Today session Objective

Interface with new Hardware which is 7-segment with GPIO peripheral, Then go deep in Interrupts.

2nd Peripheral (Interrupt)

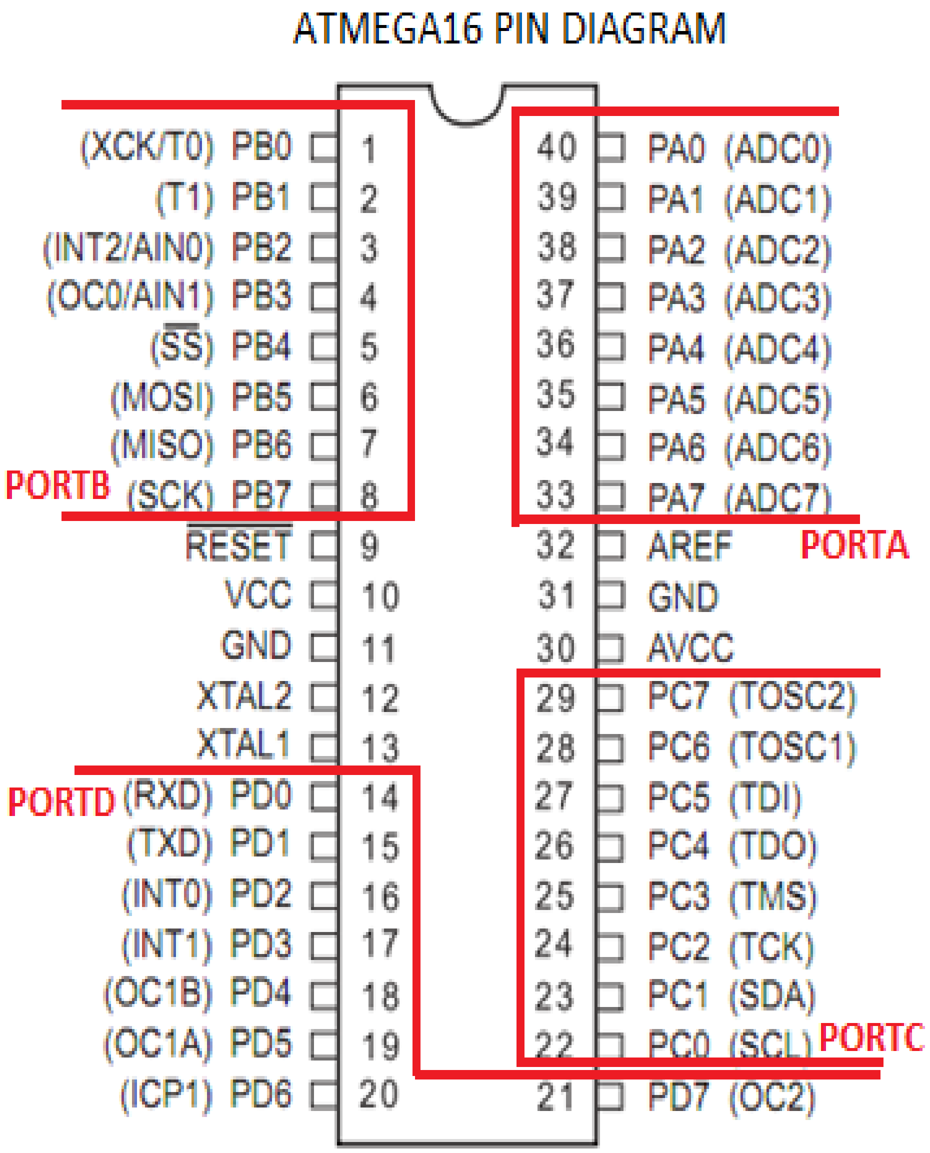
D-BUGERZ

- Recall Last session
- Interfacing with 7-Segment
- Introduction to Interrupt
- Start Coding with External Interrupt

Introduction to GPIO

1.
2.
3.
4.

- ❑ ATmega16/32 Microcontrollers has 32 programmable I/O pins constituting four ports.
- ❑ Each port has 8 pins. The pins of these four ports can be used as general-purpose inputs/outputs.
- ❑ Each I/O pin could be Input or Output (Multiplexed Direction). This could be controlled through the I/O Port direction register.
- ❑ Each Microcontroller pin could be assigned to different I/O functions. Pin could be configured for digital I/O, analog input, timer I/O, or serial I/O. For example PDO in ATmega16/32 microcontrollers could be digital I/O or UART serial input (Multiplexed Functionality).



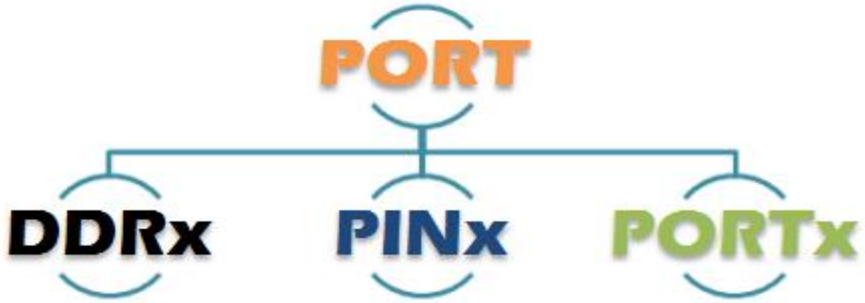
Introduction to GPIO

1.

2.

3.

4.



❑ Each PORT is controlled by 3 registers:

1. DDRX (Data Direction Register)

This register used to decide the **direction** of the pins, i.e. whether the pins will act **as input pins or as output pins**.

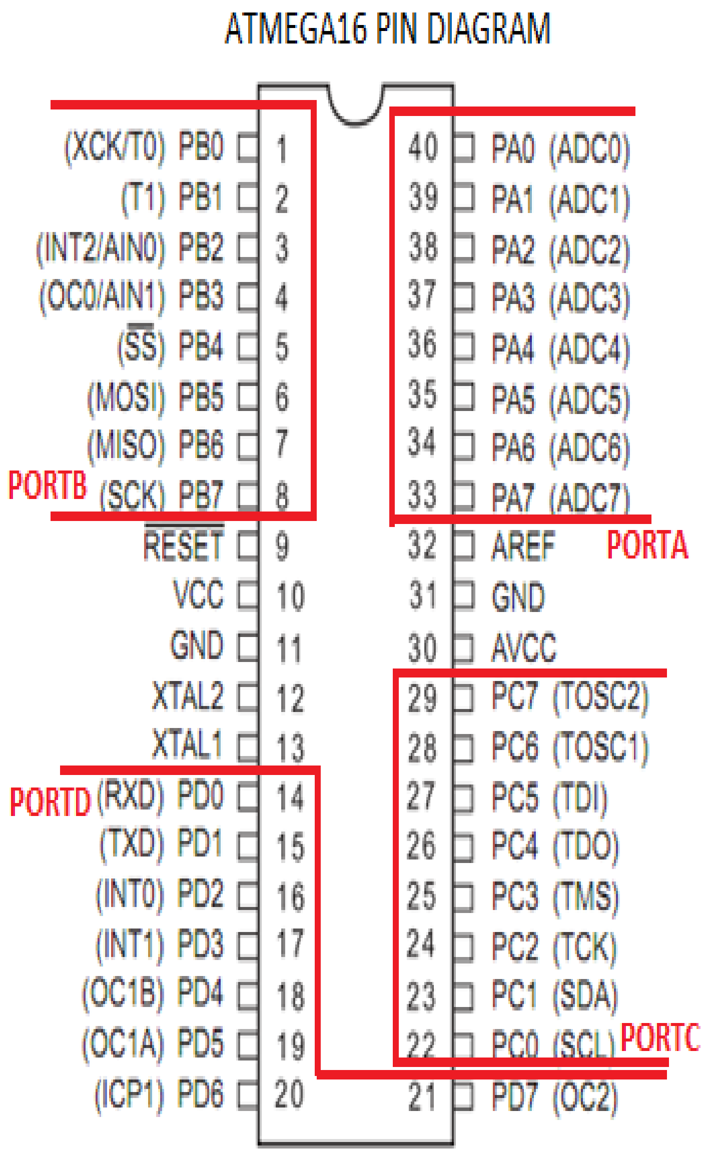
2. PORTX (Output Register)

This register is used to **set the logic** on the output pins HIGH or LOW.

3. PINX (Input Register)

This register is used to **read the logic** level on the port input pins.

- Note: x could be A, B, C, or D depending on which port registers are being addressed.



D-BUGERZ

Question??

Want Pin 7 & Pin 2 in PORTA as Output and pin 0 as input and make output 1 on Pin 7 and 0 on Pin 2 and check input on bit 0 ??

<u>Pin 7</u> DDRA = 1000 0000 (128 in decimal)(0x80 in HEXA)

<u>Pin 2</u> DDRA = 1000 0100 (132 in decimal)(0x84 in HEXA)

