the PWM pins, we use `analogWrite(Pin, Value)`, and the V**alue** should be between **0** and **255**.
- To enable internal PULLUP resistor for the buttons instead of using the PULLUP resistor externally, you should set the mode of buttons as INPUT_PULLUP, as shown: `pinMode(pin, INPUT_PULLUP).`

Part 3: Interfacing Keypad and LCD with Arduino

Keypad: The 4x3 Keypad that is depicted in the Fig. 49 below will be used. Membrane keypads are a common and affordable form of keypad used by hobbyists. When pressed, the membrane keypad's surface, which is supple and flexible, makes a connection with the underlying circuit. When a key is pressed, an inductor with a coil shape underneath it induces a current or voltage signal. Three layers make up a membrane keypad. While the center layer is non-conductive, the top and bottom layers are constructed of conductive materials. The intermediate layer stops the top and bottom from contracting when the key is not pressed or is open. But when key is pressed and closed, the middle layer creates a small hole that allows the top and bottom layer to come into contact and induce current/voltage.



Fig. 49: Keypad

As you can see from Fig. 49, the keypad has 7 pins with 4 rows and 3 columns. The keypad starts with rows and ends with the column. In the figure, the first 4 rows are labelled as R1, R2, R3 and R4 and the 3 columns are labelled as C1, C2 and C3.

How to connect Keypad with Arduino Mega:

Connect pins 7, 8, 9, and 10 of Arduino to the four-row pins R1, R2, R3, and R4 of the 4x3 keypad. Similarly, connect the Arduino pins 11, 12, and 13 to the three columns C1, C2, and C3 of the keypad.

Liquid Crystal Display (LCD): Character LCDs are available in many different sizes (16×2 or 1602, 20×4 or 2004, etc.) but most of them use the same HD44780 parallel interface LCD controller from Hitachi. 16×2 LCD has 16 pins and can be operated in 4-bit (using 4 data lines) or 8-bit mode (using 8 data lines).

16×2 LCD Pin out:

Fig. 50 shows the Pin out of the 16×2 LCD, count starts from left to right. The very first pin from the left is the VSS pin. Then VCC, V0, RS, R/W, EN, D0 to D7, and at last Anode and Cathode.

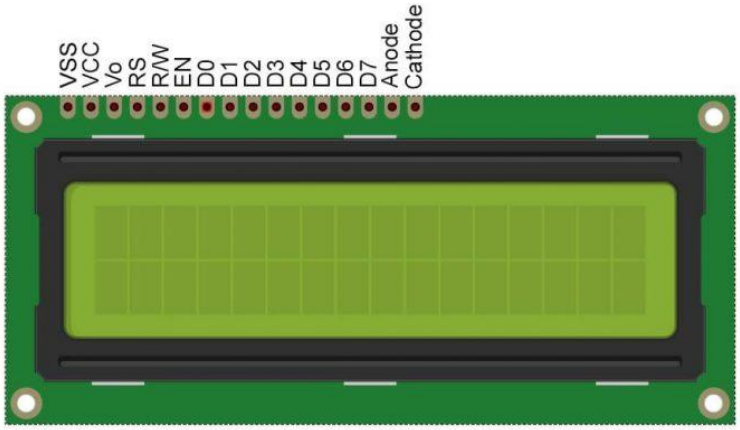


Fig. 50: LCD

VDD/VCC & VSS: These two pins are for the power supply. You have to provide a 5v power supply through these two pins. Connect VDD/VCC to +ve end of the power supply and -ve end to VSS.

V0/VEE: This pin is for controlling the contrast of the LCD. Connect a variable 10 kΩ resistor to this pin.

RS: Register select pin. Basic 16×2character LCD has two types of register, command register, and data register. Logic 0 at pin RS selects command register and Logic 1 at pin RS selects data register. If we select the data register, input through the data line (D0-D7) will be treated as data to show in the display and when we select command register input will be treated as a command.

R/W: Read/Write pin. This pin gives us an option whether we want to read the data from LCD or write to LCD.

E: Enable pin. A HIGH to LOW pulse is required at this pin to enable the LCD for any operation.

D0 – D7: Pin D0 to D7 are 8-bit data pins. The command and data are sent to the LCD through these pins. Pin D7 to D4 are used for 4-bit mode.

How to connect LCD with Arduino Mega:

Connect the pin A5 and pin 2 to the RS and E pins of the LCD respectively. And the pins 3, 4, 5, and 6 to the D4, D5, D6 and D7 of the LCD. Connect the R/W pin of the LCD is to the ground. Connect the 10KOhm potentiometer to the LCD with its middle pin to the Vee. The other sides of the POT are connected to +5V and ground respectively. The Vss and Vdd of LCD are connected to the ground and +5V supply.

Your connection should be as shown in Fig. 51:

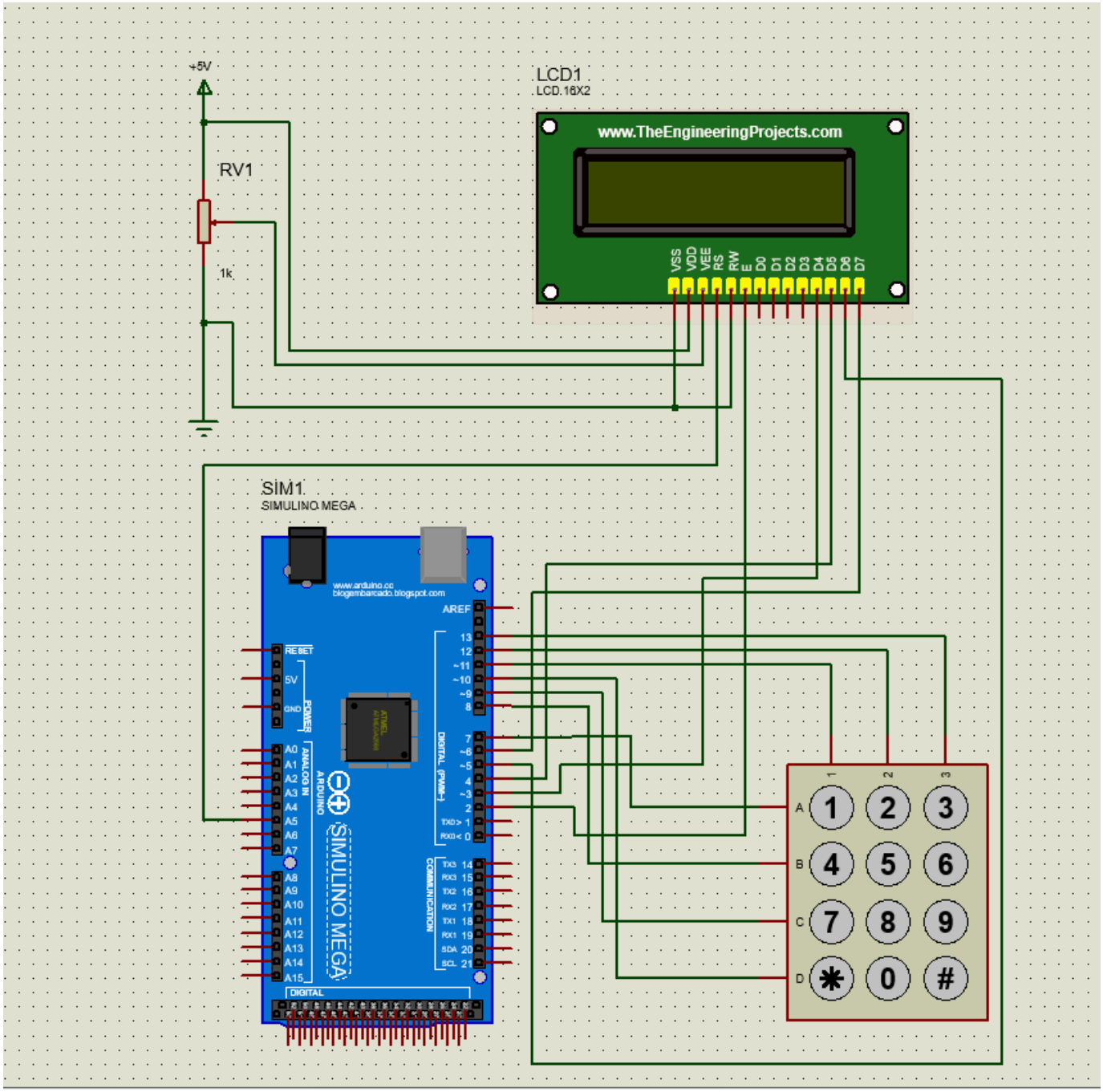


Fig. 51: Complete Circuit Connection

Now, after we connect our circuit, we need to write the program to monitor for any key press and when a key is pressed, it will be sent the pressed key to the LCD.

Follow the following steps to write your code:

1. We will include the `Keypad` and `LiquidCrystal` libraries using `#include` directive, as shown below:

```
#include <Keypad.h>
#include <LiquidCrystal.h>
```

The Keypad library isn't found by default in the Arduino IDE, and you need to install it from the library Manager as shown below:

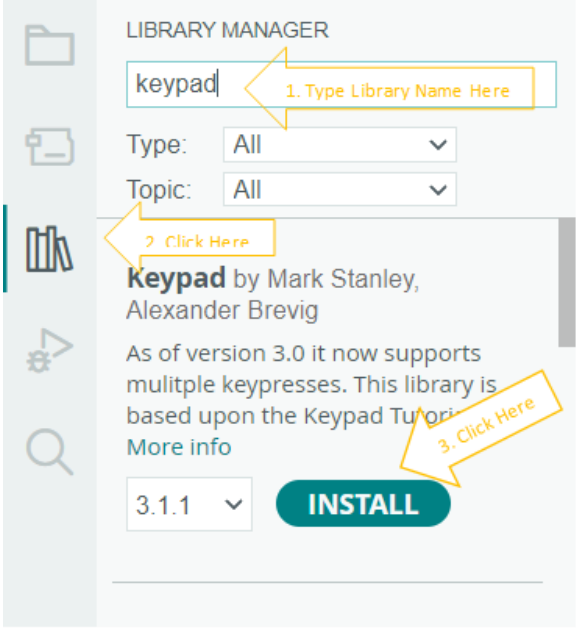


Fig. 52: Install Keypad Library

2. Initializes an object of the `LiquidCrystal` class as shown below:

```
LiquidCrystal lcd(A5, 2, 3, 4, 5, 6);
```

3. Declare variables for rows, columns, keys of the keypad, as shown below:

```
const byte rows = 4; // set display to 4 rows
const byte cols = 3; // set display to 3 columns
char keys[rows][cols] =
{{'1','2','3'},
{'4','5','6'},
{'7','8','9'},
{'*','0','#'}};
```

4. Sets up a matrix keypad using the *Keypad* library by initializing an array `rowPins` with the Arduino digital pin numbers connected to the rows of the keypad, and the same for columns with `colPins` array, and then create `keypad` object, as shown below:

```
byte rowPins[rows] = {7, 8, 9, 10};
byte colPins[cols] = {11, 12, 13};
Keypad keypad = Keypad( makeKeymap(keys), rowPins, colPins, rows, cols);
```

5. Inside `setup ()`, initializes the LCD display using `begin()` function, the parameters 16 and 2 specify the number of columns and rows on the LCD display. Use the `print()` function provided by the `LiquidCrystal` library to display the text `"Enter Number:",` then wait for 3 seconds and then clear the content on the LCD display using `clear()`.

```
void setup () {
      lcd.begin(16, 2);
      lcd.print("Enter Number:");
      delay(3000);
      lcd.clear();
}
```

6. Continuously scan the keypad in a loop and displays the pressed keys on an LCD display, as shown below:

```
void loop() {
char key = keypad.getKey();
if (key != NO_KEY){
    lcd.print(key);
   }
}
```

The results will be as shown below:

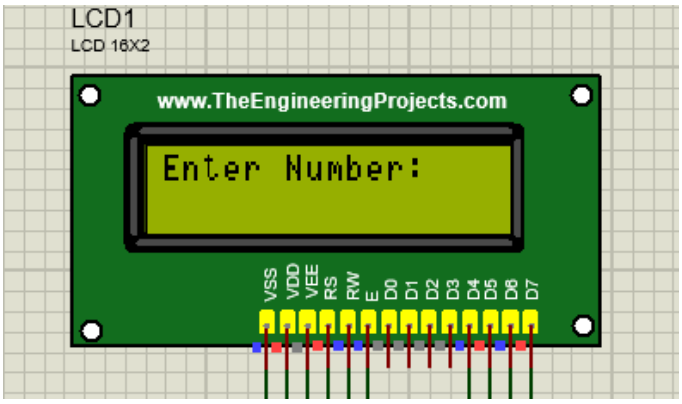


Fig. 53 (a): Program Result

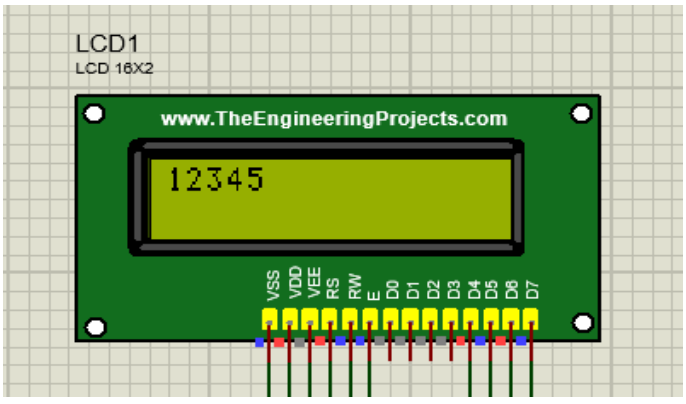


Fig. 53 (b): Program Result

TODO 4: Implement a feature that enables the user to set a 4-digit numeric password by pressing the '*' key, and save it when '#' is pressed. Afterward, prompt the user to enter the password. If the entered password is correct, display "Hello" on the LCD. Note: Do not allow the user to enter more than 4 digits.



Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

Microprocessor and Embedded Systems Lab (ENCS5101)

Experiment No. 4 - DC motor control using Arduino

4.1 Objectives

1. Use of digital controller to implement speed and position control of different types of motors.
2. Writing and uploading basic control programs to read external status (sensors and switches) and sending control sequences through controller ports.
3. Use of software and on-chip hardware timers in events control and synchronization.
4. Use the PWM unit of the digital controller to implement speed and direction control of a DC motor.

4.2 Equipment's & Tools

- Proteus v8.13
- Arduino Mega 2560
- Arduino IDE v2.1.1
- Switches
- LEDs

4.3 Prelab

1. Read the experiment, Implement, build the system, and test all the programs using Proteus.
2. Write the required tasks programs in each part of the experiment and test them using Proteus.

4.4 Introduction

4.4.1 Motors:

Motors work through a process called induction. When you can put electric charge through wire, a magnetic field is created. A coiled wire will create a stronger field, and increased current. Most motors require excitation currents greater than the controller pins driving capacity (for example, the maximum current provided by Arduino is approximately150mA). Moreover, if motors are directly connected to the output of any pin, this can damage both the pin and the motor. To prevent controller damage and ensure correct motor operation, a motor driver circuit must be included in the system. The driver acts as an amplifier and isolating buffer. Drivers can be implemented using electronic switches or mechanical relays. Protection components (such as diodes) are generally included in the motor drivers to prevent the damage of the transistors or the relays from the high back-currents that occurs during voltage transitions. There is different type of motor:

1. **DC Motor:**

   A direct current motor, sometimes known as a DC motor, is an electrical machine that converts electrical energy into mechanical energy by creating a magnetic field generated by direct current. A stator and an armature are the two main components of a direct current motor. The stator is the motor's stationary component, while the armature revolves. The rotor is usually on the inside of the motor, whereas the stator is on the outside.

   The rotor has coil windings powered by DC current, while the stator has either permanent magnets or electromagnetic windings. When DC current is used to power the motor, a magnetic field is formed within the stator, attracting, and repelling the magnets on the rotor. This causes the rotor to begin spinning. The motor features a commutator to keep the rotor moving. When the rotor aligns with the magnetic field, it stops spinning, but the commutator reverses the current through the stator, so reversing the magnetic field. This allows the rotor to continue spinning as shown in the figure below.

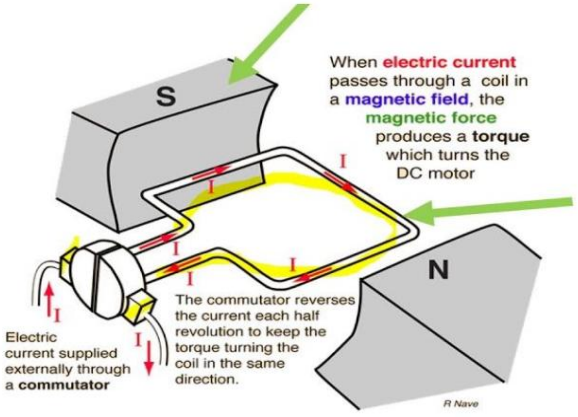


Figure 1: how DC motor works.

2. **Servo motors:**

   Unlike the DC motors, which rotate continuously if they are connected to a power source, the servo motor shown in Figure 2 is used to achieve partial, stable, and controlled rotations, for small amplitude but high precision operations: locking mechanisms, positioning of sensors, gestures, etc.



Figure 2: A servo motor.

The servomotors have three wires of different colors (depending on the manufacturer). The red color usually denotes Vcc (5V), while GND is usually black or brown. Besides the power wires, the command wire is usually yellow, orange, or white. Figure 3 shows several color schemes.

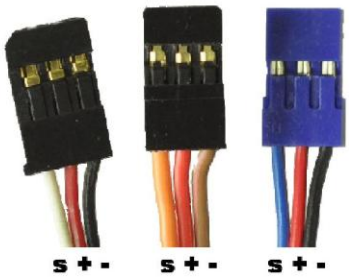


Figure 3: Color schemes used for servo motor wires.

The servomotor will not (usually) execute a complete rotation but will deviate from a neutral position with an angle controlled by the voltage applied to the signal pin. Using a PWM signal on this pin, we can control the motor's angle of rotation.

3. Stepper motors:

A stepper motor shown in Figure 4 is a unique type of brushless DC motor which position can be precisely controlled even without any feedback. The working principle of a stepper motor is based on magnetic fields. It has two main components, a stator, and a rotor. The rotor is usually a permanent magnet, and it is surrounded by some coils on the stator.

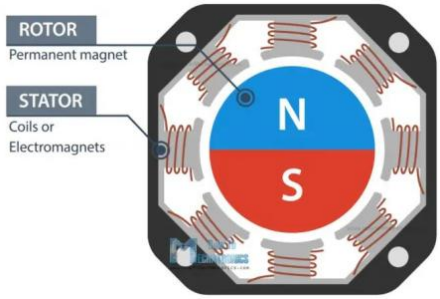


Figure 4: stepper motor components.

When we energize or let current flow through the coils, particular magnetic fields are generated in the stator that either attract or repel the rotor. By activating the coils, according to a predefined control sequence, we can achieve continuous motion of rotor, but also, we can make it stop at any position. Figure 5 shows different states for the stepper motor.



Figure 5: stepper motor works.

So, that is why these motors are called stepper motors, they move in discrete steps. By increasing the number of magnetic poles on the rotor, we can increase the number of possible stopping positions, thus increase the resolution or the precision of the motor. Here is a comprehensive overview of stepper motors:

Types of Steppers Motors:

1. Permanent Magnet (PM) Stepper Motor:
   This type has a permanent magnet rotor and multiple toothed stator poles. It is known for its simple construction and low cost.

2. Variable Reluctance (VR) Stepper Motor:
   VR stepper motors have a toothed rotor and a stator with salient poles. The rotor aligns itself with the stator poles due to magnetic reluctance, and this type is less common nowadays.

3. Hybrid Stepper Motor:
   A combination of permanent magnet and variable reluctance designs, the hybrid stepper motor offers improved performance and accuracy. It is widely used in various applications.

Stepper Motor Driving Techniques:

There are four different driving techniques for a stepper motor:

In **wave mode**, only one phase at a time is energized (see Figure 6). For simplicity, we will say that the current is flowing in a positive direction if it is going from the + lead to the - lead of a phase (e.g., from A+ to A-); otherwise, the direction is negative. Starting from the left, the current is flowing only in phase A in the positive direction and the rotor, represented by a magnet, is aligned with the magnetic field generated by it. In the next step, it flows only in phase B in the positive direction, and the rotor spins 90° clockwise to align with the magnetic field generated by phase B. Later, phase A is energized again, but the current flows in the negative direction, and the rotor spins again by 90°. In the last step, the current flows negatively in phase B and the rotor spins again by 90°.

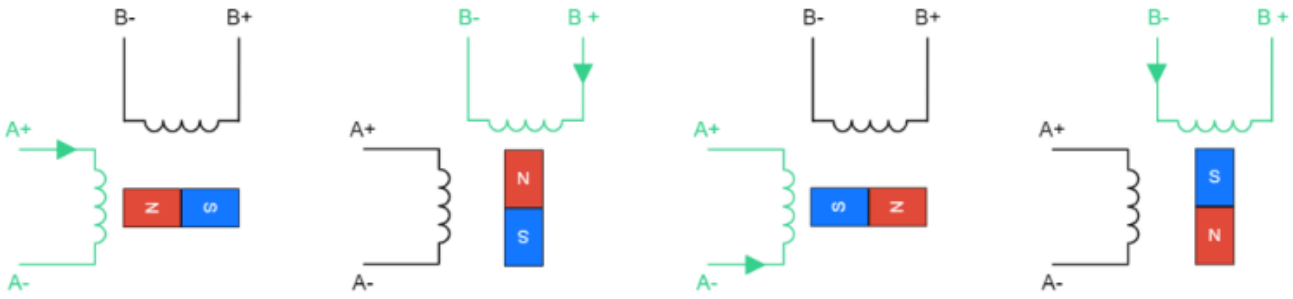


Figure 6: wave mode steps.

In **full-step mode**, two phases are always energized at the same time. Figure 7 shows the different steps of this driving mode. The steps are like the wave mode ones, the most significant difference being that with this mode, the motor can produce a higher torque since more current is flowing in the motor and a stronger magnetic field is generated.

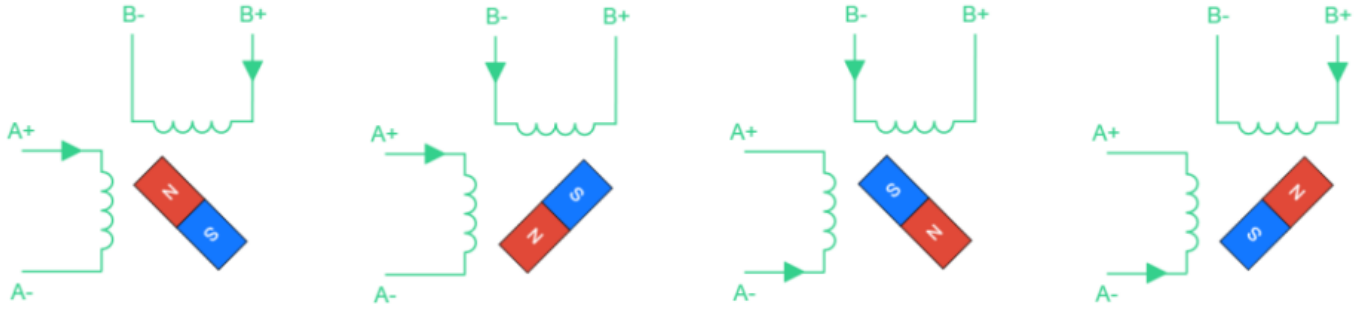


Figure 7: Full-Step Mode Steps.

Half-step mode is a combination of wave and full-step modes (see Figure 8). Using this combination allows for the step size to be reduced by half (in this case, 45° instead of 90°). The only drawback is that the torque produced by the motor is not constant, since it is higher when both phases are energized, and weaker when only one phase is energized.

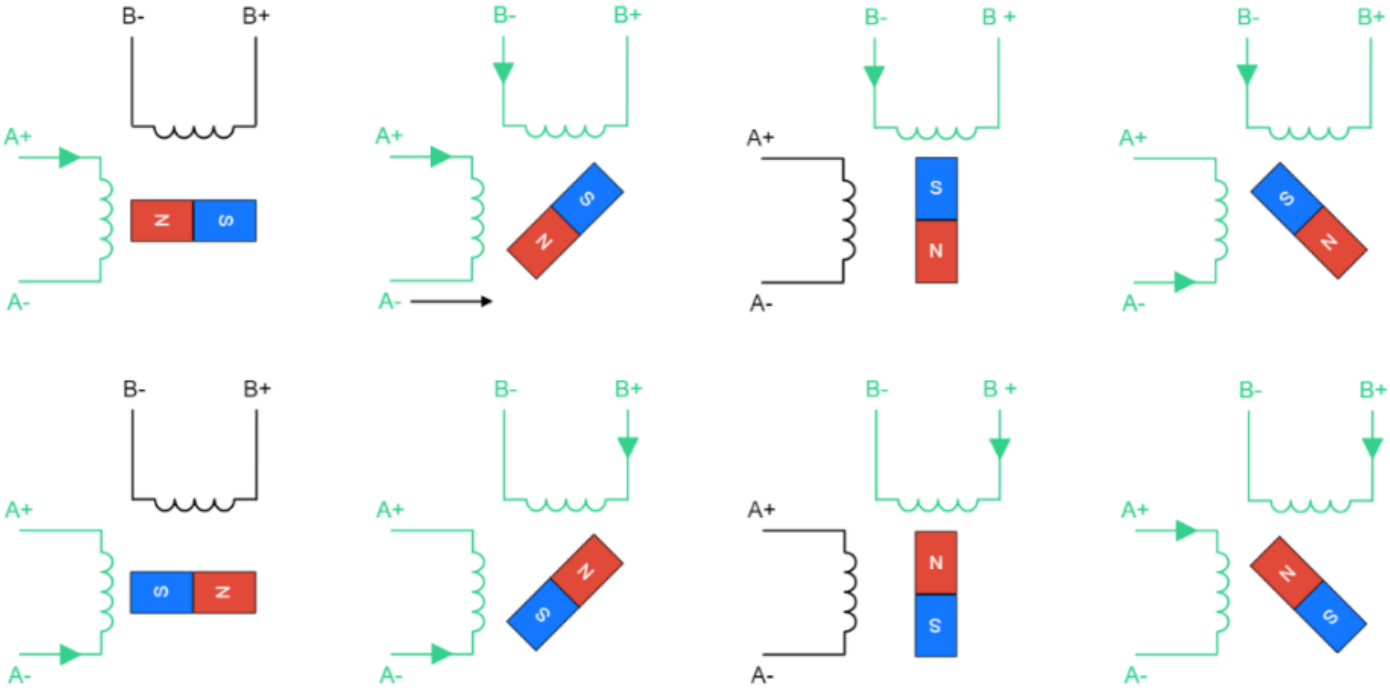


Figure 8: Half-Step Mode Steps.

4.4.2 PWM:

PWM, or pulse width modulation is a technique, which allows us to adjust the average value of the voltage that is going to the electronic device by turning on and off the power at a fast rate. The average voltage depends on the duty cycle, or the amount of time the signal is ON versus the amount of time it is OFF in a single period. PWM is a popular method for controlling DC motors with a microcontroller. The modulator voltage determines the speed of the electric motor. The higher the voltage, the faster an electric motor rotates. PWM is an effective method for controlling analog circuits in motor drives using digital outputs from a microcontroller. Duty cycle is usually stated in percentage, and it can be expressed using the equation:

$$\% \text{ Duty cycle} = \frac{\text{TON}}{\text{TON} + \text{TOFF}} * 100\%$$

Where TON is the time for which the square wave is high, and TOFF is the time for which the square wave is low. The block diagram of a typical PWM power controller scheme is shown below.

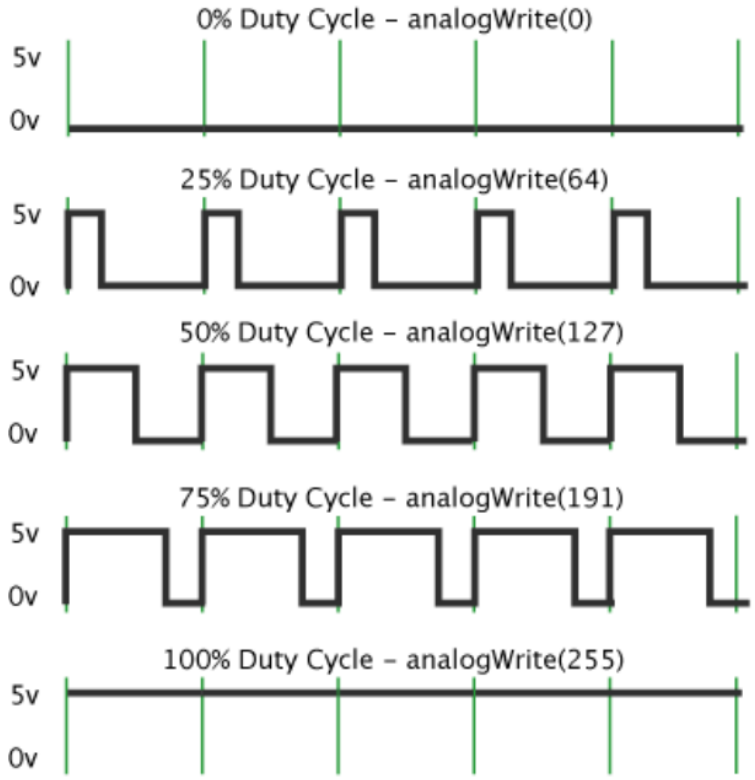


Figure 9: block diagram of a typical PWM power controller.

Note Speed is controlled by using the analogWrite function with values between 0-255 to the enabled motor on the connection pin. If your input signals, (sensor values) generate values out of this range, use the map function to scale or constrain these values.

4.4.3 H-Bridge:

H-bridge is a simple circuit that can be used as a motor driver Figure 10(a). In general, it contains four Transistor /MOSFET switches. When the proper switches (Q1 and Q4 in this case) are activated, the motor is energized and rotates in a given direction Figure 10(b). When the other pair (Q2 and Q3), the motor is energized in the reverse direction and thus the direction of rotation is reversed, Figure 10(c).

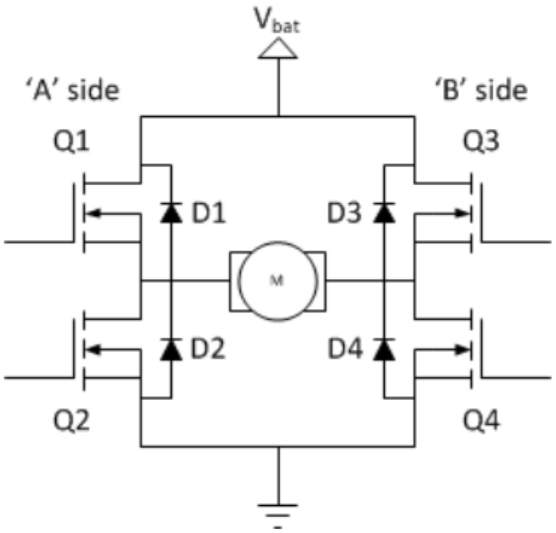


Figure 10 (a): H-bridge with MOSFETs.

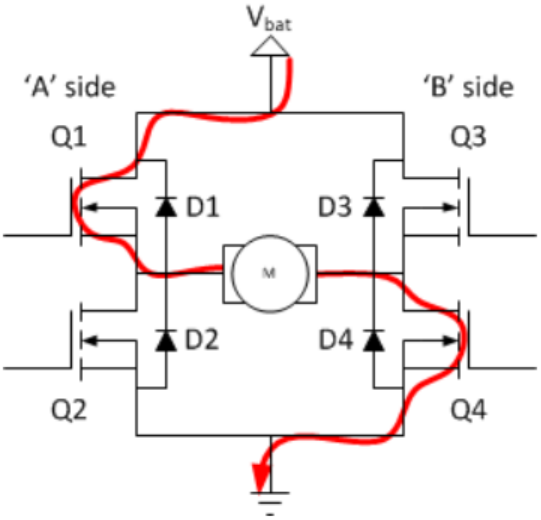


Figure 10(b): motor moves clockwise.

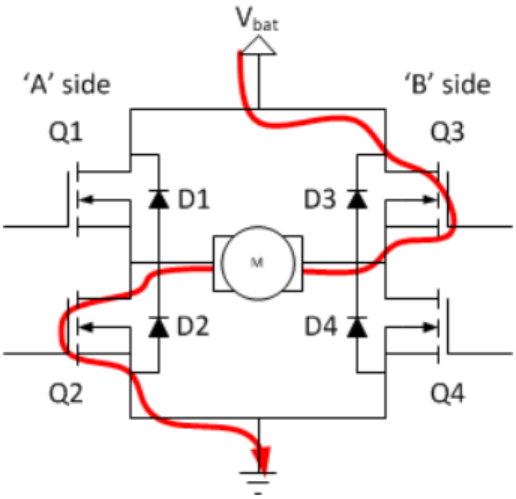


Figure 10(c): motor moves counterclockwise.

4.5 Procedure

Part1: DC motor Control by using a transistor Switch

Components Required:

- 1x Arduino Mega board
- 1x PN2222 Transistor
- 1x Small 6V DC Motor
- 1x 1N4001 diode
- 1x 270 Ω Resistor

Example1:

1. Build the system illustrated in Figure 11.
2. Edit, build and generate the binary file of program.
3. Configure the communication link to upload the code.
4. Upload and run the on the Arduino.

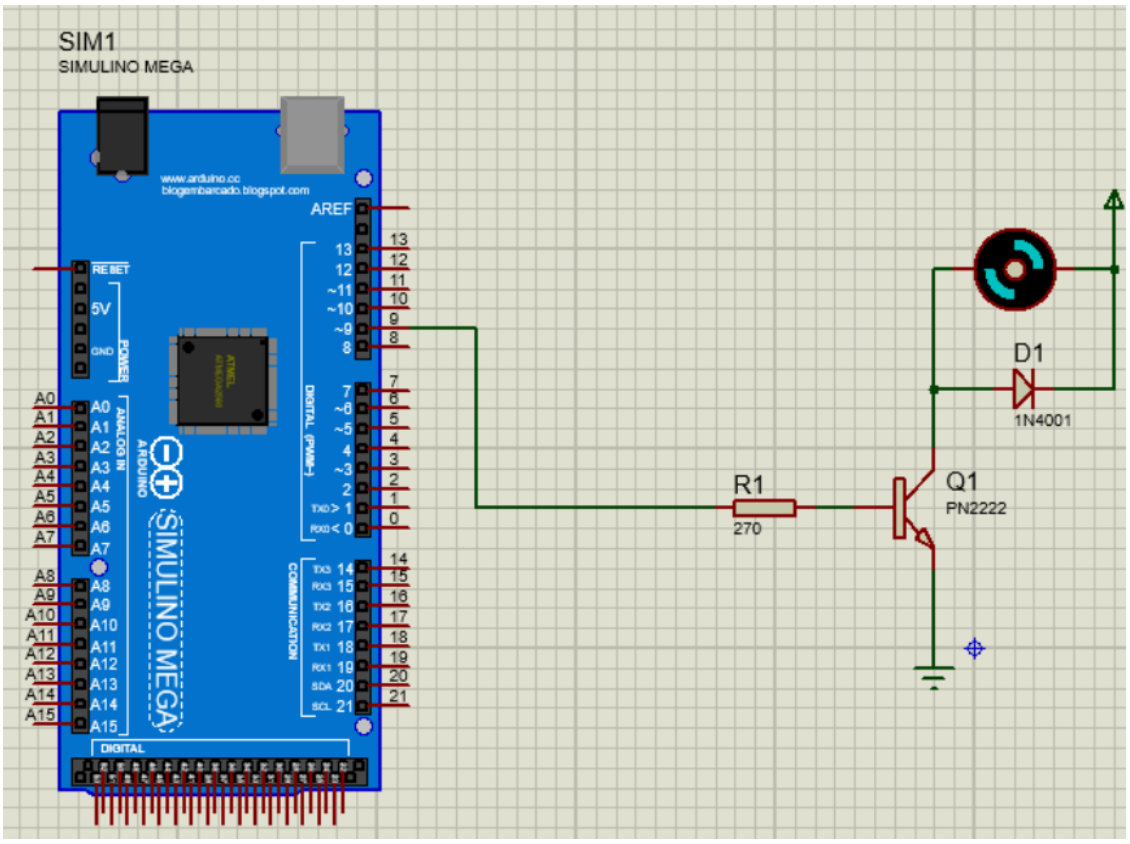


Figure 11: Example1 circuit.

