Lab 14 - Timers, Interrupts and PWM

14.1 Interrupts

Interrupts are the events that temporarily suspend the main program, pass the control to an **Interrupt Service Routine (ISR)** or **Interrupt Handler**. ISR could be a code label, block of code, or a function. After the execution, ISR passes the control to the main program where it had left off. What we are doing is called asynchronous processing. There are two types of interrupts:

- **Hardware Interrupt:** A hardware interrupt is triggered by an external hardware device, e.g., an interrupt request (IRQ) on a I/O pin of the microcontroller by a push button.
- **Software Interrupt:** A software interrupt is requested by the processor itself upon executing particular instructions or when certain conditions are met. For example, when a counter overflows, analog to digital conversion is completed, etc.

14.1.1 Interrupt Service Routine

For every interrupt, there is a fixed location in memory that holds the address of its interrupt service routine (ISR). The table of memory locations to hold the addresses of ISRs is called as the *Interrupt Vector Table*. Interrupt Vector Table of ATmega328p microcontroller is as follows:

Vector No.	Program Address	Source	Interrupt Definition	Assembly Names
1	0x0000	RESET	External pin, power-on, brown-out, and watchdog	
2	0x0002	INT0	External interrupt request 0	INT0addr
3	0x0004	INT1	External interrupt request 1	INT1addr
4	0x0006	PCINT0	Pin change interrupt request 0	PCI0addr
5	0x0008	PCINT1	Pin change interrupt request 1	PCI1addr
6	0x000A	PCINT2	Pin change interrupt request 2	PCI2addr
7	0x000C	WDT	Watchdog time-out interrupt	WDTaddr
8	0x000E	TIMER2 COMPA	Timer/Counter2 compare match A	OC2Aaddr
9	0x0010	TIMER2 COMPB	Timer/Counter2 compare match B	OC2Baddr
10	0x0012	TIMER2 OVF	Timer/Counter2 overflow	OVF2addr
11	0x0014	TIMER1 CAPT	Timer/Counter1 capture event	ICP1addr
12	0x0016	TIMER1 COMPA	Timer/Counter1 compare match A	OC1Aaddr
13	0x0018	TIMER1 COMPB	Timer/Counter1 compare match B	OC1Baddr
14	0x001A	TIMER1 OVF	TIMER1 OVF Timer/Counter1 overflow	OVF1addr
15	0x001C	TIMER0 COMPA	Timer/Counter0 compare match A	OC0Aaddr
16	0x001E	TIMER0 COMPB	Timer/Counter0 compare match B	OC0Baddr
17	0x0020	TIMER0 OVF	Timer/Counter0 overflow	OVF0addr
18	0x0022	SPI, STC	SPI serial transfer complete	SPIaddr
19	0x0024	USART, RX	USART Rx complete	URXCaddr
20	0x0026	USART, UDRE	USART, data register empty	UDREaddr
21	0x0028	USART, TX	USART, Tx complete	UTXCaddr
22	0x002A	ADC	ADC conversion complete	ADCCaddr
23	0x002C	EE READY	EEPROM ready	ERDYaddr
24	0x002E	ANALOG COMP	Analog comparator	ACIaddr
25	0x0030	TWI	2-wire serial interface	TWIaddr
26	0x0032	SPM READY	Store program memory ready	SPMRaddr

14.1.2 Triggering Methods

Each interrupt signal input is designed to be triggered by either a logic signal level or a particular signal edge (level transition). Level-sensitive inputs continuously request processor service so long as a particular (high or low) logic level is applied to the input. Edge-sensitive inputs react to signal edges: a particular (rising or falling) edge will cause an interrupt.

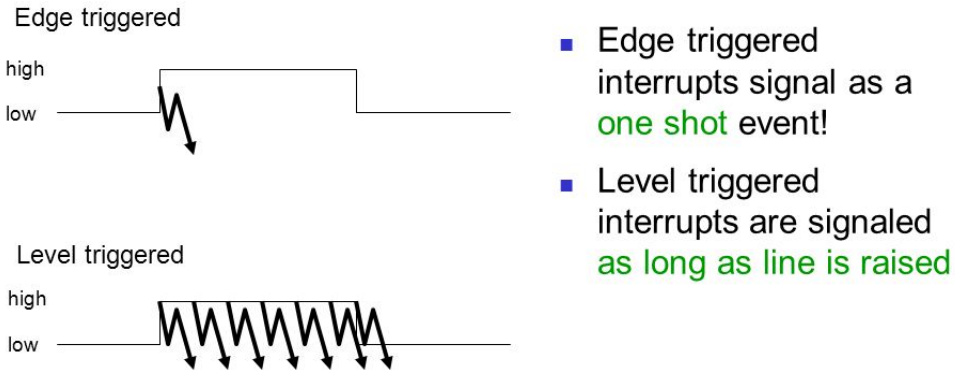
14.1.2.1 Level-Triggered

In level-triggered mode, as long as the IRQ (Interrupt Request) line is asserted, you get an interrupt request. When you serve the interrupt and return, if the IRQ line is still asserted, you get the interrupt again immediately.