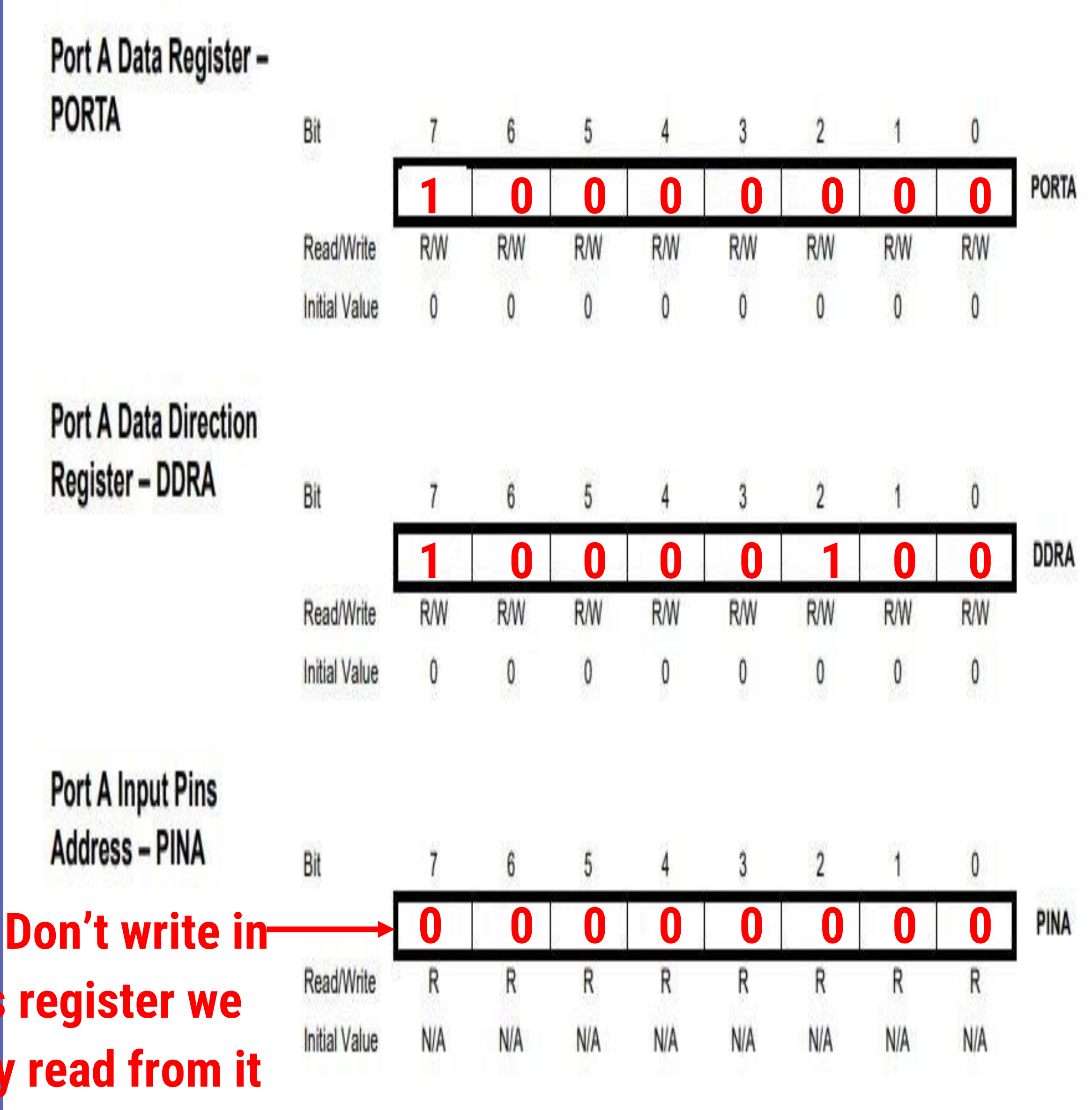
<u>Pin 0</u> DDRA = 1000 0100 (132 in decimal)(0x84 in HEXA)

<u>Pin 7</u> PORTA = 1000 0000 (128 in decimal)(0x80 in HEXA)

<u>Pin 2</u> PORTA = 1000 0000 (128 in decimal)(0x80 in HEXA)

To check on input Pin 0 :

<u>Pin 0</u> if(PINA & (1<<0));



Port A Data Register – PORTA

Bit	7	6	5	4	3	2	1	0	
	1	0	0	0	0	0	0	0	PORTA
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	

Port A Data Direction Register – DDRA

Bit	7	6	5	4	3	2	1	0	
	1	0	0	0	0	1	0	0	DDRA
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	

Port A Input Pins Address – PINA

Bit	7	6	5	4	3	2	1	0	
	0	0	0	0	0	0	0	0	PINA
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

We Don't write in this register we only read from it

Introduction to GPIO

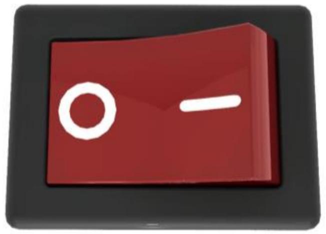
❑ Input devices

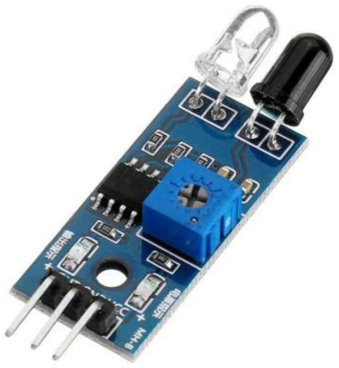
1- Switches(Push button, Keypad etc.)

2- Digital /Analogue sensor

3- Signal from another microcontroller

Electrical Switch





❑ Output devices

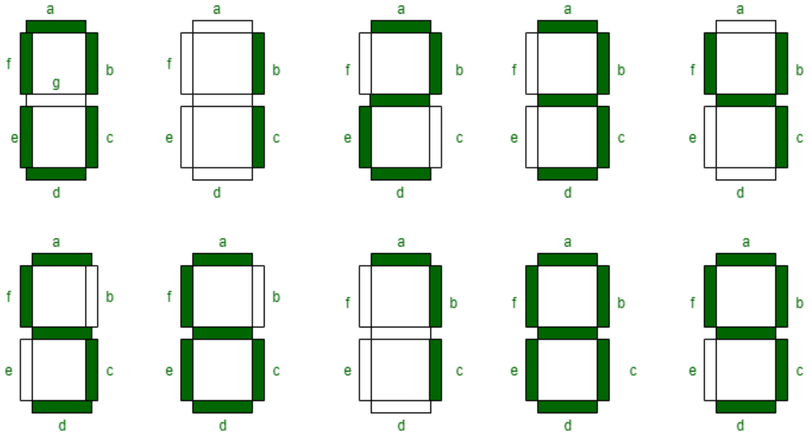
1- LED

2- 7-SEGMENT

3- LCD display

4- Motors

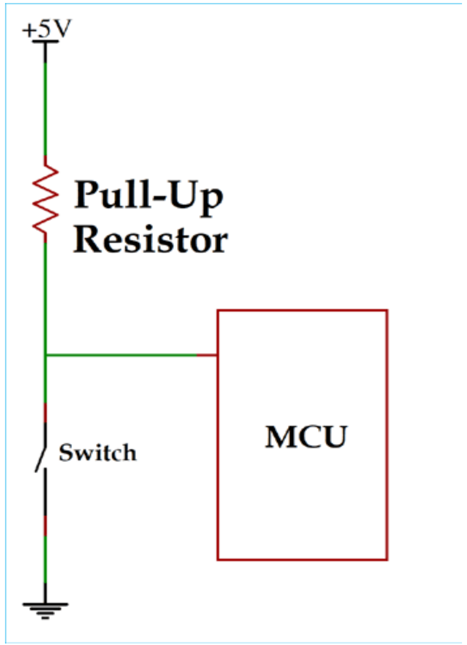
5- Buzzer



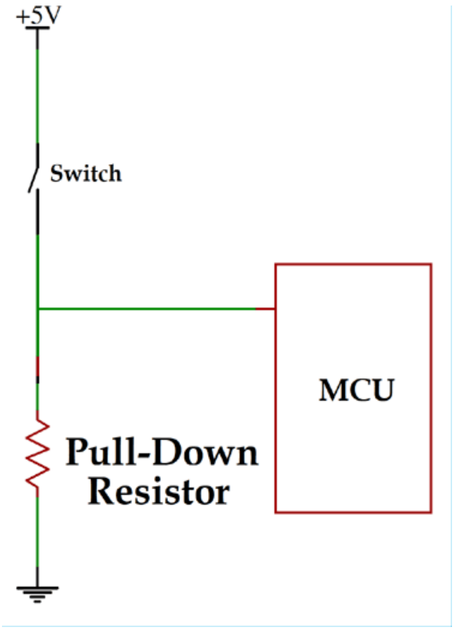
Connection

1.
2.
3.
4.

Switches(Push button, Keypad etc.)