Arduino Code (Example1)

```
int motorPin = 9; //Set pin 9 as motor control pinvoid setup() {
pinMode(motorPin, OUTPUT); //Set Motor control pin as output Pin
Serial.begin(9600); //Start Serial Communications
}
```

```
void loop() {
analogWrite(motorPin, 64); //Write the 'speed' value to motor control pin
to make it work with duty cycle 25%
delay(2000); //Wait for 2 sec
analogWrite(motorPin, 0); //motor will stop
delay(2000); //Wait for 2 sec
}
```

Q: What data pin controls the switching on and off operation of the motor, where is it considered in the code?

Example2:

If the selected pin is the analog output pin. Write a program to enable the motor and control the motor speed using the value of the analog output (if you have a multi-meter, measure the output voltage at the different speeds).

Arduino Code (Example 2)

```
int motorPin = 9; //Set pin 9 as motor control pin
void setup() {
pinMode(motorPin, OUTPUT); //Set Motor control pin as
output Pin
Serial.begin(9600); //Start Serial Communications
}
void loop() {
analogWrite(motorPin, 255); //Write the 'speed' value to motor
control pin to make it work with duty cycle 100%
delay(5000); //Wait for 5 sec
analogWrite(motorPin, 127); //Write the 'speed' value to motor
control pin to make it work with duty cycle 50%
delay(5000); //Wait for 5 sec
analogWrite(motorPin, 64); //Write the 'speed' value to motor
control pin to make it work with duty cycle 25%delay(5000); //Wait for 2 sec
analogWrite(motorPin, 0); //motor will stop
delay(2000); //Wait for 2 sec
}
```

Now analyze the programs statements by answering the following questions.

- What does the first line in loop function do?
- What is the motor speed in line three in loop function?
- What is the duty cycle of the motor driving signal according to line five in loop function?
- What does the delay function do in the code?
- Is the period of the driving signal important in the motor control or it is enough to set the duty cycle?
- Considering line seven in loop function, Should the motor speed increase or decrease?
- Which program line gives higher motor speed? Why?

Task1: Changing the motor's state:

Using the previous circuit, make the motor work at duty cycle 30% then after 3 second make it stop.

Task 2: Changing the motor's speed:

Using delay function Using the previous circuit, make the motor work at speed 15 then after 2 second the speed will increase another 15 this process will repeat until the motor speed will be 165 then it will be wait 3 second then increase speed by 30 this process will repeat until the motor work in maximum speed 255 the motor stops for 4 second then return do the same thing.

Task 3: communication through the com port to set certain rotate and stop actions

a. Using the previous circuit, Write Arduino code that asks the user to enter a string "on" to make motor work and "off to make motor Controls the speed of the motor based on the string entered (Open the serial monitor to enter the speed of motor).

Arduino Code (Task3-a)

```
const int motorPin = 9; // Connect this pin to the motor driver
int motorSpeed = 0; // Initialize motor speedString userInput = ""; // String to store user input
void setup() {
pinMode(motorPin, OUTPUT);
Serial.begin(9600); // Initialize the serial communication
Serial.println("Enter 'on' to start the motor or 'off' to stop it.");
}
void loop() {
// Check if there is any serial data available
if (Serial.available() > 0) {
// Read the user's input as a string
userInput = Serial.readStringUntil('\n');
// Remove leading and trailing whitespace
userInput.trim();
// Check the user's input
if (userInput == "on") {
digitalWrite(motorPin, HIGH);
```

```
Serial.println("Motor set ON ");
} else if (userInput == "off") {
// Motor off
digitalWrite(motorPin, LOW);
Serial.println("Motor is OFF.");
} else {
Serial.println("Invalid command. Enter 'on' or 'off'.");
}
}
}
```

b. Using the previous circuit, Write Arduino code that asks the user to enter a value between 0 and 255 and returns an error if not in this range. Controls the speed of the motor based on the value entered (Open the serial monitor to enter the speed of motor).

Arduino Code (Task3-b)

```
(initialization) //Set pin 9 as motor control pin
void setup() {
//Set Motor control pin as output Pin
//Start Serial Communications
Serial.println ("Enter a Value between 0-255");
}
void loop() {
(if condition) //If something is entered into the Serial Monitor
{
//Take the first Integer entered and set the speed variable to the
entered value
//Write the 'speed' value to motor control pin} }
```

Now analyze the programs statements by answering the following questions.

- What does the function Serial.available() do?
- What does Serial.parseInt () do?
- If user enter speed >255 what will happen?

Part2: DC-Motor by using H-bridge Motor Driver [L298]

The L298N is a dual H-Bridge motor driver which allows speed and direction control of two DC motors at the same time. The module can drive DC motors that have voltages between 5 and 35V, with a peak current up to 2A.

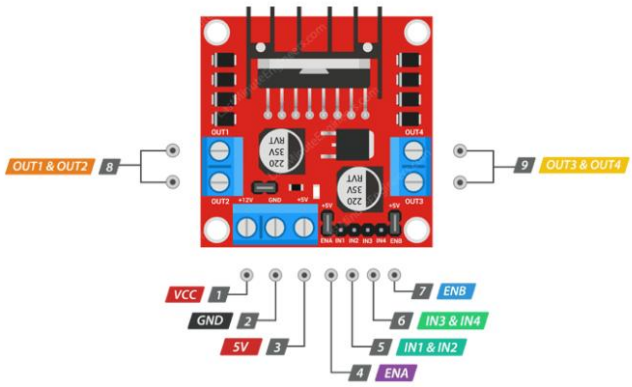


Figure 12: H-Bridge motor driver L298N.

Read about H-bridge L298N then answer the following questions:

a. What H-bridge contained?
b. how we can control what the output will have used h-bridge?

Components Required:

- 1x Arduino Mega board
- 1x L298N H-bridge
- 2x Small 6V DC Motor

Example 3:

- Build the system illustrated in Figure 13.
- Configure the communication link to upload the code.
- Edit, build, and upload the following code.

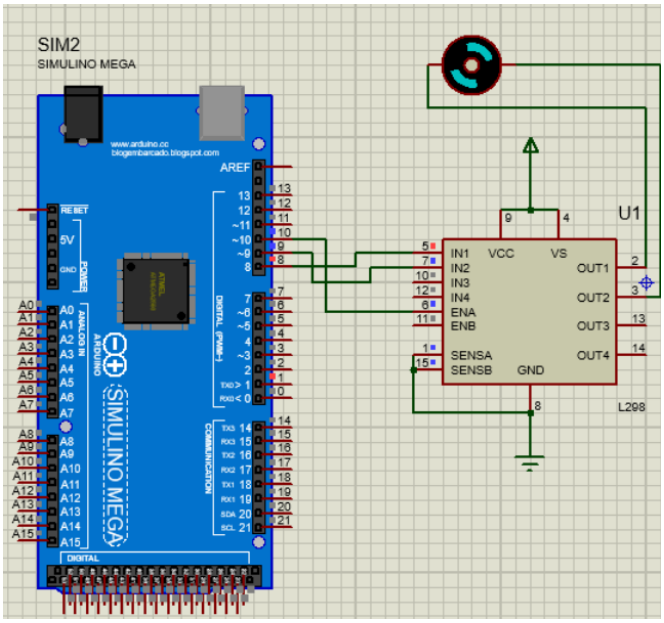


Figure 13: Example 4 circuit.

Arduino Code (Example 3)

```
int EnA= 10;
int in1=8;
int in2=9;
void setup () {pinMode(EnA, OUTPUT);
pinMode(in1, OUTPUT);
pinMode(in2, OUTPUT);
Serial.begin(9600);
}
void loop() {
analogWrite(EnA, 255);
digitalWrite(in1, LOW);
digitalWrite(in2, HIGH);
}
```

- What data pin controls the switching on and off operation of the motor, where is it considered in the code.
- How we can change the speed of motor.
- How we can change the direction of motor.

Task1: Changing the motor's speed and direction:

Write a program to control the motor operation and direction (consider PWM). Using the previous circuit, make the motor work at speed 200 and the motor move counter clockwise then after 2 second the motor will move in opposite direction then it will be wait 3 second then change the speed to 100 in the same direction the after 1 second the direction will change then repeat the process.

Arduino Code (part 2 Task 1)

```
int motorEnableA = 10;
int motorInput1 =8;
int motorInput2 =9;
int motorSpeed = 0; // Motor speed variable
bool motorDirection = HIGH; // Motor direction (HIGH =
counterclockwise, LOW = clockwise)
bool motorOn = true; // Motor state variable
void setup () {
Serial.begin(9600); // Initialize serial communication
pinMode(motorEnableA, OUTPUT); // Set motor A enable pin as output
```

```
pinMode(motorInput1, OUTPUT); // Set motor A input 1 as output
pinMode(motorInput2, OUTPUT); // Set motor A input 2 as output
analogWrite(motorEnableA, 0); // Turn off motor initially
digitalWrite(motorInput1, LOW); // Set motor A input 1 to LOW
digitalWrite(motorInput2, LOW); // Set motor A input 2 to LOW
}void loop() {
digitalWrite(motorInput1, motorDirection);
digitalWrite(motorInput2, !motorDirection);
analogWrite(motorEnableA, 200); // Set motor speed
// Change direction after 2 seconds
delay(2000);
digitalWrite(motorInput1, !motorDirection);
digitalWrite(motorInput2, motorDirection);
delay(3000);
analogWrite(motorEnableA, 100); // Set motor speed
delay(1000);
digitalWrite(motorInput1, motorDirection);
digitalWrite(motorInput2, !motorDirection);
}
```

Part3: stepper motor

Components Required:

- 1x Arduino Mega board
- 1x small bipolar stepper Motor as shown in the image given below
- ULN2003AN driving IC



Figure 14: bipolar stepper Motor.

- Build the system illustrated in Figure 15.
- Configure the communication link to upload the code.
- Edit, build, and upload the following code.

Task1: Changing the motor's state:

Using the below circuit, make the motor on for 0.5 second then after make it stop for 0.5 second.

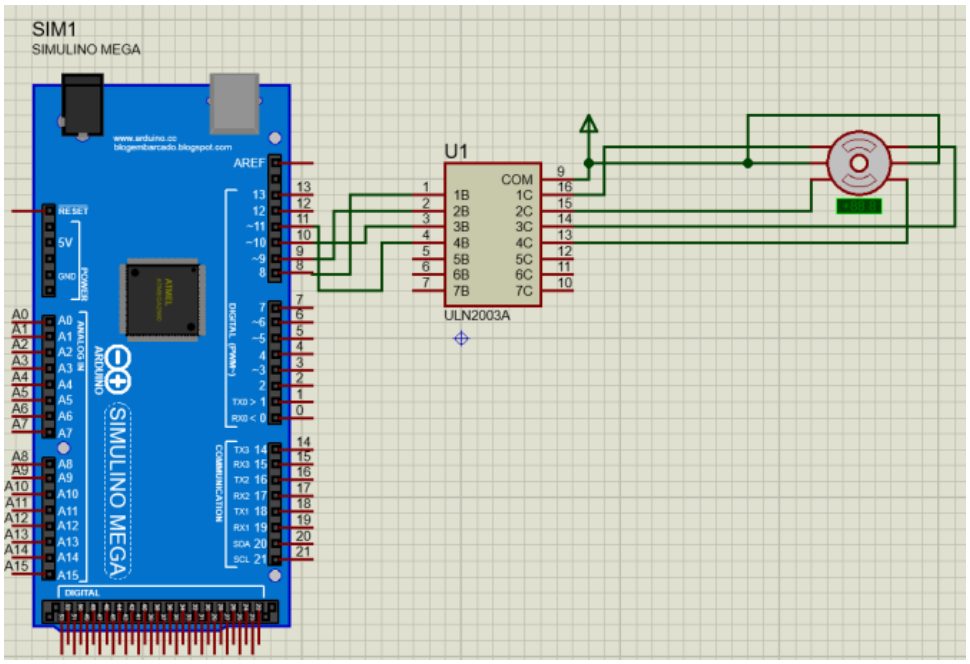


Figure 15: example circuit using proteus.

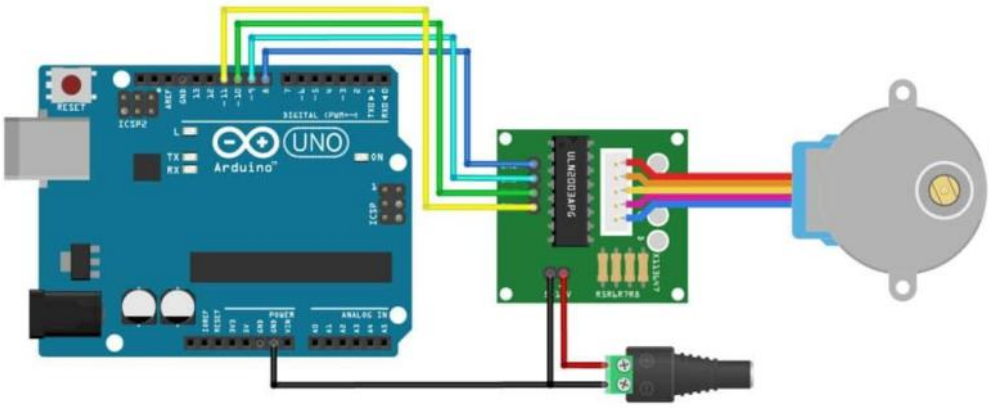


Figure 16: example circuit hardware.

Arduino Code (Part3 Task1)

```
// Include the Arduino Stepper.h library:
#include <Stepper.h>
const int stepsPerRevolution = 2048;// Define number of steps per
rotation:
// Wiring:
// Pin 8 to IN1 on the ULN2003 driver
// Pin 9 to IN2 on the ULN2003 driver
// Pin 10 to IN3 on the ULN2003 driver
// Pin 11 to IN4 on the ULN2003 driver
Stepper myStepper = Stepper(stepsPerRevolution, 8, 10, 9, 11);// Create
stepper object called 'myStepper', note the pin order:
void setup() {
myStepper.setSpeed(5);// Set the speed to 5 rpm:
Serial.begin(9600);// Begin Serial communication at a baud rate of
9600:
}
void loop() {
// Step one revolution in one direction:
Serial.println("clockwise");
myStepper.step(stepsPerRevolution);
delay(500);
// Step one revolution in the other direction:
```

```
Serial.println("counterclockwise");
myStepper.step(-stepsPerRevolution);
delay(500);
}
```

Task 2: Changing the motor's speed (Using delay function):

Using the previous circuit, make the motor work at speed 5 then wait 2 second the direction will change wait 3 second then change speed to 3 wait 4 second the motor stops for 2 second then return do the same thing.

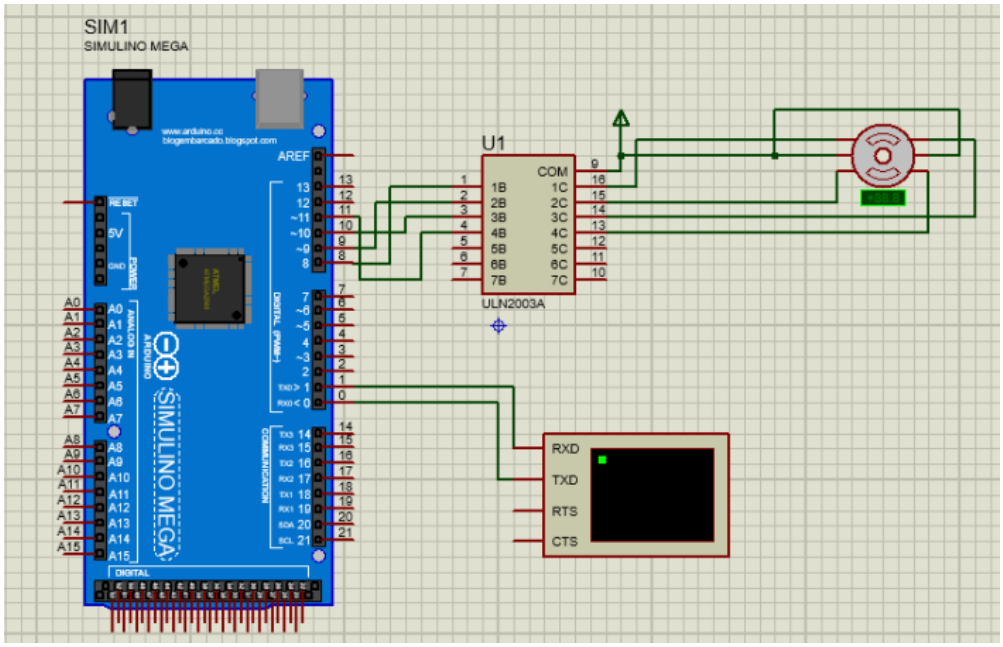


Figure 20: part 3 Task 2 circuit.



Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

Microprocessor and Embedded Systems Lab (ENCS5101)

Experiment No. 5 - Assembly Programming using Arduino

5.1 Objectives

1. Learn about ATmega2560 architecture, memory, and addressing modes.
2. Learn about ports and their mapping on Arduino Mega.
3. Learn about AVR assembly language and its instruction set.
4. Learn to program Arduino Mega ports and access control, status and data registers and ports using AVR assembly language.

5.2 Equipment's & Tools

- Proteus v8.13
- Arduino Mega
- Arduino IDE v2.1.1

5.3 Prelab

Read the theory section to get the required knowledge and be ready for doing the experiment.

5.4 Theory

5.4.1 ATmega2560 Architecture:

The ATmega2560 is a high-performance, low-power CMOS 8-bit microcontroller based on the Advanced Virtual RISC (AVR) enhanced Reduced Instruction Set Computing (RISC) architecture. Fig. 1 displays a simplified block diagram of the ATmega2560 microcontroller that highlights its key parts.

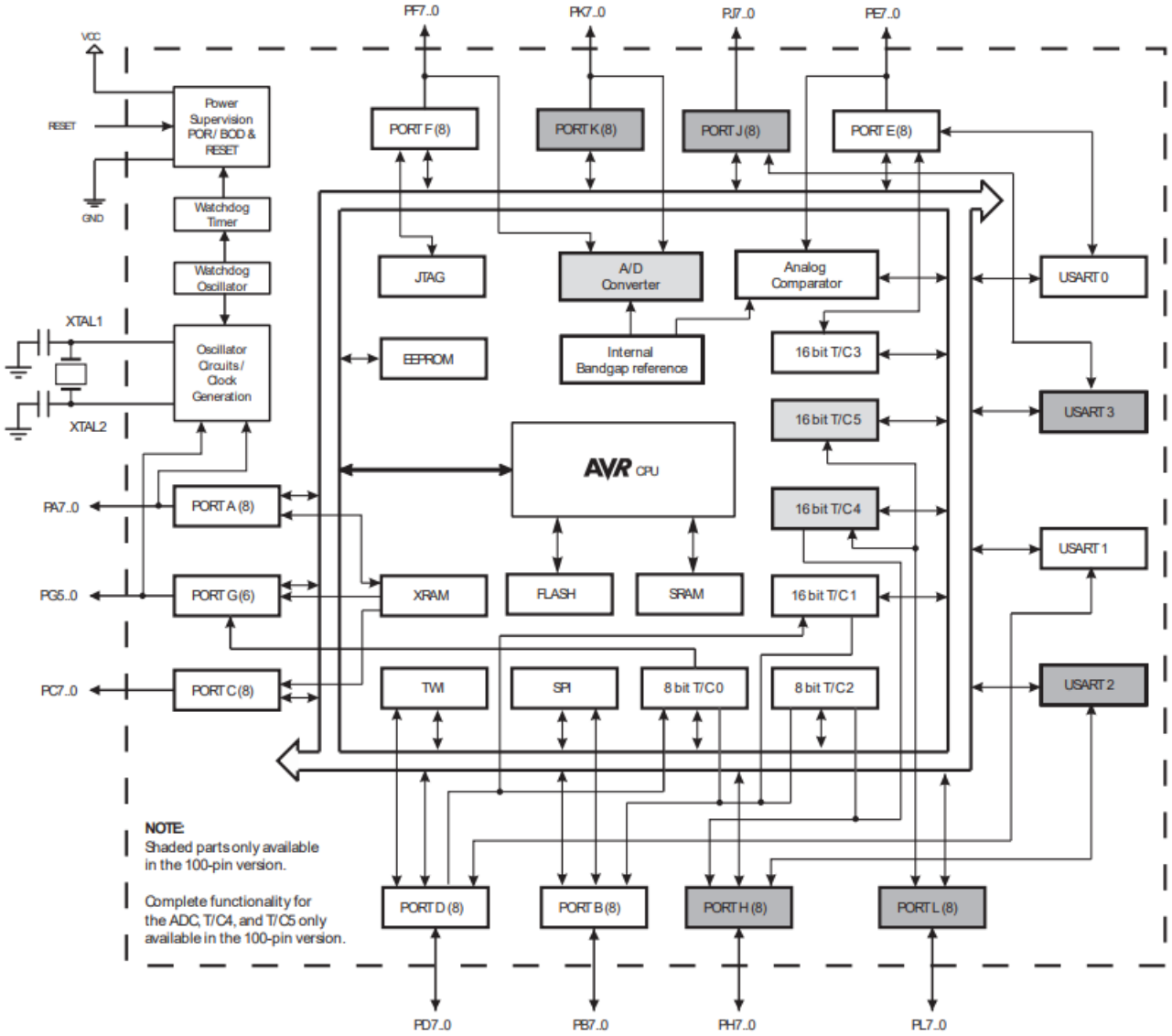


Fig. 1: ATmega2560 Block Diagram (ATmega2560 Datasheet)

5.4.1.1 ATmega2560 Memory:

Arduino Mega's ATmega2560 employs a modified Harvard architecture which means that program memory and data memory are in two different address spaces. The ATmega2560 has 256 KB of Flash memory, is used to store programs, and has 8 KB taken up by the bootloader. Additionally, it contains 8 KB of SRAM for data storage. Moreover, it features 4 KB of EEPROM to store any data that programs may wish to retain after power is cycled.

SRAM Data Memory: The common data bus can be used to access data. A supplementary In/Out bus provides quick, direct access to select locations. **Data memory** consists of:

- Registers
- I/O Memory
- Extended I/O Memory (Device dependent)
- Internal SRAM

Register space: Consists of 32 general-purpose working 8-bit registers (**R0-R31**).
I/O Memory: Contains addressable space for peripheral functions, such as control registers and other I/O functions.
Extended I/O Memory: Some AVR microcontrollers with more peripherals need more space than the I/O memory can address. So some of the SRAM is used as Extended I/O memory to handle the extra peripherals control registers and other I/O functions.
Internal SRAM (Data Memory): is used for temporarily storing intermediate results and variables within a software application.

In/Out Data Bus:
This data bus has a direct access to the 64-byte I/O Memory section (not Extended) using a 0x00 to 0x3F address. This memory can also be accessed by the standard data bus using a 0x20 address offset in the access command. The 64 I/O registers can be accessed over the I/O bus with **in** and **out** instructions using an I/O address that is lower by 0x20 than the regular memory address.

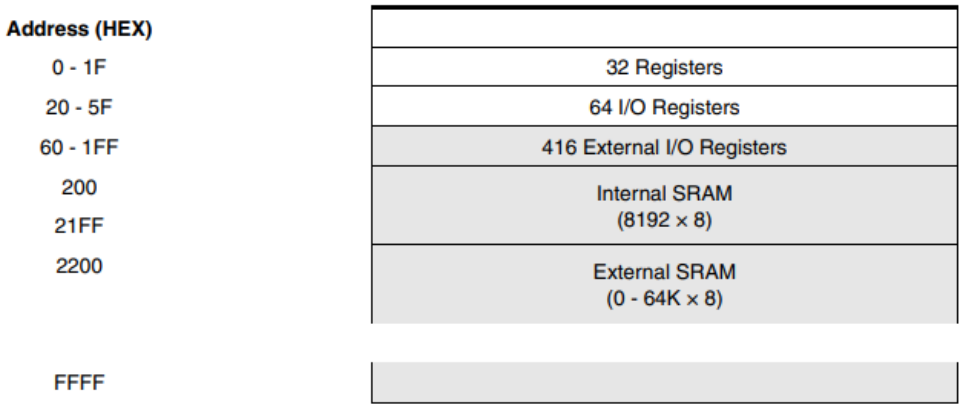


Fig. 2: Data Memory (ATmega2560 Datasheet)

5.4.1.2 Addressing Modes:

Table 1. ATmega328P Addressing Modes

Addressing Mode	Definition	Example	Example Explanation
Direct register addressing, single register	The operand is contained in one register, and the result of applying the instruction is stored in the same register	`neg r0`	replaces the contents of register r0 with its two's complement
Direct register addressing, two registers	The operands are contained in two registers. If the instruction has a result, then it is stored in the first register.	`sub r1, r2`	subtracts the contents of r2 from r1 and places the result in r1
Immediate value	One operand is an immediate value that is part of the instruction. The other one is in a register (or register pair). Depending on the instruction, different ranges of values are permitted.	`adiw r30, 5`	adds the value 5 (0-63 is allowed) to the register pair r30/r31
I/O direct addressing	A byte from a register is read from one I/O register or written to it	`in r0, 0x05`	loads one byte from I/O register 0x05 to register r0
Direct data addressing	One operand is a byte in data memory addressed by a 16-bit address given in the instruction	`lds r2, 0x05FF`	loads the byte contained in the byte addressed by 0x05FF into r2
Indirect data addressing	One operand is a byte in data memory addressed by the X-, Y- or Z-register	`st X, r5`	stores the contents of r5 to the memory addressed by the X-register
Indirect data addressing with displacement	One operand is a byte in data memory addressed by the Y- or Z-register and a constant displacement of 0 up to 63	`ldd r0, Y+5`	loads from the address pointed at by the Y register plus 5 into register r0
Indirect data addressing with pre-decrement	One operand is a byte in data memory addressed by the X-, Y– or Z-register, and the address register is decremented before access	`st -Y, r1`	stores the byte contained in register r1 to the location which is addressed after the Y register is decremented
Indirect data addressing with post-increment	One operand is a byte in data memory addressed by the X-, Y– or Z-register, and the address register is incremented after access	`ld r0, X+`	loads r0 with the byte pointed at by the X register, which is incremented afterwards

5.4.1.3 ATmega2560 AVR CPU Core:

The 32, 8-bit general purpose working registers are directly connected to the ALU. The AVR ALU operates in direct connection with all the 32 general purpose working registers. Within a single clock cycle, arithmetic operations between general purpose registers or between a register and an immediate are executed. The ALU operations are divided into the three main categories **arithmetic**, **logical**, and **bit-functions**. After an arithmetic operation, the status register (SREG) is updated to reflect information about the result of the operation. Fig. 3 shows the AVR CPU block diagram.

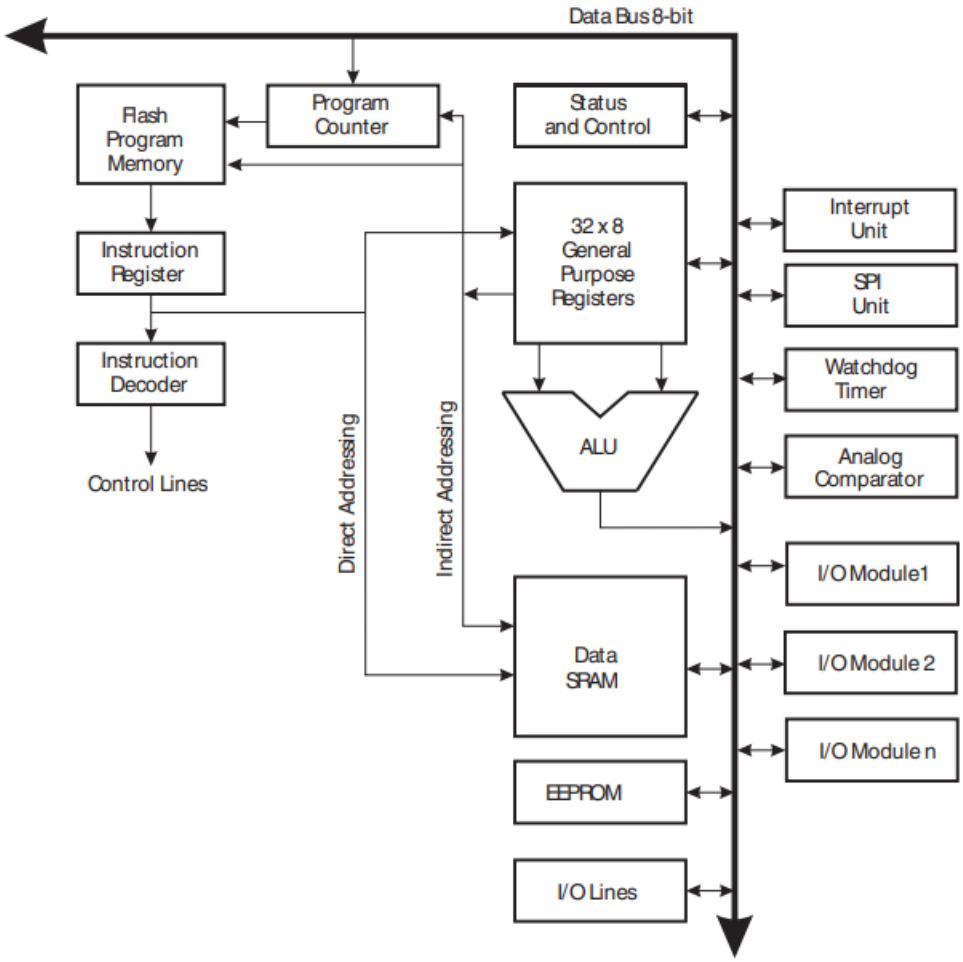


Fig. 3: AVR CPU Block Diagram (ATmega2560 Datasheet)

While all registers are 8-bit registers, the 6-upper registers form three pairs of 16-bit registers that can be used for indirect addressing. **R26/R27** is also called **X**-register, **R28/R29 Y**-register, and **R30/R31 Z**-register. AVR MCUs are little-endian systems, meaning the byte with the lower address contains the least significant byte. So, if we want to store 0x0F12 in the **Z**-register, **R30** would contain 0x12, and **R31** would contain 0x0F.

5.4.1.3 ATmega2560 Ports & Pin Mapping:

AVR (Atmega2560) microcontroller which has 11 I/O ports (PORTA, PORTB, PORTC, PORTD, PORTE, PORTF, PORTG, PORTH, PORTJ, PORTK, and PORTK). Each of these ports typically consists of 8 pins, except for Port G, which has 6 pins.

Fig. 4 shows the Arduino Mega2560 Pinout for all digital IO pins and their mapping to the AVR Atmega2560 microcontroller.

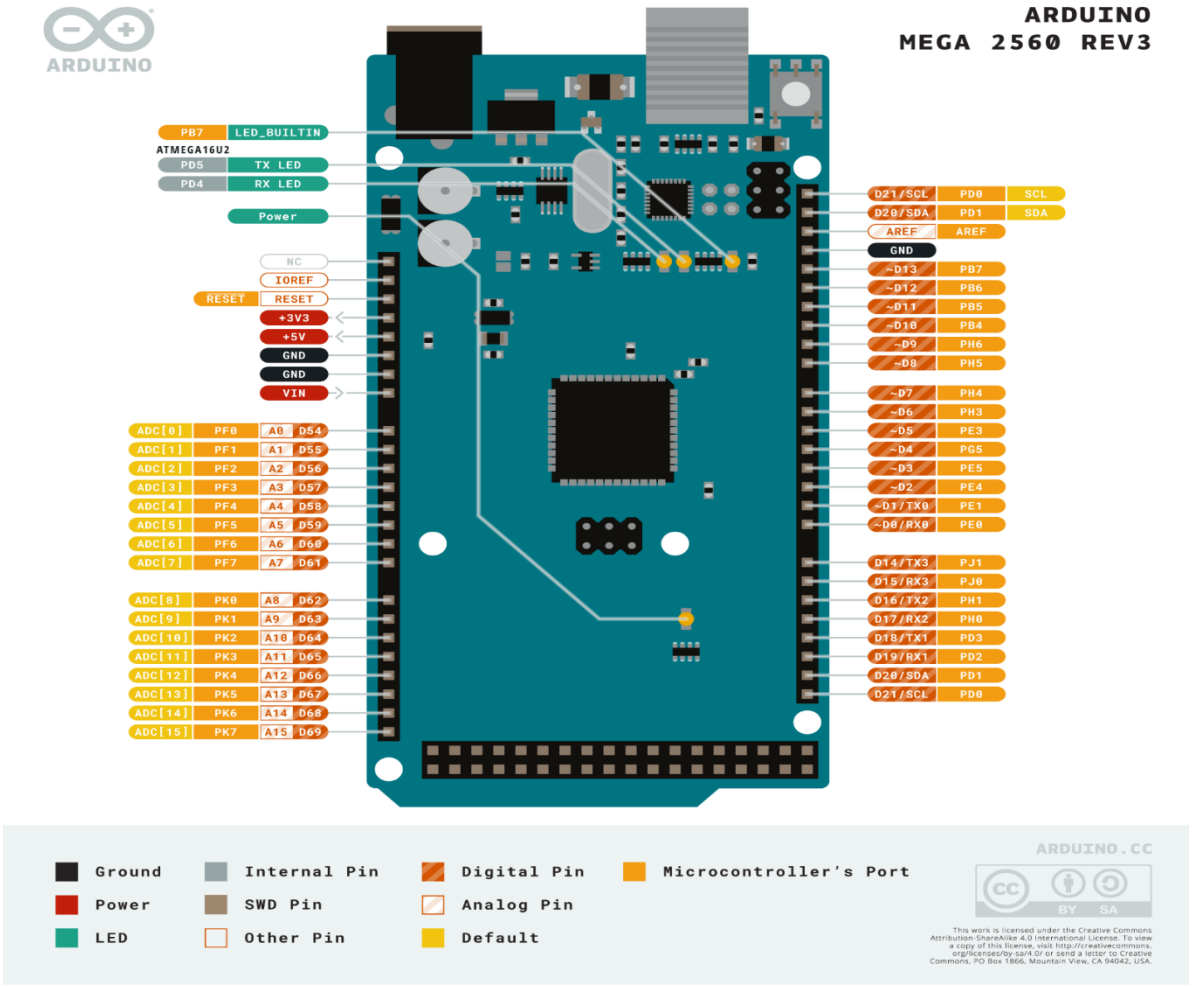


Fig. 4: Arduino Mega Pinout and their Mapping to the AVR Atmega2560 Microcontroller (Arduino Mega2560 Rev3 Documentation)

Check this: Arduino Mega 2560 PIN mapping table.