14.1.2.2 Edge-Triggered

You get an interrupt when the signal pulse changes from inactive to active state, but only once. To get a new request, the pulse must go back to inactive and then to active again.

An edge-triggered interrupt is signaled by a level transition on the interrupt pin, either a falling edge (high to low) or a rising edge (low to high).



14.1.3 Interrupt Priority

Priority for the interrupts is determined by the interrupt vector address. An interrupt with lowest interrupt vector address has the highest priority. So RESET has the highest priority followed by INT0, then INT1 and so on. If two interrupts occur simultaneously, then the interrupt with higher priority is served first.

14.1.4 How to Define an ISR?

We are simply making a general AVR function somewhere in the AVR's FLASH memory space, and "linking " it to a specific interrupt source by placing a JMP or RJMP instruction to this function at the address specified in the interrupt vector. At the start of the AVR's FLASH memory space lies the Interrupt Vector Table, which is simply a set of hardwired addresses which the AVR's processor will jump to when each of the interrupt fire. By placing a jump instruction at each interrupt's address in the table, we can make the AVR processor jump to our ISR which lies elsewhere.

For an ISR to be called, we need three conditions to be true:

Firstly, the AVR's global Interrupts Enable bit (I) must be set in the AVR Status Register (SREG). This allows the AVR's core to process interrupts via ISRs when set. It defaults to being cleared on power up, so we need to set it.

Secondly, the individual interrupt source's enable bit must be set. Each interrupt source has a separate interrupt enable bit in the related peripheral's control registers, which turns on the ISR for that interrupt. This must also be set, so that when the interrupt event occurs the processor runs the associated ISR.

Thirdly, the condition for the interrupt must be met - for example, for the ADC conversion complete interrupt.

When all three conditions are met, the AVR will fire our ISR each time the interrupt event occurs.

```
.org 0x{Vector Address}
      jmp MyISRHandler

      ...

      MyISRHandler:
            ; ISR code to execute here
      reti
```

Note: We need to add a "reti " instruction at the end of our interrupt instead of the usual "ret ". this special instruction has the dual function of exiting the ISR, and automatically re-enabling the *Global Interrupt Enable* bit.

Next, we also need to enable a specific interrupt source, to satisfy the second condition for firing an ISR. Let's set an interrupt on the ADC Conversion Complete (ADIE) interrupt.

```
      in r16, ADCSRA      ; Loading ADCSRA register into r16
      sbr r16, ADIE       ; setting bit number 3 – ADC Interrupt Enable (ADIE ) bit
      out ADCSRA, r16
```

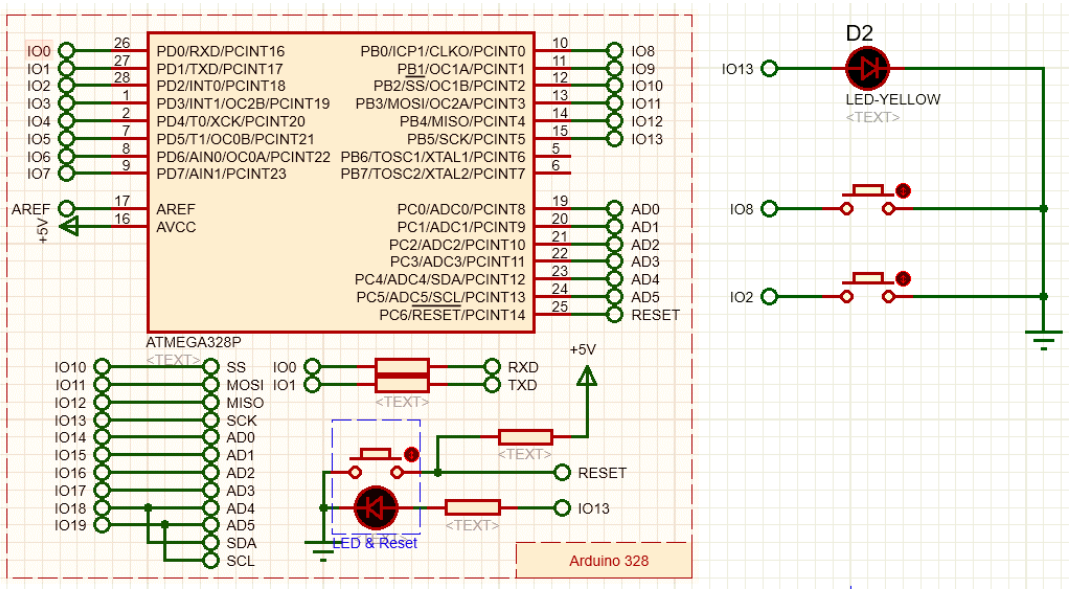
That should be enough to make the AVR's execution jump to the appropriate ISR when the ADC Conversion Complete interrupt occurs.