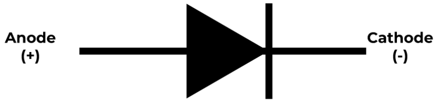


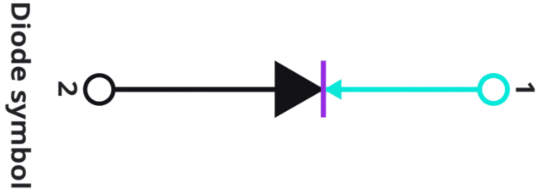
Pull UP Resistor

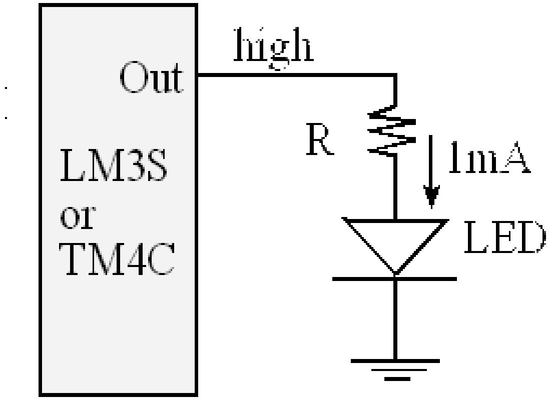


Pull Down Resistor

LED

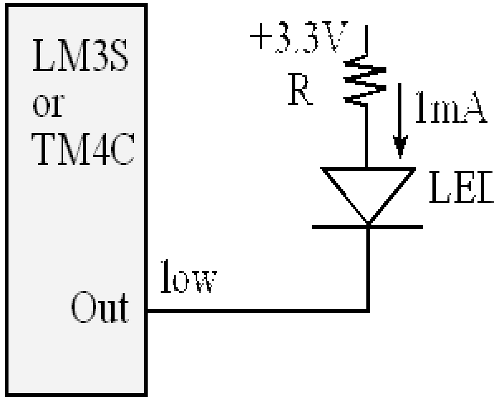






(b) Positive logic interface

Positive Logic



(c) Negative logic interface

Negative Logic

Recall Last session

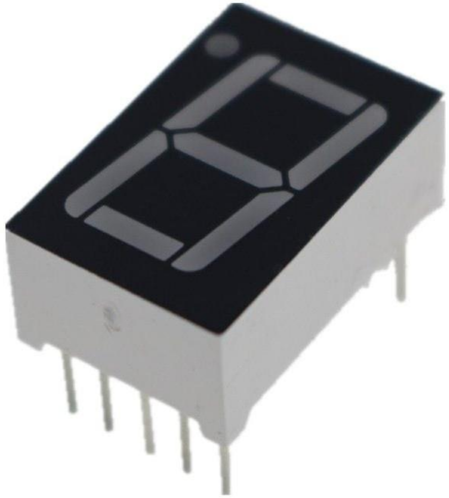
Interfacing with 7-Segment

Introduction to Interrupt

Start Coding with External Interrupt

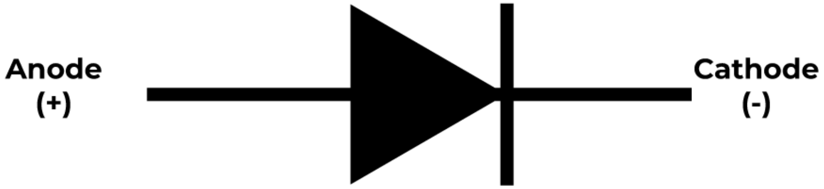
Interfacing With 7-Segment

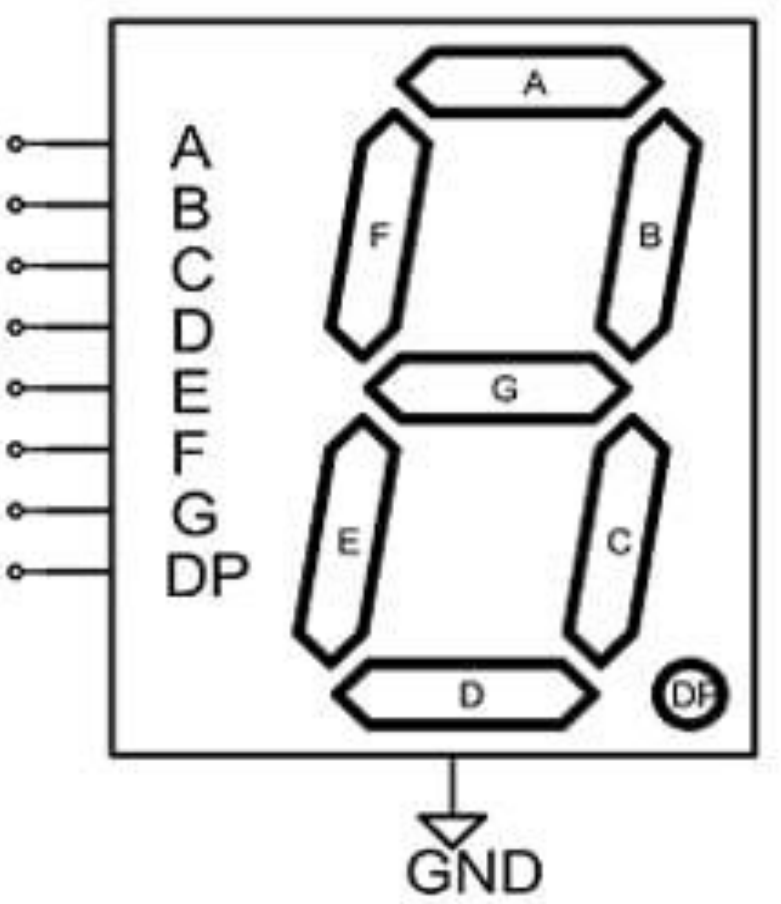
- 7-segment displays are made up of 8 LED segments 7 of these LED segments are in the shape of a line, whereas 1 segment is circular
- The 7 line shaped LED segments are used for displaying numbers 0 to 9 and a few letters like A, B, C, D, E, F, G, etc. The circular segment is used for displaying decimal point
- Used in: Digital Clocks and Timers, Instrumentation and Measurement Devices , Information Displays



Interfacing With 7-Segment

- ❑ Each of the 8 elements has a pin associated with it which can be driven HIGH or LOW according to type of display and the number to be displayed
- ❑ There are two types of 7-segments Common Anode and Common Cathode



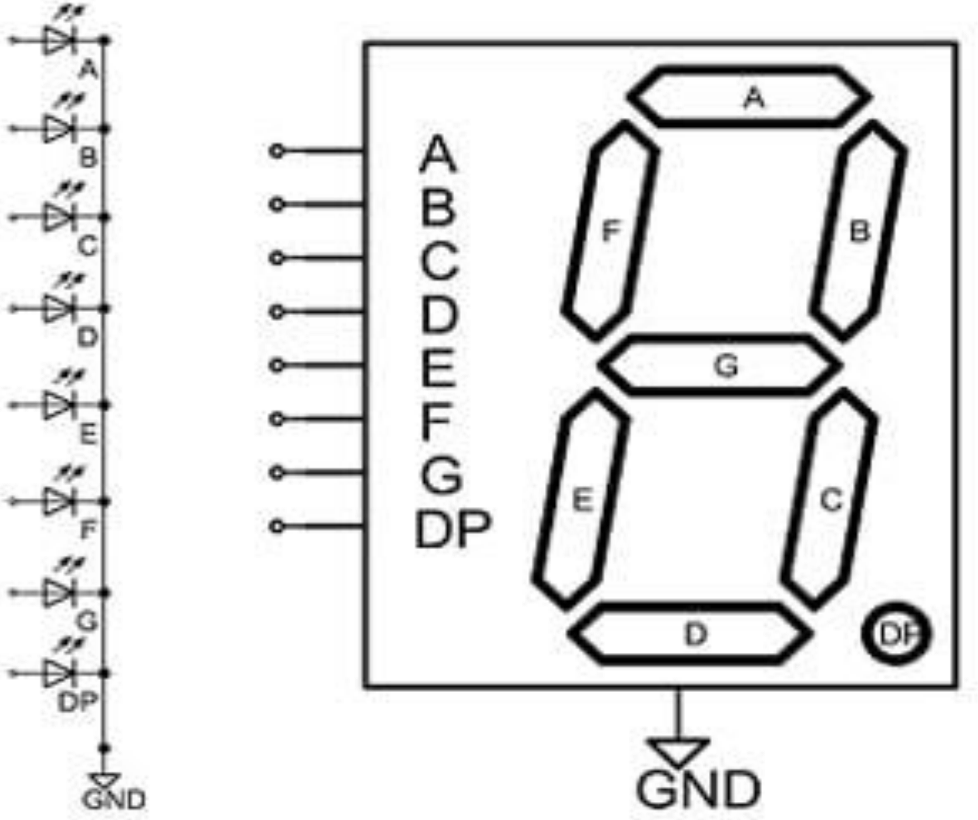


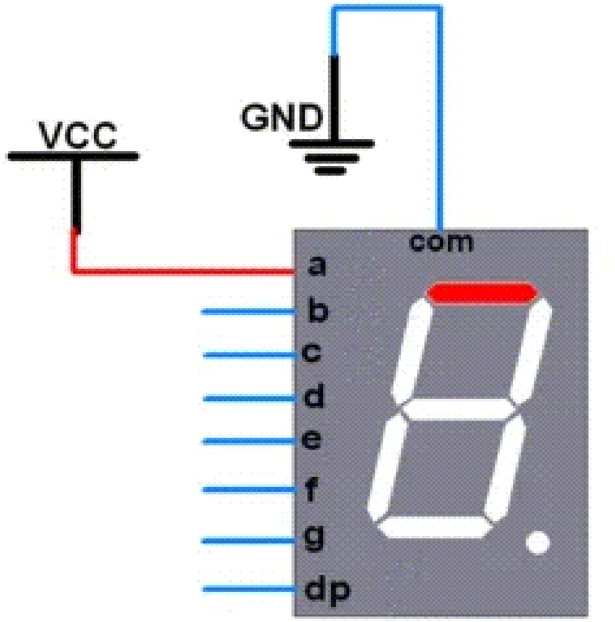
Interfacing With 7-Segment

1.
2.
3.
4.

Common Cathode

- ❑ All the cathode connections of the LED segments are joined together to logic "0" or ground. The individual segments are illuminated by application of a "HIGH", or logic 1 signal via a current limiting resistor to forward bias the individual Anode terminals(a-g)





Common Cathode

Interfacing With 7-Segment

D-BUGERZ

1.

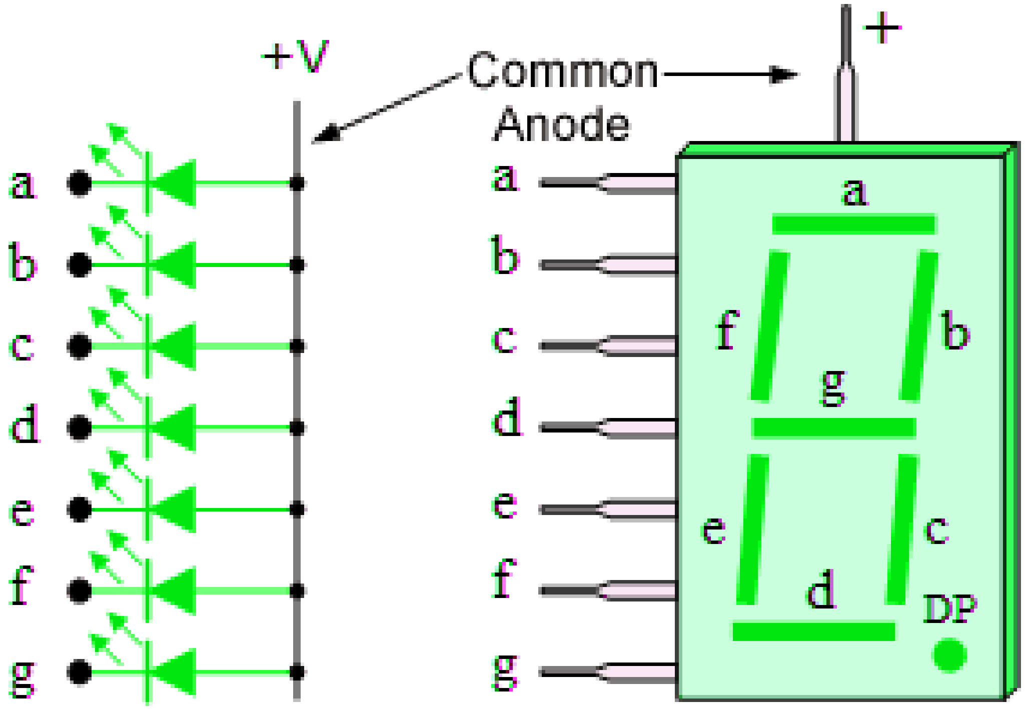
2.

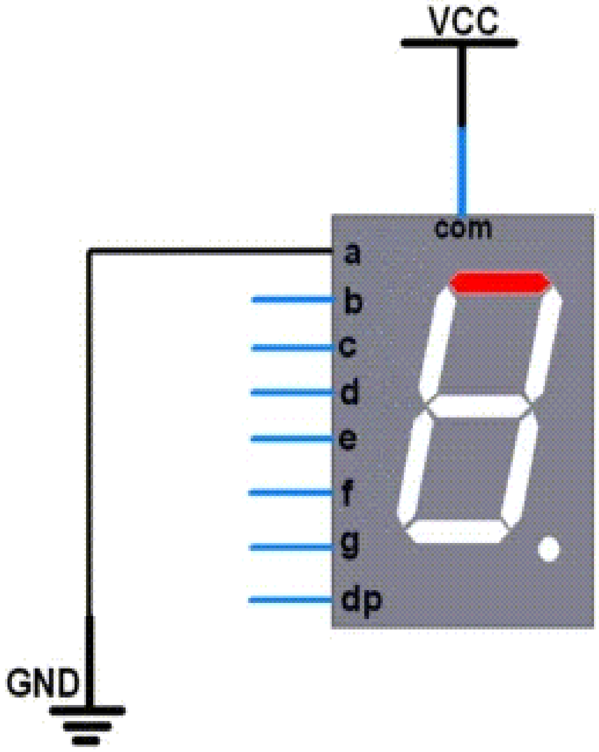
3.