Each port consists of three registers to control its pins, which are:

1. **Data Direction Registers (DDRx)**: It is used to set the port/pin mode (data direction) whether a specific pin is going to be an output or input pin.

Name: DDRB
Offset: 0x24
Reset: 0x00
Property: When addressing as I/O Register: address offset is 0x04

Bit	7	6	5	4	3	2	1	0
	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Bits 7:0 – DDRBn: Port B Data Direction [n = 7:0]

Fig. 5: DDRB Register (Microchip Developer Help)

2. **Digital Output Registers (PORTx)**: It is used to set (write) the digital output state of the port/pin.

Name: PORTB
Offset: 0x25
Reset: 0x00
Property: When addressing as I/O Register: address offset is 0x05

Bit	7	6	5	4	3	2	1	0
	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Bits 7:0 – PORTBn: Port B Data [n = 0:7]

Fig. 6: PortB Register (Microchip Developer Help)

3. **Digital Input Registers (PINx)**: Which is the input buffer register that we can use to read the digital status of the port/pin.

Name: PINB
Offset: 0x23
Reset: N/A
Property: When addressing as I/O Register: address offset is 0x03

Bit	7	6	5	4	3	2	1	0
	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	x	x	x	x	x	x	x	x

Bits 7:0 – PINBn: Port B Input Pins Address [n = 7:0]

Fig. 7: PINB Register (Microchip Developer Help)

The Port Input Pins I/O location is read only, while the Data Register and the Data Direction Register are read/write. Each bit of every register affects a single pin (some bits of some ports might be unused) as follows:

- Setting a bit in a **Data Direction Register** would put its corresponding pin in output mode and clearing the same would put the pin in input mode.
- Setting a bit in a **Digital Output Register** would set its corresponding pin to a high value and clearing the bit would set the pin to a low value, this if the pin is in output mode. And if the pin is in input mode, setting/clearing the same bit enables/disables the built-in pull-up resistor on the pin.
- When a pin is in input mode, its corresponding **Digital Input** bit can be read to get the state of the pin.

ATmega2560 Registers Access:

Let's say we'd like to set the Arduino **pin 10** to be an output pin and directly write to it **HIGH** and **LOW**. Here is how to do it by port registers manipulation. First, we need to recall the Arduino-Atmega2560 pin mapping diagram that shown in Fig. 4. From the pin mapping diagram, we can see that Arduino **pin 10** maps to pin **PB4** (in **PORTB**) on the Atmega2560 microcontroller. Therefore, to set its mode to be an output pin.

We'll do the following:

```
DDRB = 0b00010000;
```

Or we can write it in HEX format like this:

```
DDRB = 0x10;
```

Both lines are perfectly the same and do the same function of setting the Arduino pin 10 (PB4) to be an output pin. This is equivalent to the functionality of the pinMode function:

```
pinMode(10, OUTPUT);
```

Now, in order to control the pin's digital state, we can use the **PORTB** register to directly write to the pin HIGH or LOW. Here is how it's done:

```
PORTB = 0b00010000;  // Sets The Pin10 (PB4) HIGH
PORTB = 0b00000000;  // Sets The Pin10 (PB4) LOW
```

This is equivalent to the functionality of the digitalWrite function:

```
digitalWrite(10, 1); // Sets The Pin10 (PB4) HIGH

digitalWrite(10, 0); // Sets The Pin10 (PB4) LOW
```

5.4.2 Introduction to Assembly Programming:

Assembly language is a low-level programming language that is specific to a particular computer architecture or microprocessor. It serves as a bridge between machine code (binary code that the CPU can directly execute) and high-level programming languages like C, C++, or Python.

Assembler

To convert assembly code into machine code, an assembler is required. The assembler translates human-readable assembly instructions into the binary format that the CPU understands.

Basic elements of assembly language are:

1. **Labels and symbols:** Are used for marking specific locations in your code or representing memory addresses. They help you create more readable and maintainable code. Labels are often used in conjunction with jump instructions (e.g., "JMP" or "BR") to control program flow.

2. **Instructions:** Assembly language instructions are the basic building blocks of a program. Each instruction represents a specific operation that the microcontroller can perform, such as arithmetic, data movement, control flow, and I/O operations. Examples of instructions include "MOV" for moving data between registers and memory, "ADD" for addition, and "JMP" for jump instructions to control program flow.
3. **Operands:** Are the instruction elements for the instruction is being executed. They are usually **registers** or **variables** or **constants**.
4. **Comments:** Comments in assembly language commonly start with a semicolon (;) and are used to provide explanations and documentation for your code. They are not executed by the microcontroller and help make your code more understandable.
5. **Directives:** Directives are special instructions for the assembler, not executed by the microcontroller, but used for specifying aspects of the assembly process. Common directives include ".ORG" (origin) to set the program memory location, ".DB" (define byte) to define data values, and ". EQU" (equation) to define constants.

Table. 3 shows the most common directives for programming the ATmega328P:

AVR Directives:

Table 3. AVR Directives

Directive	Description
.DEVICE	Defines the type of the target processor and the applicable set of instructions (illegal instructions for that type trigger an error message, syntax: .DEVICE AT90S8515)
.DEF	Defines a synonym for a register (e.g. .DEF MyReg = R16)
.EQU	Defines a symbol and sets its value (later changes of this value remain possible, syntax: .EQU test, 1234567)
.SET	Fixes the value of a symbol (later redefinition is not possible)
.INCLUDE	Includes a file and assembles its content, just like its content would be part of the calling file (typical e.g. including the header file: .INCLUDE "C:\avrtools\appnotes\8515def.inc")
.CSEG	Start of the code segment (all that follows is assembled to the code segment and will go to the program space)
.DB	Inserts one or more constant bytes in the code segment (could be numbers from 0..255, an ASCII-character like 'c', a string like 'abcde' or a combination like 1,2,3,'abc'. The number of inserted bytes must be even, otherwise an additional zero byte will be inserted by the assembler.)
.DW	Insert a binary word in the code segment (e.g. produces a table within the code)
.LISTMAC	Macros will be listed in the .LST-file. (Default is that macros are not listed)
.MACRO	Beginning of a macro (no code will be produced, call of the macro later produces code, syntax: .MACRO macroname parameters, calling by: macroname parameters)
.ENDMACRO	End of the macro
.ESEG	Assemble to the EEPROM-segment (the code produced will go to the EEPROM section, the code produces an .EEP-file)
.DSEG	Assemble to the data segment (here only BYTE directives and labels are valid, during assembly only the labels are used)

.BYTE	Reserves one or more bytes space in the data segment (only used to produce correct labels, does not insert any values!)
.ORG	Defines the address within the respective segment, where the assembler assembles to (e.g. .ORG 0x0000)
.LIST	Switches the listing to the .LST-file on (the assembled code will be listet in a readable text file .LST)
.NOLIST	Switches the output to the .LST-file off, suppresses listing.
.IF, .ELSE, .END IF	These conditional directives allow you to conditionally assemble portions of code based on specified conditions. For example, .IF defined(MY_FLAG) can be used to conditionally include code based on whether MY_FLAG is defined.
.REPT and .ENDR	These directives are used for repeating blocks of code a specified number of times. They are often used to generate repetitive code sequences.
.END	This directive indicates the end of the assembly code. It is not always required, but some assemblers may use it to mark the end of the source file.
.DATA and .CODE	These directives can be used to switch between data and code sections in the assembly code, allowing you to define data separately from program instructions.

AVR Instruction Set:

Instructions can be of different types based on what they accomplish. Some of them just move data around, others perform arithmetic. The AVR instructions can be classified into the following types of instructions:

- **Arithmetic and logic instructions:** These are instructions that perform logic and arithmetic operations. Almost all of them work on two registers, some of them take an immediate value. They all affect the status register, e.g., zero- and carry flags are set.
- **Branch instructions**: Branch instructions are all instructions that change the linear flow of execution. For example, **call** instruction belongs to this set and also the **ret** instruction, which returns from a subroutine call. These two are unconditional. There is also a large set of conditional branch instructions, such as **breq** (branch if equal), which only jumps to the specified program address if the Zero-flag is set.
- **Data transfer instructions:** Data transfer instructions move data around, e.g., from data memory to registers (**ld**) and vice versa (**st**). Then there are also two instructions that move data between registers (**mov** and **movw**), and instructions that manipulate the stack (**push** and **pop**). And there are, of course, the instructions mentioned above that read from I/O registers (**in**) and that write to them (**out**). All of them do not change any flags in the status register.
- **Bit manipulation instructions:** These are instructions that manipulate bits in one of the general registers, in the I/O registers, or in the status register. An example for the first kind of instructions is **rol**, which rotates the contents of a register to the left, shifting the carry flag into the least significant bit and moving the most significant bit into the carry flag. An example for the second kind of instructions is **sbi** that sets a particular bit in an I/O register to one. Finally, an example for the third type is **clc**, which clears the carry flag.
- **MCU control instructions:** The instructions that control the operation of the microcontroller, **nop** is an example of this type, which does nothing and takes one clock cycle to do so. This instruction is often used for delaying execution in order to time a loop.
- **Word instructions:** Orthogonally to the above classification, one can classify instructions along the dimension of how many bytes are operated on. There are actually only three instructions that operate on

words instead of on bytes. These are **movw**, to move a word from one register pair to another one, **adiw**, to add an immediate value (0-63) to a word in register pair, and **sbiw**, which subtracts an immediate value.

Following table shows the most common used instructions in AVR Assembly:

Table 4. AVR Instruction Set

Instruction	Syntax	Description	Action	Limitations
Arithmetic and logic instructions				
ADD	ADD Rx, Ry	Add register	Rx ← Rx + Ry	
ADC (Add with Carry)	ADDC Rx, Ry	Add register and carry	Rx ← Rx + Ry + C	
SUB	SUB Rx, Ry	Subtract register	Rx ← Rx - Ry	
SBC (Subtract with Carry)	SUBC Rx, Ry	Subtract register and carry	Rx ← Rx - Ry - C	
SUBI	SUBI Rx, K	Subtract constant	Rh ← Rh - K	R: 16 to 31
CP	CP Rx, Ry	Compare register	Rx - Ry	R: 16 to 31
CPI		Compare with constant	Rx - K	R: 16 to 31, K: 0 to 255
INC	INC Rx	Increase by one	Rx ← Rx + 1	
DEC	DEC Rx	Decrease by one	Rx ← Rx - 1	
AND	AND Rx, Ry	Binary AND register	Rx ← Rx AND Ry	
ANDI	ANDI Rx, K	Binary AND with constant	Rx ← Rx AND K	R: 16 to 31, K: 0 to 255
OR	OR Rx, Ry	Binary OR register	Rx ← Rx OR Ry	
ORI	ORI Rx, K	Binary OR with constant	Rh ← Rh OR K	R: 16 to 31, K: 0 to 255
EOR	EOR Rx, Ry	Exclusive-OR	Rx ← Rx XOR Ry	
COM	COM Rx	One's complement	Rx ← 255 - Rx	
NEG	NEG Rx	Two's complement	Rx ← 256 - Rx	
MUL	MUL Ry1, Ry2	Multiply 8 bits	R1:R0 ← Ry1 * Ry2	
MULS	MULS Ry1, Ry2	Multiply signed	R1:R0 ← Ry1 * Ry2	Ry1,Ry2: 16 to 31
Jump Instructions				
JMP	JMP label	Direct Jump	(PC) ← K	K: 0 to 65535
CALL	CALL label	Call Subroutine	(Stack) ← (PC), (PC) ← K	
RET	RET	Return from Subroutine	(PC) ← (Stack)	
RJMP	RJMP offset	Relative Jump	(PC) ← (PC) +/- K	K: -2048 to 2047
** BRxx	BRxx label	Branch if xx Condition Met	(PC) ← (PC) +/- K	K: -63 to + 64

Data copy and load instructions				
MOV	MOV Rd, Copy to SRAMRr	Copy register	Rx ← Ry	
LDI	LDI Rx, K	Load constant	Rh ← K	R: 16 to 31, K: 0 to 255
LDS	LDS Rx, data_address	Load Direct from SRAM	Rx ← (data_address)	R: 16 to 31
LD	LD Rx, (X or Y or Z)	Load Indirect from SRAM	Rx ← (X or Y or Z)	
LDD	LDD Rd, (X or Y or Z) + K	Load Indirect with Displacement	Rx ← (X or Y or Z +K)	K: 0 to 63
STS	ST data_address, Rr	Store Direct to SRAM	(data_address) ← Rx	
ST	ST (X or Y or Z), Rx	Store Indirect to SRAM	(X or Y or Z) ← Rx	
STD	ST (X or Y or Z) + K, Rx	Load Indirect with Displacement	(X or Y or Z +K) ← Rx	K: 0 to 63
IN	IN Rx, io_address	Copy port byte	Rx ← P	P: 0 to 63
OUT	OUT io_address, Rx	Copy byte to port	P ← Rx	P: 0 to 63
PUSH	PUSH Rx	Copy to stack and decrement (SP)	(Stack) ← Rx, SP = SP - 1	
POP	POP Rx	Copy from stack and increment (SP)	Rx ← (Stack), SP = SP + 1	
Bit Operations				
LSL	LSL Rx	Logical shift left	Rx ← Rx * 2	0 ≤ x ≤ 31
LSR	LSR Rx	Logical shift right	Rx ← Rx / 2	0 ≤ x ≤ 31
ROL	ROL Rx	Binary rotate left with C	Rx ← Rx * 2 with Bit 0 = C/C = Bit 7	0 ≤ x ≤ 31
ROR	ROR Rx	Binary rotate right with C	Rx ← Rx / 2 with Bit 7 = C/C = Bit 0	0 ≤ x ≤ 31
SWAP	SWAP Rx	Exchange upper and lower nibble	Rx ← (7:4) ↔ (3:0)	0 ≤ x ≤ 31
SBI	SBI PL,B	Set bit in port	PL ← PL OR (1<<B)	PL: 0 to 31, B: 0 to 7
CBI	CBI PL,B	Clear bit in port	PL ← PL AND (255-(1<<B))	PL: 0 to 31, B: 0 to 7
Word instructions				
ADIW	ADIW Rd+1:Rd, K	Add constant to register pair	Rd+1:Rd ← Rd+1:Rd + K	d ∈ {24,26,28,30}, 0 ≤ K ≤ 63
SBIW	SBIW Rd+1:Rd, K	Subtract constant from register pair	Rd+1:Rd ← Rd+1:Rd - K	d ∈ {24,26,28,30}, 0 ≤ K ≤ 63
MOVW	MOVW Rd+1:Rd, Rr+1:Rr	Copy register pair	Rd+1:Rd ← Rr+1:Rr	d ∈ {0,2,...,30}, r ∈ {0,2,...,30}

MCU control instructions				
BREAK	BREAK	Stop execution, control to debugger		
NOP	NOP	Do nothing		
SLEEP	SLEEP	Sleep		

** xx may be one of these conditions:

- **EQ (Branch if Equal):** Branches to the specified label if the Zero Flag (Z) in the SREG is set (indicating that the result of the previous operation was zero).
- **NE (Branch if Not Equal):** Branches to the specified label if the Zero Flag (Z) in the SREG is clear (indicating that the result of the previous operation was not zero).
- **CS (Branch if Carry Set):** Branches to the specified label if the Carry Flag (C) in the SREG is set (indicating a carry or borrow in the previous operation).
- **CC (Branch if Carry Clear):** Branches to the specified label if the Carry Flag (C) in the SREG is clear (indicating no carry or borrow in the previous operation).
- **MI (Branch if Minus):** Branches to the specified label if the Negative Flag (N) in the SREG is set (indicating a negative result).
- **PL (Branch if Plus):** Branches to the specified label if the Negative Flag (N) in the SREG is clear (indicating a non-negative result).
- **VS (Branch if Overflow Set):** Branches to the specified label if the Overflow Flag (V) in the SREG is set (indicating an overflow condition).
- **VC (Branch if Overflow Clear):** Branches to the specified label if the Overflow Flag (V) in the SREG is clear (indicating no overflow condition).
- **LT (Branch if Less Than, Signed):** Branches to the specified label if the Negative Flag (N) and Overflow Flag (V) are different (indicating a signed less-than condition).
- **GE (Branch if Greater Than or Equal, Signed):** Branches to the specified label if the Negative Flag (N) and Overflow Flag (V) are the same (indicating a signed greater-than-or-equal condition).

To learn more about AVR instruction set, check this: AVR Instruction Set Manual.

5.5 Procedure

Part 1: Control LEDs Programmatically

Example 1: Blinking LED

Let's begin our exploration of Arduino Assembly programming with the blinking LED example, as we did in the previous experiments. Write an assembly program to make the LED turn on for 200ms second and then turn off for the 200ms second too, and so on.

Components of the project:

The components used in the blinking of an LED are listed below:

a. 1 x Arduino Mega2560 board.
b. 1 x Breadboard
c. 2 x Jump wires
d. 1 x LED
e. 1 x Resistor of 220 Ohm

Procedure:

Firstly, connect the circuit as you did in the previous experiments, and as shown below:

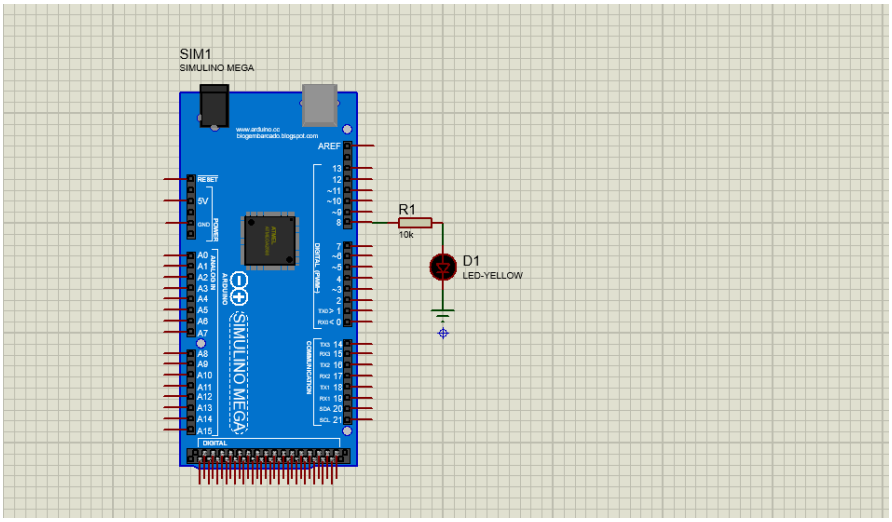


Fig. 8: Complete Circuit Connection

We will use the Arduino IDE as editor, and to allow an assembly program to compile in the Arduino IDE, we need two files with the same name and reside in the same project folder, one with ***.ino*** extension and the other with the ***.S*** extension. Follow the following steps:

1. Create new sketch and giving it the name **Example1.ino**.
2. In the top right corner of the IDE, click the three-dot menu and choose *New Tab*, name it **Example1.S** (capital S), as shown below:

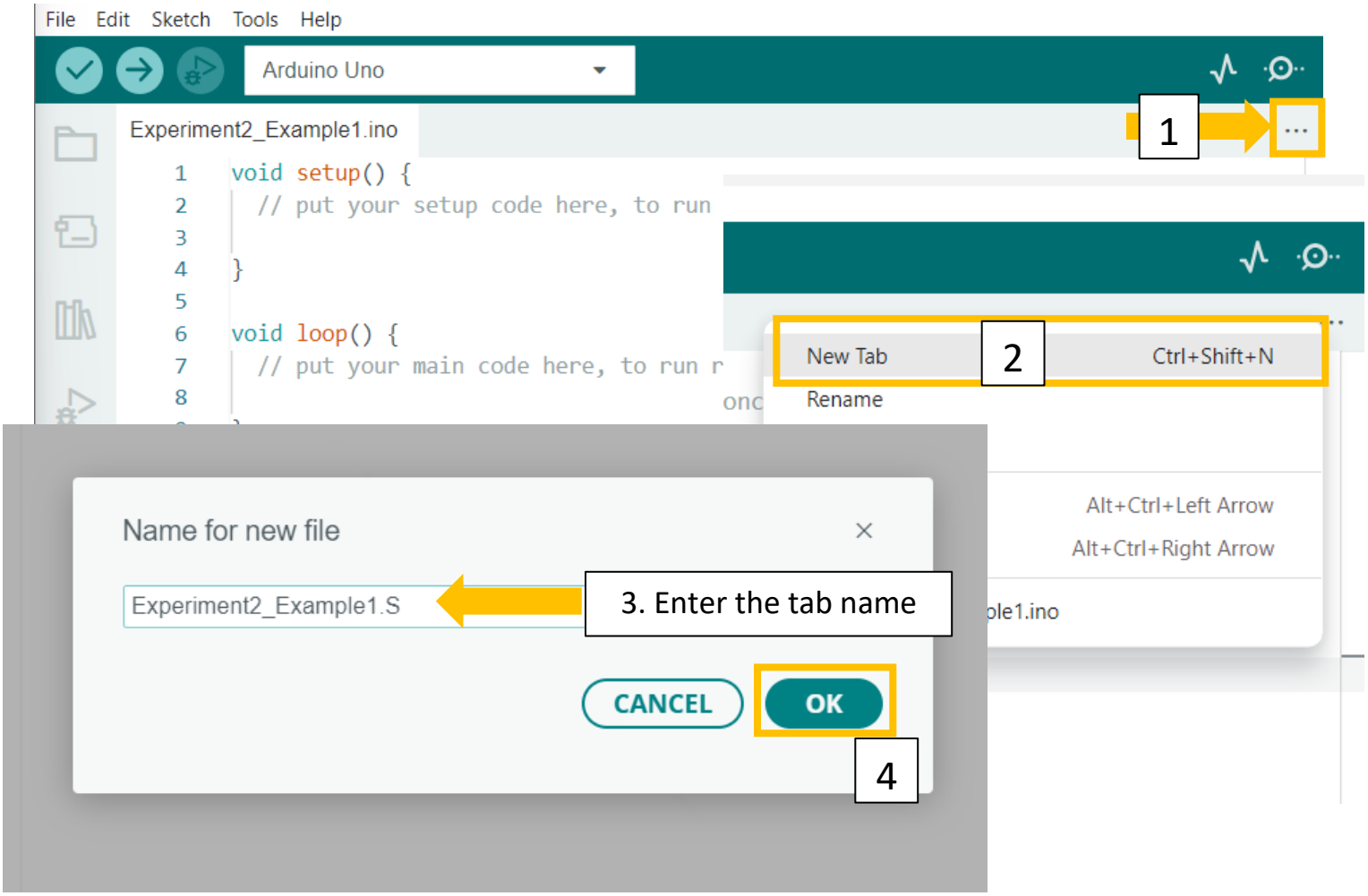


Fig. 9: Create New Tab for Assembly Code

3. In the .ino file, add this code. We are using a small sketch to manage our app:

Code Snippet 1: C Code for Blinking LED

```
//-------------------------
// C Code for Blinking LED
//-------------------------
extern "C" //the following will be located externally to this ino file
{
  void start(); //define our function prototypes here.
  void blink(byte);
}
//-----------------------------------------------------
void setup()
{
  start();
```

```
}
//-----------------------------------------------------
void loop()
{
  blink(1);
  delay(200);
  blink(0);
  delay(200);
}
```

Let's discuss the code before go forward in the assembly:

`extern "C"`: This tells the IDE compiler we are using code outside/externally to this C code sketch. It defines the two assembly functions we will be using: namely start() and blink(byte).

```
void start();
void blink(byte);
```

These are the prototypes of our functions and tells the compiler what the signature of the functions will be. In other words, besides its name, it defines the parameters and data types the functions will use. We must define our functions in C/C++ before using them.

The `void setup()` manages the app just like a regular sketch. The difference is, we have encapsulated our functions into another file which happens to be assembly code. We could do the same thing with C++ code to separate our code into logical sections. `setup()` calls the start() function inside the .S file to initialize our app.

As usual, the `loop()` function will make our program run in a loop. But in this case, the `blink()` function is in the .S file. The `blink()` function accepts an integer as a parameter. This value will be used to turn the on-board (D13 or PB7) LED ON and OFF.

`blink(1)`: This calls the `blink()` and passes 0x01 (0000 0001) as an argument to turn LED ON.
`blink(0)`: This calls the `blink()` function again but passes a 0x00 (0000 0000) as the argument to turn the LED OFF.

`delay(200)`: Here we are using the standard delay() function we are accustomed to in our sketches.

4. Now, open the .S tab, and copy the following code:

Code Snippet 2: Assembly Code for Blinking LED

```
/*
;_____________________________________
; Assembly Code Used to toggle the On-Board LED on Pin 13
```

```
;_____________________________________
NOTES:
Use actual pin# to toggle LED. In this case PB7.
*/
;tells where to start storing code in IO registers
#define __SFR_OFFSET 0x00
#include "avr/io.h"
;this is used to upload code to the Mega using AVRDUDE which comes with the IDE

  .global start ;declare global variables so it can be accessed from INO code
  .global blink

;-----------------------------------------------------------------------
start:
SBI DDRB, 7 ;SBI(Set Bit in IO Register) sets PB7 (D13) to output mode so we can
write.
RET ;return to setup() function
;-----------------------------------------------------------------------
blink: ;the argument sent to blink()in the INO code is stored in R24.
CPI R24, 0x00 ;Compare if the argument of blink() is 0 or not
BREQ ledOFF ;BREQ(Branch If Equal to 0). If R24 is equal to 0 jump to subroutine
ledOFF
SBI PORTB, 7 ;SBI (Set Bit in IO Register) D13 to high
RET ;return to loop() function
;-----------------------------------------------------------------------
ledOFF:
CBI PORTB, 7 ;set D13 to low
RET ;return to loop() function
```

Notice comments start with a ";" in assembly not "//".

Let's discuss how the previous code work:

1. First of all, we need to assign the direction of the ports, i.e., whether they are going to be used as INPUT or OUTPUT. So, PB7 has to be configured as O/P. The DDR register is used to specify whether the pin is going to be an I/P or an O/P, we can do that in two ways:

Method I: Assign the entire pins in the port as output, as follow:

```
LDI R16, 0xFF     ; 0x is to indicate that the following number is in Hexa
OUT DDRB, R16    ; Set the DDRB Register
```

In AVR Assembly, 0 in a data direction register specifies that it's an I/P and 1 specifies that it's an O/P.

FF in hexadecimal is 1111 1111 in binary. Which corresponds to assigning:

DDRB7	DDRB7	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
1	1	1	1	1	1	1	1

So now all pins are assigned as O/P

The statement `LDI R16, 0xFF` means to load the value FF into the register R16 and the statement `OUT DDRB, R16` means to assign the respective direction bits that specify the I/P and O/P.

Method II: Assign the desired pins only, as follow:

```
SBI DDRB, 7    ; Set PB7 as Output
```

The above code does the following changes in the Data Direction Register:

DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
1	X	X	X	X	X	X	X

The only change is that the bits other than 7 are not defined.

2. To check if the `blink()` argument is 1 or 0, we use `CPI R24, 0x00` which compare the content of R24 if 0, then `BREQ ledOFF` check if the zero flag is set. If so, it will branch to `ledOFF` label which will clear the PB7 and turn the LED off using `CBI PORTB, 7`.
3. If R24 doesn't equal 0, then the condition will not satisfy, and the `SBI PORTB, 7` will set the PB7 and turn the LED on.
4. RET returns to the void `loop()` of the main program and hence this code will start from the beginning.

Keep in mind that these codes are being called/controlled by the loop function inside the INO file. The return (RET) statements direct the execution instructions to jump/go back to where these functions were called just like standard logic in high-level languages!

TODO 1: Modify your work by using another LED. Write a program to make one LED turn on for 1 second while the other remains off, then switch their states so that the previously off LED turns on and the initially illuminated LED turns off for another 1-second cycle, and so on. Follow the steps we explained and mentioned previously to do the task.

Part 2: Control LEDs Using Push Button:

Example 2: Light a LED using a pushbutton

Components of the project:

The components used in the blinking of an LED are listed below:

a. Arduino Mega2560 board.
b. 1 x 220-ohm resistor
c. 1 x 10 Kohm resistor
d. Jump wires
e. 1 x pushbutton
f. 1 x red LED (we can take LED of any color)
g. 1 x Breadboard

Procedure:

- Connect the negative pin of the LED to the Ground, and the positive one to the resistor to PIN 13.
- Connect a 10 Kohm resistor to one end of the pushbutton to 5V.
- Connect PIN 4 of the Arduino Mega2560 to the 10k resistor to 5V.
- Connect the other end of the pushbutton to the ground.

Your connection should be as shown in Fig. 10:

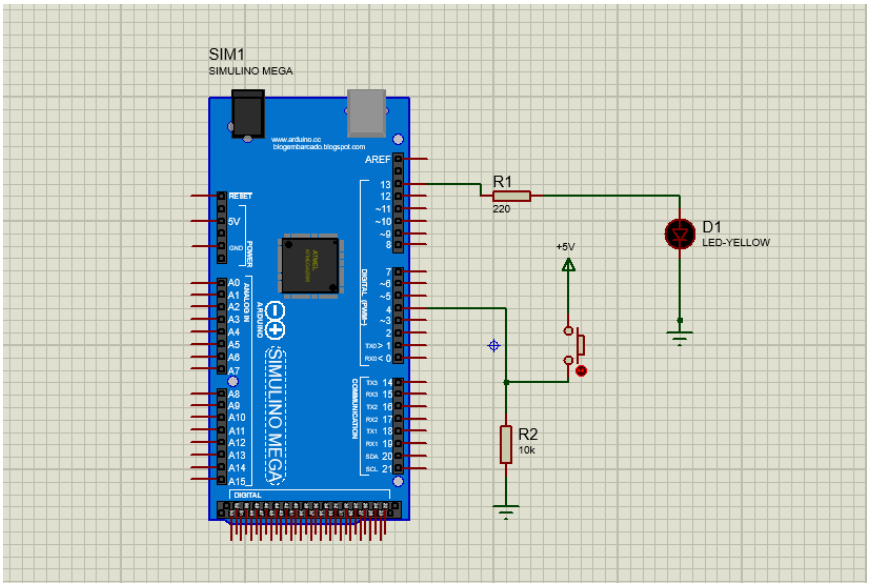


Fig. 10: Complete Circuit Connection

Now, after we connect our circuit, we need to write assembly program to make the LED turn on when the pushbutton is pressed and turn off when it is released.

1. First of all, create new project with two files, name them as Example2.ino and Example2.S.

2. Copy the following code in the Example2.ino file:

Code Snippet 3: C Code for Control LED using Pushbutton

```
extern "C" {
  // function prototypes
  void start();
  void blink();
}

void setup() {
  start();
}

void loop() {
  blink();
}
```

3. Copy the following code in the Example2.S file:

Code Snippet 4: Assembly Code for Control LED using Pushbutton

```
#define __SFR_OFFSET 0

#include "avr/io.h"

.global start
.global blink

start:
  SBI DDRB, 7   ; SET PB7 AS OUTPUT
  CBI DDRG, 5   ; SET PG4 AS INPUT
  SBI PORTG, 5  ; ENABLE PULL-UP RESISTER FOR PG4

blink:
  L2: SBIS PING, 5
      RJMP L1
      CBI PORTB, 7
      SBIC PING, 5
      RJMP L2
  L1: SBI PORTB, 7
```

```
    RET
```

Let's discuss some lines of the previous code work:

1. We configured PB7 as O/P, and PG5 as I/P:

```
SBI DDRB, 7   ; SET PB5 AS OUTPUT
CBI DDRG, 5   ; SET PD4 AS INPUT
```

The above code does the following changes in the Data Direction Register of PORTB and PORTD:

DDRB7	DDRB7	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
1	X	X	X	X	X	X	X

DDRG7	DDRG6	DDRG5	DDRG4	DDRG3	DDRG2	DDRG1	DDRG0
X	X	0	X	X	X	X	X

2. We enable the pull-up resistor of PG5 using `SBI PORTG, 5.`
3. The `SBIS PING, 5` skip the next line if the bit in I/O Register (PG5 in this example) is set, and `SBIC PING, 5` skip the next line if the bit in I/O Register (PG5 in this example) is cleared. We set the PB7 when PG5 is cleared since we use the pushbutton with pull-up resistor which means that PG5 will be 1 when the pushbutton is pressed and 0 when it isn't pressed.

TODO 2: Modify your code so you have two LEDs, one connected to PA3 and the other to PA5, and a push-button connected to PA1. Then write an assembly code such that when the push button is not pressed, the LED connected to PA3 will turn on and the LED connected to PA5 will turn off. When the push button is pressed, the LED connected to PA5 will turn off and the LED connected to PA3 will turn on.

Part 3: Control LED Intensity Using PWM (Dimming LED)

Components of the project:

The components used in the dimming LED are listed below:

a. 1 x Arduino Mega2560 board.
b. 1 x Breadboard
c. 2 x Jump wires
d. 1 x LED
e. 1 x Resistor of 220 Ohm

Procedure:

- Connect the negative pin of the LED to the Ground, and the positive one to the resistor to PIN 13.

Your connection should be as shown in Fig. 11:

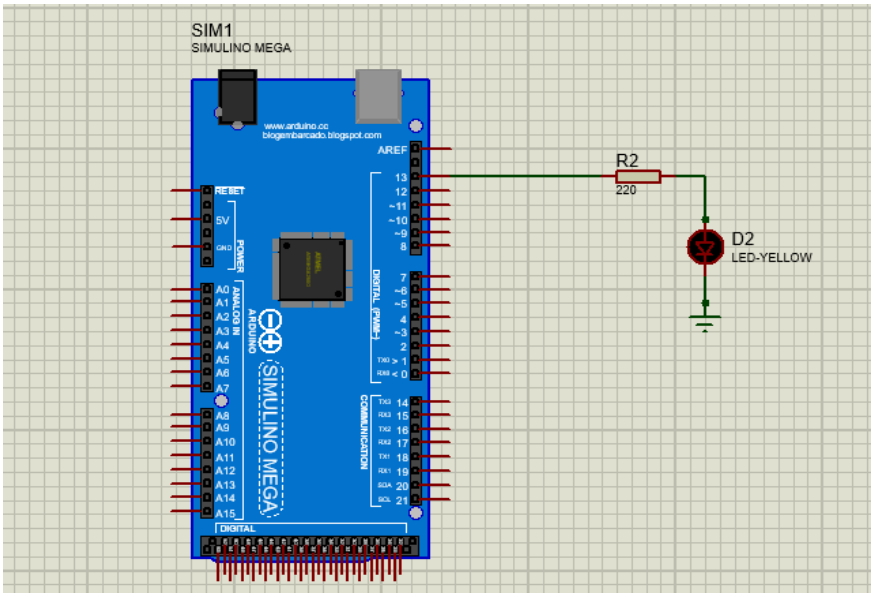


Fig. 11: Complete Circuit Connection

Now, after we connect our circuit, we need to write assembly program to control the LED brightness by generating PWM signal by a timer. In this example we will use Timer 0, you will learn more about timers in the next experiments.