14.1.5 Interrupt Example: Press Push Buttons to Turn ON /OFF the LED

These are the instruction to turn ON or OFF all the interrupts of the microcontroller:

Instruction	Function	Description
CLI	Clears the I flag of SREG register	Turn OFF all the interrupts globally
SEI	Sets the I flag of SREG register	Turn ON all the interrupts globally

So before writing any code that involves the interrupts, we have to enable interrupts globally using SEI instruction. Now attach two push buttons to ATmega328p. One to PB0 pin and other to PD2 pin.



Then upload the following code to the MCU:

```
; This example program uses interrupts to detect the push buttons.
; Interruput PCINT0 is attached to a push button on pin D8 (PB0) of Arduino UNO.
; Interruput PCINT18 is attached to a push button on pin D2 (PD2) of Arduino UNO.

.include "m328pdef.inc"
.include "delay_Macro.inc"
.cseg

; Interrupt Vector
.org 0x0000                     ; Main program start
      jmp RESET
.org PCI0addr                   ; pin change interrupt on PB0 pin
      jmp PCINT0_ISR
.org PCI2addr                   ; pin change interrupt on PD2 pin
      jmp PCINT18_ISR

RESET:
      LDI r16, high(RAMEND)     ; Set Stack Pointer to end of the RAM
      OUT SPH, r16              ; it is necessary when using interrupt vectors
      LDI r16, low(RAMEND)
      OUT SPL, r16

      ;I/O pins configuration
      SBI DDRB, PB5             ; PB5 set as OUTPUT Pin (LED)
      CBI DDRD, PD2             ; PD2 set as INPUT pin (push button1)
      SBI PORTD, PD2            ; Enable internal pull-up resistor
      CBI DDRB, PB0             ; PB0 set as INPUT pin (push button2)
      SBI PORTB, PB0            ; Enable internal pull-up resistor

      ; configure pin change interrupts on PB0 and PD2 pins
      LDI r16, 0b00000101       ; enabling PCIE0 and PCIE2 interrupts
      STS PCICR, r16
      LDI r16, 0b00000001       ;enabling PCINT0 interrupt (PB0 Pin)
      STS PCMSK0, r16
      LDI r16, 0b00000100       ;enabling PCINT18 interrupt (PD2 Pin)
      STS PCMSK2, r16

      sei                       ; Enable interrupts globally

loop:
      ; your other code, etc.
      nop                       ; no operation (just waste 1 clock cycle)
rjmp loop

; Interrupt PCINT0 ISR function
PCINT0_ISR:
      SBI PORTB, PB5      ; LED ON
reti

; Interrupt PCINT18 ISR function
PCINT18_ISR:
      CBI PORTB, PB5      ; LED OFF
reti
```

14.2 Timers

A timer is a simple counter. Its advantage is that the input clock and operation of the timer is independent of the program execution. The deterministic clock makes it possible to measure time by counting the elapsed cycles and take the input frequency of the timer into account.

The Atmega328p has a total of three timer/counters:

Name	Size	Counting Range
Timer/Counter 0	8-bits	0-255
Timer/Counter 1	16-bits	0-65535
Timer/Counter 2	8-bits	0-255

At power on, or reset, all timer/counters are disabled and must be enabled in software.

Timer/Counters are so called because they have two separate functions.

- Count up, and down, depending on the mode, with a regular clock source based off of the AVR microcontroller's system clock. This is when it acts as a timer.
- Count up, and down, based on an external rising or falling edge attached to a specific pin. This is when it acts as a counter.