4.

Common Anode

- ❑ All the anode connections of the LED segments are joined together to logic "1". The individual segments are illuminated by applying a ground, logic "0" or "LOW signal via a suitable current limiting resistor to the Cathode of the particular segment (a-g)

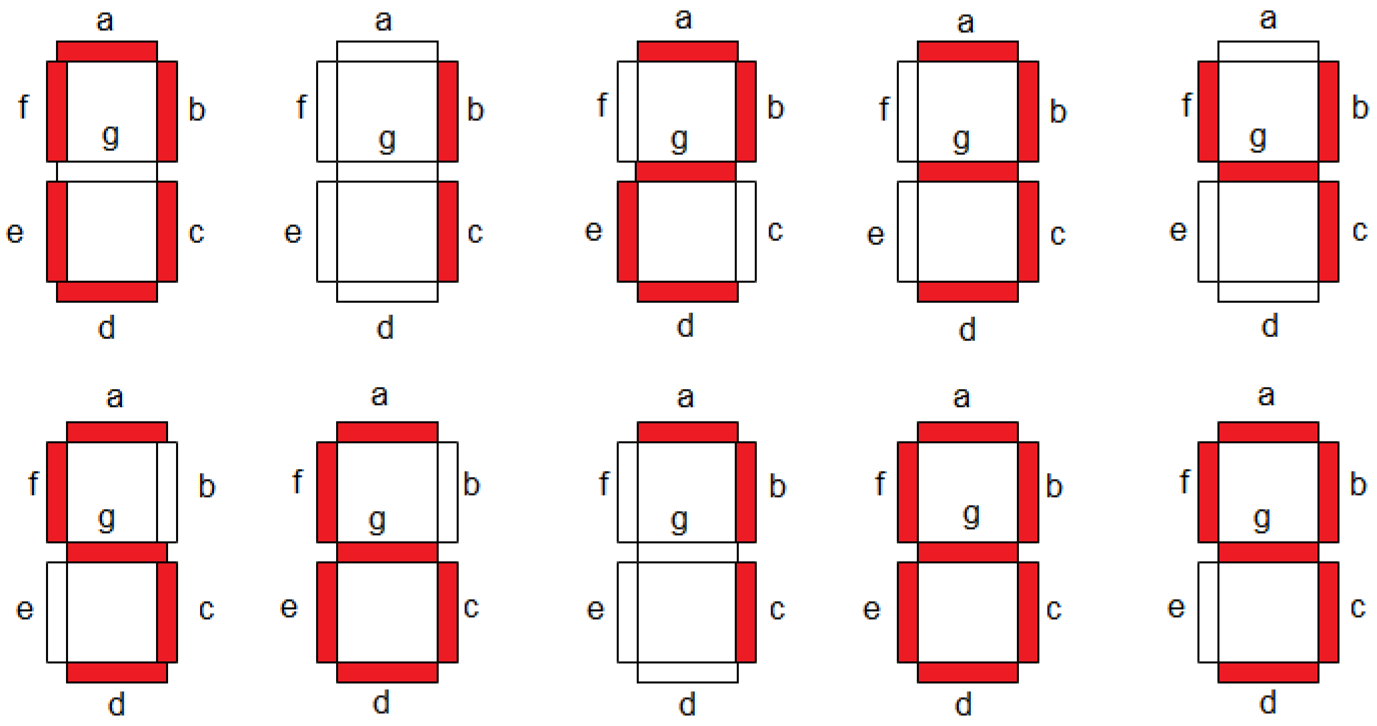


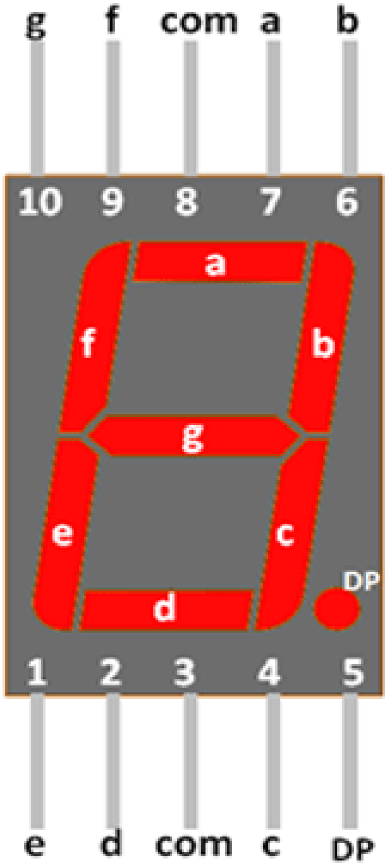


Common Anode

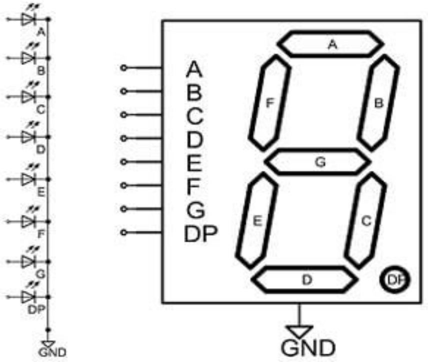
Interfacing With 7-Segment

- ❑ Depending upon the decimal digit to be displayed, the particular set of LEDs is turned on. For example, to display the numerical digit 0, we will need to light up six of the LED segments corresponding to a, b, c, d, e and f. so the various digits from 0 through 9 can be displayed using a 7-segment display as shown



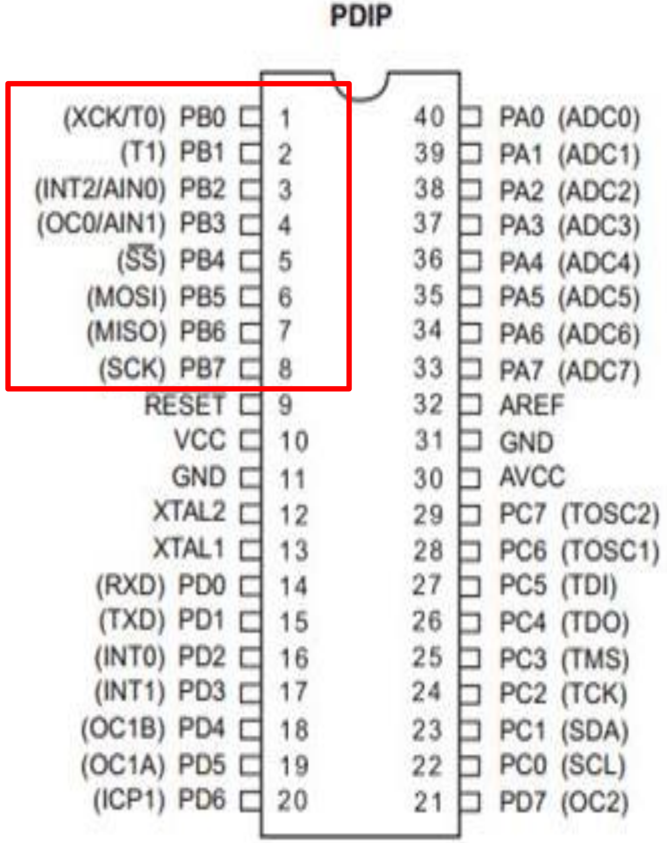


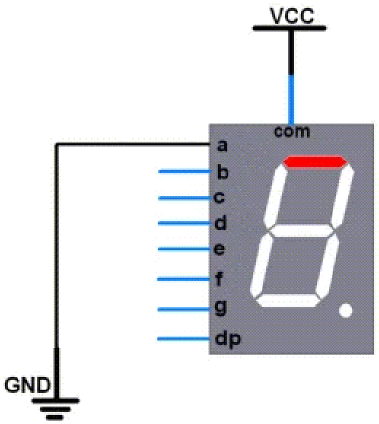
Interfacing With 7-Segment



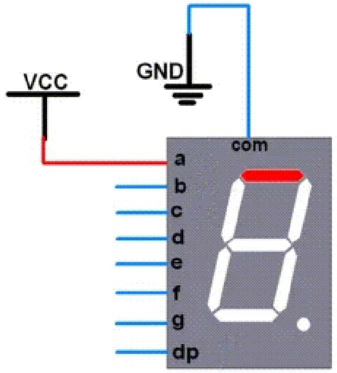
Problems with interfacing with 7-segment:

- ❑ Need Full port for each 7-segment and this consume much pins
- ❑ Need put 1 and 0 manually for each pin
- ❑ Code depend on type of 7-segment





Common Anode



Common Cathode

1.