1. Create new project with two files, name them as Example3.ino and Example3.S.

2. Copy the following code in the Example3.ino file:

Code Snippet 3: C Code for Control LED Brightness using PWM

```
extern "C" //the following will be located externally to this ino file
{
  void start(); //define our function prototypes here.
  void dimming();
}
//-----------------------------------------------------
void setup()
{
  start();
}
//-----------------------------------------------------
void loop()
{
  dimming();
}
```

3. Copy the following code in the Example3.S file:

Code Snippet 6: Assembly Code for Control LED Brightness using PWM

```
#define __SFR_OFFSET 0
#include "avr/io.h"

.equ LED_PIN, 7     ; Define the digital pin for LED (PB7)

.global start
.global dimming

start:
    ; Set LED_PIN (PB7) as an output
    sbi DDRB, LED_PIN

    ; Configure Timer0 for Fast PWM mode, non-inverted on OC0A (PB7)
    ldi r16, 0x83 ;(1 << COM0A1) | (1 << WGM01) | (1 << WGM00)
    out TCCR0A, r16
    ; No prescaler
    ldi r16, 0x01 ;(1 << CS00)
    out TCCR0B, r16

dimming:
    ; Increase brightness (fade in)
```

```
    ldi r18, 0
increase_brightness:
    out OCR0A, r18      ; Set PWM duty cycle for OC0A
    call delay
    inc r18             ; Increment brightness
    cpi r18, 255        ; Check if brightness reaches maximum
    brne increase_brightness  ; Continue increasing brightness until 255

    call delay

    ; Decrease brightness (fade out)
    ldi r18, 255
decrease_brightness:
    out OCR0A, r18      ; Set PWM duty cycle for OC0A
    call delay
    dec r18             ; Decrease brightness
    cpi r18, 0          ; Check if brightness reaches minimum
    brne decrease_brightness  ; Continue decreasing brightness until 0

    call delay

    rjmp dimming   ; Repeat loop

delay:
    ldi r20, 100  ; Adjust this delay for fade speed
delay_ms:
    ; Assuming F_CPU = 16MHz, delay 1ms at 16MHz clock
    ldi r21, 20
loop_delay_ms:
    dec r21
    brne loop_delay_ms
    dec r20
    brne delay_ms
    ret
```

Let's discuss some lines of the previous code work:

1. We configured PB7 as O/P:

```
SBI DDRB, 7   ; SET PB5 AS OUTPUT
```

2. We are going to use Timer0 Fast PWM mode, non-inverted on output A (OC0A/PB7). Fast PWM mode is activated by setting the Wave Generation Mode bits **WGM02**, **WGM01** and **WGM00** to **011** on Timer0 Timer/Counter control Registers (**TCCR0A** and **TCCR0B**). And to provide non-inverted PWM mode for output A, we should set **COM0A1**, **COM0A0** to **10** on **TCCR0A**, as following:

```
ldi r16, 0x83 ;(1 << COM0A1) | (1 << WGM01) | (1 << WGM00)
out TCCR0A, r16
```

3. Setting the **CS** bits to **001** on **TCCR0B** sets the prescaler to divide the clock by 1 i.e., no prescaler, as following:

```
ldi r16, 0x01 ;(1 << CS00)
out TCCR0B, r16
```

4. The following code segment will increase the brightness of the LED repeatedly from 0 until it reaches to the maximum value which is set to 255 by adjusting the value in `OCR0A` to control the duty cycle of the PWM signal and thus control the brightness of an LED:

```
; Increase brightness (fade in)
    ldi r18, 0
increase_brightness:
    out OCR0A, r18      ; Set PWM duty cycle for OC0A
    call delay
    inc r18             ; Increment brightness
    cpi r18, 255        ; Check if brightness reaches maximum
    brne increase_brightness  ; Continue increasing brightness until 255
```

5. The following code segment will decease the brightness of the LED repeatedly from 255 until it reaches to the minimum value which is set to 0 by adjusting the value in `OCR0A` to control the duty cycle of the PWM signal and thus control the brightness of an LED:

```
; Decrease brightness (fade out)
    ldi r18, 255
decrease_brightness:
    out OCR0A, r18      ; Set PWM duty cycle for OC0A
    call delay
    dec r18             ; Decrease brightness
    cpi r18, 0          ; Check if brightness reaches minimum
    brne decrease_brightness  ; Continue decreasing brightness until 0
```

6. The following code segment for delay:

```
delay:
    ldi r20, 100  ; Adjust this delay for fade speed
delay_ms:
    ; Assuming F_CPU = 16MHz, delay 1ms at 16MHz clock
```

```
    ldi r21, 20
loop_delay_ms:
    dec r21
    brne loop_delay_ms
    dec r20
    brne delay_ms
    ret
```

Please note that the delays in assembly language may not be precisely the same as the delay functions used in Arduino.

You can refer to ATmega2560 datasheet to learn more about Timer/Counter.

TODO 3: Modify your code by adding another LED and connected it to PG5 (OC0B) i.e., output B of Timer 0, and make this LED intensity decrease while the first LED (PB7) intensity increasing and vice versa when the first LED intensity decreasing.

جامعة بيرزيت
BIRZEIT UNIVERSITY

Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

Microprocessor and Embedded Systems Lab (ENCS5101)

Experiment No. 6 - Registers, Timers, Interrupts, and ADC

6.1 Objectives

1. Configure some registers to set ports as input/output and to configure timers.
2. Introduce timers, hardware and software interrupts, and Analog to Digital Converter (ADC).
3. Interface some sensors with Arduino Mega2560.

6.2 Tools & Components

- Proteus v8.13
- Arduino IDE v2.1.1
- Arduino Mega 2560
- 7-Segment Display
- Light Dependent Resistor (LDR)
- Temperature Sensor (TMP36)

6.3 Theory

6.3.1 Configuring Ports Using Registers:

As mentioned in previous experiments, Arduino Mega 2560 which has ATmega2560 microcontroller, 11 I/O ports (PORTA, PORTB, PORTC, PORTD, PORTE, PORTF, PORTG, PORTH, PORTJ, PORTK, and PORTL). Each of these ports typically consists of 8 pins, except for Port G, which has 6 pins as Fig.1 shown:

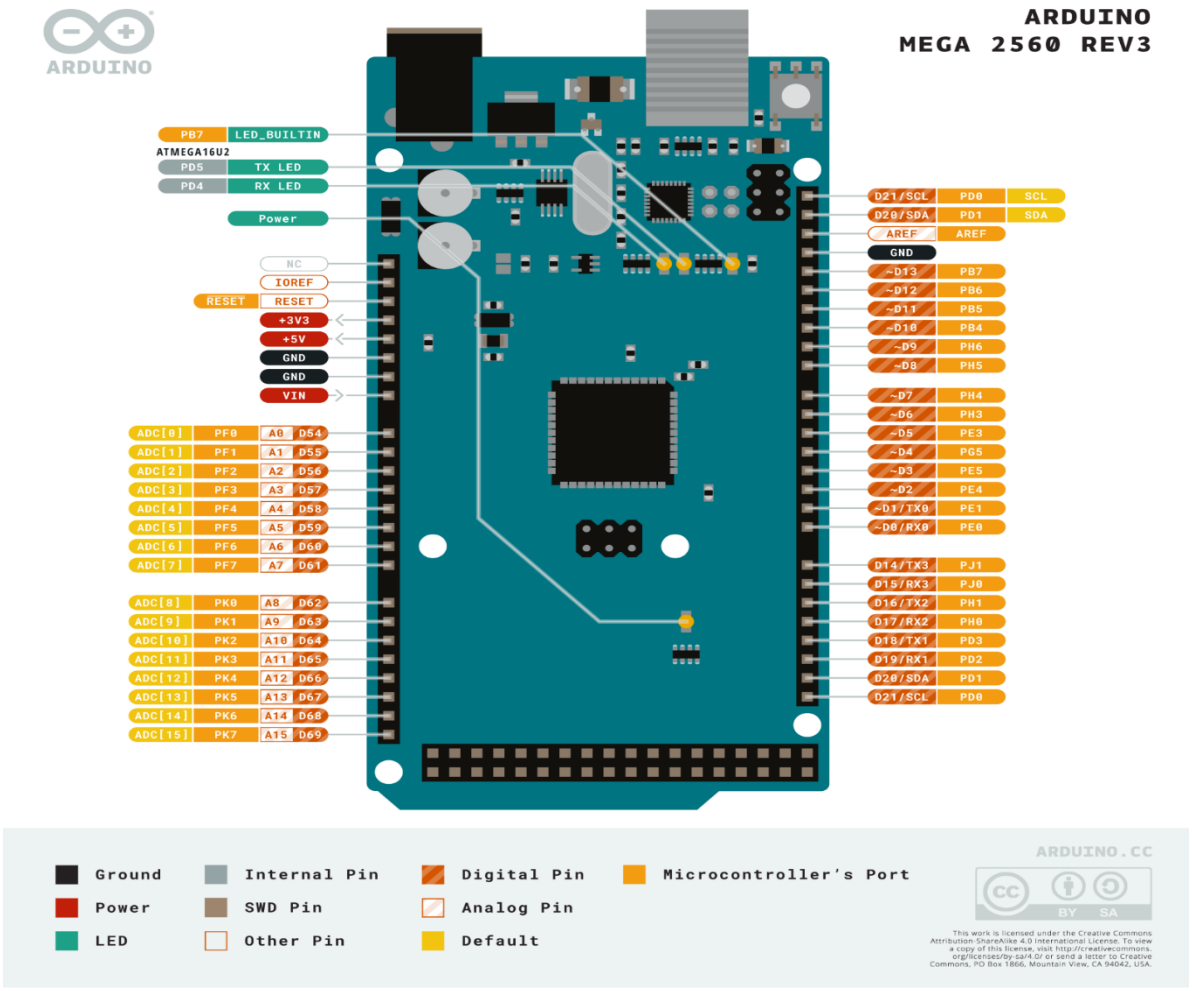


Fig. 1: Arduino Mega Pinout and their Mapping to the AVR Atmega2560 Microcontroller (Arduino Mega2560 Rev3 Documentation)

In the previous experiments, you have learnt how to set a pin as input or output using *PinMode* command. Using some registers, it is possible to set a port as input or output and in fact do more things. Three registers can be used to configure or access a pin in a port.

1. **Data Direction Register (DDRx)**
2. **Data Register (PORTx)**
3. **Input Pins Register (PINx)**

Where x refers to the port symbol such as, A, B, etc.

To set a pin to output, Data Direction Register for that pin should be "1" and the data itself should be in Data Registers ("1" > High "5 Volt") and "0" > Low "0 Volt".

For instance, DDRB is the Data Direction Register for Port B and PORTB is the Data Register for Port B. For example, if DDRB and PORTB as shown below:

DDRB

DDRB7	DDRB7	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
1	1	1	1	1	1	1	1

PORTB

PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
1	1	1	1	0	0	0	0

Then, DDRB=0xFF and PORTB=0xF0 which mean all 8 pins of Port B are Output. The most significant 4-bits are "HIGH" and the least significant 4-bits are "LOW". To set a pin to input, then Data Direction register for that pin should be "0".

Here we have two choices:

1- To enable the internal pull-up register, the Data register for that pin should be "1".
2- To disable the internal pull-up register, the Data register for that pin should be "0".

For example, DDRB=0x00, PORTB=0XF0. This set all pins on PORTB to input. The most significant 4-bits are with internal pull-up resistor a and the least significant 4-bits are without pull-up resistor.

Example: Consider PORTD, we want to have bit0 and bit2 as input with pullup resistors, bit5 and bit6 as input without pullup, and bit3 as output, the we will set the DDRD and PORTD registers as follow:

DDRD = 0x08

DDRB7	DDRB7	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
0	0	0	0	1	0	0	0

PORTD = 0x05

PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
0	0	0	0	0	1	0	1

6.3.2 Timers/Counters:

Several registers can be used to configure a timer:

1. **TCCRx (Timer/Counter Control Register):** The prescaler can be configured here.
2. **TCNTx (Timer/Counter Register):** The actual timer value is stored here.
3. **OCRx (Output Compare Register)**.
4. **TIMSKx (Timer/Counter Interrupt Mask Register)**: To enable/disable timer interrupts. **Where x represents a timer number.**

ATmega2560 has six timers, timer 0, 1, 2, 3, 4, and 5. Timer/Counter 0 and Timer/Counter 2 are 8-bit timers while Timer/Counter 1, Timer/Counter 3, Timer/Counter 4, and Timer/Counter 5 are 16-bit timers that allows accurate program execution timing.

For Timer 1, TCCR1A is used to set the mode. We have different modes such as normal mode and Clear Timer on Compare Match (CTC) mode. When WGM1[3:0] are 0000 then the timer uses in the normal mode, in the normal mode, an interrupt will occur when the timer reaches its maximum value. Because Timer 1 is

16-bit, the timer will overflow when it reaches 0xFFFF i.e., 65,536. Here, the timer overflow interrupt service routine ISR(TIMER1_OVF_vect) will be called. Fig 2 (a) and Fig.2 (b) shows the Timer/Counter Control Registers:

TCCR1A – Timer/Counter 1 Control Register A

Bit	7	6	5	4	3	2	1	0	
(0x80)	COM1A1	COM1A0	COM1B1	COM1B0	COM1C1	COM1C0	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig. 2 (a): Timer/Counter Control Register A (ATmega2560 Documentation)

TCCR1B – Timer/Counter 1 Control Register B

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig. 2 (b): Timer/Counter Control Register B (ATmega2560 Documentation)

To enable overflow interrupt, we should set bit0 of TIMSK1 register.

TIMSK1 – Timer/Counter 1 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6F)	–	–	ICIE1	–	OCIE1C	OCIE1B	OCIE1A	TOIE1	TIMSK1
Read/Write	R	R	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 5 – ICIEn: Timer/Countern, Input Capture Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Countern Input Capture interrupt is enabled. The corresponding Interrupt Vector (see "Interrupts" on page 101) is executed when the ICFn Flag, located in TIFRn, is set.

- **Bit 3 – OCIEnC: Timer/Countern, Output Compare C Match Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Countern Output Compare C Match interrupt is enabled. The corresponding Interrupt Vector (see "Interrupts" on page 101) is executed when the OCFnC Flag, located in TIFRn, is set.

- **Bit 2 – OCIEnB: Timer/Countern, Output Compare B Match Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Countern Output Compare B Match interrupt is enabled. The corresponding Interrupt Vector (see "Interrupts" on page 101) is executed when the OCFnB Flag, located in TIFRn, is set.

- **Bit 1 – OCIEnA: Timer/Countern, Output Compare A Match Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Countern Output Compare A Match interrupt is enabled. The corresponding Interrupt Vector (see "Interrupts" on page 101) is executed when the OCFnA Flag, located in TIFRn, is set.

- **Bit 0 – TOIEn: Timer/Countern, Overflow Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Countern Overflow interrupt is enabled. The corresponding Interrupt Vector (see "Interrupts" on page 101) is executed when the TOVn Flag, located in TIFRn, is set.

Fig. 3: Timer/Counter 1 Interrupt Mask Register (ATmega2560 Documentation)

Arduino Mega runs on 16MHz, so each pulse it 0.0625 us. Therefore, Timer1 needs only about 4ms to count from 0 to 0xFFFF (i.e. 65535 x 0.0625 us ≈ 4 ms) and to slow down the timer, we need to use a prescaler. We can set the Clock Select (CS12, CS11, CS10) bits of TCCR1B to set the prescaler. The table below shows different prescaler values.

• **Bit 2:0 – CSn2:0: Clock Select**

The three clock select bits select the clock source to be used by the Timer/Counter, see Figure 17-10 and Figure 17-11 on page 152.

Table 17-6. Clock Select Bit Description

CSn2	CSn1	CSn0	Description
0	0	0	No clock source. (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/1 (No prescaling
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on Tn pin. Clock on falling edge
1	1	1	External clock source on Tn pin. Clock on rising edge

If external pin modes are used for the Timer/Countern, transitions on the Tn pin will clock the counter even if the pin is configured as an output. This feature allows software control of the counting.

Fig. 4: Clock Select Bit Description (ATmega2560 Documentation)

If we want to divide the frequency by 1024 then we should set **CS12**, **CS11**, **CS10** to **101**. This way the new frequency will be 16MHz/1024 = 15625 Hz and hence it will take the Timer 1 about 4 seconds to count from 0 to 0xFFFF.

Example: We want to call a function every 0.25 seconds:

If we selected 256 to be a prescaler, then:

- New Frequency = 16M / 256 = 62500 Hz
- Now the period T = 1/ 62500 =16 us
- Count = 0.25 seconds / 16 us =15625
- The counter will overflow at 0xFFFF = 65535
- So, the start value of the counter should be 65536 – 15625 = 49911

So, if the counter starts with 49911, it will overflow after 0.25 seconds and hence it will call the interrupt service routine ISR(TIMER1_OVF_vect).

Questions:

Study the following Code, and then answer these questions:

1. Why do we have 49911 inside the interrupt service routine?

2. Use Serial.print to find the values of CS12 and TOIE1.

3. What does "<<" mean?

4. What does "|" mean?

5. What does "^" mean?

Code Snippet 1: Timer Code

```
#define ledPin 13
void setup(){
  pinMode(ledPin, OUTPUT);
  // initialize timer1
  noInterrupts();           // disable all interrupts
```

```
  TCCR1A = 0;
  TCCR1B = 0;
  TCNT1 = 49911;
  TCCR1B |= (1 << CS12);    // 256 prescaler
  TIMSK1 |= (1 << TOIE1);   // enable timer overflow interrupt
  interrupts();             // enable all interrupts
}
ISR(TIMER1_OVF_vect)        // interrupt service routine
{
  TCNT1 = 49911;            // preload timer
  digitalWrite(ledPin, digitalRead(ledPin) ^ 1);
}
void loop(){  // your program here...
}
```

6.3.3 Hardware Interrupt

Hardware interrupts occurs as a result of a signal generated by a hardware component such as a switch or a sensor. When an interrupt occurs, the normal execution of a program gets interrupted and an Interrupt Service routine will be executed. When the execution of the interrupt service routine is completed, the execution of the user program is resumed as Fig. 5 shows:

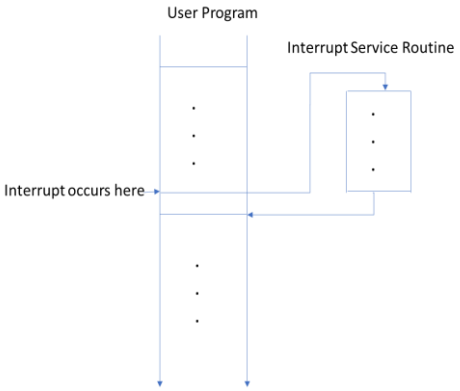


Fig. 5: Interrupt Execution

Arduino Mega has six external interrupts **INT0**, **INT1**, **INT2**, **INT3**, **INT4**, and **INT5** which are mapped to Arduino pins **2**, **3**, **21**, **20**, **19**, and **18** respectively. Arduino uses ***attachinterrupt*** to connect an external interrupt to an ISR as shown in the following:

```
attachInterrupt(InterruptNumber, ISR, MODE)
```

where **InterruptNumber** either 0 to 5 (INT0, INT1, INT2, INT3, INT4, or INT5), **ISR** is the name of Interrupt Service Routine, and **MODE** can be one of the following:

- **LOW** to trigger the interrupt whenever the pin is low.
- **CHANGE** to trigger the interrupt whenever the pin changes value.
- **RISING** to trigger when the pin goes from low to high.
- **FALLING** to trigger when the pin goes from high to low.

As shown below:

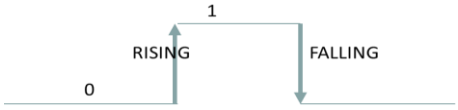


Fig. 6: Interrupt Triggering Modes

Example:

Code Snippet 2: Hardware Interrupt Code

```
const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;
void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(0, blink, RISING);
}

void loop() {
  digitalWrite(ledPin, state);
}

void blink() {
  state = !state;
}
```

ISR can't have any parameter and should not return any value. They should be simple. Some instructions such as *delay()* and *millis()* do not work inside an ISR. Variables shared between the main and ISR should be declared volatile.

6.3.4 Analog to Digital Converter (ADC)

An ADC converts an analog signal into N-bit digital number. For instance, Arduino Mega has a 10-bit ADC with adjustable reference voltage (Vref). The default reference voltage (Vref) is 5 Volt. This way a 0-volt analog input give 0 digital value and 5 volt input gives 1023 digital value.

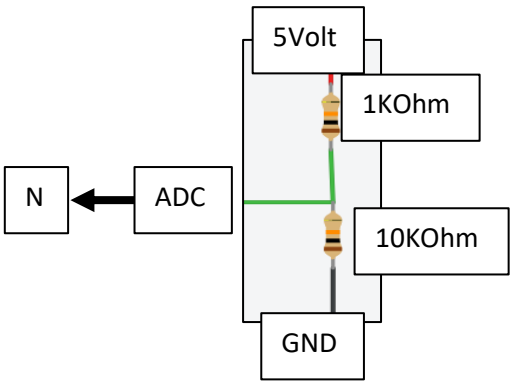
X= Vi/Vref*1023, where **X** is the digital value that is equivalent to the analog voltage**.** For instance, If the analog input is 2.3 volt then the digital value is 2.3/5 *1023=470. One important characteristic of an ADC is the resolution which is the smallest measurable voltage change. Therefore, the Voltage resolution (**Vres**) of a **N**-bit ADC with **Vref** is **Vres= Vref/(2^N-1)**.

Example: What is the resolution of a 10-bit ADC with Vref=5 Volt?

Vres = 5 / 1023= 4.88 mV

Question:

Consider the figure with 10-bit ADC and 5 volt Vref:
What is the value of the ADC reading (N)?



6.4 Procedure

Part 1: Registers

Study the following code, and then connect the circuit shown in Fig.7. After that run the code on that circuit and answer the following questions:

a. Explain the code.
b. What is the result of executing this code?

Code Snippet 3: Registers Example Code

```
void setup() {
  DDRA=0xFF; //ALL pins are output
}

void loop() {
  PORTA=B00000111;
  delay(1000);
  PORTA=0x00; //all pins are OFF
  delay(1000);
}
```

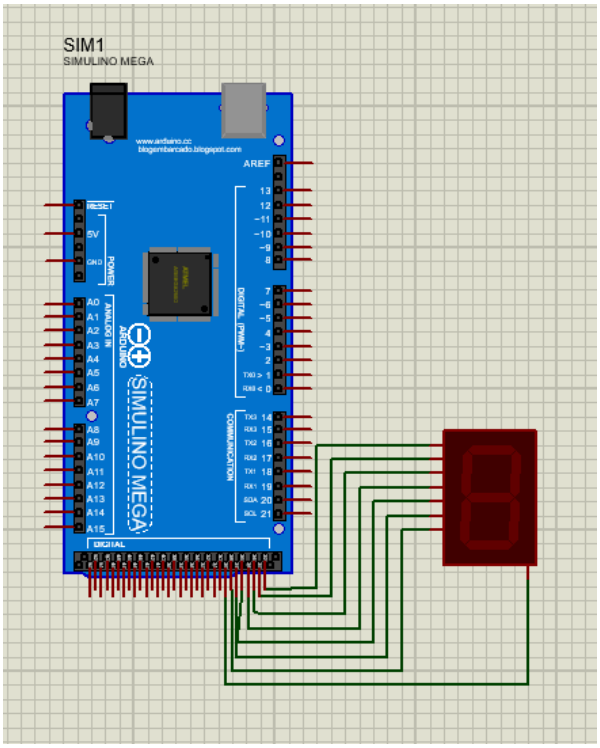


Fig. 7: 7-Segment Connection

Note that the 7-segment used in this example is a common cathode, so the common pin value (which is connected to D29 (PA7)) should have a value of 0, and if you want any LED of the 7-segment to turn on then the corresponding pin should have a value of 1.

TODO 1: using a 7-segment display, write a program to count from 0 to 9 with 1.5 sec delay between the counts.

Part 2: Timers

Study the following code, and then connect the circuit shown in Fig.8. After that run the code on that circuit:

Code Snippet 4: Timer Example Code

```
#define ledPin 13
void setup(){
  pinMode(ledPin, OUTPUT);
  // initialize timer1
  noInterrupts();           // disable all interrupts
  TCCR1A = 0;
  TCCR1B = 0;
  TCNT1 = 49911;
  TCCR1B |= (1 << CS12);    // 256 prescaler
  TIMSK1 |= (1 << TOIE1);   // enable timer overflow interrupt
  interrupts();             // enable all interrupts
}
ISR(TIMER1_OVF_vect)        // interrupt service routine
{
  TCNT1 = 49911;            // preload timer
  digitalWrite(ledPin, digitalRead(ledPin) ^ 1);
}
void loop(){  // your program here...
}
```

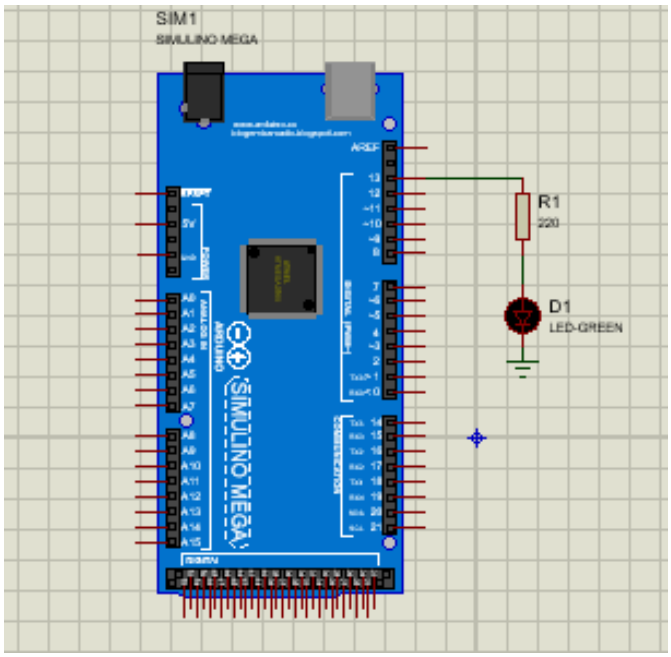


Fig. 8: Example 2 Circuit Connection

Part 3: Hardware Interrupts

Study the following code, and then connect the circuit shown in Fig.9. After that run the code on that circuit:

Code Snippet 5: Hardware Interrupts Example Code

```
// Hardware interrupt example
int pin = 13;
const int interruptPin = 2;
volatile int state = LOW;
void setup()
{
  pinMode(pin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}
void loop()
{
  digitalWrite(pin, state);
}
void blink()
{
  state = !state;
}
```

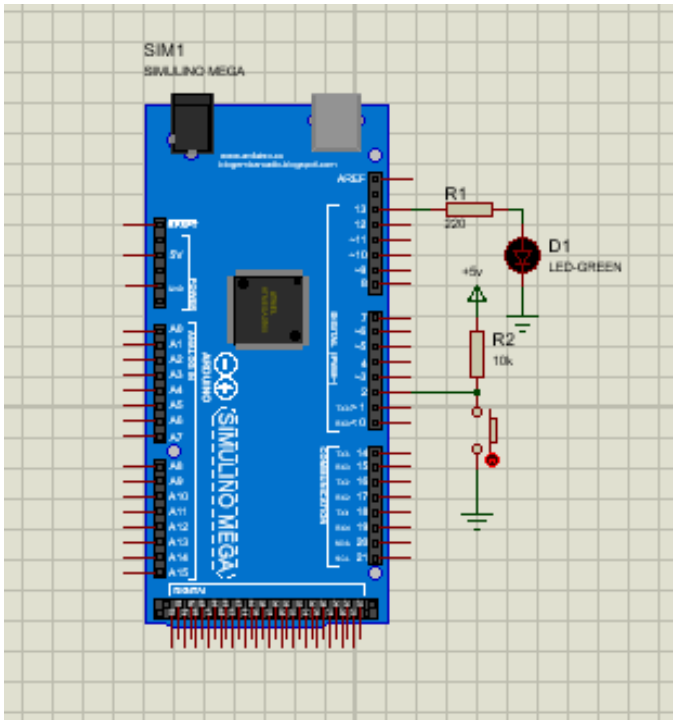
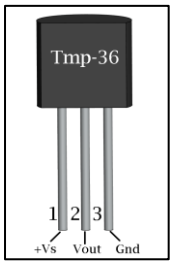


Fig. 9: Example 3 Circuit Connection

TODO 2: Using the interrupt, implement a counter that stops counting whenever a bush button is pressed and then resumes counting whenever the bush button is pressed again. You can display the counts either on a 7-segment display or on the serial monitor.

Part 4: Analog to Digital Converter



A. Measuring the temperature:

Connect the temperature sensor TMP36 as shown in Fig. 10. After that run the code on that circuit:

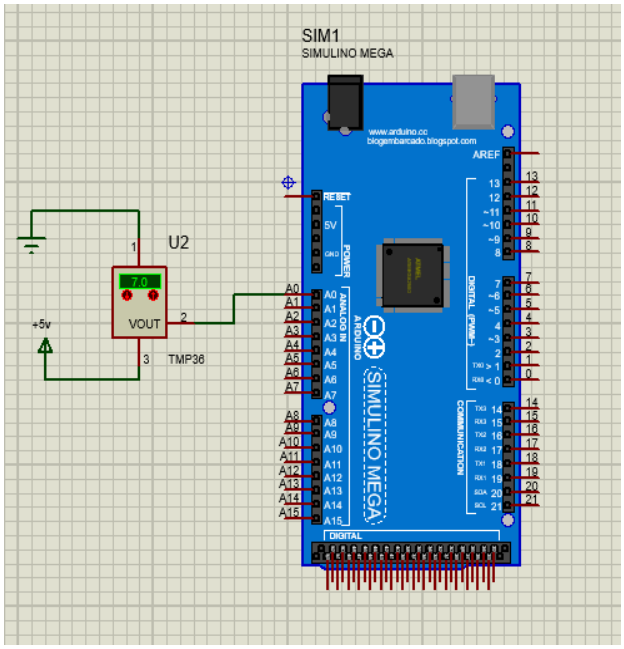
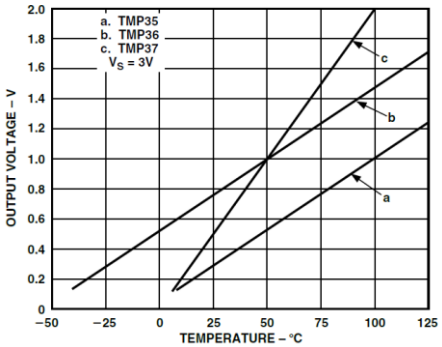


Fig. 10: Example 4 Circuit Connection

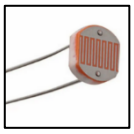
Code Snippet 6: Temperature Sensor Example Code

```
void setup() {
  Serial.begin(9600);}
void loop() {
  int reading = analogRead(0);
  float voltage = reading * 5.0/1024;
  Serial.print(reading);
  Serial.println(" reading");
  Serial.print(voltage);
  Serial.println(" volts");
  float temperatureC = (voltage - 0.5) * 100;
  Serial.print(temperatureC);
  Serial.println(" degrees C");
  delay(3000);}
```



TPC 1. Output Voltage vs. Temperature

B. In this part, we are going to use Light Dependent Resistor (LDR), the resistance of this sensor depends on light.



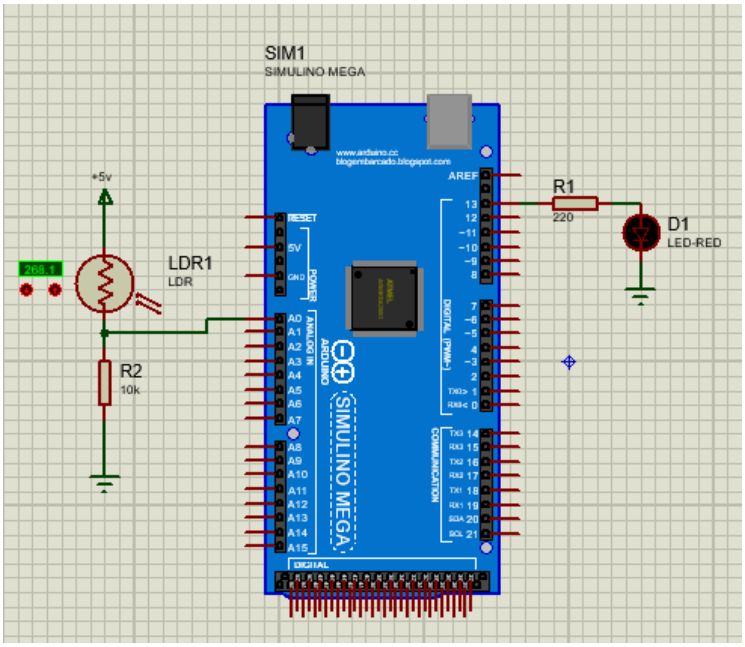


Fig. 10: TODO 4a Circuit Connection

Code Snippet 7: LDR Example Code

```
#define LED 13
#define LDR 0

void setup()
{
  Serial.begin(9600);
  pinMode(LED, OUTPUT);
}

void loop()
{
  if (analogRead(LDR) < 700){
    digitalWrite(LED,HIGH);
    Serial.println(analogRead(LDR));
  }
  else digitalWrite(LED,LOW);
  delay(200);
}
```

TODO 3: Measure the resistance of the LDR when it is dark. Then focus a light on the LDR (using your smartphone) and measure the resistance.



Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

Microprocessor and Embedded Systems Lab (ENCS5101)

Experiment No. 7 - Serial Communication

7.1 Objectives

Understand the basic concepts of serial communication and to practically apply it using Arduino.

7.2 Tools & Components

1. Proteus v8.13
2. Arduino IDE v2.1.1
3. 2 Arduino Mega 2560
4. Push button and LEDs
5. I2C LCD display

7.3 Theory

7.3.1 Universal Asynchronous Receiver/Transmitter (UART)

UART stands for **Universal Asynchronous Receiver/Transmitter**, it's not a communication protocol like SPI and I2C, but a physical circuit in a microcontroller, or a stand-alone IC. A UART's main purpose is to transmit and receive serial data.

In UART communication, two UARTs communicate directly with each other. The transmitting UART converts parallel data from a controlling device like a CPU into serial form, transmits it in serial to the receiving UART, which then converts the serial data back into parallel data for the receiving device. **Only two wires are needed to transmit data between two UARTs**. Data flows from the **Tx** pin of the transmitting UART to the **Rx** pin of the receiving UART.

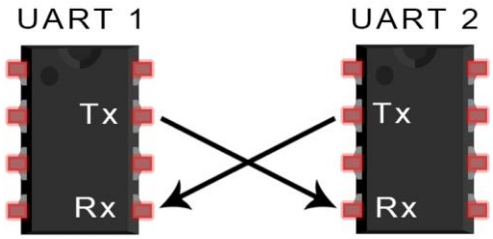


Fig. 1: UART Communication

UARTs transmit data **asynchronously**, which means there is no clock signal to synchronize the output of bits from the transmitting UART to the sampling of bits by the receiving UART. Instead of a clock signal, the transmitting UART adds **start** and **stop** bits to the data packet being transferred. These bits define the beginning and end of the data packet so the receiving UART knows when to start reading the bits.