There are three terms *TOP*, *MAX*, and *BOTTOM*. These are definitions that are used in the datasheet for the ATmega328p and refer to the following:

BOTTOM: It is always zero.

MAX: It is always the maximum value that can be held in the timer/counter's *TCNTn* register.

For timer/counters 0 and 2, this is always 8 bits, and so MAX always equals 255. For Timer/counter 1, MAX varies as follows:

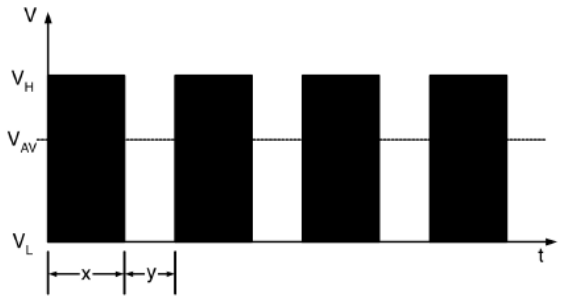
- In 8-bit mode, MAX = 255.
- In 9-bit mode, MAX = 511.
- In 10-bit mode, MAX = 1,023.
- In 16-bit mode, MAX = 65,535.

TOP: It depends on the timer/counter's mode and is either MAX for some modes or as defined by various other timer/counter registers such as *OCRnA*, *OCRnB*, *ICR1*, etc.

14.3 PWM (Pulse Width Modulation)

PWM is an abbreviation for Pulse Width Modulation. In this mode, the timer acts as an up/down counter. This means that the counter counts up to its maximum value and then clears to zero. The advantage of the PWM is that the duty cycle relation can be changed in a phase consistent way.

If the PWM is configured to toggle the Output Compare pin (OCx), the signal at this pin can look like shown in the following figure.



Were:

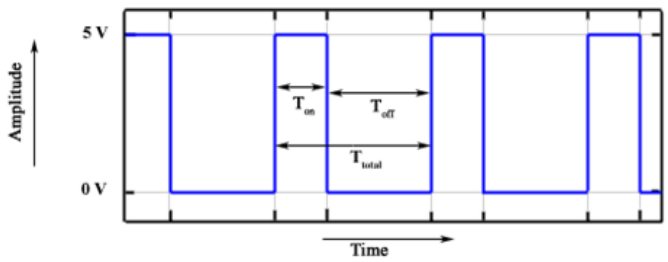
- **VH:** Output Voltage high level
- **VL:** Output Voltage low level
- **VAV:** Average Output Voltage level
- **x:** Duty cycle high level
- **y:** Duty cycle low level

PWM signals when operates at different duty cycle it gives a varying voltage at the output. Voltage regulation method is used in various areas of application like:

- Audio
- LED dimmers
- Analog signal generation (DAC – Digital to Analog Conversion)
- Switching regulators
- and many more…

14.3.1 AVR Timers as PWM

Mostly in AVR Microcontroller the on-chip PWM channel is available which makes the PWM usage much simpler and highly accurate. AVR timers and counters can be used in PWM mode of operation without disturbing the basic timer function.



There are various terms related with the Pulse Width Modulation (PWM):

- Off-Time: Duration of time period when the signal is low.
- On-Time: Duration of time period when the signal is high.
- Duty cycle: It is the percentage of the time period when the signal remains ON during the period of the pulse width modulation signal.
- Period: It is the sum of off-time and on-time of pulse width modulation signal.

Duty Cycle:

Calculation of duty cycle is done by calculating the ON-time from total period of time. It is a ratio between ON-time and total time period of the signal using period calculation, duty cycle is calculated as shown in the equation below:

$$D = \frac{T_{on}}{(T_{on} + T_{off})} = \frac{T_{on}}{T_{total}}$$

Period:

As represented in the above figure, T_{off} represents the off-time and T_{on} represents the on-time of a signal. Period is a sum of both on and off times and period is calculated as shown in the equation below:

$$T_{total} = T_{on} + T_{off}$$

PWM: Voltage Regulation

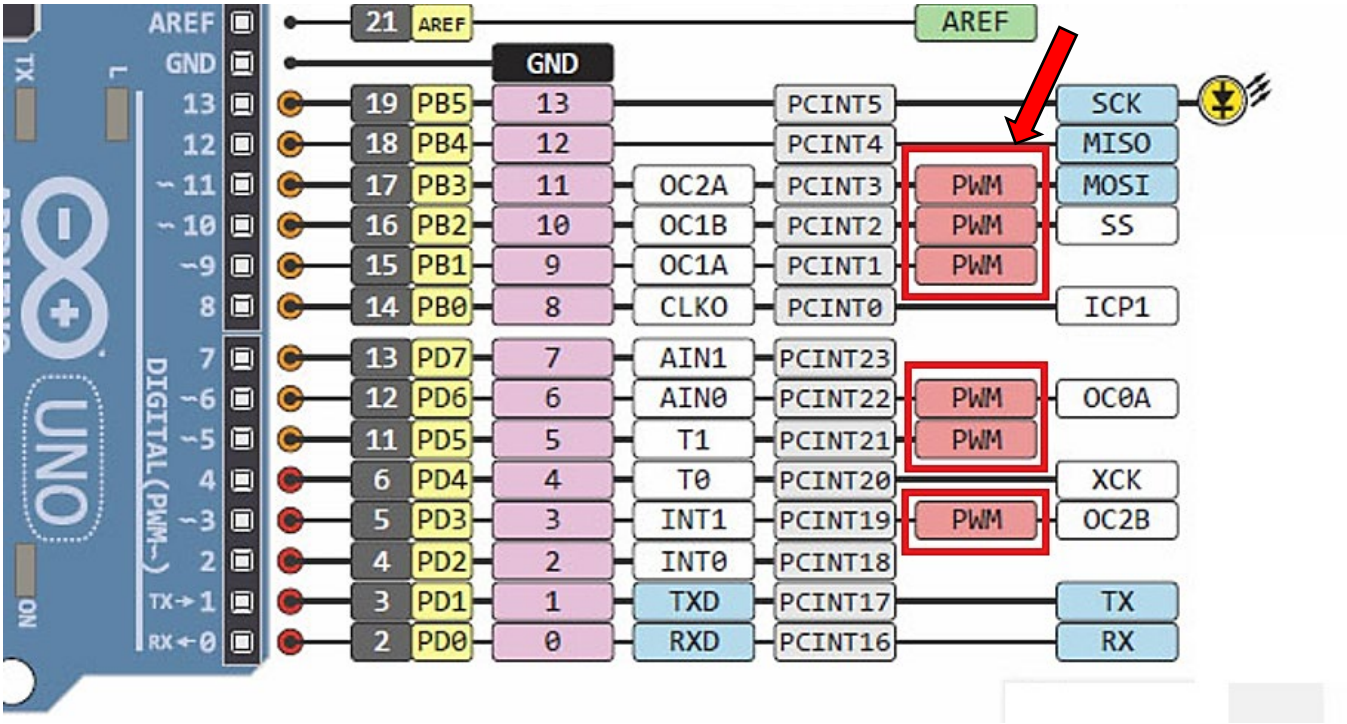
Voltage regulation operation is performed by averaging the PWM signal. Output voltage is calculated as shown in the equation below:

$$V_{out} = D \times V_{in} \quad \boxed{V_{out} = \frac{T_{on}}{T_{total}} \times V_{in}}$$

The output voltage can be directly varied by varying ON time of the pulse width modulated signal.

14.3.2 Analog Write

Analog write in actually done using PWM (Pulse Width Modulation) and it is only available on the pins of ATmega328p having PWM support. On most Arduino, the PWM pins are identified with a ~ sign, like ~3, ~5, ~6, ~9, ~10 and ~11.



PWM Supported Pin of ATmega328p	Pin Label on Arduino UNO Board
PD3	~3
PD5	~5
PD6	~6
PB1	~9
PB2	~10
PB3	~11

Important: A0 to A5 pins on Arduino (PC0 to PC5 pins of ATmega328p) do not support analog write (PWM) functionality. They only support Analog to Digital Conversion (ADC).