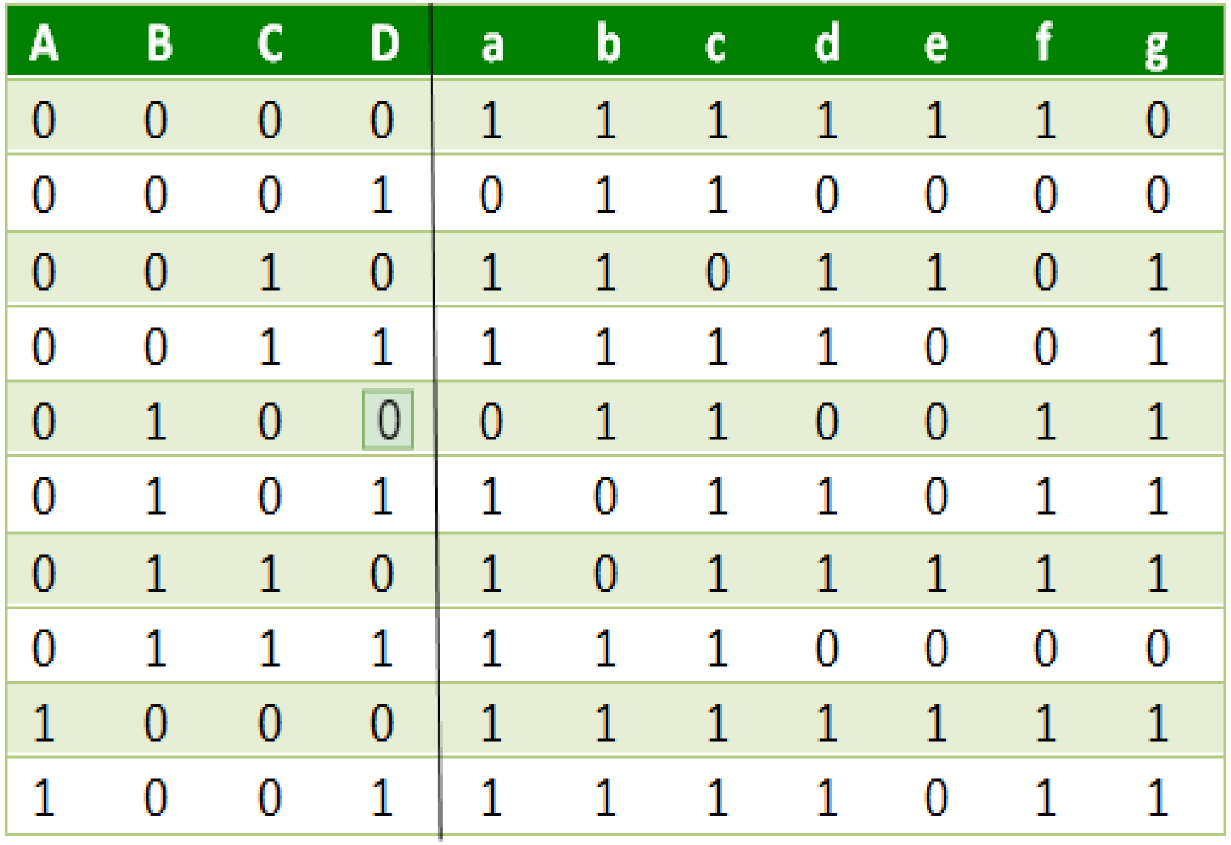
2.

3.

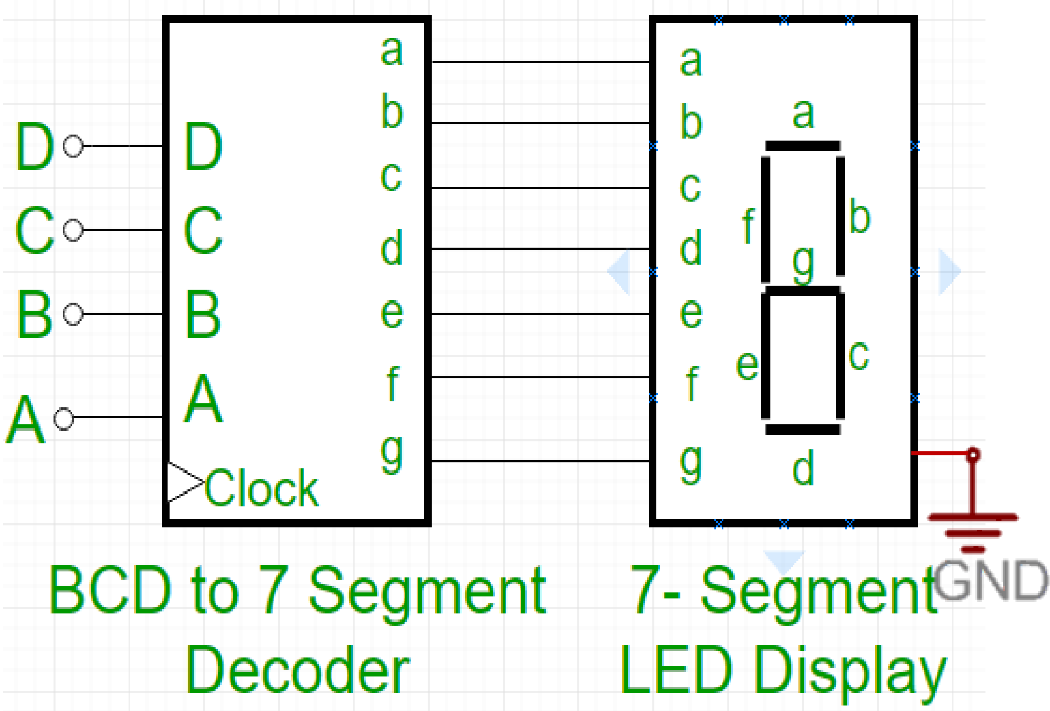
4.

Common Cathode BCD

- ❑ **BCD to 7-Segment Display Decoders:** A binary coded decimal (BCD) to 7-segment display decoder such as 74LS48 for common cathode 7-segment, have 4 BCD inputs and 7 output lines, one for each LED segment. This allows a smaller 4-bit binary number to be used to display all the numbers from 0 to 9.



A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

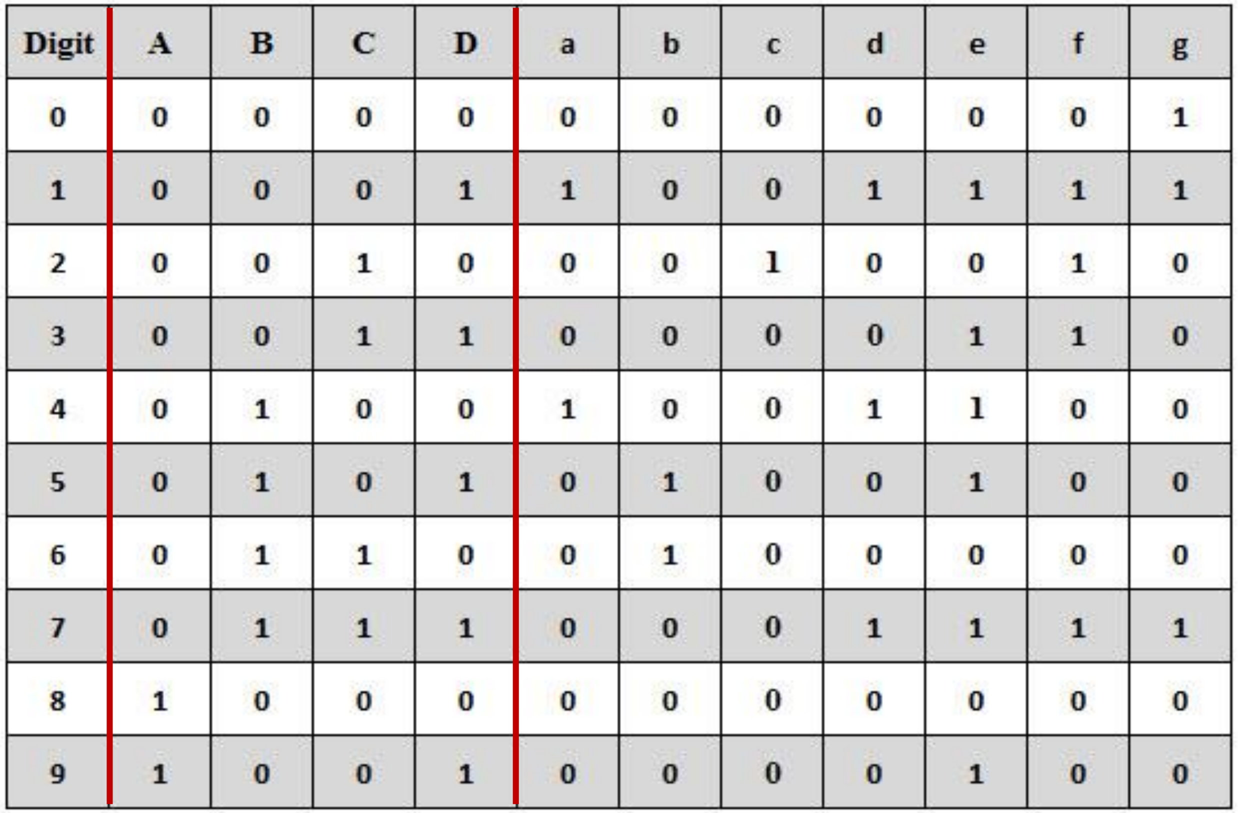


Interfacing With 7-Segment

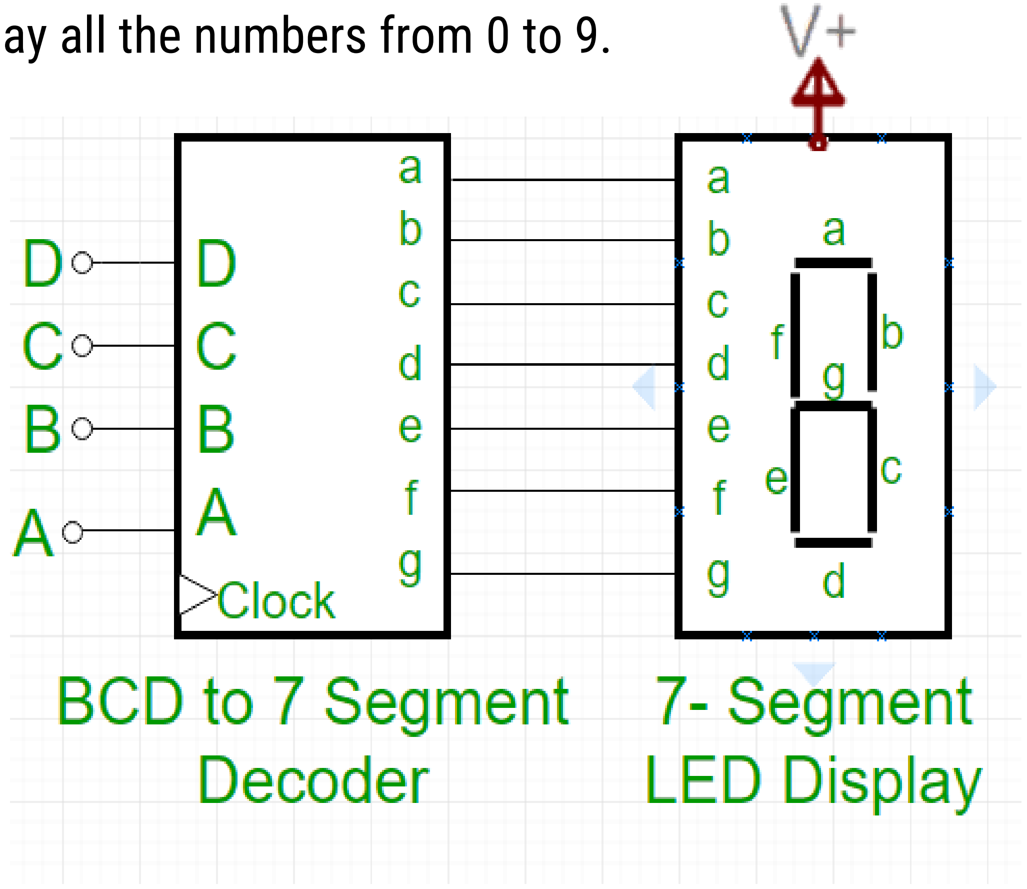
1.
2.
3.
4.
5.