When the receiving UART detects a start bit, it starts to read the incoming bits at a specific frequency known as the baud rate. **Baud rate** is a measure of the speed of data transfer, expressed in bits per second (bps). Both UARTs must operate at about the same baud rate.

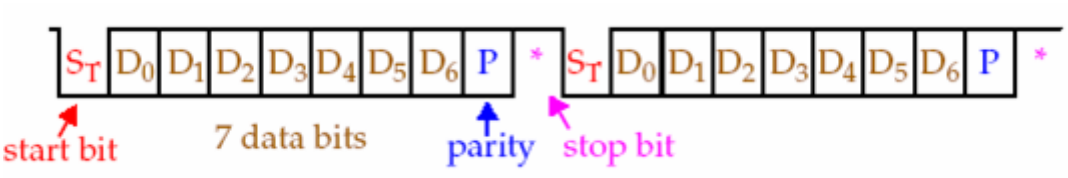


Fig.2: Serial Communication

- **START BIT**
  The UART data transmission line is normally held at a high voltage level when it's not transmitting data. To start the transfer of data**,** the transmitting UART pulls the transmission line from high to low for one clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.

- **DATA FRAME**
  The data frame contains the actual data being transferred. It can be **5 bits** up to **8 bits** long if a parity bit is used. If no parity bit is used, the data frame can be 9 bits long. In most cases, the data is sent with the least significant bit first.
- **PARITY**
  Parity describes the **evenness** or **oddness** of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long-distance data transfers. After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the 1-bits in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1-bits in the data frame should total to an odd number. When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd; or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.
- **STOP BITS**
  To signal the end of the data packet**,** the sending UART drives the data transmission line from a low voltage to a high voltage for at least two-bit durations.

7.3.1.1 Serial communication with Arduino [T_x, R_x]

Serial communication on pins TX/RX uses TTL logic levels (5V or 3.3V depending on the board). Don't connect these pins directly to an RS232 serial port; they operate at +/- 12V and can damage your Arduino board.

Serial is used for communication between the Arduino board and a computer or other devices. All Arduino boards have at least one serial port (also known as a UART or USART): Serial. It communicates on digital pins 0 (**RX**) and 1 (**TX**) as well as with the computer via USB. Thus, if you use these functions, you cannot also use pins 0 and 1 for digital input or output.

You can use the Arduino environment's built-in serial monitor to communicate with an Arduino board. Click the serial monitor button in the toolbar and select the same baud rate used in the call to ***begin()***.



Fig.3: Serial Monitor in Arduino IDE

In order to handle the serial communication, a lot of built-in functions are implemented to make your life easier. The following functions are used mainly. For extra information, please refer to (https://docs.arduino.cc/language-reference/en/functions/communication/serial/)

- ***Serial.available()***

Get the number of bytes (characters) available for reading from the serial port. This is data that's already arrived and stored in the serial receive buffer.

- ***Serial.begin()***

Sets the data rate in bits per second (baud) for serial data transmission. For communicating with Serial Monitor, make sure to use one of the baud rates listed in the menu at the bottom right corner of its screen. You can, however, specify other rates - for example, to communicate over pins 0 and 1 with a component that requires a particular baud rate. [**Must be used in setup()**]

An optional second argument configures the data, parity, and stop bits. **The default is 8 data bits, no parity, one stop bit.**

Syntax:

Serial.begin(speed)
Serial.begin(speed, config)

Where:

speed: in bits per second (baud). Allowed data types: long, e.g. 4800, 9600, 19200.
config: sets data, parity, and stop bits. Valid values are:

No Parity:
SERIAL_5N1
SERIAL_6N1
SERIAL_7N1
SERIAL_8N1 (the default)
SERIAL_5N2
SERIAL_6N2
SERIAL_7N2
SERIAL_8N2

Even Parity:
SERIAL_5E1
SERIAL_6E1
SERIAL_7E1
SERIAL_8E1
SERIAL_5E2
SERIAL_6E2
SERIAL_7E2
SERIAL_8E2

Odd Parity:
SERIAL_5O1
SERIAL_6O1
SERIAL_7O1
SERIAL_8O1
SERIAL_5O2
SERIAL_6O2
SERIAL_7O2
SERIAL_8O2

Read Data:

- ***Serial.read()***

Reads incoming serial data.
Returns: The first byte of incoming serial data available (or -1 if no data is available).
Data type: int.

- ***Serial.readBytes(buffer, length)***

Reads characters from the serial port into a buffer. The function terminates if the determined length has been read, or it times out.
Returns: The number of bytes placed in the buffer.
Data type: size_t.

- ***Serial.readString()***

Reads characters from the serial buffer into a String. The function terminates if it times out.
Returns: A String read from the serial buffer.

- ***Serial.parseInt()***

Looks for the next valid integer in the incoming serial.

- ***Serial.parseFloat()***

Returns the first valid floating-point number from the Serial buffer. It is terminated by the first character that is not a floating-point number

Write Data:

- ***Serial.print()***

Prints data to the serial port as human-readable ASCII text. This command can take many forms, numbers are printed using an ASCII character for each digit, floats are similarly printed as ASCII digits, defaulting to two decimal places, bytes are sent as a single character, and characters and strings are sent as is.

- ***Serial.write()***

Writes binary data to the serial port. This data is sent as a byte or series of bytes; to send the characters representing the digits of a number use the ***print()*** function instead.

7.3.2 Inter-Integrated Circuit (I2C)

7.3.2.1 I2C Protocol Overview

A typical embedded system consists of one or more microcontrollers and peripheral devices like memories, converters, I/O expanders, LCD drivers, sensors, matrix switches, etc. The complexity and the cost of connecting all those devices together must be kept to a minimum. The system must be designed in such a way that slower devices can communicate with the system without slowing down faster ones. To satisfy these requirements a serial bus is needed. A bus means specification for the connections, protocol, formats, addresses and procedures that define the rules on the bus. This is exactly what I2C bus specifications define.

I2C combines the best features of other serial communication protocols like SPI and UARTs. With I2C, you can connect multiple slaves to a single master and you can have multiple masters controlling single, or multiple slaves. This is really useful when you want to have more than one microcontroller logging data to a single memory card or displaying text to a single LCD. Like UART communication, I2C only uses two wires to transmit data between devices as Fig.4 shows.

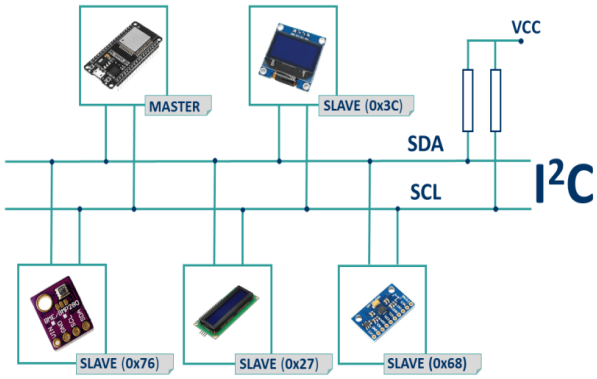


Fig.4: I2C connection

The wires used are:

SDA (Serial Data): The line for the master and slave to send and receive data.

SCL (Serial Clock): The line that carries the clock signal.

Both signals (SCL and SDA) are bidirectional. They are connected via resistors to a positive power supply voltage, which means that when the bus is free, both lines are high.

I2C is a serial communication protocol, so data is transferred bit by bit along a single wire (the SDA line). **I2C** is synchronous, so the output of bits is synchronized to the sampling of bits by a clock signal shared between the master and the slave. The clock signal is always controlled by the master.

Moreover, here are some of the I2c terminologies:

- **Transmitter**: This is the device that transmits data to the bus.
- **Receiver**: This is the device that receives data from the bus.
- **Master**: This is the device that generates clock, starts communication, sends I2C commands and stops communication.
- **Slave**: This is the device that listens to the bus and is addressed by the master.
- **Multi-master**: I2C can have more than one master and each can send commands.
- **Arbitration**: A process to determine which of the masters on the bus can use it when more masters need to use the bus.
- **Synchronization**: A process to synchronize clocks of two or more devices.

7.3.2.2 I2C Data Transfer and Timing

1. Bit Transfer:

For each clock pulse one bit of data is transferred. The SDA signal can only change when the SCL signal is low – when the clock is high the data should be stable, as Fig.5 shows:

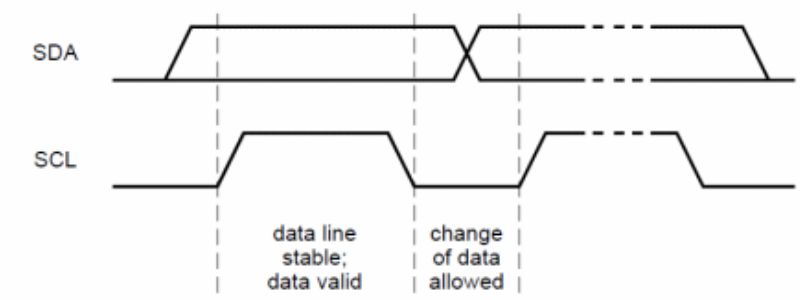


Fig.5: I2C bit transfer

2. Start and stop conditions:

Each I2C command initiated by master device starts with a **START condition** and ends with a **condition**. For both conditions SCL has to be high. A high to low transition of SDA is considered as **START** and a low to high transition as **STOP**, as Fig.6 shows:

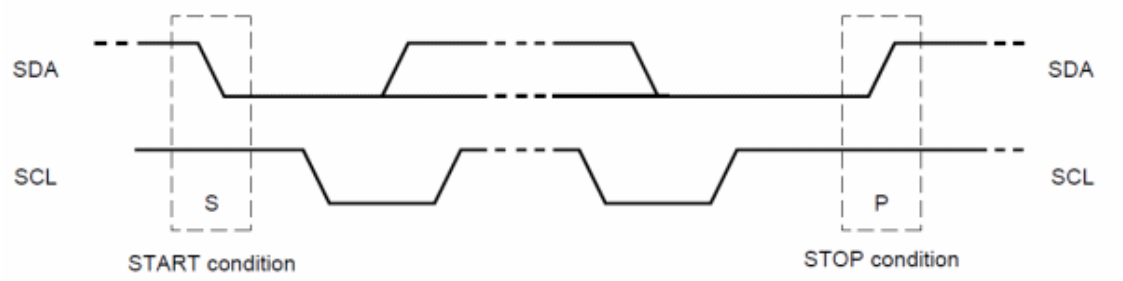


Fig.6: I2C Start and stop bits transfer

After the **START** condition the bus is considered as busy and can be used by another master only after a **STOP** condition is detected. After the **START** condition the master can generate a repeated **START**. This is equivalent to a normal **START** and is usually followed by the slave I2C address.

Microcontrollers that have dedicated I2C hardware can easily detect bus changes and behave also as I2C slave devices. However, if the I2C communication is implemented in software, the bus signals must be sampled at least two times per clock cycle in order to detect necessary changes.

3. I2C Data Transfer:

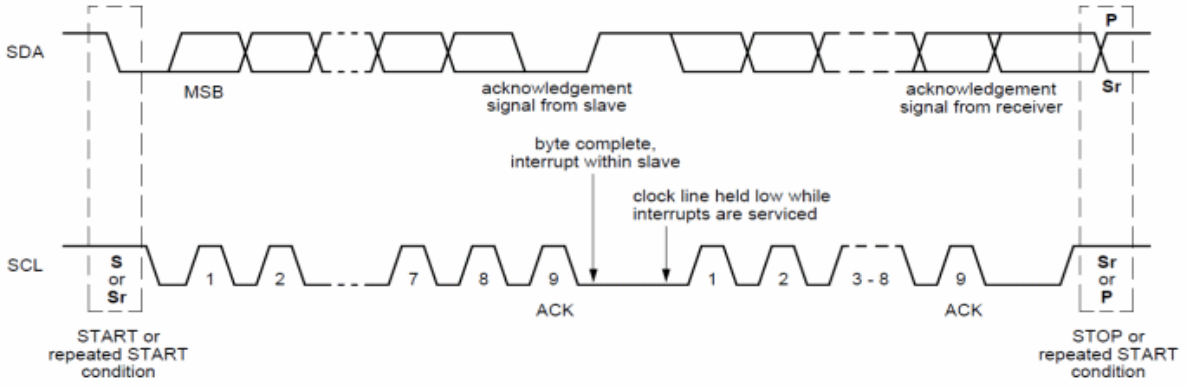


Fig.7: I2C data transfer

Fig.7 shows the data flow for the I2C on both SDA and SCL. Data on the I2C bus is transferred in 8-bit packets (bytes). There is no limitation on the number of bytes, however, each byte must be followed by an Acknowledge bit. This bit signals whether the device is ready to proceed with the next byte. For all data bits including the Acknowledge bit, the master must generate clock pulses. If the slave device does not acknowledge transfer this means that there is no more data or the device is not ready for the transfer yet. The master device must either generate Stop or Repeated Start condition. (see Fig.8 for the timing of the acknowledgment bit).

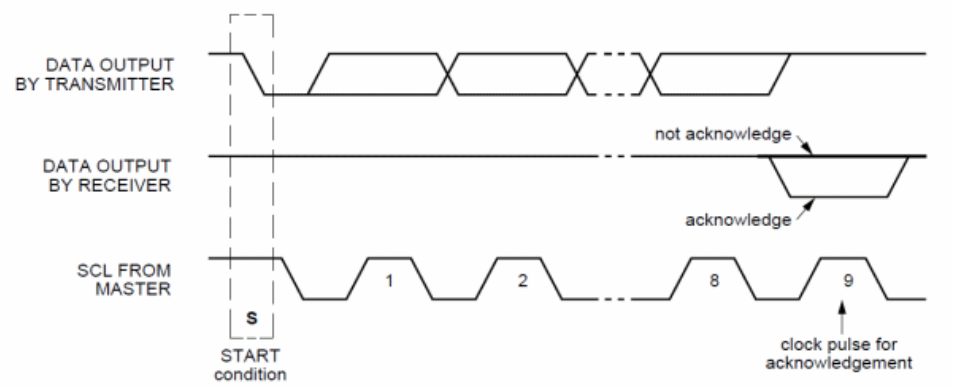


Fig.8: I2C ACK diagram

4. I2C Synchronization:

Each master must generate its own clock signal and the data can change only when the clock is low. For successful bus arbitration a synchronized clock is needed. Once a master pulls the clock low it stays low until all masters put the clock into high state. Similarly, the clock is in the high state until the first master pulls it low. This way by observing the SCL signal, master devices can synchronize their clocks.

5. Arbitration:

For normal data transfer on the I2C bus only one master can be active. If for some reason two masters initiate I2C command at the same time, the arbitration procedure determines which master wins and can continue with the command. Arbitration is performed on the SDA signal while the SCL signal is high. Each master checks if the SDA signal on the bus corresponds to the generated SDA signal. If the SDA signal on the bus is low but it should be high, then this master has lost arbitration. Master I2C device that has lost arbitration can generate SCL pulses until the byte ends and must then release the bus and go into slave mode. The arbitration procedure can continue until all the data is transferred. This means that in multi-master system each I2C master must monitor the I2C bus for collisions and act accordingly.

6. Communication with 7-Bit I2C Addresses:

Each slave device on the bus should have a unique 7-bit address. The communication starts with the Start condition, followed by the 7-bit slave address and the data direction bit. If this bit is 0 then the master will write to the slave device. Otherwise, if the data direction bit is 1, the master will read from slave device. After the slave address and the data direction is sent, the master can continue with reading or writing. The communication is ended with the Stop condition which also signals that the I2C bus is free. If the master needs to communicate with other slaves it can generate a repeated start with another slave address without generation Stop condition. All the bytes are transferred with the MSB bit shifted first. Fig.9 shows the full timing for the I2C protocol signals including the start and stop bits.

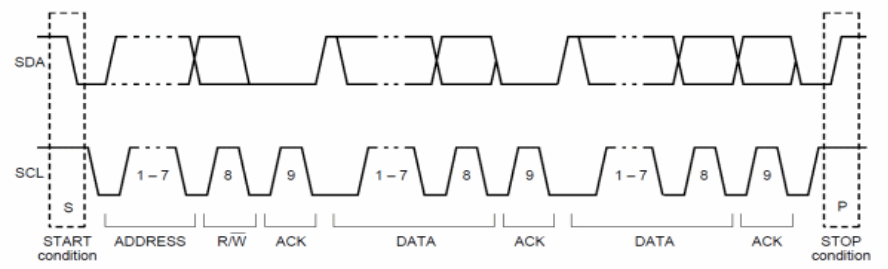


Fig.9: Full timing for the I2C data flow

If the master only writes to the slave device, then the data transfer direction is not changed, as Fig.10 shows.

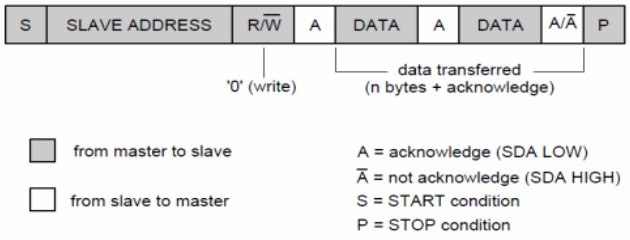


Fig.10: Master writes on slave

If the master only needs to read from the slave device, then it simply sends the I2C address with the R/W bit set to read. After this the master device starts reading the data, as Fig.11 shows.

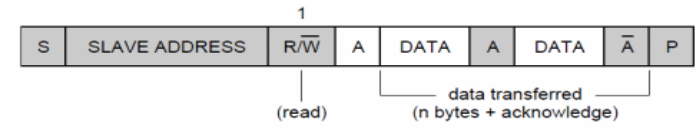


Fig.11: Master reads from slave

Sometimes the master needs to write some data and then read from the slave device. In such cases it must first write to the slave device, change the data transfer direction and then read the device. This means sending the I2C address with the R/W bit set to write and then sending some additional data like register address. After writing is finished the master device generates repeated start condition and sends the I2C address with the R/W bit set to read. After this the data transfer direction is changed and the master device starts reading the data, as Fig.12 shows.

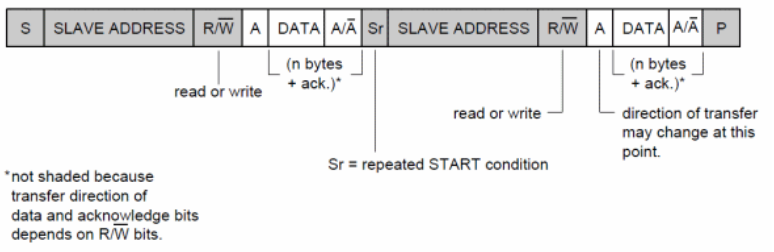


Fig.12: Master reads and writes to the slave device

7.4 Procedure

1. Universal Asynchronous Receiver/Transmitter (UART) Examples & Tasks:

The following parts are different applications on using Serial communication between 2 devices.

Part1: Basic communication between Arduino & PC

The idea of this part is sending a simple information between the Arduino & PC **"Hello World!"** in order to become familiar with the connections & the setup.

I. Components of the project:

a. PC
b. Arduino Board
c. USB Cable

II. Circuit



Fig.13: Arduino-PC Serial Communication

III. Code

The following code will receive data from the PC and then write it back to the serial port.

Code Snippet 1: Arduino-PC Serial Communication Code

```
/* Use a variable called byteRead to temporarily store
   the data coming from the computer */
byte byteRead;

void setup() {
// Turn the Serial Protocol ON
  Serial.begin(9600);
}

void loop() {
```

```
   /*  check if data has been sent from the computer: */
  if (Serial.available()) {
    /* read the most recent byte */
    byteRead = Serial.read();
    /*ECHO the value that was read, back to the serial port. */
    Serial.write(byteRead);
  }
}
```

IV. Download & Run

- Connect the Arduino with the PC using the USB cable and then download the code.
- Open the serial monitor.



Fig.14: Serial Monitor in Arduino IDE

- Send a value to Arduino as seen in the following screenshot:

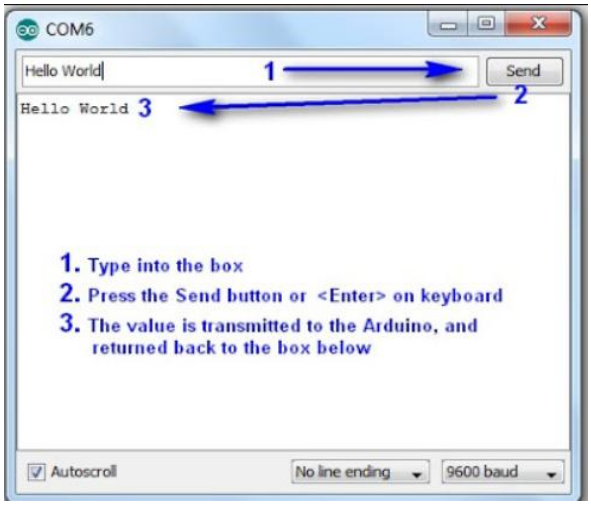


Fig.15: Send Data to the Arduino via USB

Part2: Basic communication between 2 Arduinos

The idea of this part is sending a simple information between the 2 Arduinos [“Hello World!”] in order to become familiar with the connections & the setup.

I. Circuit

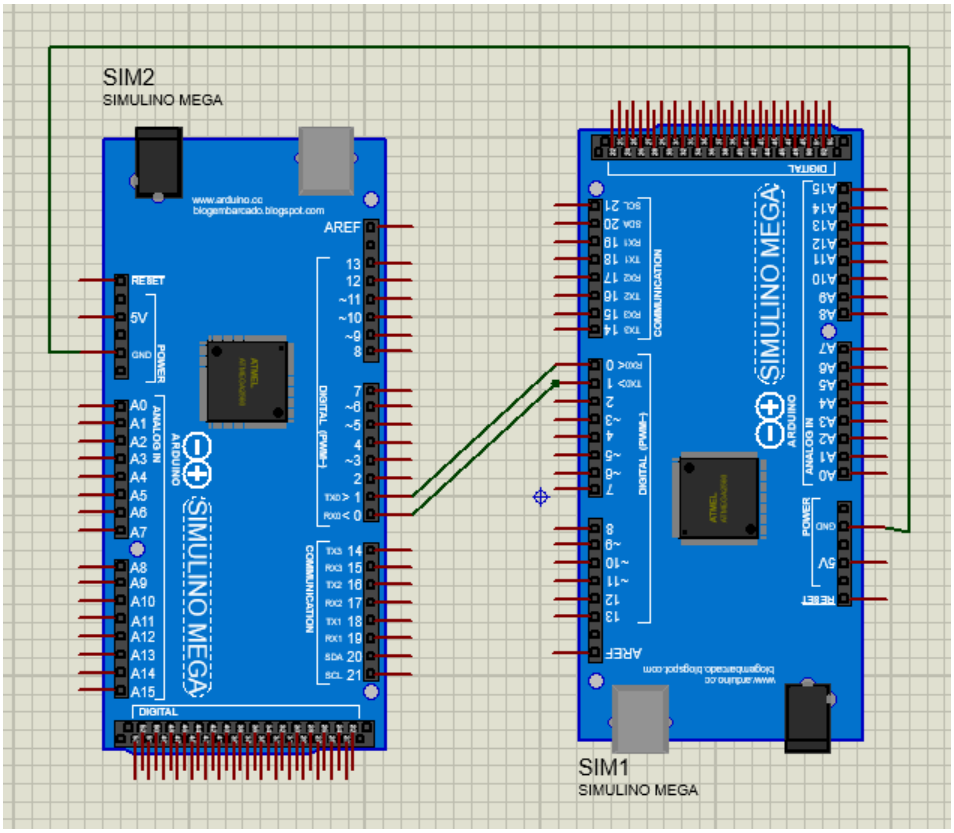


Fig. 16: Connecting 2 Arduinos using TX & RX

We connect **TX** in the First Arduino with **RX** in the second one and Vice versa [$\mathbf{TX_1 \leftrightarrow RX_2}$ **&** $\mathbf{RX_1 \leftrightarrow TX_2}$]. The ground must be common between both of them.

II. Code

Since we are dealing with 2 Arduinos, a code for each is needed.

Sender Code:

The sender will be responsible of sending “Hello World!” to the receiver Arduino. First, we should specify the serial communication setup [Baud rate, stop bits, …] and then send!

Code Snippet 2: Sender Code

```
void setup() {
  // Setup the Serial at 9600 Baud
  Serial.begin(9600);
}

void loop() {
  Serial.println("Hello World!"); //Write the serial data
  delay(1000);
}
```

Receiver Code:

The receiver will be responsible of receiving the data and then displaying it. First, we should specify the serial communication setup [Baud rate, stop bits, …] and then read the data!

Code Snippet 3: Receiver Code

```
void setup() {
  // Begin the Serial at 9600 Baud
  Serial.begin(9600);
}

void loop() {
   if (Serial.available()) {
    String data = Serial.readString();//Read the serial data and store in
var
    Serial.println(data);//Print data on Serial Monitor
  }
}
```

III. Download & Run

1. Download Both codes on the Transmitter & Receiver Arduinos.

Note: If it shows error while uploading code to Arduino. Then, disconnect the connection between Arduino and try to upload program on it.

2. Open Serial monitor on the receiver and observe the printed data.

Part3: Push Button & LED using 2 Arduinos

The idea of this part is to connect a push button with **Arduino1** and whenever it's pushed, the value **1** is sent to the **Arduino2**. In its part, **Arduino2** receives the data and whenever logic **1** is read, a LED will be turned on.

I. Components:

a. Arduino Mega2560 Board X 2
b. Push Button
c. LED
d. Resistance X 2

II. Circuit

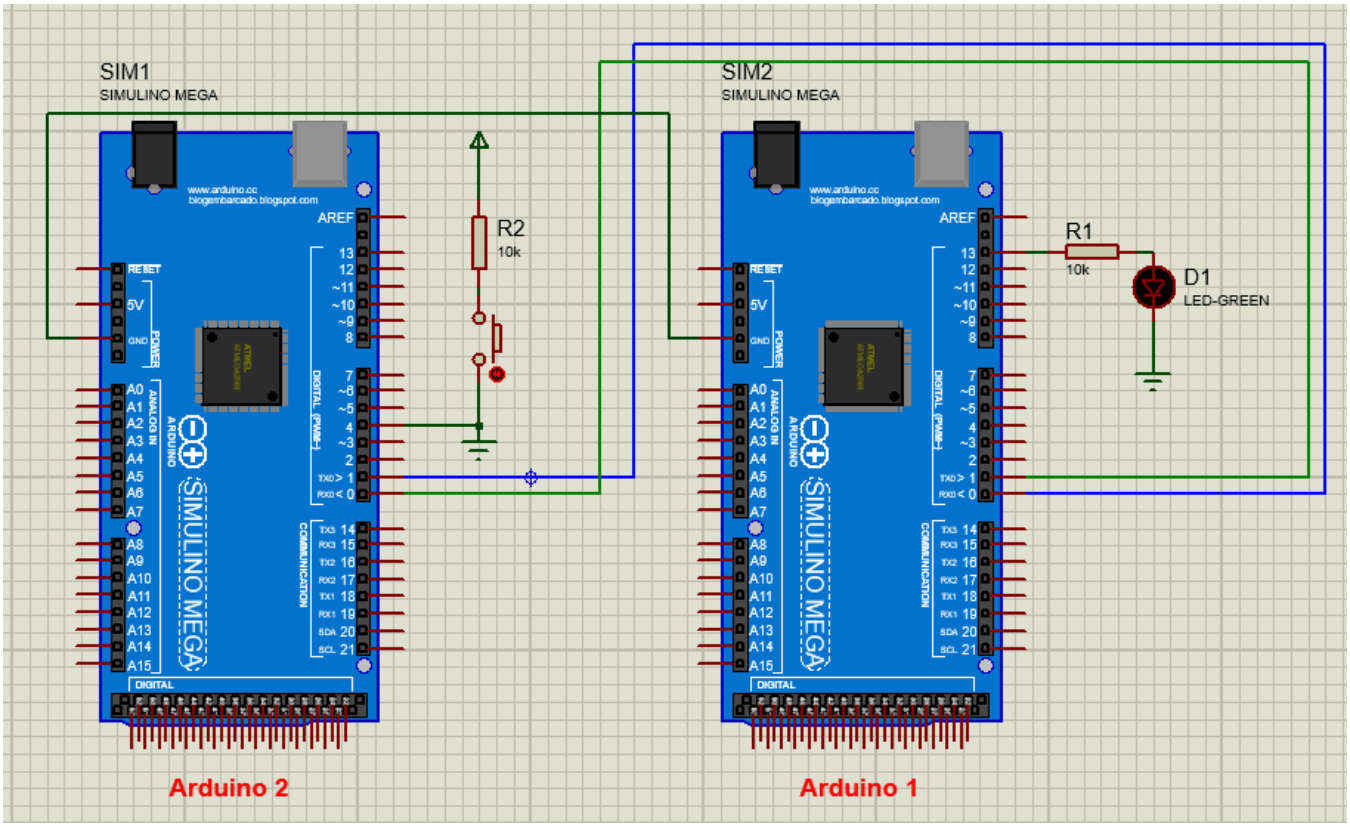


Fig. 17: Complete Circuit Connection

Note: We connect **TX** in the First Arduino with **RX** in the second one and Vice versa [**TX_1 ↔ RX_2 & RX_1 ↔ TX_2**]. The ground must be common between both of them.

III. Code

Todo1: Complete the following code snippets, then download and run it on the previous circuit.

Sender Code:

Arduino1 should send 1 to **Arduino2** whenever the push button is pushed. If not pushed, nothing should be sent.

Code Snippet 4: Sender Code

```
void setup() {
  // TODO[1]: Setup
the Serial at 9600 Baud

}
void loop() {
  // TODO[2]: Check if the push button is pushed & Send 1 if pushed

}
```

Receiver Code:

Arduino2 should receive 1 from **Arduino1** whenever the push button is pushed. In this case, the Led must be turned on.

Code Snippet 5: Receiver Code

```
void setup() {
  // TODO[1]: Setup the Serial at 9600 Baud

}
void loop() {
/* Check if data has been sent from the computer: */
  if (Serial.available()) {
  // TODO[2]: Check if 1 is sent, and turn on the LED if yes.

  }
}
```

IV. Download & run

1. Download both codes on the Transmitter & Receiver Arduinos.
2. Open the serial monitor to make sure that the receiver sends the data correctly.

2. I2C Examples & Tasks:

Part 1.1: Writing the LCD to Arduino and I2C scanning

This display uses I2C communication, which makes wiring really simple.

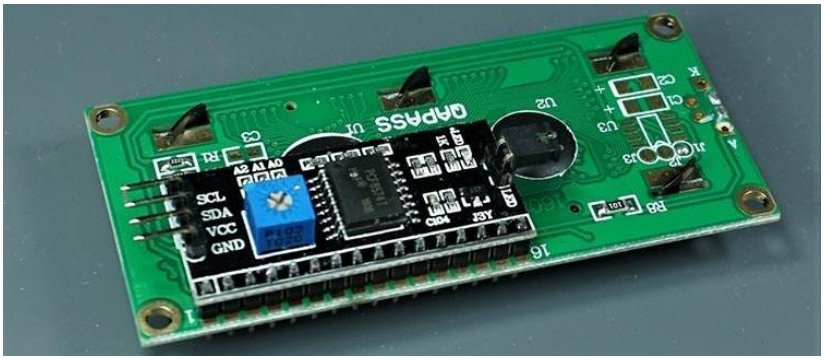


Fig.19: I2C module interface LCD pins to I2c protocol

I. Components:

a. Arduino Mega2560
b. LCD

II. Circuit

Wire your LCD to the Arduino by following the schematic diagram in Fig.20. We're using the Arduino default I2C pins (SDA and SCL).

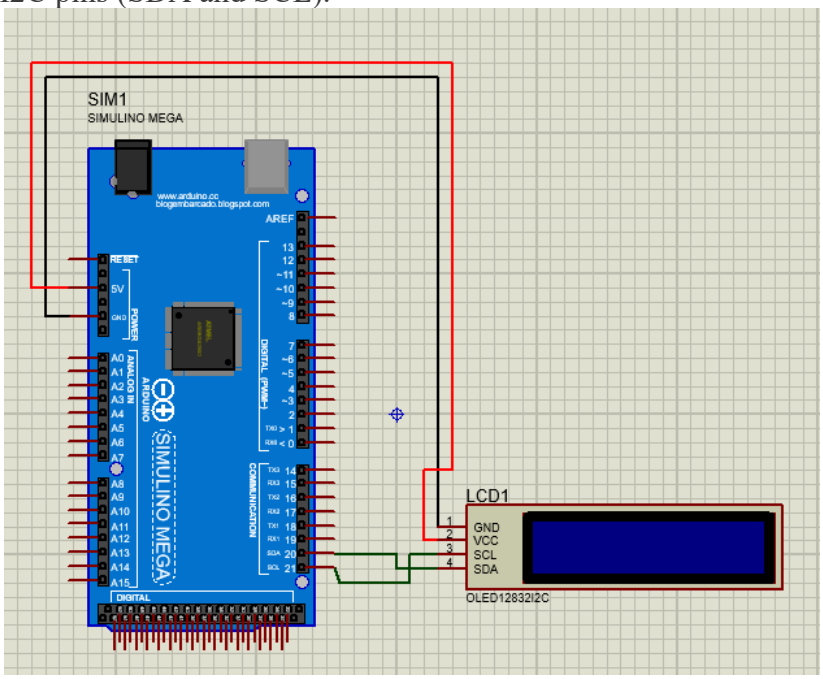


Fig.20: I2C module connection between Arduino and LCD

You can also use the following table as a reference.

I2C LCD	Arduino
GND	GND
VCC	5V
SDA	SDA (D20)
SCL	SCL (D21)

Preparing the Arduino IDE

Before proceeding with the project, you need to install the Arduino add-on in the Arduino IDE. Follow one of the next guides to prepare your Arduino to use the LCD functions:

Installing the LiquidCrystal_I2C Library

There are several libraries that work with the I2C LCD. We're using (https://github.com/marcoschwartz/LiquidCrystal_I2C) library by Marco Schwartz. Follow the next steps to install the library:

1. Click (https://github.com/marcoschwartz/LiquidCrystal_I2C/archive/master.zip) to download the **LiquidCrystal_I2C** library. You should have a *.zip* folder in your Downloads
2. Unzip the *.zip* folder and you should get **LiquidCrystal_I2C-master** folder.
3. Rename your folder from **LiquidCrystal_I2C-master** to **LiquidCrystal_I2C**.
4. Move the **LiquidCrystal_I2C** folder to your Arduino IDE installation **libraries** folder.
5. Finally, re-open your Arduino IDE.

Getting the LCD Address (Address scanning)

1. Before displaying text on the LCD, you need to find the LCD I2C address. With the LCD properly wired to the Arduino, upload the following I2C Scanner sketch.