14.3.3 Example: LED Dimming using PWM

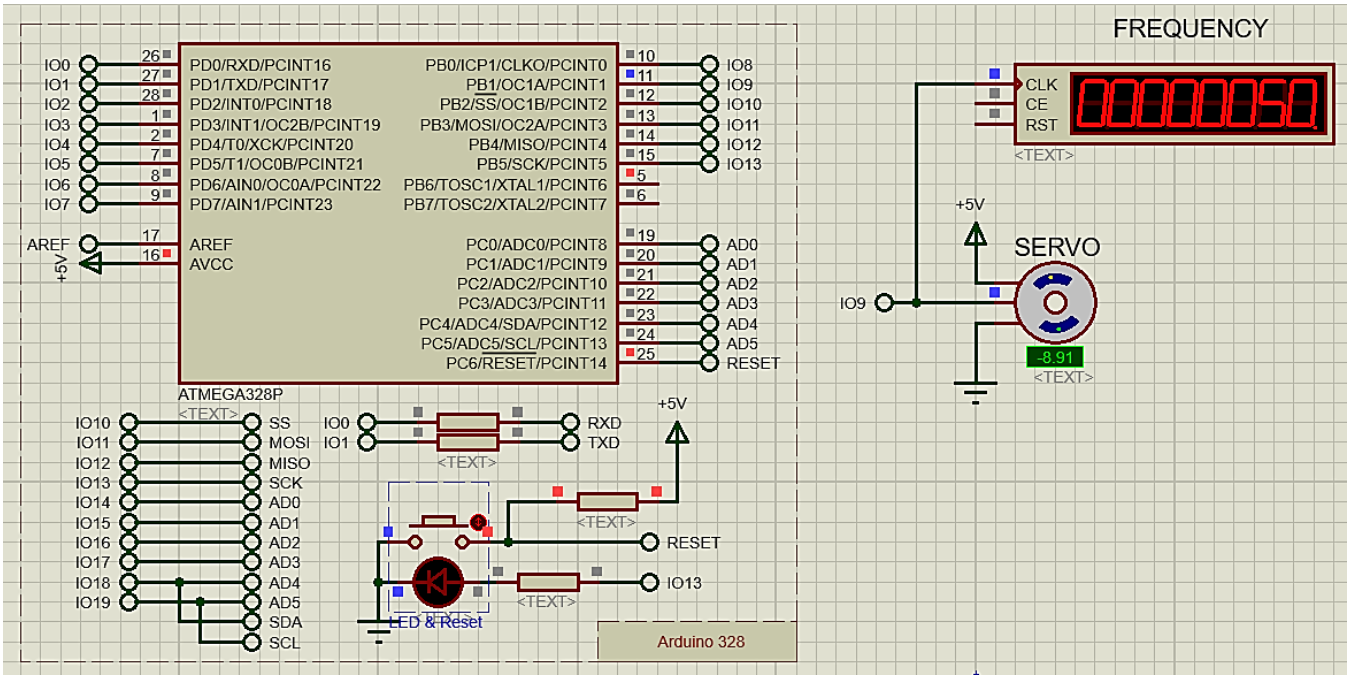
```
.include "m328pdef.inc"
.include "delay_Macro.inc"
.cseg

; Macro to calculate the value for PWM dutycycle and output it on PWM pin
.macro PWM_set_dutycycle
	PUSH r16
	PUSH r17

	; formula  = dutycycle * 256 / 100
	;	ldi r16, dutycycle * 256 / 100
	;	out OCR0A,r16
	; where dutycycle could be from 10 to 90

	; r17 contains the input value
	mov r17, @0
	ldi r16, 2		; 256/100=2.56
	mul r16, r17		; Multiply r16 by r17, result in r1:r0
	mov r16, r0		; Copy the low byte of the result to r16
	mov r17, r1		; Copy the high byte of the result to r17

	; At this point, r16 contains the result of the expression (dutycycle*256/100)
	; So set the PWM dutycycle
	out OCR0A,r16

	POP r17
	POP r16
.endmacro


.org 0x0000

; set output pin PD6
sbi DDRD,PD6			; OC0A (PD6 pin) as PWM output pin

; Timer 0 in Fast PWM mode, output A low at cycle start
ldi r16,(1<<COM0A1)|(1<<COM0A0)|(1<<WGM01)|(1<<WGM00)
out TCCR0A,r16			; to timer control port A

; Start Timer 0 with prescaler = 1
ldi r16,1<<CS00			; Prescaler = 1
out TCCR0B,r16			; to timer control port B

loop:
	; set brightness value from 10 to 90 in r17
	; 10 gives ~maximum duty cycle (maximum LED brightness)
	; 90 gives ~minumum duty cycle (minumum LED brightness)
	ldi r16, 90
	PWM_set_dutycycle r16
	delay 500

	ldi r16, 80
	PWM_set_dutycycle r16
	delay 500
```

```
        ldi r16, 50
        PWM_set_dutycycle r16
        delay 500

        ldi r16, 30
        PWM_set_dutycycle r16
        delay 500

        ldi r16, 10
        PWM_set_dutycycle r16
        delay 500
rjmp loop
```

The above code will change brightness on a LED. You can change the highlighted value from 0 to 99 to get different brightness levels.