Common Anode BCD

- ❑ **BCD to 7-Segment Display Decoders:** A binary coded decimal (BCD) to 7-segment display decoder such as the TTL 74LS47 for common anode 7-segment , have 4 BCD inputs and 7 output lines, one for each LED segment. This allows a smaller 4-bit binary number to be used to display all the numbers from 0 to 9.



Digit	A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	1	1	1	1
2	0	0	1	0	0	0	1	0	0	1	0
3	0	0	1	1	0	0	0	0	1	1	0
4	0	1	0	0	1	0	0	1	1	0	0
5	0	1	0	1	0	1	0	0	1	0	0
6	0	1	1	0	0	1	0	0	0	0	0
7	0	1	1	1	0	0	0	1	1	1	1
8	1	0	0	0	0	0	0	0	0	0	0
9	1	0	0	1	0	0	0	0	1	0	0



Interfacing With 7-Segment

1.
2.
3.
4.

❑ Check on 4 bits I need to write

```
&   0000 1001  (9)
    0000 1111  (0x0F)
  ---------------------
    0000 1001  (9)
```

NOTES:

❑ Any insert with OR (|) and Any check with AND(&)

❑ You must zero the PORT first then insert to make correct inset in desired bits in port

1- Delete port to write new number on PORT

```
&   0000 1111  (PORTC)
    1111 0000  (0xF0)
  ---------------------
    0000 0000  (PORTC)
```

2- Write the number you need on PORT

```
|   0000 1001  (Number you need ex:9)
    0000 0000  (PORTC)
  ---------------------
    0000 1001  (PORTC)
```

1.

2.

3.

4.

Exercise 4

1.
2.
3.
4.

Write Embedded C code using ATmega16/32 µC to control a 7-Segment using a Push Button.

Requirements:

1- Configure the µC clock with internal 1Mhz Clock
2- The Push Button is connected to pin 4 in PORTD.
3- Connect the Push Button using Pull Down configuration.
4- The 7-Segment is connected to first 4-pins of PORTC.
5- The 7-Segment type is common anode.
6- If the Push Button is pressed just increase the number appeared in the 7-Segment display, and if the 7-segment reaches the maximum number (9) overflow occurs.

Interfacing With 7-Segment

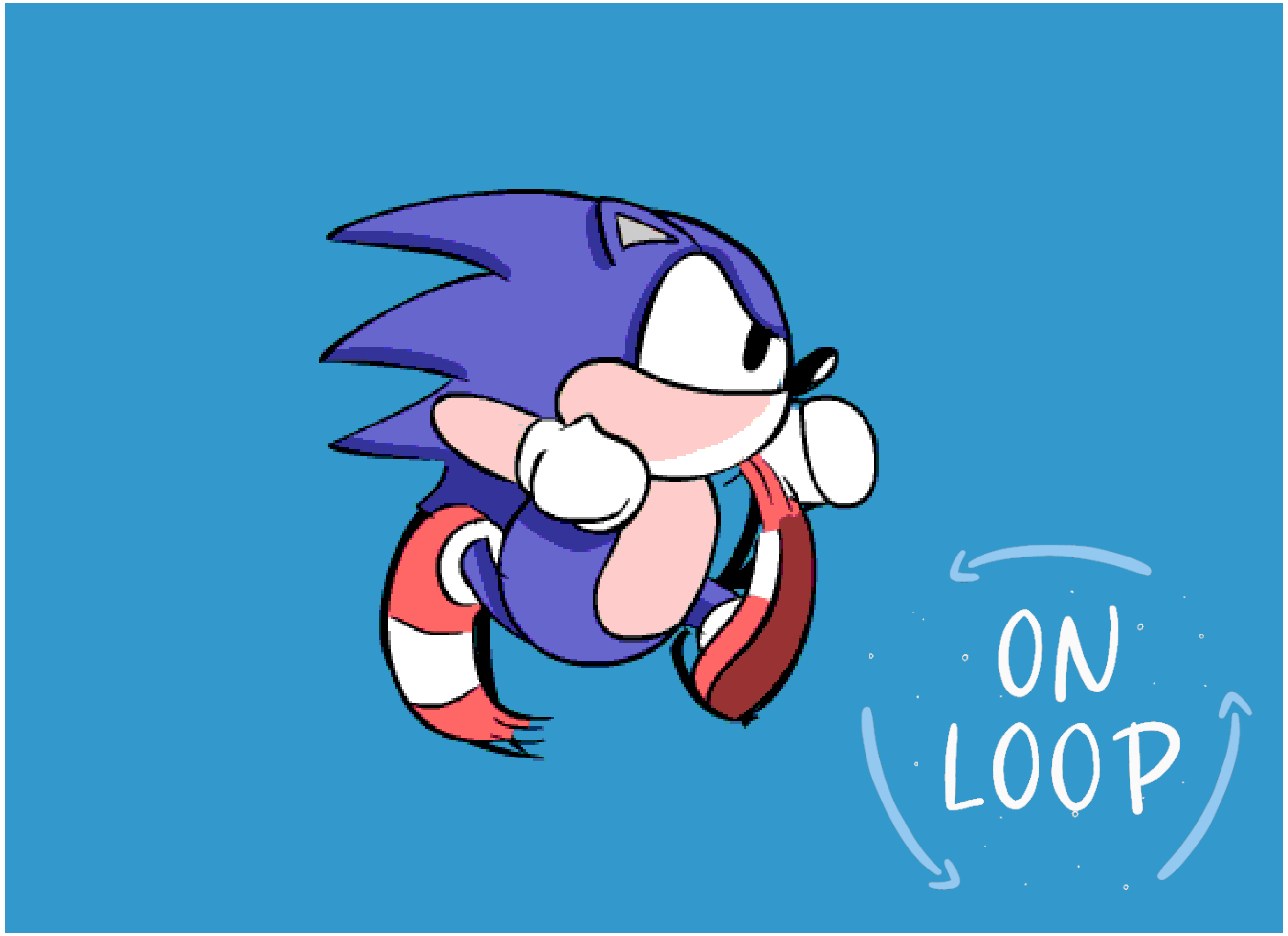
1.

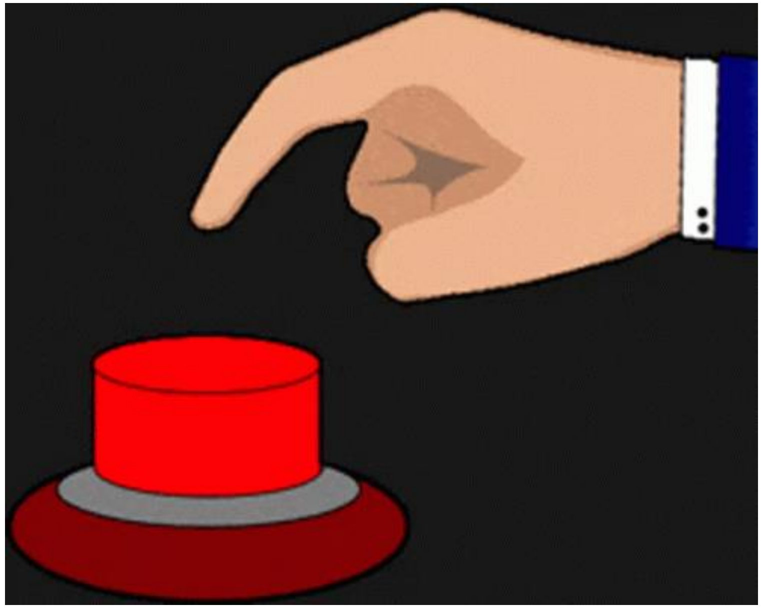
2.

3.

4.

```
while( PIND & (1<<PD4) ){} // wait until switch is released you will still here if the button is pressed
```





1.

2.

3.

4.

Challenge 3

D-BUGERZ

Write Embedded C code using ATmega16/32 µC to control a 7-Segment using two Push Buttons.

Requirements:

- ✓ Configure the µC clock with 16Mhz Crystal Oscillator
- ✓ The two Push Buttons are connected to pin 2 and 3 in PORTD
- ✓ Connect the Push Button using Pull Up configuration
- ✓ The 7-Segment is connected directly to PORTA from PA1 PA7 without a decoder
- ✓ The 7-Segment type is common cathode and enable the first 7-segment which its common pin controlled by PC6 pin
- ✓ If the Push Button 1 is pressed just increase the number appeared in the 7-Segment display, and if the number reaches the maximum number (9) then do nothing
- ✓ If the Push Button 2 is pressed just decrease the number appeared in the 7-Segment display, and if the number reaches the minimum number (0) then do nothing

BREAK 10 MIN

Recall Last session

Interfacing with 7-Segment

Introduction to Interrupt

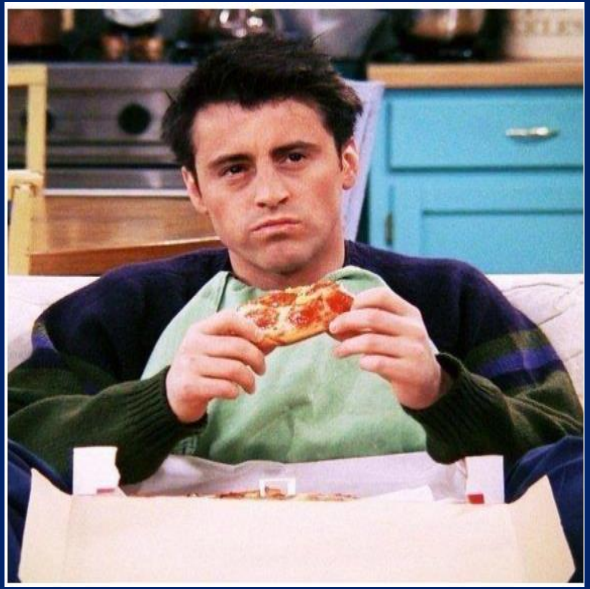
Start Coding with External Interrupt

D-BUGERZ

Imagine that you are eating your favourite meal and then your doorbell rang ! This is the interrupt.