Code Snippet 6: I2C Scanner

```
#include <Wire.h>

void setup() {
  Wire.begin();
  Serial.begin(9600);
  Serial.println("\nI2C Scanner");
}
```

```
void loop() {
  byte error, address;
  int nDevices;
  Serial.println("Scanning...");
  nDevices = 0;
  for(address = 1; address < 127; address++ ) {
    Wire.beginTransmission(address);
    error = Wire.endTransmission();
    if (error == 0) {
      Serial.print("I2C device found at address 0x");
      if (address<16) {
        Serial.print("0");
      }
      Serial.println(address,HEX);
      nDevices++;
    }
    else if (error==4) {
      Serial.print("Unknow error at address 0x");
      if (address<16) {
        Serial.print("0");
      }
      Serial.println(address,HEX);
    }
  }
  if (nDevices == 0) {
    Serial.println("No I2C devices found\n");
  }
  else {
    Serial.println("done\n");
  }
  delay(5000);
}
```

---

2. After uploading the code, open the Serial Monitor at a baud rate of 9600. Press the Arduino reset button. The I2C address should be displayed in the Serial Monitor, as Fig.21 shows:

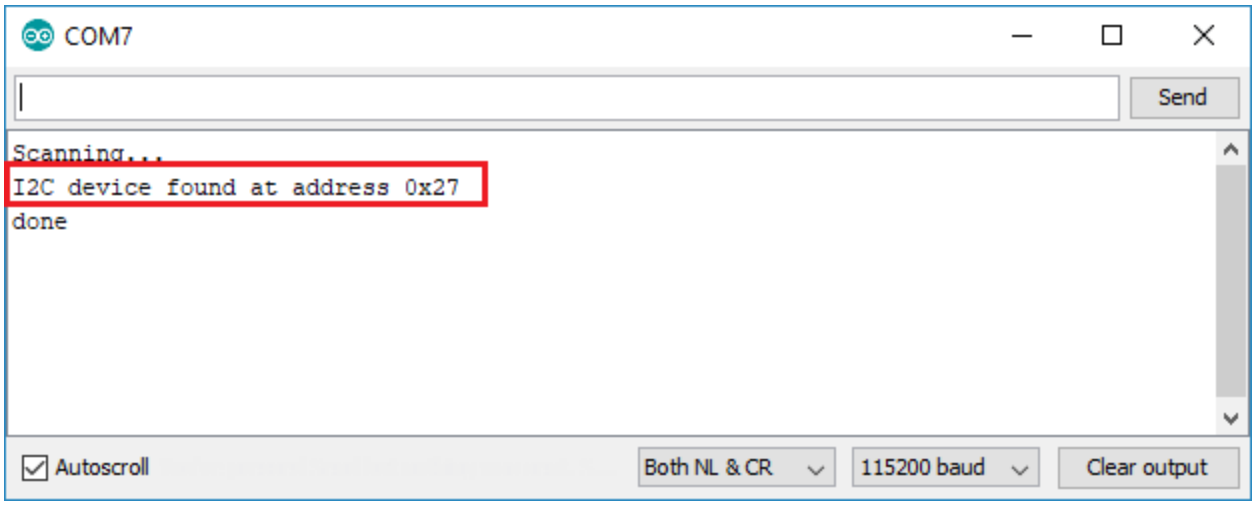


Fig.21: I2C Scanner Output

In this case the address is **0x27, but it could be any other address**. Write down the address of your I2C module.

Part 1.2: Display Static Text on the LCD

Displaying static text on the LCD is very simple. All you have to do is select where you want the characters to be displayed on the screen, and then send the message to the display. Here's a very simple sketch example that displays "Welcome TO BZU ".

Code Snippet 7: Display Text on the LCD Example

```
#include <LiquidCrystal_I2C.h>

// set the LCD number of columns and rows
int lcdColumns = 16;
int lcdRows = 2;

// set LCD address, number of columns and rows
// if you don't know your display address, run an I2C scanner sketch
LiquidCrystal_I2C lcd(0x27, lcdColumns, lcdRows);

void setup(){
  // initialize LCD
  lcd.init();
  // turn on LCD backlight
  lcd.backlight();
}
```

```
void loop(){
  // set cursor to first column, first row
  lcd.setCursor(0, 0);
  // print message
  lcd.print("Welcome TO");
  delay(1000);
  // clears the display to print new message
  lcd.clear();
  // set cursor to first column, second row
  lcd.setCursor(0,1);
  lcd.print("BZU”);
  delay(1000);
  lcd.clear();
}
```

Replace the display address in the above code which is x27 with the address you got after running the I2C scanner sketch.

The previous example displays the message in the first row, and then in the second row. In this simple code we show you the most useful and important functions from the LiquidCrystal_I2C library. So, let's take a quick look at how the code works.

How the code works

First, you need to include the LiquidCrystal_I2C library.

```
#include <LiquidCrystal_I2C.h>
```

The next two lines set the number of columns and rows of your LCD display. If you're using a display with another size, you should modify those variables.

```
int lcdColumns = 16;
int lcdRows = 2;
```

Then, you need to set the display address, the number of columns and number of rows. You should use the display address you've found in the previous step.

```
LiquidCrystal_I2C lcd(0x27, lcdColumns, lcdRows);
```

In the ***setup()***, first initialize the display with the ***init()*** method.

```
lcd.init();
```

Then, turn on the LCD backlight, so that you're able to read the characters on the display.

```
lcd.backlight();
```

To display a message on the screen, first you need to set the cursor to where you want your message to be written. The following line sets the cursor to the first column, first row.

```
lcd.setCursor(0, 0);
```

Note: (0, 0) corresponds to the first row, first column, (0, 1) to the first row, second column, and so on.

Then, you can finally print your message on the display using the ***print()*** method.

```
lcd.print("Welcome TO");
```

Wait one second, and then clean the display with the ***clear()*** method.

```
lcd.clear();
```

After that, set the cursor to a new position: first column, second row.

```
lcd.setCursor(0,1);
```

Then, the process is repeated.

So, here's a summary of the functions to manipulate and write on the display:

- lcd.init(): initializes the display.
- lcd.backlight(): turns the LCD backlight on.
- lcd.setCursor(int column, int row): sets the cursor to the specified column and row.
- lcd.print(message): displays the message or variable on the display.
- lcd.clear(): clears the display.

This example works well to display static text no longer than 16 characters.

Part 2.1: I2C communication between 2 Arduinos

Fig.22 show you how to connect 2 devices to use I2C communication, in this case we set one of them as master and other as slave.

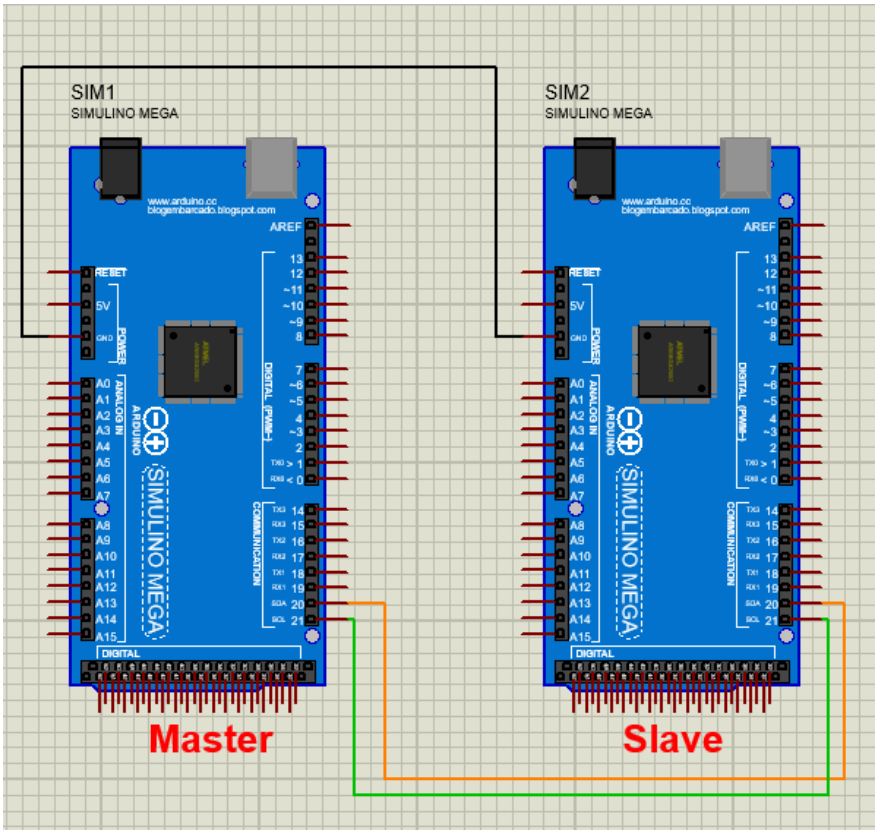


Fig.22: I2C Connections

Our example will send data from slave to master. The data is characters starting from '0' and ending with 'z'.

To run the codes:

1. Do the schematic in Fig.22. (make sure to make the GND common).
2. Compile and upload the master code to the first Arduino.
3. Compile and upload the slave code to the second Arduino.
4. Open the serial monitor for the master (make sure to change it since the monitor could be on the slave Arduino).

Master code:

Code Snippet 8: Master Code

```
// master side code
#include <Wire.h>
void setup()
{
 Wire.begin();         // join i2c bus (address optional for master)
 Serial.begin(9600);  // start serial for output
 Serial.print("master sleeping...");
```

```
 delay(2000);
 Serial.println("go");
}

void loop()
{
 Wire.requestFrom(2, 1);    // request data from slave device #2

 while(Wire.available())
 {
   char c = Wire.read(); // receive a byte as character
   Serial.print(c);         // print the character
 }

 delay(100);
}
```

Slave code:

Code Snippet 9: Slave Code

```
// slave side code
#include <Wire.h
void setup()
{
 Wire.begin(2);                // join i2c bus with address #2
 Wire.onRequest(requestEvent); // register event
 Serial.begin(9600);           // start serial for output
}

void loop()
{
 delay(100);
}

// function that executes whenever data is received from master
// this function is registered as an event, see setup()
void requestEvent()
{
 static char c = '0';

 Wire.write(c++);
 if (c > 'z')
```

```
    c = '0';
}
```

Part 2.2: I2C communication between 2 Arduinos and LCD

In this part we want to read analog value from slave Arduino and write the value on LCD connected to Arduino master device:

Steps:

1. Do the schematic in Fig.23.
2. To connect LCD to the master, flip the Arduino and try to find SCL and SDA written on the back of some Arduino pins. Connect them to LCD.
3. Connect a potentiometer to the slave device on A0.
4. Compile and upload the master code to the first Arduino.
5. Compile and upload the slave code to the second Arduino.
6. Open the serial monitor for the master (make sure to change it since the monitor could be on the slave Arduino).

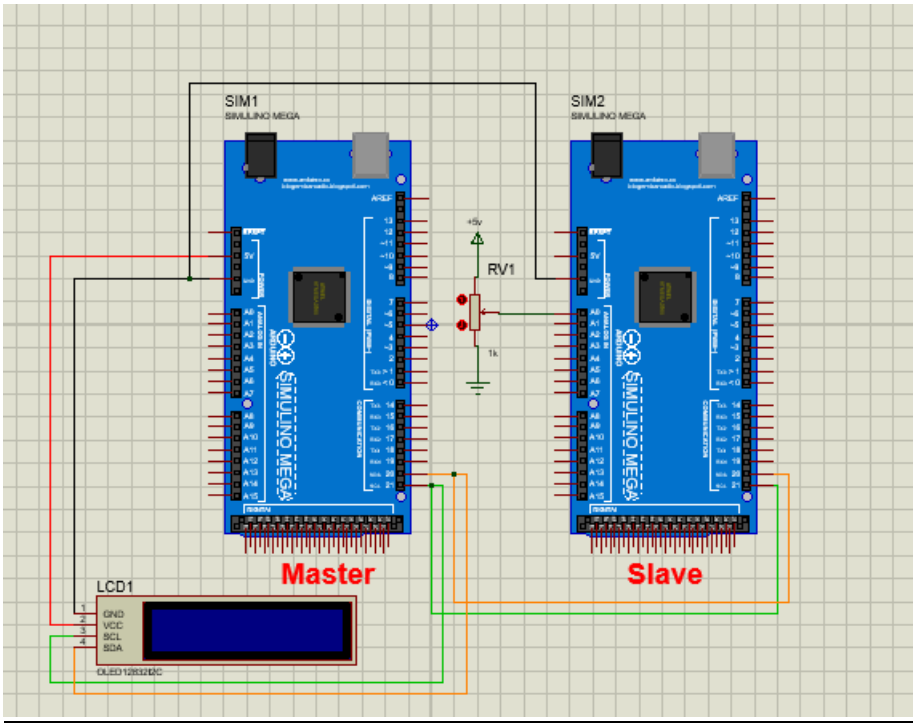


Fig.23: I2C Connections

Master code:

Code Snippet 10: Master Code

```
// master side code
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

// set the LCD number of columns and rows
int lcdColumns = 16;
int lcdRows = 2;

// set LCD address, number of columns and rows
// if you don't know your display address, run an I2C scanner sketch
LiquidCrystal_I2C lcd(0x27, lcdColumns, lcdRows);

void setup()
{
  Wire.begin();     // join i2c bus (address optional formaster)
  Serial.begin(9600);  // start serial for output
  Serial.print("master sleeping...");
  delay(2000);
  Serial.println("go");
  // initialize LCD
  lcd.init();
  // turn on LCD backlight
  lcd.backlight();
}

void loop()
{
  Wire.requestFrom(50, 2);    // request data from slave device #2
  int res;//result from I2c reading
  byte MSB = Wire.read(); /* receive a byte MSB(NOTE: the function is
blocking, so it would not continue the code until a byte reach)*/
  byte LSB = Wire.read(); /* receive a byte LSB(NOTE: the function is
blocking, so it would not continue the code until a byte reach)*/
  res = ((MSB << 8) | LSB);
  Serial.print("MSB = ");
  Serial.print(MSB);
  Serial.print(" , LSB = ");
  Serial.print(LSB);
  Serial.print(" , analog value = ");
```

```
  Serial.println(res);         // print the analog value

  // set cursor to first column, first row
  lcd.setCursor(0, 0);

  // print message
  lcd.print("Reading = ");
  lcd.print(res);
  delay(1000);
  lcd.clear();

}
```

Slave code:

Code Snippet 11: Slave Code

```
// slave side code
#include <Wire.h>
void setup()
{
 Wire.begin(50);                 // join i2c bus with address #2
 Wire.onRequest(requestEvent); // register event
 Serial.begin(9600);           // start serial for output
}

void loop()
{
 delay(100);
}

// function that executes whenever data is received from master
// this function is registered as an event, see setup()
void requestEvent()
{
  int value = analogRead(A0);
  Wire.write(value >> 8); // send the MSB
  Wire.write(value & 0x00ff); // send the LSB
}
```

BIRZEIT UNIVERSITY

Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

Microprocessor and Embedded Systems Lab (ENCS5101)

Experiment No. 8 – LabVIEW based applications

8.1 Objectives

1. Increasing familiarity and experimenting with Lab-View by implementing different systems by using LabVIEW, and by trying different components in it.
2. Learn how to interact with your system by using LINX in LabVIEW with Arduino.

8.2 Overview

8.2.1 Introduction to LabVIEW:

LabVIEW (Laboratory Virtual Instrument Engineering Workbench) is a graphical programming environment. So it relies on graphical components to implement different systems and different programing scripts. There are many applications use the Lab-View in different fields: science, education and industry. In this Experiment, the students will write programs to become familiar with some of the basics of LabVIEW.

8.2.2 What is LINX Toolkit?

LINX is an open source project by Digilent and is designed to make it easy to develop embedded applications using LabVIEW. LINX includes VIs for over 30 of the most common embedded sensors as well as hardware agnostic APIs for accessing peripherals like digital I/O, analog I/O, PWM, I2C, SPI, and UART. By using LINX toolkit, developers can acquire data from microcontroller and process it in the LabVIEW Graphical Programming environment.

8.2.3 Why LINX?

There are many reasons LabVIEW makes you more productive when using Arduino:

- Interact with your system through a graphical user interface.
- Streamline your design process with intuitive graphical programming.
- Improve your debugging experience with interactive tools.
- Leverage built in resources/functions for implementing simple to complex tasks.
- Open API allows for complete customization (customize your programs to fit your application).

8.3 Procedure

8.3.1 Design a Simple Alarm System:

We need to design a control system with the following specifications:

1. The system has two inputs: pressure and temperature.
2. There is an alarm when the temperature exceeds 100° C or the pressure exceeds 15KPa.
3. The transfer function of the temperature and pressure transducers are 3.2mV/ °C, 0.3V/KPa respectively.

This example is very simple, try to draw a simple design before starting with LabVIEW, and try to recognize the required components of the system.

The following steps describe how to assemble a VI that implements Simple Alarm System. When this VI is completed, you will be able to test the functionality of the system. Inspect Figure 1 and Figure 2 which represent the front panel and block diagram of the system.

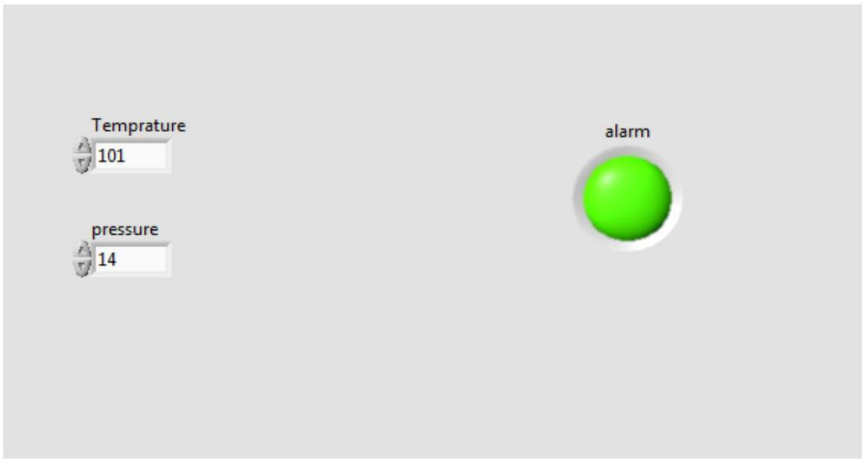


Figure 1: Simple Alarm System VI Front Panel

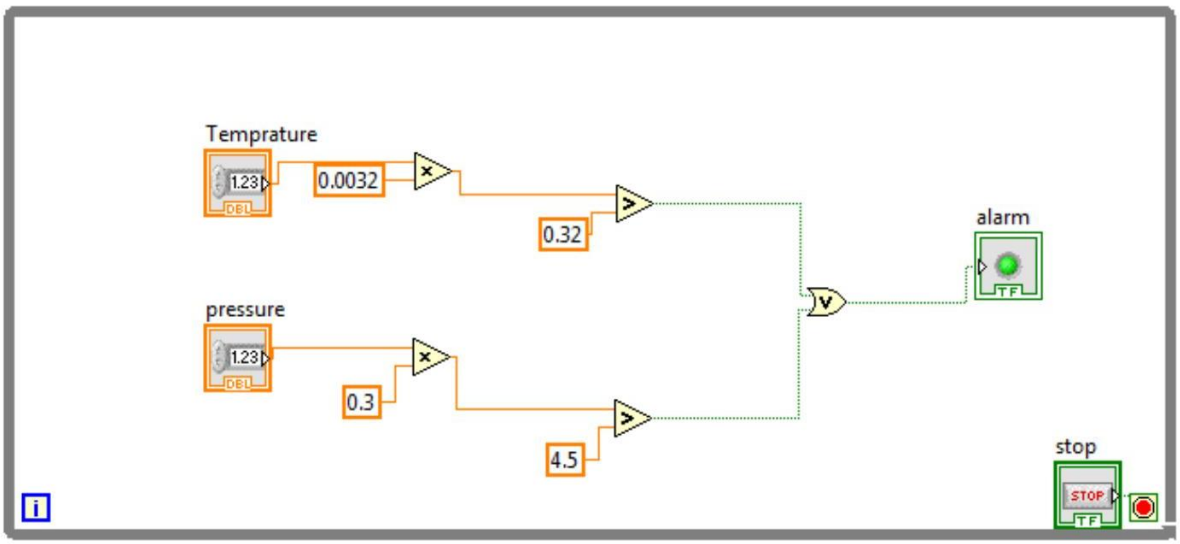
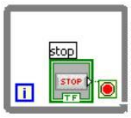


Figure 2: Simple Alarm System VI Block Diagram

The block diagram consists of a while loop which contains the components that change their values while the program is running.

To find any component in the LABVIEW, simply make a right click, the you can search on it. Also to find the components of this system, you can follow these steps.

1. On the block diagram, choose from the Functions palette>Express>Exec Control>while loop. Expand the while loop block to proper size. Note that a stop button will appear on the front panel which stops the execution of the whole VI when pressed. (To obtain the Functions palette, right click on any blank space in the block diagram window). All controls and indicators that appears on the block diagram window should be placed in the while loop.



2. On the front panel, let us build the VI block diagram shown in Figure 1 above. First, place a led and rename it as required (Control palette>Express>lEDs).



3. Then, place two "Num Ctrl" VI. Rename and resize them as required (Control palette>Express>Num Ctrls).



4. Up to this point, the front panel should look like Figure 1.
5. On block diagram, drag and drop all the blocks into the while loop.



6. Place a "Multiply" from Functions>Mathematics>Numeric VI on the block diagram. Also wire a constant value by right click on the component.
7. Place a "Comparator" VI (Function>Express>Arithmetic&Comparison) on the block diagram and wire the output of the multiply function from the previous step into the comparator input. Also wire a constant value by a right click on the component.



Exercise: Simple Liquid Store System

Design a simple liquid store system in the LABVIEW with the following specifications:

1. The system has four inputs (volume of the liquid, temperature of the liquid and two enables) and two outputs (led and screen).
2. When the volume of the liquid exceeds 6 liters and led enable is on, the led will turn into Red color.
3. When the temperature of the liquid exceeds 60° C and temperature enable is on, the message "The temperature of the liquid is high" should be displayed on the screen.

The front panel of the system should be like Figure 3 and Figure 4.

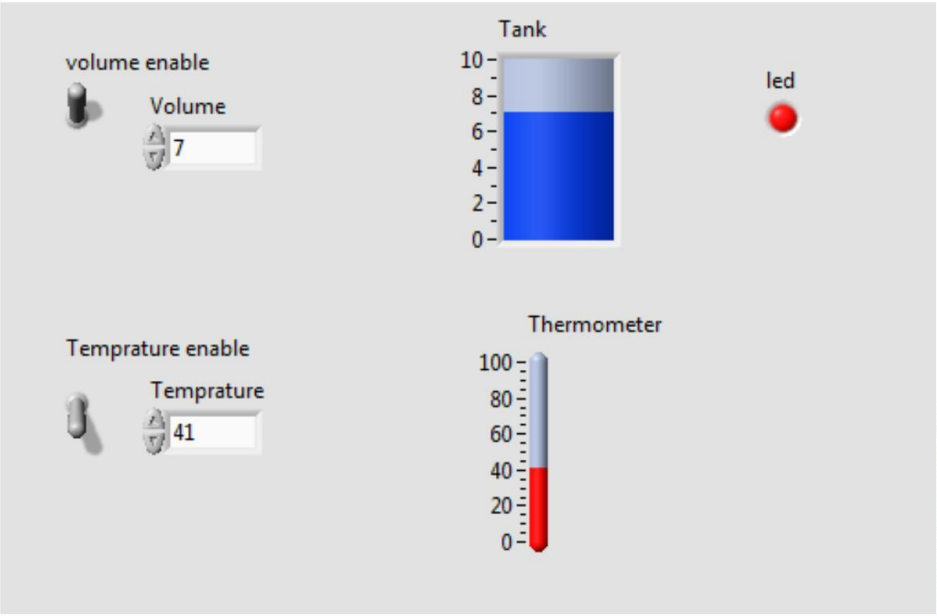


Figure 3: front panel of the system (condition #2)

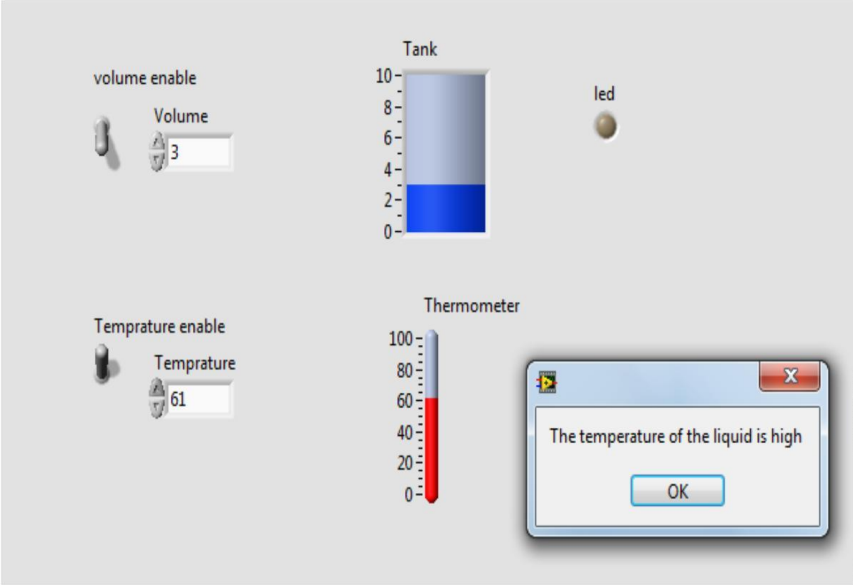


Figure 4: front panel of the system (condition #3)

8.3.2 Arduino with LabVIEW:

In the previous, there was a toolkit called LIFA (LabVIEW Interface for Arduino). The author of LIFA decided to expand the interface to more devices (and make other improvements). So, a few years ago, LIFA was replaced by LINX ("LabVIEW INterface for X"; where X stands for the various devices that it supports like Arduino, chipKIT, etc.). In addition, LINX is solved many problems that were in LIFA, so there were significant communication efficiency improvements because the protocol was redesigned so most of the functions that communicate with the device are much more efficient.

Step by Step Startup:

Here is a step by step process to get up and running with Arduino and LabVIEW:

1. Install the LINX. The LINX is available as a VI package through the LabVIEW Tools Network. You must first install VIPM from the following link (http://jki.net/vipm/download). Then download the LINX package as in Figure 5.

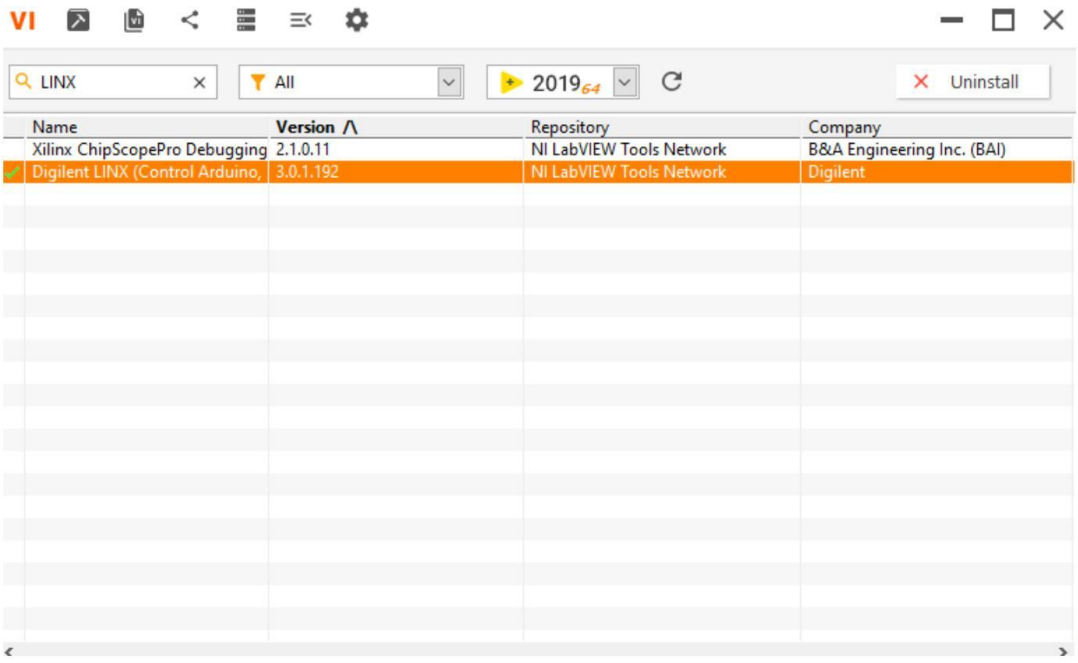


Figure 5: Install LINX

After select the Digilent LINX press the install icon.

2. Install NI-VISA Drivers. To LabVIEW, the Arduino appears as a serial instrument device. To communicate with serial instruments in LabVIEW, you need to have the latest version of the NI-VISA driver. You can get the latest NI_VISA driver from "http://search.ni.com/nisearch/app/main/p/bot/no/ap/tech/lang/en/pg/1/sn/catnav:du,n8:3.1637,ssnav:sup ".
3. Use the following steps to connect LabView with Arduino:
   - Go to Tools → MakerHub → LINX → LINX Firmward Wizard.
   - Then select the following:

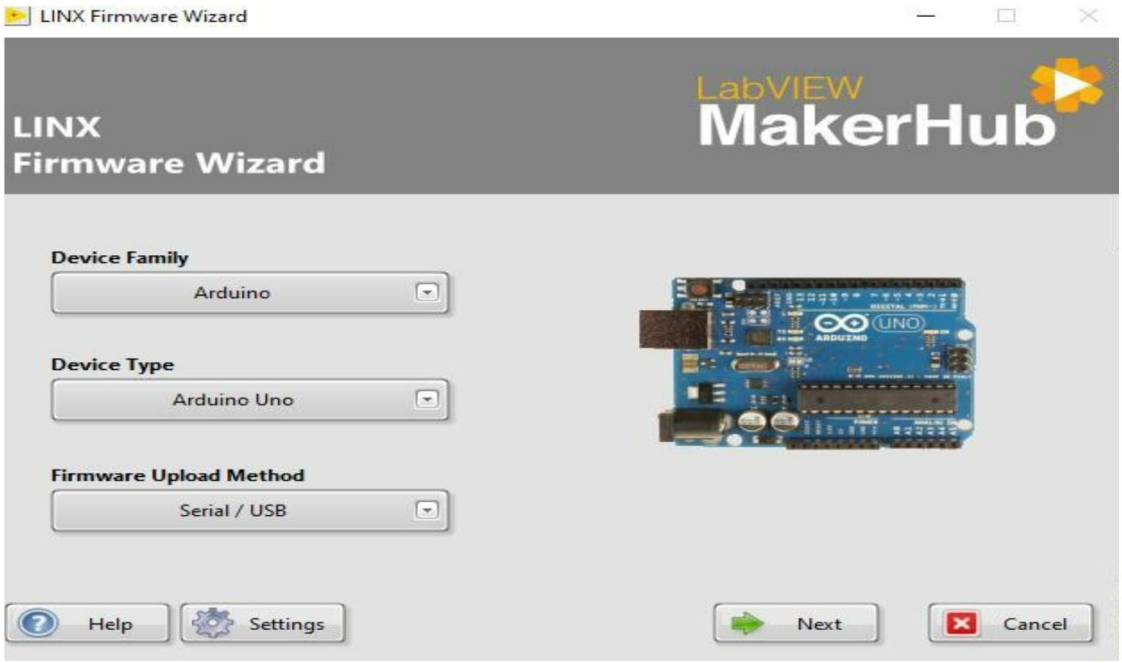


Figure 6: Define the device type

Then, click Next and select Arduino port:

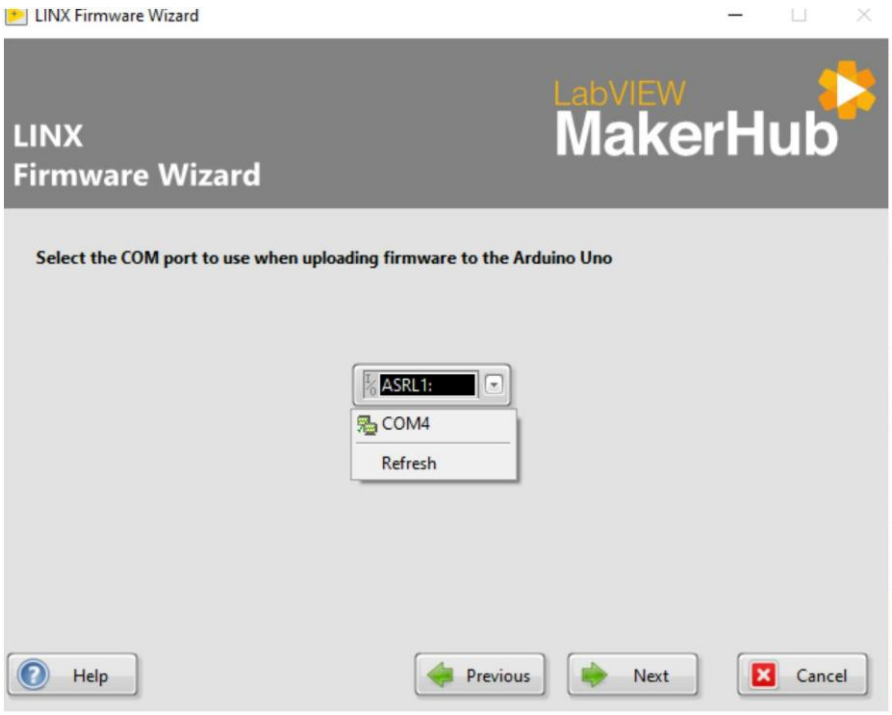


Figure 7: Specify the port

Then, click Next:

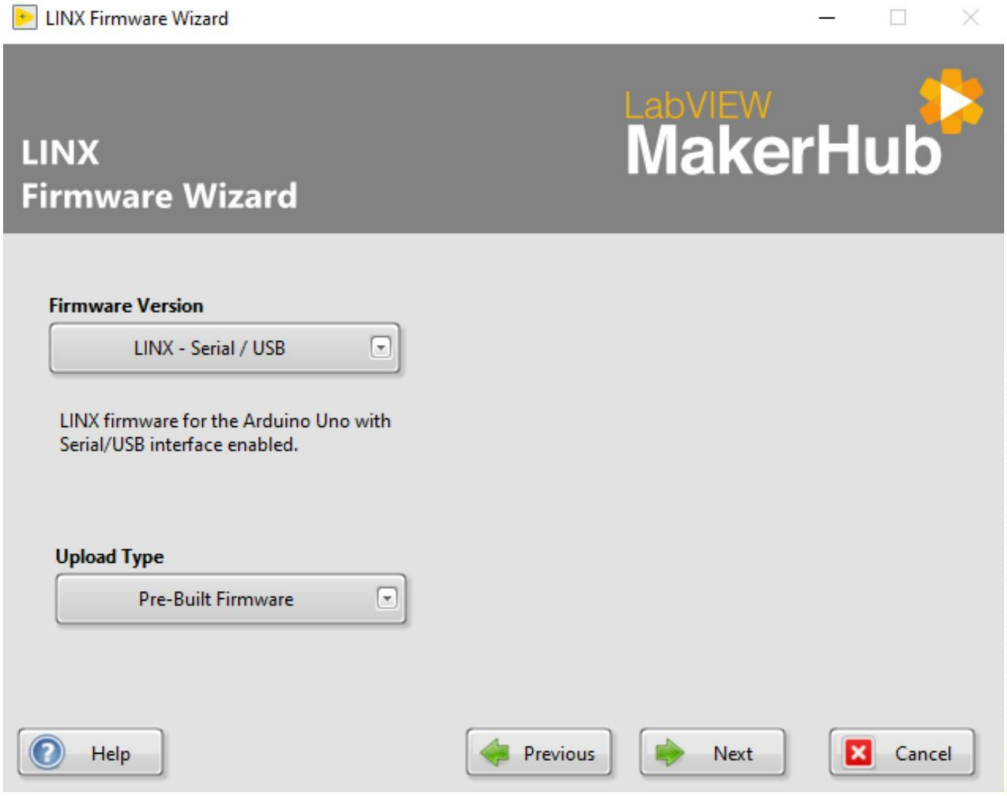


Figure 8: Firmware version

After that LINX will start to make the connection. Once this step done, press finish and now you are ready to use Arduino with LabVIEW.