14.3.4 Example: Servo Motor Controlling using PWM

To control a servo motor just a control signal needs to be feed to the servo to position it in any specified angle. The frequency of the control signal is 50hz (i.e., the period is 20ms) and the width of positive pulse controls the angle.




```
.include "m328pdef.inc"
.include "delay_Macro.inc"
.include "16bit_reg_read_write_Macro.inc"
.cseg
.org 0x0000
	SBI DDRB, PB1	; Set pin PB1 as output for OC1A (for Servo Motor)

	; Timer 1 setup PWM
	; Set OC1A and WMG11.
	LDI r16, 0b10100010
	STS TCCR1A, r16
	; Set WMG12 and WMG13 and Prescalar to 8
	LDI r16, 0b00011010
	STS TCCR1B, r16
	; clear counter
	LDI r16, 0
	STSw TCNT1H, r16, r16
	; count of 40000 for a 20ms period or 50 Hz cycle
	LDI r16, LOW(40000)
	LDI r17, HIGH(40000)
	STSw ICR1H,r17,r16
loop:
	; use value from 900 to 4900 to change the angle of the servo accordingly

	; 0 degree
	LDI r16, LOW(900)
	LDI r17, HIGH(900)
	STSw OCR1AH,r17,r16
	delay 1000
```

```
        ; 90 degree
        LDI r16, LOW(2900)
        LDI r17, HIGH(2900)
        STSw OCR1AH,r17,r16
        delay 1000

        ; 180 degree
        LDI r16, LOW(4900)
        LDI r17, HIGH(4900)
        STSw OCR1AH,r17,r16
        delay 1000

rjmp loop
```

The above code will rotate the 9g servo to different positions. You can change the highlighted value from 900 to 4900 to get different angles of the servo motor.

The *16bit_reg_read_write_Macro.inc* file contains these macros to read and write the 16-bit registers.

```
; Macros to Read/Write 16-bit registers

; Usage example:
;       LDSw r17, r16, TCNT1H     ; Reads the counter value (high, low, adr)
;       STSw TCNT1H, r17, r16     ; Writes the counter value (adr, high, low)

.macro STSw
        cli
        STS @0, @1
        STS @0-1, @2
        sei
.endmacro

.macro LDSw
        cli
        LDS @1, @2-1
        LDS @0, @2
        sei
.endmacro
```

14.4 Division in AVR Assembly

AVR does not have the ability to do hardware division (i.e., there are no division instructions), so subroutines or macros must be written using other instructions to accomplish this task.

An 8-bit unsigned division macro is shown below:

```
.macro div
;***************************************************************************
;*
;* "div8u" - 8/8 Bit Unsigned Division
;*
;* This macro divides the two register variables "r16" (dividend) and
;* "r17" (divisor). The result is placed in "r16" and the remainder in
;* "r15".
;*
;***************************************************************************
;* Register Variables:
;      r15          ;remainder
;      r16          ;result
;      r16          ;dividend
;      r17          ;divisor
;      r18          ;loop counter

       push r18
div8u:
       sub r15,r15         ;clear remainder and carry
       ldi r18,9           ;init loop counter
d8u_1:
       rol r16             ;shift left dividend
       dec r18             ;decrement counter
       brne d8u_2          ;if done
       rjmp exit           ;return
d8u_2:
       rol r15             ;shift dividend into remainder
       sub r15,r17         ;remainder = remainder - divisor
       brcc d8u_3          ;if result negative
       add r15,r17         ;restore remainder
       clc                 ;clear carry to be shifted into result
       rjmp d8u_1          ;else
d8u_3:
       sec                 ;set carry to be shifted into result
       rjmp d8u_1
exit:
       pop r18
.endmacro
```