You will open your door to see who is in then you will return back to continue eating your favourite meal

☺ This is the interrupt service routine.



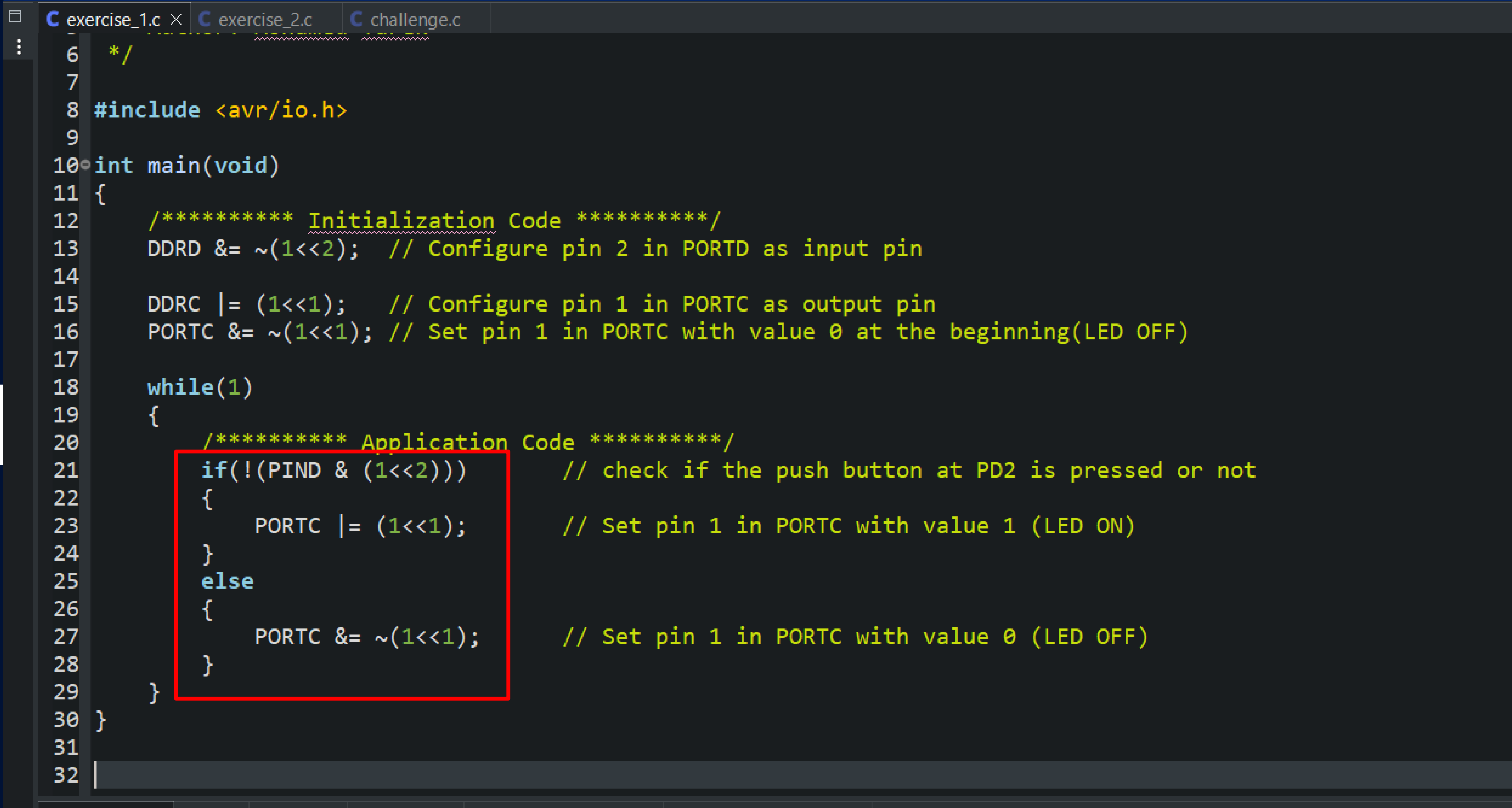
Code Techniques

1.

2.

3.

4.



```
 */

#include <avr/io.h>

int main(void)
{
    /********** Initialization Code **********/
    DDRD &= ~(1<<2);  // Configure pin 2 in PORTD as input pin

    DDRC |= (1<<1);   // Configure pin 1 in PORTC as output pin
    PORTC &= ~(1<<1); // Set pin 1 in PORTC with value 0 at the beginning(LED OFF)

    while(1)
    {
        /********** Application Code **********/
        if(!(PIND & (1<<2)))       // check if the push button at PD2 is pressed or not
        {
            PORTC |= (1<<1);       // Set pin 1 in PORTC with value 1 (LED ON)
        }
        else
        {
            PORTC &= ~(1<<1);      // Set pin 1 in PORTC with value 0 (LED OFF)
        }
    }
}

```

Code Techniques

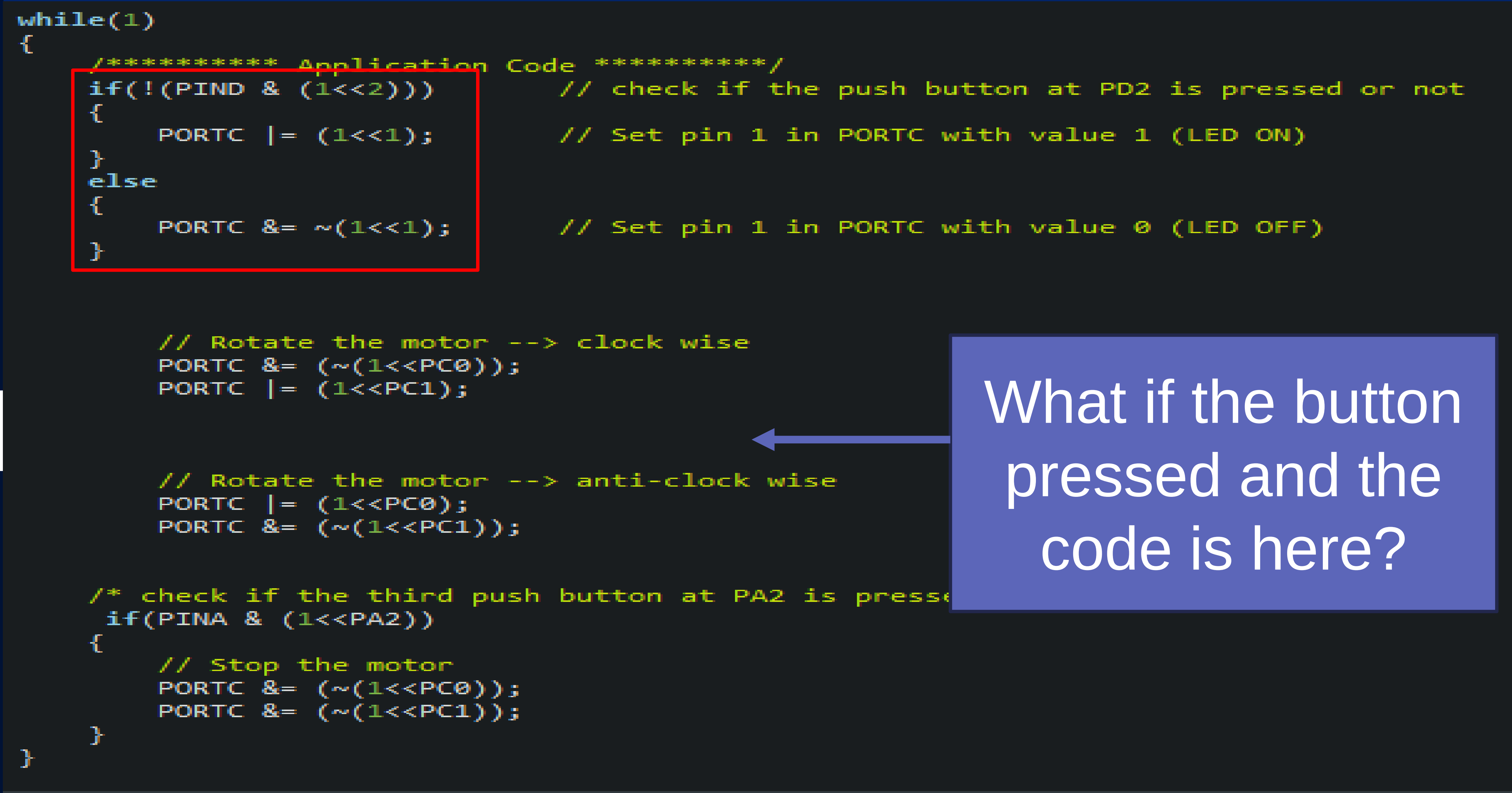
D-BUGERZ

1.

2.

3.

4.



```
while(1)
{
	/********** Application Code **********/
	if(!(PIND & (1<<2)))			// check if the push button at PD2 is pressed or not
	{
		PORTC |= (1<<1);		// Set pin 1 in PORTC with value 1 (LED ON)
	}
	else
	{
		PORTC &= ~(1<<1);		// Set pin 1 in PORTC with value 0 (LED OFF)
	}


		// Rotate the motor --> clock wise
		PORTC &= (~(1<<PC0));
		PORTC |= (1<<PC1);


		// Rotate the motor --> anti-clock wise
		PORTC |= (1<<PC0);
		PORTC &= (~(1<<PC1));

	/* check if the third push button at PA2 is presse
	 if(PINA & (1<<PA2))
	{
		// Stop the motor
		PORTC &= (~(1<<PC0));
		PORTC &= (~(1<<PC1));
	}
}
```

What if the button pressed and the code is here?

1.