8.3.2.1 Control LED 13 in Arduino by LabVIEW:

In this example, we will control LED 13 in Arduino by using LabVIEW, so as shown in Figure 9, we used a digital write box that has three inputs, the first input comes from init box, this input used to know that the initialization done successfully, the second input is a channel number, in other word, the digital port that we need to write on it, and the last input is a switch in order to control the led.

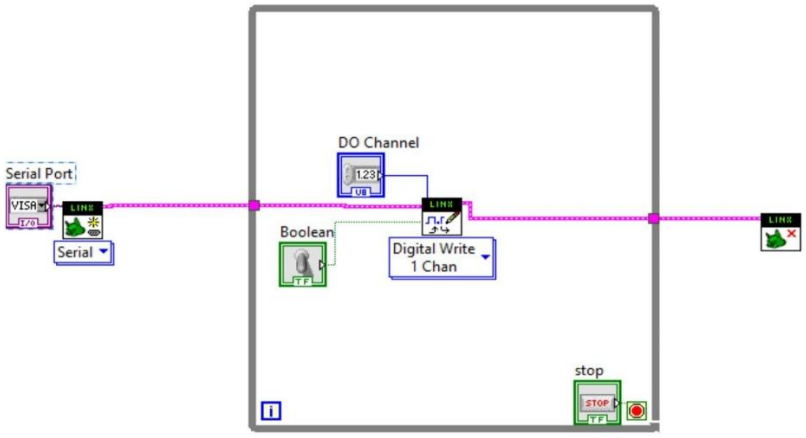


Figure 9: Control LED 13 example

The front-end side will look like this:

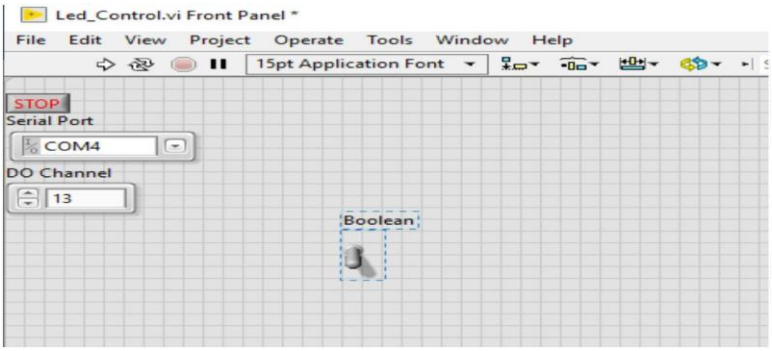


Figure 10: Front end side for the example

8.3.2.2 Array example:

In this example you will use two binary arrays in as indicator and the other as control.

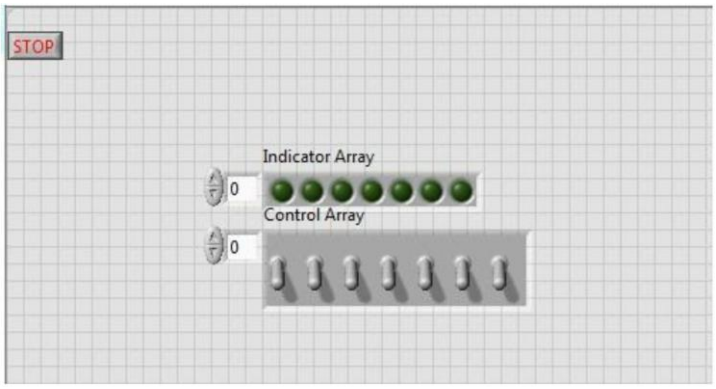


Figure 11: Array example

- To add array: Front panel >> Modern >> Array, Matrix >> array.
- To make it a control binary array drags and drops a switch in it.
- To make it an indicator binary array drags and drops a switch in it.
- To resize the array graphically.

8.3.2.3 Read analog value (LDR):

Build the following LDR voltage divider circuit in to read its output voltage using Arduino and print it out on the LabVIEW.

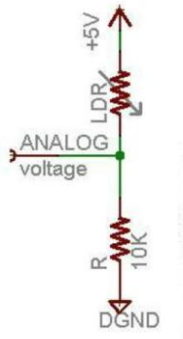


Figure 12: LDR voltage divider

Connect the output voltage of the LDR voltage divider circuit to analog bin 0 of Arduino. Then build the following example.

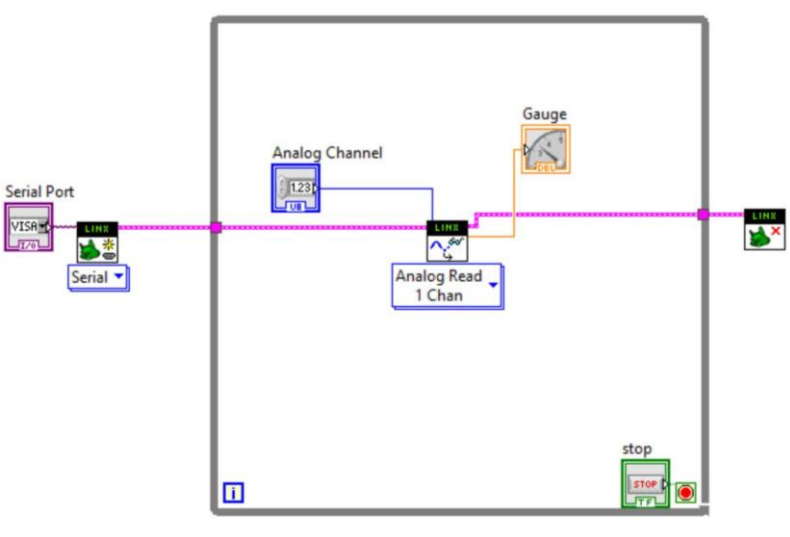


Figure 13: Read example

To add the "init, close, and read analog pin" block diagram >>LINX.

Exercise: Lights detection security key (LDSK)

In this part, you have to build a 12 bits' light detection security key using a LDR with Arduino and LabVIEW, where you will insert the password using the LDR if it detects a light it will be a one otherwise it will be zero.

- Use for loop to get the data from Arduino and store them in binary array.
- One bit every second.
- System needs a trigger to start reading. (you can use an Ok button with case structure).
- If the password is correct, then a green light is on. Else a red one is on.



Faculty of Engineering and Technology

Electrical and Computer Engineering Department

Microprocessor and Embedded Systems Lab (ENCS5101)

EXP. No. 9. Elevator Control System

9.1 Objectives

1. To understand the construction and operating principle of the elevator.
2. To design and implement a prototype of an elevator control system using Arduino microcontroller and other electrical elements.
3. To learn the numerical control operation principles, the interfacing techniques of DC motor, and the interfacing techniques of photo interrupters.
4. To understand the mechanical design related to this system (i.e. motor, sensors, terminals, motor driver, and others).
5. To be familiar with the Arduino IDE and C programming language.

9.2 Equipment Required

1. Arduino Mega.
2. MDA-007 Kit.
   a) Lift power; DC motor
   b) Position sensor; photo interrupter
   c) Control system; DC motor and shaft
   d) Power; DC 5V 3A, 12V 1A
   e) Switch; Limit switch x 2, push button switch x 4
3. Dupont cables.

9.3 Pre-Lab

Prepare all tasks and work on all required designs.

9.4 Overview

9.4.1 Elevator System

The elevator is a vertical transport system that efficiently transfers people or goods between floors of a building. They are generally powered by electric motors and work just like a pulley system with a developed mechanism. They are used in apartments, malls, also in areas of agriculture, manufacturing etc. The elevator as a control system has a number of components, these can basically be divided into the following:

1. Inputs:
   A- Sensors: photo electric position sensor, infrared sensor, weight sensor.
   B- Buttons: hall buttons, floor request buttons, open door buttons, emergency stop button, emergency bell button
   C- Key controls: fireman's service key, inspector's key.
   D- System controls: they are used to turn the elevator system on or off and are only accessible from an elevator control room.
2. Outputs:
   A- Actuators: door opening device, electric motor, brakes.
   B- Bells: emergency bell, load bell.
   C- Displays: position display, direction display.
3. Controllers: controller is a device which manages the visual monitoring, interactive command control and traffic analysis system to ensure the elevators are functioning efficiently like Relay based controller and PLC based controller.

9.4.2 System Components:

To implement the elevator system, we are going to use the following parts. Here are some details about the structure and how to use them to build the whole elevator system.

A. MDA-007 elevator kit

Figure 1 shows the schematic of the MDA-007 kit; it consists of the four main parts along with the power supply board.

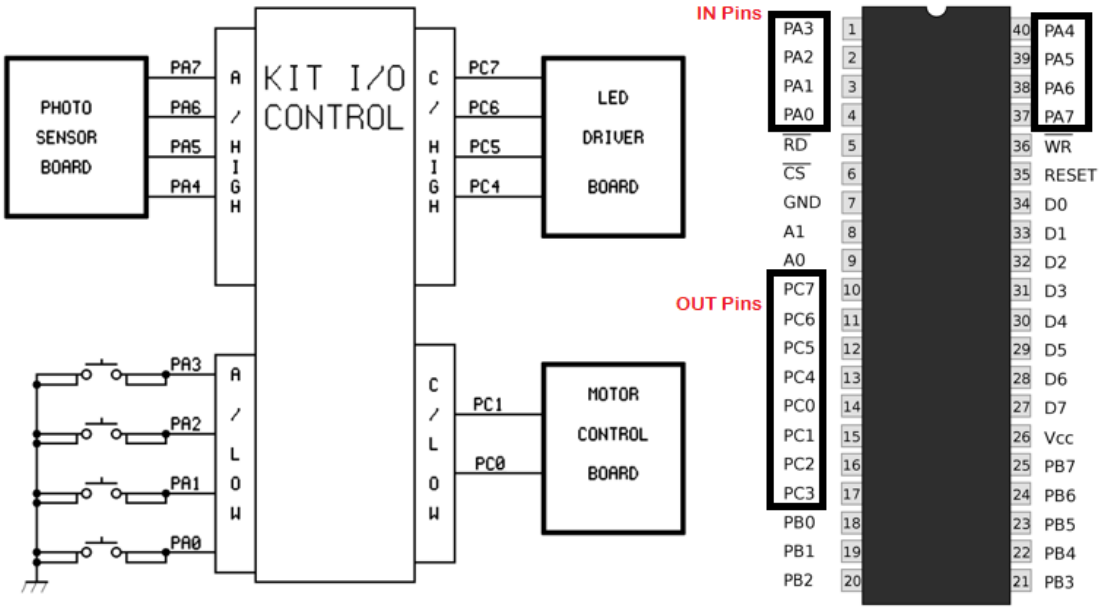


Figure 1: MDA-007 kit boards.

Here are the descriptions and the details of these parts:

1. Power supply board:

   This board supplies the MDA-007 with two DC voltages: 5V 3A and 12V 1A. These two values provide the ICs with the required VCC and GND. The 12 V is required to drive the motor and move the elevator up and down based on the user request.

2. Photo sensor board:

   A photo-interrupters, also known as an opto-isolator, is a device in which light-emitting device and light -receiving device are fabricated in on package, accomplishing the desired light transmission and providing complete electrical-isolation.

   Most-emitting devices use gallium arsenide (GaAs) infrared LEDs, the light-receiving devices use different types depending on application purpose. If the analog signal is used, cadmium sulfide (CdS) or light dependent resistor (LDR) or photo-transistors or photo logic gates are used instead. The photo-interrupters introduced in this manual deal with digital signals, so photo-transistors are used as light-receiving devices. The output voltage is taken across the collector -emitter terminals, allowing for driving different kinds of electronic circuits. **Figure 2** shows the diagram of the photo sensor.

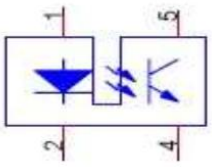


Figure 2: The diagram of photo-interrupters.

As in **Figure 3**, there are four photo sensors, each one serves one floor. There are used to detect the position of the elevator and feed the circuit with the suitable input value (i.e. HIGH, LOW). These sensors are considered as inputs to the controller unit (i.e. PA7 for 4F, PA6 for 3F, PA5 for 2F, and PA4 for 1F).

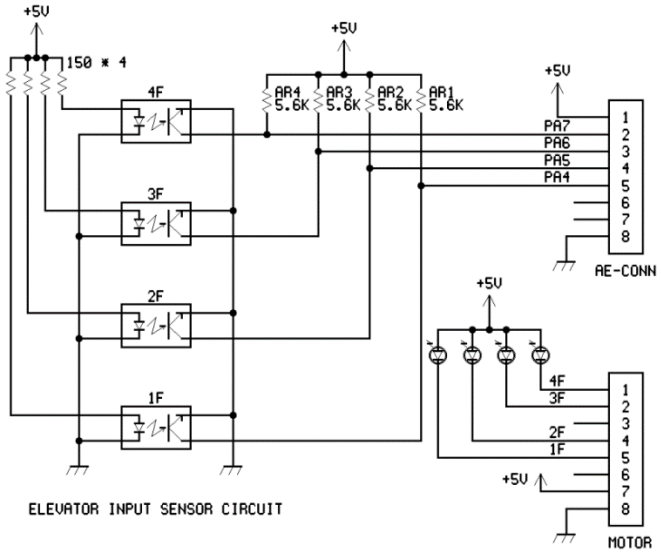


Figure 3: Elevator input sensor circuit.

3. Switch input circuit:

   Based on **Figure 4**, the elevator system serves four floors (i.e. 1F, 2F, 3F, and 4F). Each floor can be requested using a push button.

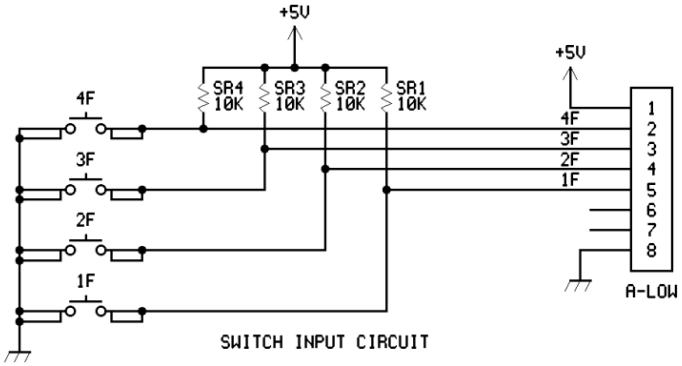


Figure 4: Switch input circuit.

4. LED driver board:

   There are four LEDs based on **Figure 5**, each one is linked with one floor (i.e. PC7 for 4F, PC6 for 3F, PC5 for 2F, and PC4 for 1F).

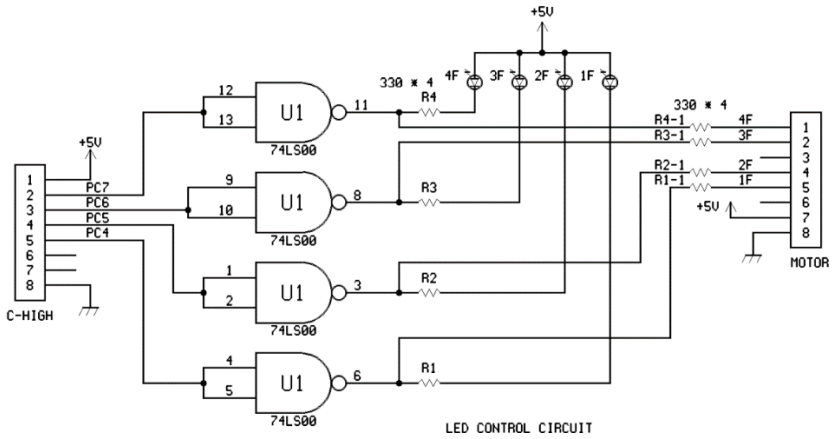


Figure 5: LED control circuit.

5. Motor control board:

   Figure 6 shows how the motor driver 6209 is connected in the elevator system in order to move the elevator up and down.

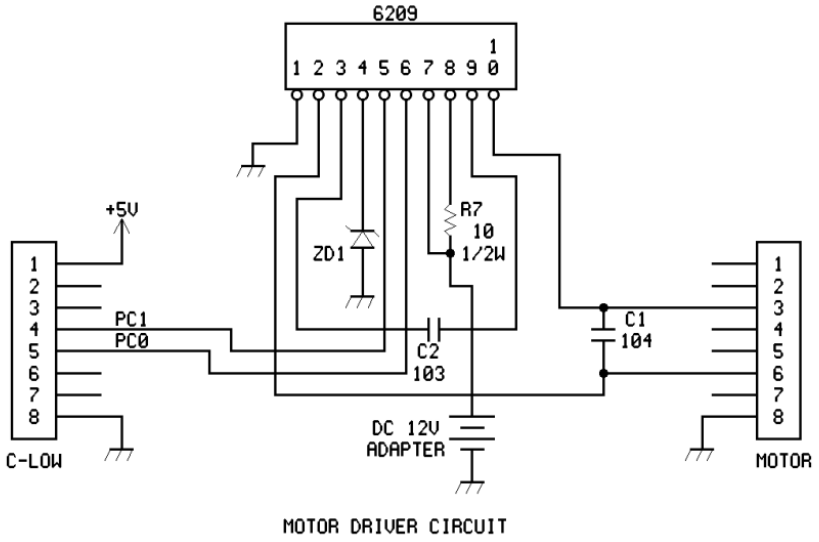


Figure 6: Motor driver circuit.

Also, **Figure 7** shows the pin diagram of the driver 6209, its truth table, and the block diagram. Forward / reverse control, forced stop, and rush current absorption are controlled by the combination of FIN and RIN input states mentioned in the truth table. The operation of this driver as follows:

- Forward / reverse control circuit: when FIN is HIGH and RIN is LOW, current flows from OUT1 to OUT2. When FIN is LOW and RIN is HIGH, current flows from OUT2 to OUT1.
- Forced stop circuit: by setting RIN and FIN both HIGH or both LOW, power supply to the motor is shut down and a brake is applied by absorbing the motor counter-electromotive force.
- Rush current absorption circuit: when a high voltage (caused by such as a motor reversal) is generated on OUT1 and OUT2, an internal comparator detects the high voltage and turns on an internal circuit that absorbs rush currents.
- Drive circuit: the forward direction of the motor connected between OUT1 and OUT2 corresponds to the current flow from OUT1 to OUT2, and the reverse direction corresponds to the current flow from OUT2 to OUT1.

(a) Pin description

Pin No.	Pin name	Function
1	GND	GND
2	OUT 1	Motor output
3	V_{Z1}	Capacitor connection pin for preventing both output transistors being turned on at the same time
4	V_{ref}	Output HIGH voltage setting
5	F_{IN}	Logic input
6	R_{IN}	Logic input
7	V_{CC1}	Control circuit power supply
8	V_{CC2}	Output power supply
9	V_{Z2}	Capacitor connection for preventing both output transistors being turned on at the same time
10	OUT 2	Motor output

(b) Input/Output truth table

Input / output truth table

Input		Output	
F_{IN}	R_{IN}	OUT1	OUT2
L	L	L	L
H	L	H	L
L	H	L	H
H	H	L	L

(c) block diagram

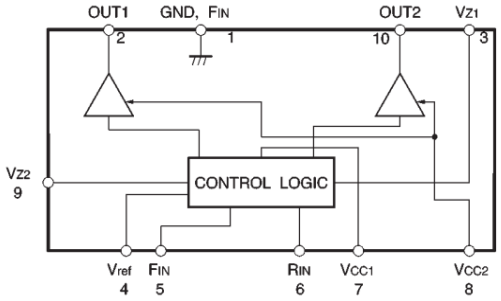


Figure 7: Motor driver 6209 (a) Pins, (b) truth table and (c) block diagram.

B. Arduino Microcontroller

Arduino is a user-friendly electronics platform that operates on open-source principles, encompassing both hardware and software components. Arduino boards possess the capability to interpret inputs, both digital and analog, received from sensors. Utilizing Arduino, you have the ability to create and upload computer code to a physical programmable circuit board, often referred to as a microcontroller.

The Arduino Mega 2560 is a microcontroller board based on the ATmega2560. It has 54 digital input/output pins (of which 15 can be used as PWM outputs), 16 analog inputs, 4 UARTs (hardware serial ports), a 16 MHz crystal oscillator, a USB connection, a power jack, an ICSP header, and a reset button.

This process is facilitated by a software application known as the Arduino IDE (Integrated Development Environment), which operates on your computer. Arduino stands as the leading open-source microcontroller board as shown in **Figure *8***, extensively employed in a multitude of electronics and DIY endeavors. A straightforward example of an Arduino project is the traffic light controller. Nowadays, there is a widespread desire for personal vehicle ownership, leading to a continuous increase in the volume of cars on the roads, consequently leading to traffic congestion.

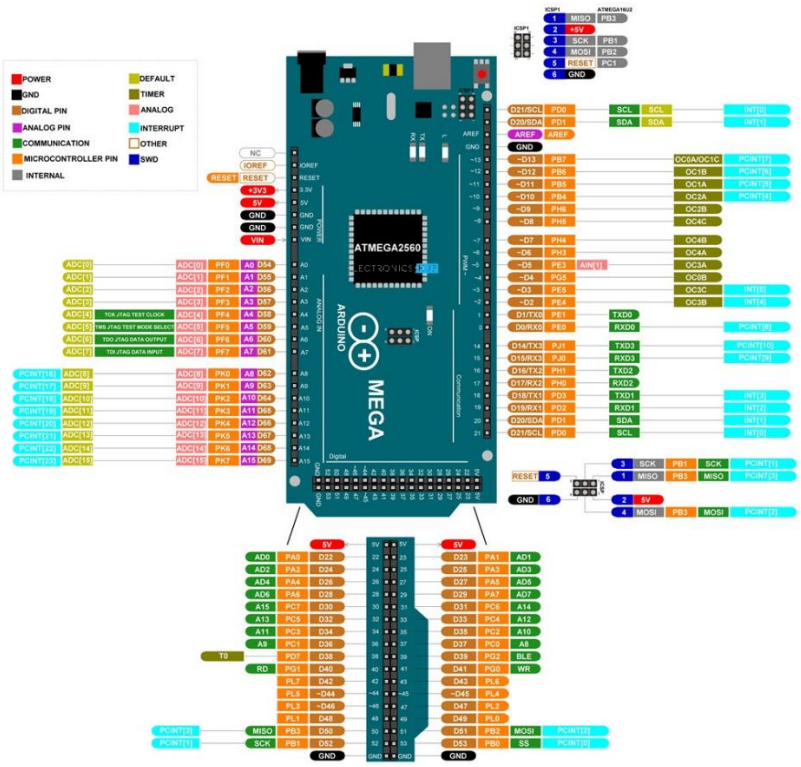


Figure 8: Arduino Mega Pinout.

C. Arduino IDE

The Arduino IDE is an open-source software application designed for the purpose of composing and uploading code to Arduino boards. This IDE software is compatible with various operating systems, including Windows, Mac OS X, and Linux. It accommodates programming languages like C and C++.

The code created within the Arduino IDE is commonly referred to as a "sketch." To upload this sketch to the Arduino board, a connection with the IDE is necessary. The sketch is typically saved with the '.ino' file extension. The Arduino IDE will appear as in **Figure *9***.

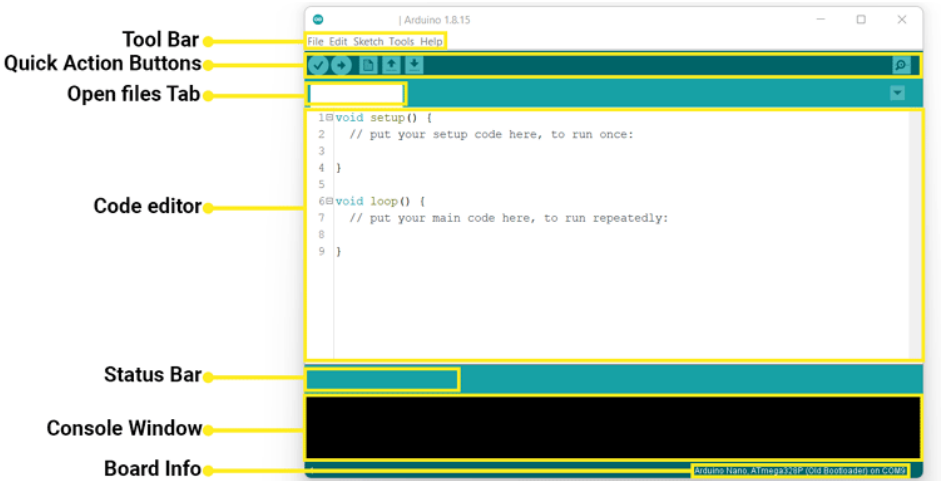


Figure 9: Arduino IDE.

9.5 Procedure

9.5.1 Task 1: The Basics of MDA-007 Kit and the MEGA Arduino

A. Write a code that turns the square LED ON (UP(green)/DOWN(red)) based on your selection of the push button (1F/2F). Here is a sample code and you need to make the suitable connections between the MDA-007 and the MEGA:

```
int PA_0 = 22;
int PA_1 = 23;
int PC_2 = 24;
int PC_3 = 25;
void setup() {
// put your setup code here, to run once:
Serial.begin(9600);
pinMode(PA_0,INPUT);
pinMode(PA_1,INPUT);
pinMode(PC_2,OUTPUT);
pinMode(PC_3,OUTPUT);
}
void loop() {
digitalWrite(PC_2, HIGH);
digitalWrite(PC_3, HIGH);
// put your main code here, to run repeatedly:
if (digitalRead(PA_0) == LOW){
digitalWrite(PC_3, LOW);
}
if (digitalRead(PA_1) == LOW){
digitalWrite(PC_2, LOW);
}
}
```

B. Write a code that moves the motor of the elevator (UP/DOWN) based on your selection of the push button (1F/2F). Here is a sample code and you need to make the suitable connections between the MDA-007 and the MEGA:

```
int PA_0 = 22;
int PA_1 = 23;
int PC_0 = 24;
int PC_1 = 25;
void setup() {
// put your setup code here, to run once:
Serial.begin(9600);
pinMode(PA_0,INPUT);
pinMode(PA_1,INPUT);

pinMode(PC_0,OUTPUT);
pinMode(PC_1,OUTPUT);
}
void loop() {
digitalWrite(PC_0, HIGH);
digitalWrite(PC_1, HIGH);
// put your main code here, to run repeatedly:
```

```
if (digitalRead(PA_0) == LOW){
digitalWrite(PC_0, LOW);
}
if (digitalRead(PA_1) == LOW){
digitalWrite(PC_1, LOW);
}
}
```

C. Write a code that moves the motor of the elevator (UP/DOWN) based on your selection of the push button (1F/2F) and turns the square LED ON (UP(green)/DOWN(red)). Here is a sample code and you need to the suitable connections between the MDA-007 and the MEGA:

```
int PA_0 = 22;
int PA_1 = 23;
int PC_0 = 24;
int PC_1 = 25;
int PC_2 = 28;
int PC_3 = 29;
int PC_4 = 26;
int PC_5 = 27;
void setup() {
// put your setup code here, to run once:
Serial.begin(9600);
pinMode(PA_0,INPUT); // F1
pinMode(PA_1,INPUT); // F2
pinMode(PC_0,OUTPUT); // UP motor
pinMode(PC_1,OUTPUT); // Down motor
pinMode(PC_2,OUTPUT); // Down led
pinMode(PC_3,OUTPUT); // Up led
pinMode(PC_4,OUTPUT); // Led F1
pinMode(PC_5,OUTPUT); // Led F2
}
void loop() {
digitalWrite(PC_0, HIGH);
digitalWrite(PC_1, HIGH);
digitalWrite(PC_2, HIGH);
digitalWrite(PC_3, HIGH);
digitalWrite(PC_4, HIGH);

digitalWrite(PC_5, HIGH);
// put your main code here, to run repeatedly:
if (digitalRead(PA_0) == LOW){
digitalWrite(PC_0, LOW);
digitalWrite(PC_3, LOW);
digitalWrite(PC_4, LOW);
}
if (digitalRead(PA_1) == LOW){
digitalWrite(PC_1, LOW);
digitalWrite(PC_2, LOW);
digitalWrite(PC_5, LOW);
}
}
```

9.5.2 Task 2: Build an elevator for two floors

Build an elevator system for two floors which should include the following:

1. Turning the green square LED ON when the elevator is going UP and turning the red square LED ON when the elevator is going DOWN.

2. Pressing on the push buttons (1F/2F) to visit the requested floor, this also will turn ON the floor led and will turn it OFF when reaching this floor.

3. Stopping the elevator on the requested floor by using the photo sensor of each floor.

Here is the sample code and you are required to make the suitable connections between the MDA-007 and the Arduino MEGA:

```
int PA_0 = 22;
int PA_1 = 23;
int PA_4 = 30;
int PA_5 = 31;
int PC_0 = 24;
int PC_1 = 25;
int PC_2 = 28;
int PC_3 = 29;
int PC_4 = 26;
int PC_5 = 27;
int floor_last = 1;
int floor_1sense = 0;
int floor_2sense = 0;
void setup() {
// put your setup code here, to run once:

Serial.begin(9600);
pinMode(PA_0,INPUT); // F1
pinMode(PA_1,INPUT); // F2
pinMode(PA_4,INPUT); // F1_sensor
pinMode(PA_5,INPUT); // F2_sensor
pinMode(PC_0,OUTPUT); // UP motor
pinMode(PC_1,OUTPUT); // Down motor
pinMode(PC_2,OUTPUT); // Down led
pinMode(PC_3,OUTPUT); // Up led
pinMode(PC_4,OUTPUT); // Led F1
pinMode(PC_5,OUTPUT); // Led F2
}
void loop() {
digitalWrite(PC_0, HIGH);
digitalWrite(PC_1, HIGH);
digitalWrite(PC_2, HIGH);
digitalWrite(PC_3, HIGH);
digitalWrite(PC_4, HIGH);
```

```
digitalWrite(PC_5, HIGH);
// put your main code here, to run repeatedly:
if (digitalRead(PA_0) == LOW){
digitalWrite(PC_4, LOW);
floor_1_func();
}
if (digitalRead(PA_1) == LOW){
digitalWrite(PC_5, LOW);
floor_2_func();
}
}
void floor_1_func(){
if(floor_last != 1){
while( floor_1sense == LOW ){
digitalWrite(PC_1, LOW);
digitalWrite(PC_2, LOW);
floor_1sense = digitalRead(PA_4);
}
}
floor_last = 1;
floor_1sense = 0;
digitalWrite(PC_4, HIGH);
}
void floor_2_func(){
if(floor_last > 2){
while( floor_2sense == LOW){
digitalWrite(PC_1, LOW);
digitalWrite(PC_2, LOW);
floor_2sense = digitalRead(PA_5);
}

}
else if(floor_last < 2){
while( floor_2sense==LOW){
digitalWrite(PC_0, LOW);
digitalWrite(PC_3, LOW);
floor_2sense = digitalRead(PA_5);
}
}
floor_last = 2;
floor_2sense = 0;
digitalWrite(PC_5, HIGH);
}
```

9.5.3 Task 3: Build an elevator for four floors

Build an elevator system for four floors which should include the following:

1. Turning the green square LED ON when the elevator is going UP and turning the red square LED ON when the elevator is going DOWN.

2. Pressing on the push buttons (1F/2F/3F/4F) to visit the requested floor, this also will turn ON the floor led and will turn it OFF when reaching this floor.

3. Stopping the elevator on the requested floor by using the photo sensor of each floor.

Here is the sample code and you are required to make the suitable connections between the MDA-007 and the Arduino MEGA:

```
int PA_0 = 22;
int PA_1 = 23;
int PA_2 = 34;
int PA_3 = 35;//
int PA_4 = 30;
int PA_5 = 31;
int PA_6 = 32;
int PA_7 = 37;//
int PC_0 = 24;
int PC_1 = 25;
int PC_2 = 28;
int PC_3 = 40;//
int PC_4 = 26;
int PC_5 = 27;
int PC_6 = 33;
int PC_7 = 36;//

int floor_last = 4;
int floor_1sense = 0;
int floor_2sense = 0;
int floor_3sense = 0;
int floor_4sense = 0;
void setup() {
// put your setup code here, to run once:
Serial.begin(9600);
pinMode(PA_0,INPUT); // F1
pinMode(PA_1,INPUT); // F2
pinMode(PA_2,INPUT);
pinMode(PA_3,INPUT);
pinMode(PA_4,INPUT); // F1_sensor
pinMode(PA_5,INPUT); // F2_sensor
pinMode(PA_6,INPUT);
pinMode(PA_7,INPUT);
pinMode(PC_0,OUTPUT); // UP motor
pinMode(PC_1,OUTPUT); // Down motor
pinMode(PC_2,OUTPUT); // Down led
pinMode(PC_3,OUTPUT); // Up led
pinMode(PC_4,OUTPUT); // Led F1
pinMode(PC_5,OUTPUT); // Led F2
pinMode(PC_6,OUTPUT);
pinMode(PC_7,OUTPUT);
}
```

```
void loop() {
digitalWrite(PC_0, HIGH);
digitalWrite(PC_1, HIGH);
digitalWrite(PC_2, HIGH);
digitalWrite(PC_3, HIGH);
digitalWrite(PC_4, HIGH);
digitalWrite(PC_5, HIGH);
digitalWrite(PC_6, HIGH);
digitalWrite(PC_7, HIGH);
// put your main code here, to run repeatedly:
if (digitalRead(PA_0) == LOW){
digitalWrite(PC_4, LOW);
floor_1_func();
}
if (digitalRead(PA_1) == LOW){
digitalWrite(PC_5, LOW);
floor_2_func();
}
if (digitalRead(PA_2) == LOW){
digitalWrite(PC_6, LOW);
floor_3_func();
}

if (digitalRead(PA_3) == LOW){
digitalWrite(PC_7, LOW);
floor_4_func();
}

}
void floor_1_func(){
if(floor_last != 1){
while( floor_1sense == LOW ){
digitalWrite(PC_1, LOW);
digitalWrite(PC_2, LOW);
floor_1sense = digitalRead(PA_4);
delay(10);
}
}
floor_last = 1;
floor_1sense = 0;
digitalWrite(PC_4, HIGH);
}
void floor_2_func(){
if(floor_last > 2){
while( floor_2sense == LOW){
digitalWrite(PC_1, LOW);
digitalWrite(PC_2, LOW);
floor_2sense = digitalRead(PA_5);
delay(10);
}
}
else if(floor_last < 2){
while( floor_2sense==LOW){
digitalWrite(PC_0, LOW);
```

```
digitalWrite(PC_3, LOW);
floor_2sense = digitalRead(PA_5);
delay(10);
}
}
floor_last = 2;
floor_2sense = 0;
digitalWrite(PC_5, HIGH);
}
void floor_3_func(){
if(floor_last < 3){
while( floor_3sense==LOW){
digitalWrite(PC_0, LOW);
digitalWrite(PC_3, LOW);
floor_3sense = digitalRead(PA_6);
delay(10);
}
}
else if(floor_last > 3){

while( floor_3sense==LOW){
digitalWrite(PC_1, LOW);
digitalWrite(PC_2, LOW);
floor_3sense = digitalRead(PA_6);
delay(10);
}
}
floor_last = 3;
floor_3sense = 0;
digitalWrite(PC_6, HIGH);
}
void floor_4_func(){
if(floor_last != 4){
while( floor_4sense==LOW){
digitalWrite(PC_0, LOW);
digitalWrite(PC_3, LOW);
floor_4sense = digitalRead(PA_7);
delay(10);
}
}
floor_last = 4;
floor_4sense = 0;
digitalWrite(PC_7, HIGH);
}
```

9.6 To Do

Build an elevator for four floors as in the previous task and turn on the seven segment to show the floor number. You might use the pins (PB0 to PB7) to do this.