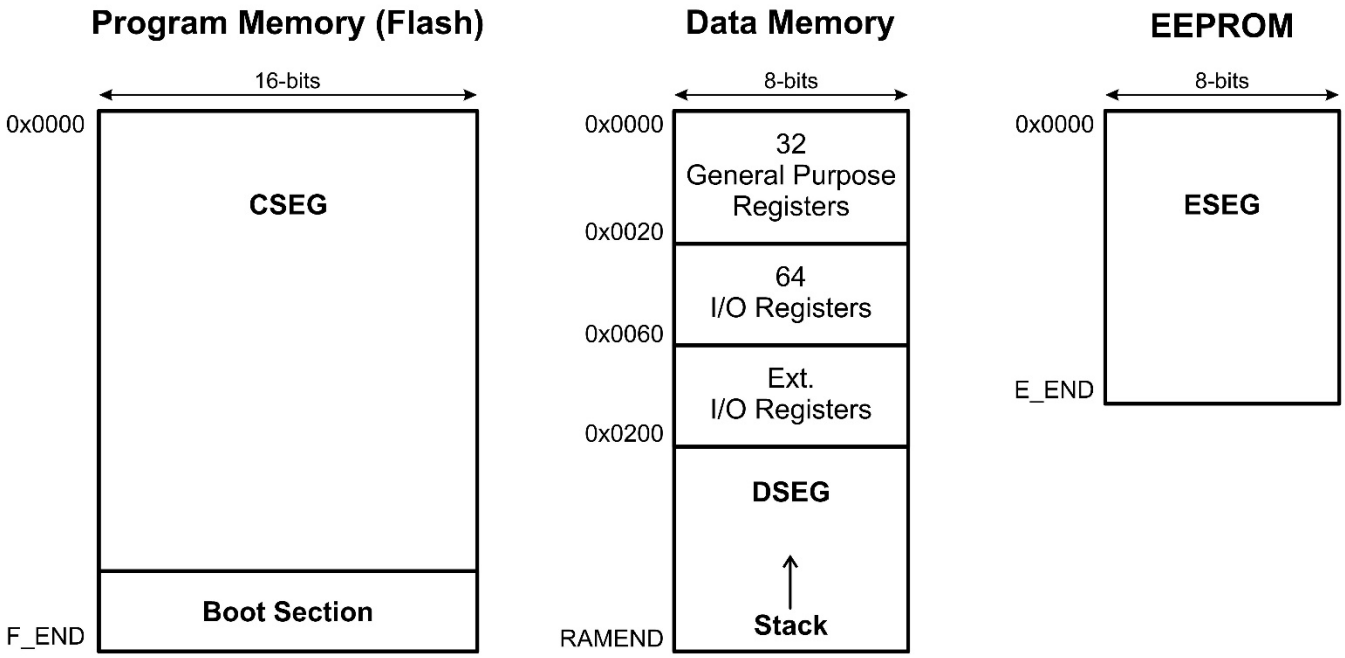
In this macro, the dividend is passed in register r16, and the divisor is passed in register r17. When this macro is called, the division result will be stored in r16, and the remainder will be in r15.

```
       ldi r16, 125        ; dividend
       ldi r17, 10         ; divisor
       div                 ; div macro
```

After the execution, r16 will hold the result 12 while r15 will hold the modulus 5.

14.5 SRAM Data Read/Write (Variables)

There are three types of memories in AVR microcontrollers. The memories of ATmega328p are as follows:



When accessing the Extended I/O Registers and SRAM, a more general set of instructions must be used, shown in the table below.

Mnemonic	Description
LDS	load direct from data space
STS	store direct to data space

14.5.1 Declaring Variable in SRAM

Space can be allocated with the directive *.byte*, followed by the number of bytes to allocate, e.g.

```
.dseg
      var1:  .byte  1            ; allocate 1 byte in RAM
```

It is recommended to declare the variables in the RAM from its starting address *SRAM_START* (an address defined in the include file). For that purpose, use *.org* directive before declaring variables.

```
.dseg
.org SRAM_START
      var1:  .byte  1            ; allocate 1 byte in RAM
```

Note: We cannot initialize the variables in the Data Memory when programming, we can only allocate space for them and initialize them at runtime. We will insert the data in these variables (memory addresses) at runtime.

For example:

```
.include "m328pdef.inc"
.include "delay_Macro.inc"
.include "UART_Macros.inc"
.include "div_Macro.inc"

.dseg
.org SRAM_START
      ; Allocate 1 BYTE in RAM (Note: this is not the value of the register)
      var1:  .byte  1
      var2:  .byte  1
      sum:   .byte  1

.cseg
.org 0x0000

      Serial_begin              ; initilize UART serial communication

      ldi    r16, 5
      ldi    r17, 6

      sts    var1, r16          ; store r16 in variable var1
      sts    var2, r17          ; store r17 in variable var2

      lds    r18, var1          ; load value of var1 into r18
      lds    r19, var1+1        ; another way to access var2 by offset from var1

      add r18, r19
      sts    sum, r18           ; store sum into sum variable

loop:
      Serial_writeReg_ASCII r18       ; print the sum on Serial Terminal
      Serial_writeNewLine
      delay 1000
rjmp loop
```