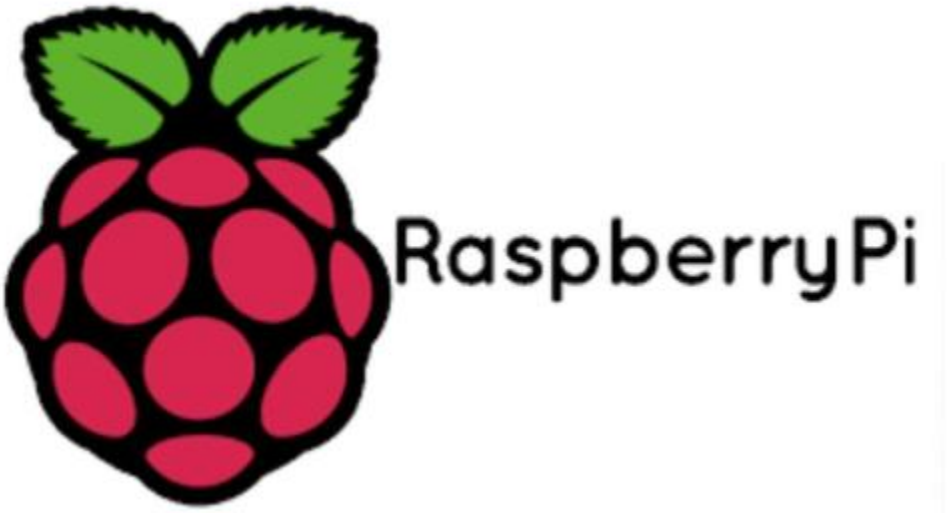
جامعة بيرزيت
BIRZEIT UNIVERSITY

Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

Microprocessor and Embedded Systems Lab (ENCS5101)

Experiment No. 10 - Introduction to Raspberry Pi



Introduction

This is an introductory experiment about the credit card sized computer Raspberry Pi. This experiment is ideal for those who are interested in exploring the possibilities of Raspberry Pi as a computer. The experiment does not assume any prior knowledge on Raspberry Pi programming. However, knowledge of Linux operating system and Python programming language would greatly help you in getting up to speed.

What is Raspberry Pi?

The Raspberry Pi is a computer, very like the computers with which you're already familiar. It uses a different kind of processor you can install several versions of the Linux operating system that look and feel very much like Windows. If you want to, you can use the Raspberry Pi to surf the internet, send an email or write a letter using a word processor. But you can also do so much more.

This little board here is low cost, it's easily accessible, it's very simple to use. When you power it up you get a nice little desktop environment, it includes all of the things that you need to do to get started to learn programming. There's lots of information on the internet that you can take away and start programming code in to make things happen.

The great thing about these boards as well is in addition to software, you can play with hardware. So these little general purpose pins here allow access to the processor and you can hang off little hardware projects that you build and you can control via the code you are writing through the software application. So, this is a great tool for kids to learn how computers work at a grassroots level.

So there are various versions available of the Raspberry Pi currently. The original Raspberry Pi is based on the Broadcom BCM2835 system on a chip (SoC), which includes an ARM1176JZF-S 700 MHz processor, VideoCore IV GPU, and was originally shipped with 256 megabytes of RAM, later upgraded (models B and B+) to 512 MB. The system has Secure Digital (SD) (models A and B) or Micro SD (models A+ and B+) sockets for boot media and persistent storage.

In early February 2015, the next-generation Raspberry Pi, Raspberry Pi 2, was officially announced. The new computer board will initially be available only in one configuration (model B) and features a Broadcom BCM2836 SoC, with a quad-core ARM CortexA7 CPU 900MHz and a VideoCore IV dual-core GPU; 1 GB of RAM with remaining specifications being similar to those of the previous generation model B+.

Raspberry Pi Board:

In this experiment we will use the most recent board which is the Raspberry Pi 2 model B. Figure 1 below shows Raspberry Pi 2 model B board components.

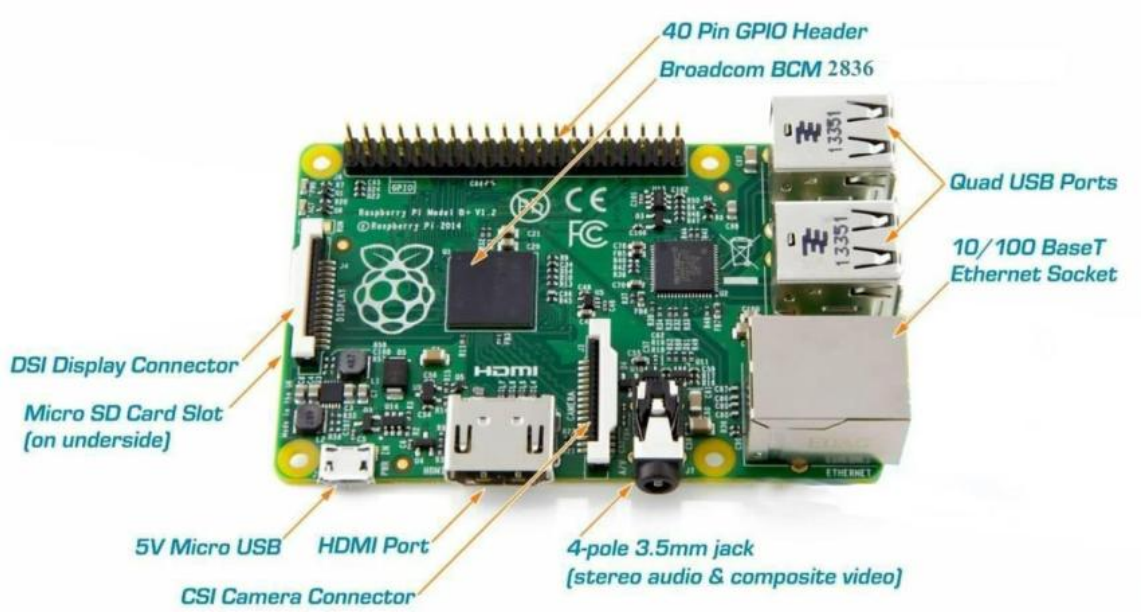


Figure 1: Raspberry Pi 2 board components.

As shown the board consists of the following:

2 USB ports, Ethernet out, camera connector, HDMI out, micro USB power, SD card, display connector, GPIO pins, video out, audio out and status LEDs.

And it has the following main features compared to model B+ and model A+:

	Raspberry Pi 2	Model B+	Model A+
SoC/CPU	BCM2836 Quadcore 900 MHz ARMv7	BCM2835 700 MHz ARMv6k	BCM2835 700 MHz ARMv6k
GPU	Broadcom VideoCore IV @ 250 MHz	Broadcom VideoCore IV @ 250 MHz	Broadcom VideoCore IV @ 250 MHz
RAM	1GB	512 MB	256 MB
Storage	MicroSD	MicroSD	MicroSD
USB	4	4	1
Ethernet	1	1	0
Video output	HDMI/Composite via RCA jack	HDMI/Composite via RCA jack	HDMI/Composite via RCA jack
Audio output	3.5 mm jack	3.5 mm jack	3.5 mm jack
GPIO	40	40	40

Figure 2: Comparison between models.

GPIO PINs:

One powerful feature of the Raspberry Pi is the row of GPIO (general purpose input/output) pins along the edge of the board, next to the video out socket.

These pins are a physical interface between the Raspberry Pi and the outside world. At the simplest level, you can think of them as switches that you can turn on or off (input) or that the Raspberry Pi can turn on or off (output). 26 of the 40 pins are GPIO pins; the others are power or ground pins.

GPIO voltage levels are 3.3v and are not 5v tolerant. There is no over-voltage protection on the board. A voltage near 3.3 V is interpreted as a logic one while a voltage near zero volts is a logic zero.

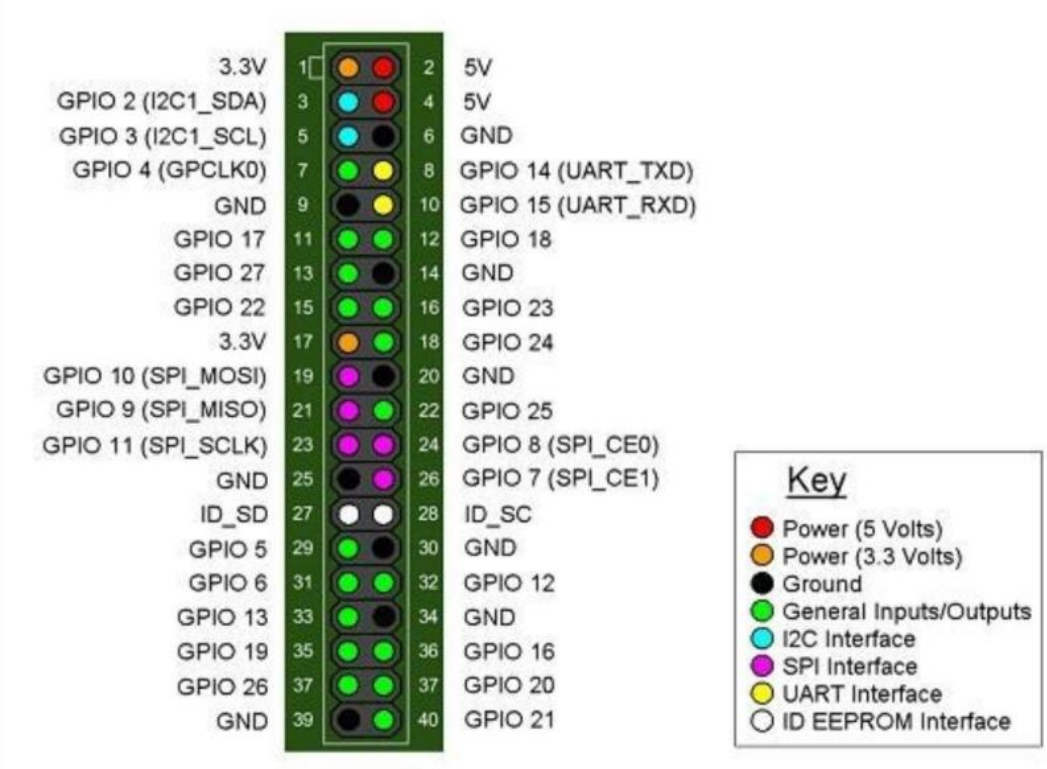


Figure 3: Raspberry Pi 2 GPIO pins.

What are they for?

What can I do with them? You can program the pins to interact in amazing ways with the real world. Inputs don't have to come from a physical switch; it could be input from a sensor or a signal from another computer or device, for example. The output can also do anything, from turning on an LED to sending a signal or data to another device. If the Raspberry Pi is on a network, you can control devices that are attached to it from anywhere and those devices can send data back. Connectivity and control of physical devices over the internet is a powerful and exciting thing, and the Raspberry Pi is ideal for this.

Raspberry Pi Programming Language:

The Raspberry Pi Foundation recommends Python as a language for learners. Any language which will compile for ARMv6 can be used with the Raspberry Pi, though; so you are not limited to using Python. C, C++, Java, Scratch, and Ruby all come installed by default on the Raspberry Pi.

Python is a dynamic, interpreted (bytecode-compiled) language. There are no type declarations of variables, parameters, functions, or methods in source code. This makes the code short and flexible, and you lose the compile-time type checking of the source code. Python tracks the types of all values at runtime and flags code that does not make sense as it runs.

To learn how to program using Python languages see this link: http://www.tutorialspoint.com/python/index.htm

PIR sensor:

PIR sensors allow you to sense motion, almost always used to detect whether a human has moved in or out of the sensors range. They are small, inexpensive, low-power, easy to use and don't wear out. For that reason, they are commonly found in appliances and gadgets used in homes or businesses. They are often referred to as PIR, "Passive Infrared", "Pyroelectric", or "IR motion sensors".

PIRs are basically made of a pyroelectric sensor, which can detect levels of infrared radiation. Everything emits some low level radiation, and the hotter something is, the more radiation is emitted. The sensor in a motion detector is actually split in two halves.

The reason for that is that we are looking to detect motion (change) not average IR levels. The two halves are wired up so that they cancel each other out. If one half sees more or less IR radiation than the other, the output will swing high or low.



Figure 4: PIR (motion detector) sensor.

Gas sensor:

This is a simple-to-use liquefied petroleum gas (LPG) sensor, suitable for sensing LPG (composed of mostly propane and butane) concentrations in the air. The MQ-6 can detect gas concentrations anywhere from 200 to 10000ppm.

This sensor has a high sensitivity and fast response time. The sensor's output is an analog resistance. The drive circuit is very simple; all you need to do is power the heater coil with 5V, add a load resistance, and connect the output to an ADC.



Figure 5: Gas sensor.

Operating System:

1. Installing operating system:

In this experiment we will be using the raspberry pi with the Rasbian as the operating system for the raspberry pi. In order to do that, we need to go through a few easy steps.

We need 8 GB class 4 (or higher) memory card, card reader, laptop or computer connected to internet, mouse, keyboard and screen.

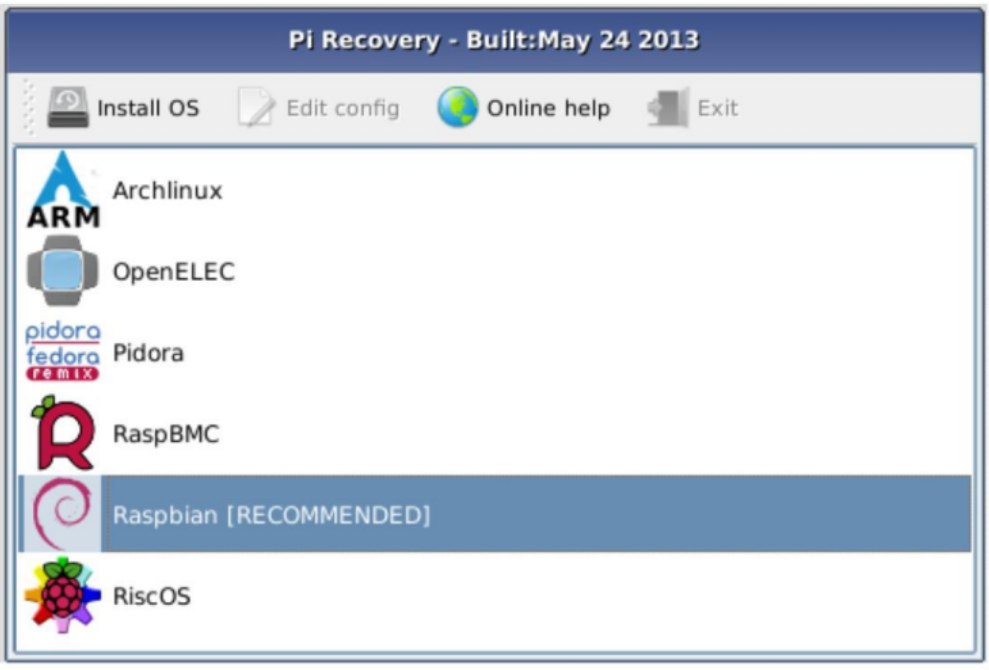
a. Formatting the SD card:
   We need to clear all the files from the SD card; this can be done by downloading a program on the computer from www.sdcard.org. Just go to the download page and select "SD Formatter for Windows or MAC Download".
   After finishing the download and the program setup insert the SD card in the card reader and format it using the program.

b. Downloading NOOBS:
To setup Raspbian, NOOBS must be downloaded first (also you can download Raspbian without the NOOBS).
Go to the download page of www.raspberrypi.org and download NOOBS on your computer. Then open the file and copy all the files from the NOOBS folder to the SD card, this can take up to 20 minutes.

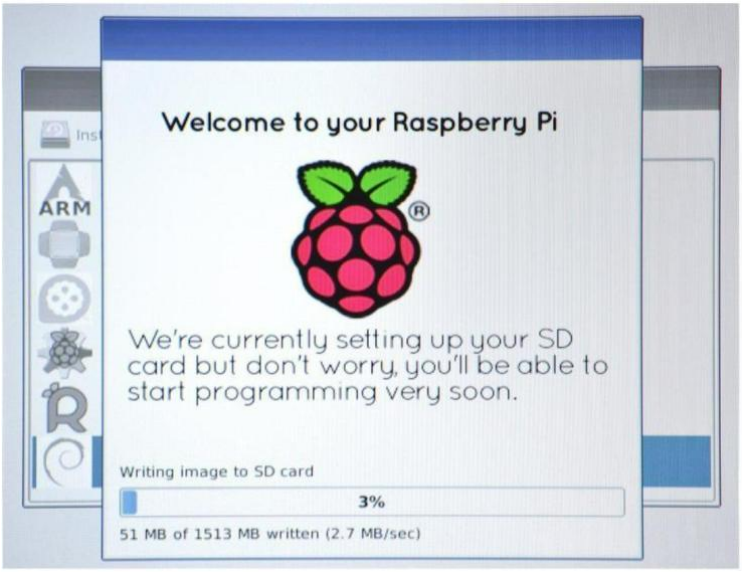


c. Installing Raspbian:
After copying the files, safely remove the SD card and insert it in its place in the raspberry pi. Connect the mouse, keyboard and screen to the raspberry pi then power it. Raspberry pi will start booting.

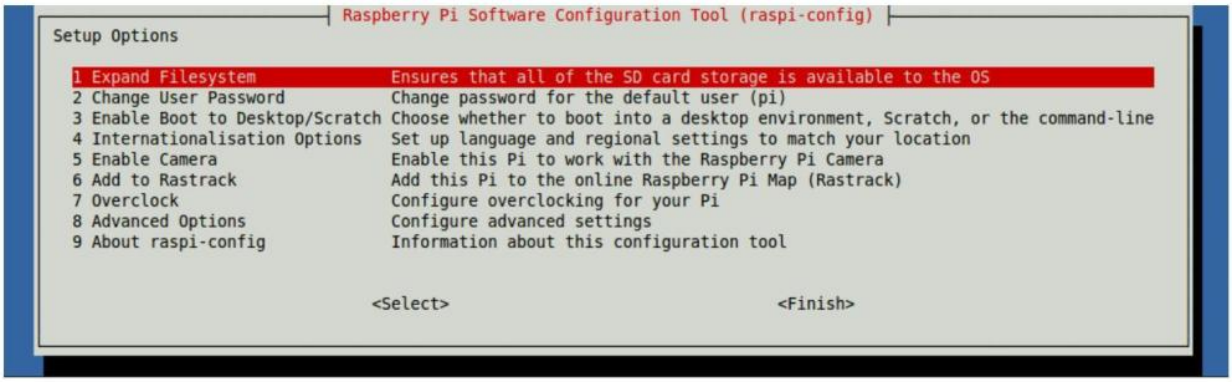




After the boot you will see a list on the screen of the possible operating systems than can be setup (as the above picture). Choose Raspbian and press OK. This should start the setup. It will take up to 30 minutes.



Note: while setup Raspbian it worth to see the notes which will appear on the screen.



Finishing the setup will show a config. List, you will be able to select the time zone and date. Do that by using the keyboard. After that press on “tab or back space” then press enter. Now the prompt command line will appear” pi@raspberrypi ~ $”, type “startx” or any word and press enter. Now you see the Raspbian interface screen. The raspberry pi is ready for use now.

2. Install RPi.GPIO Python Library:

The RPi.GPIO Python library allows you to easily configure and read-write the input/output pins on the Pi’s GPIO header within a Python script. Thankfully, this library is now included in the standard Raspbian image available from the Foundations Download Page.

Install from repository

If the package exists in the Raspbian repository is can be installed using apt-get. First you need to update the available package versions:

```
sudo apt-get update
```

Then attempt to install the RPi.GPIO package:

```
sudo apt-get install python-dev python-rpi.gpio
```

If it isn’t already installed it will be installed. If it is already installed it will be upgraded if a newer version is available.

Procedure:

PIR Motion Detector

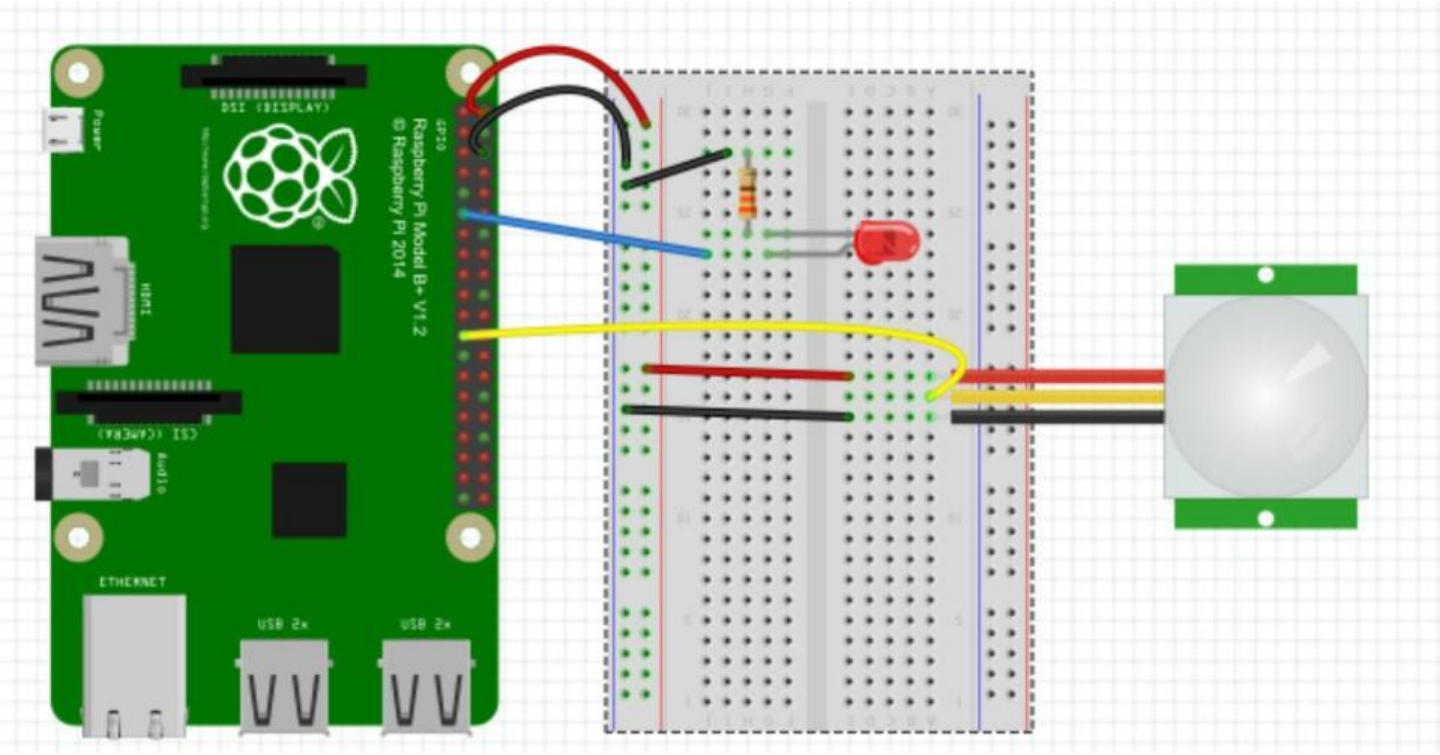
In the first part we will connect the PIR and led to the raspberry pi, our goal is to turn on the led when a motion is detected.

In the second part we will add LDR to the circuit, this will turn on the led if there is a motion detected and there is no light detected.

1- PIR with led:

Connect PIR sensor and a led to the raspberry pi as in the picture:

- Connect vcc on pin#1
- Connect gnd on pin#6
- Connect output pin of PIR to pin#23
- Connect 300ohm resistor with the led(+leg) then connect it with pin#11
- Connect the short (-) leg to GND



- Then open LXterminal from tools bar to run the following program, to do this go to the location of the saved code which is:

```
Cd /home/pi
```

Writing the code:

- Create new folder

```
mkdir “folder name”
```

- Enter the folder

```
cd “folder name”
```

- Open the editor

```
sudo nano “program name”.py
```

- Write the following code in the editor:

```
import time                 #import time library
import RPi.GPIO as GPIO     #import GPIO library
GPIO.setmode(GPIO.BOARD)    #configurations GPIO pins
GPIO.setup(23,GPIO.IN)      #define pin #23 as input
GPIO.setup(11,GPIO.OUT)     #define pin #11 as output

while True:
    if(GPIO.input(23)==True): #true means high
        print("Motion Detected")
        GPIO.output(11,1)     #turn on led
    else:
        print("There is no motion")
        GPIO.output(11,0)     #turn off led
    time.sleep(1)             #delay for 1 second
```

- To save the code press “ctrl+x”, then press “y” then enter.

To run the code write the following command:

```
sudo python program_Name.py
```

2- PIR with led and LDR:

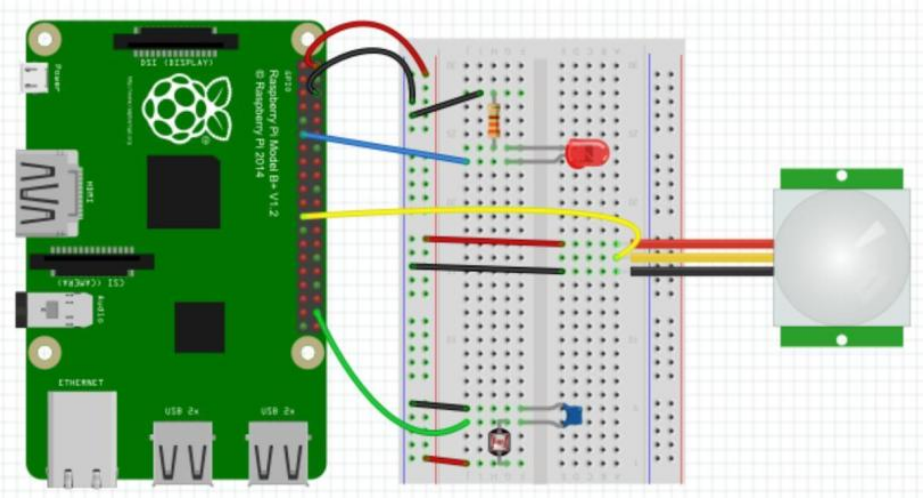
Since raspberry pi can’t read analog signal and because LDR only gives analog, output we need to find a way to connect the LDR to it without using ADC.

The way we do this is by taking advantage of a basic electronic property of resistors and capacitors. It turns out that if you take a capacitor that is initially storing no voltage, and then connect it to power (like 3.3V) through a resistor, it will charge up to the power voltage slowly. The bigger the resistor, the slower it is.

Note:

This technique only works with sensors that act like resistors. It cannot be used with sensors that have a pure analog output like IR distance sensor. It's not nearly as precise as an ADC and it’s a little flaky (since it depends on the Pi timing itself which can vary based on how 'busy' the computer is).

Connect the LDR and 1uf capacitor as in the following picture:



Then run code as you did in the first code (in last step):

```
import time
import RPi.GPIO as GPIO
import os
DEBUG = 1

GPIO.setmode(GPIO.BOARD)
GPIO.setup(11, GPIO.OUT)

def RCtime (RCpin):
        reading = 0
        GPIO.setup(RCpin, GPIO.OUT)
        GPIO.output(RCpin, GPIO.LOW)
        time.sleep(0.1)

        GPIO.setup(RCpin, GPIO.IN)
        # This takes about 1 millisecond per loop cycle
        while (GPIO.input(RCpin) == GPIO.LOW):
                reading += 1
        return reading

def motion (mpin) :
    GPIO.setup(mpin, GPIO.IN)
    motionstate = False
    motionstate = GPIO.input(mpin)
    return motionstate
```

```
while True:
    if (motion(23) == True and RCtime(38)>20):
        # print " Motion Detected "  +"Light Off "
        GPIO.output(11,True)
        time.sleep(2)
    elif motion(23) == True and RCtime(38)<20:
        # print " Motion Detected "+"Light On"
            GPIO.output(11,False)

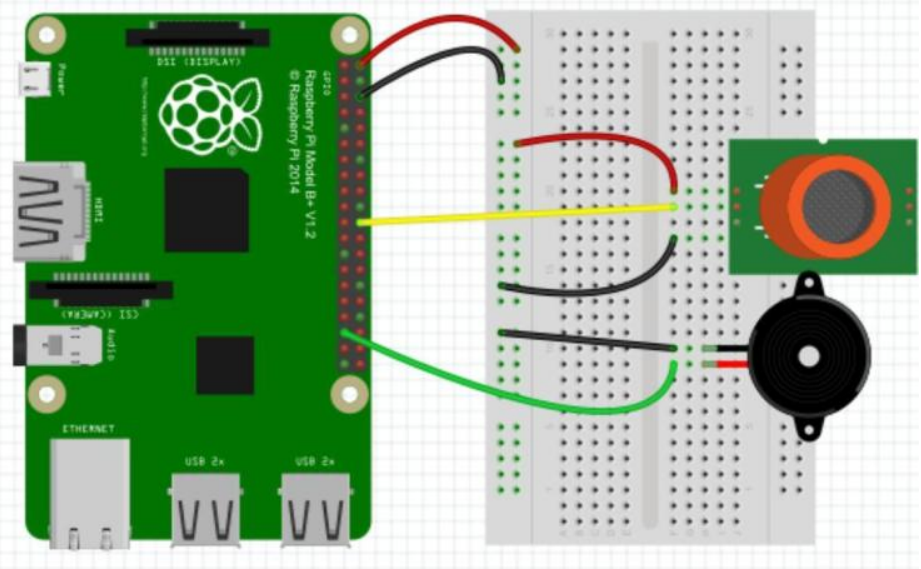
    elif motion(23) == False and RCtime(38)>20:
        # print " There is No Motion "+"Light Off"
            GPIO.output(11,False)

    time.sleep(0.1)
```

Gas Sensor/any other available sensor

The second program will be to run a gas sensor, the object of this program is turn on a fan if the gas sensor detect gas.

To do that, connect the components as following:

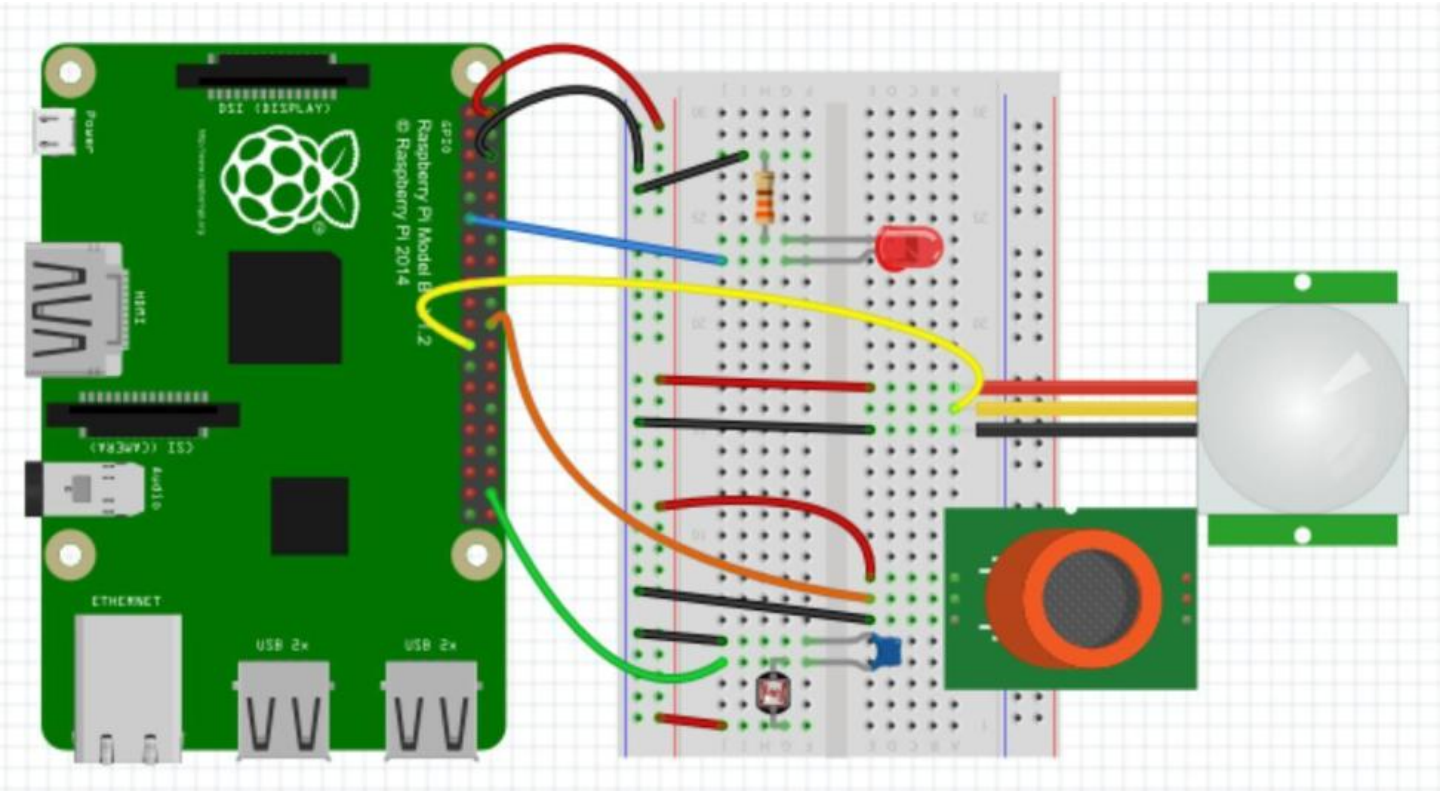


Complete the following code and run it as you did with the first code:

```
1   import time
2   import RPi.GPIO as GPIO
3
4
5   GPIO.setmode(GPIO.BOARD)
6   GPIO.setup(   , GPIO.OUT)
7
8   def GasDet (Gaspin) :
9       GPIO.setup(Gaspin, GPIO.IN)
10      Gasstate = False
11      Gasstate = GPIO.input(Gaspin)
12      return Gasstate
13
14  while True:
15
16          time.sleep(1)
17
```

Running the two programs together:

Now we will build the two circuits together and run the two programs. Just connect the components on the same pins as before. The full circuit should be as following:



In order to run two programs (or more) you should type the name of the program that you wrote and saved using the command “sudo” and “&” as following:

```
Sudo python motion.py&
Sudo python gas.py&
```

To stop any program you should write the command

```
Sudo kill “number of program”
```

To stop all the programs write this command

```
Sudo killall sudo python programName.py
```