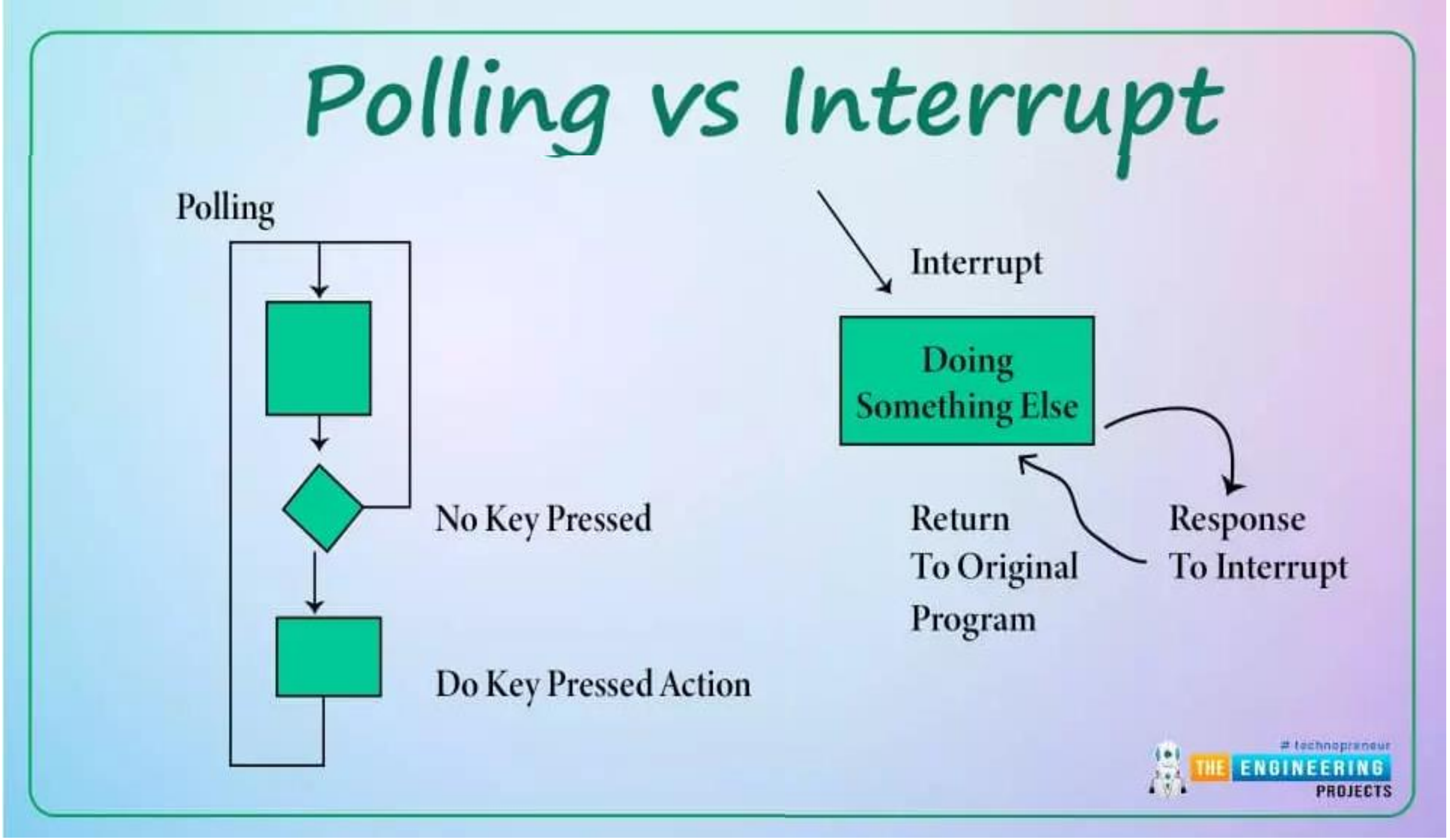
2.

3.

4.

Code Techniques

D-BUGERZ



Interrupt Overview

- ❑ Interrupts Overview Interrupts are basically events that require **immediate attention by the microcontroller.**
- ❑ An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention
- ❑ When an interrupt event occurs the microcontroller pause its current task and handle the interrupt by executing the **Interrupt Service Routine (ISR)** and at the end of the ISR the microcontroller returns to its interrupted task and continue its normal operations.
- ❑ Interrupts are what make embedded systems truly responsive and ready for Real Time Applications.
- ❑ Interrupts are a common feature supported by almost all microcontrollers.

Interrupt Overview

1.
2.
3.
4.

Such an event may be :

1. An peripheral device has finished the last task it was given and is now ready for another. For example ADC may be generate interrupt after finishing the conversion from analog to digital.

2. External Interrupt event on I/O Pin.

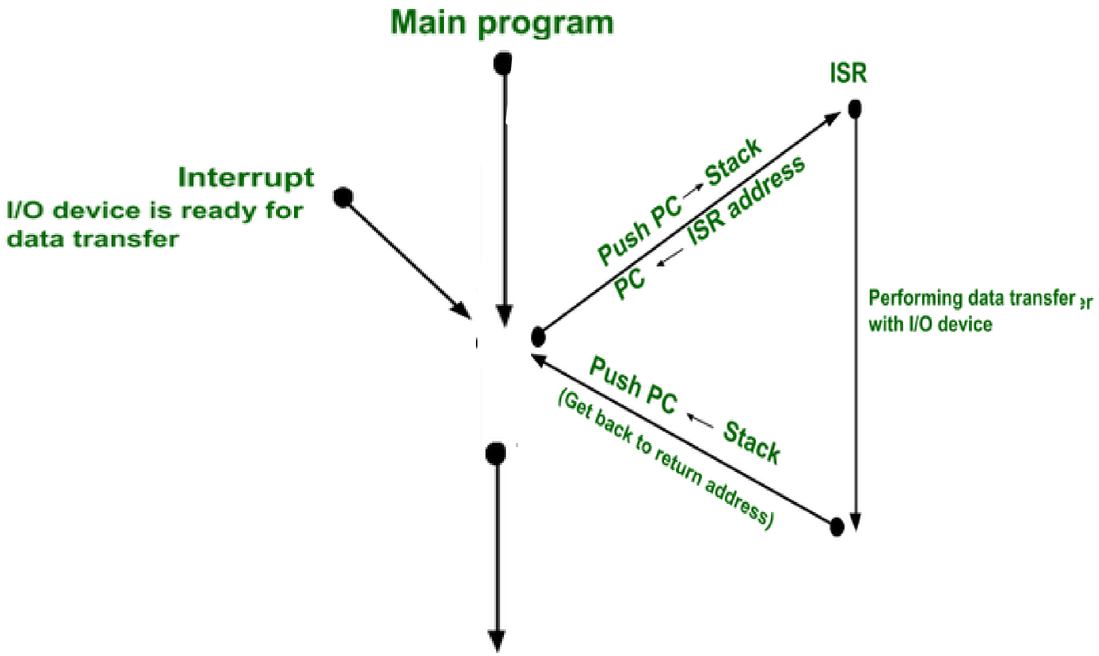
3. An error from a peripheral. You can think of it as a hardware-generated function call

interrupt → make code more responsive.

Interrupt Overview

- ❑ Interrupts are typically generated in most cases by hardware, for example peripherals or external input pins. When a peripheral or hardware needs service from the processor, this will lead to changes in program flow control outside the normal programmed sequence. Typically, the following sequences would occur

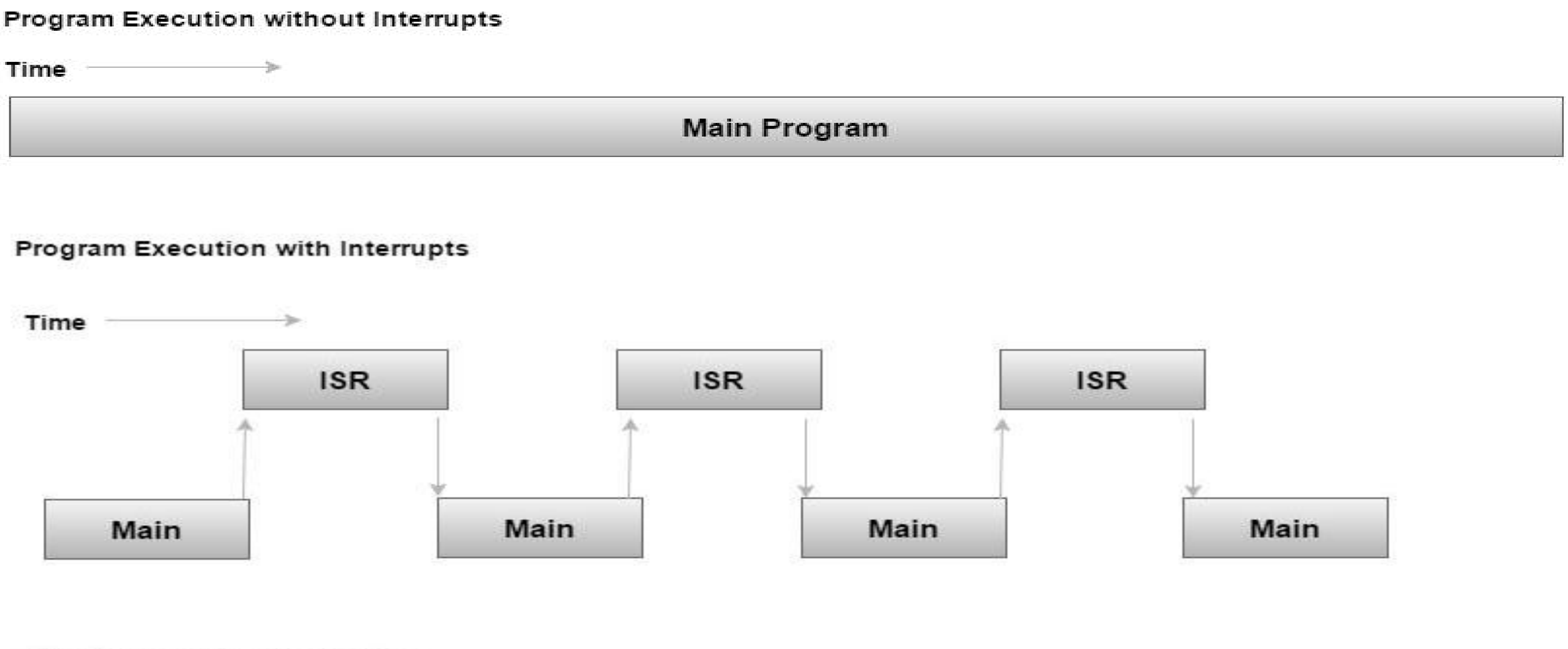
1. The peripheral asserts an interrupt request to the processor.
2. The currently running task is suspended by the processor
3. The processor executes an Interrupt Service Routine (ISR) to service the peripheral
4. The processor resumes the previously suspended task.


Main program
ISR
Interrupt
I/O device is ready for data transfer
Push PC → Stack
ISR address
PC ←
Performing data transfer with I/O device
Push PC ← Stack
(Get back to return address)

Interrupt Overview

1.
2.
3.
4.

1. The peripheral asserts an interrupt request to the processor.
2. The currently running task is suspended by the processor
3. The processor executes an Interrupt Service Routine (ISR) to service the peripheral
4. The processor resumes the previously suspended task.


Program Execution without Interrupts
Time
Main Program
Program Execution with Interrupts
Time
ISR
ISR
ISR
Main
Main
Main
Main

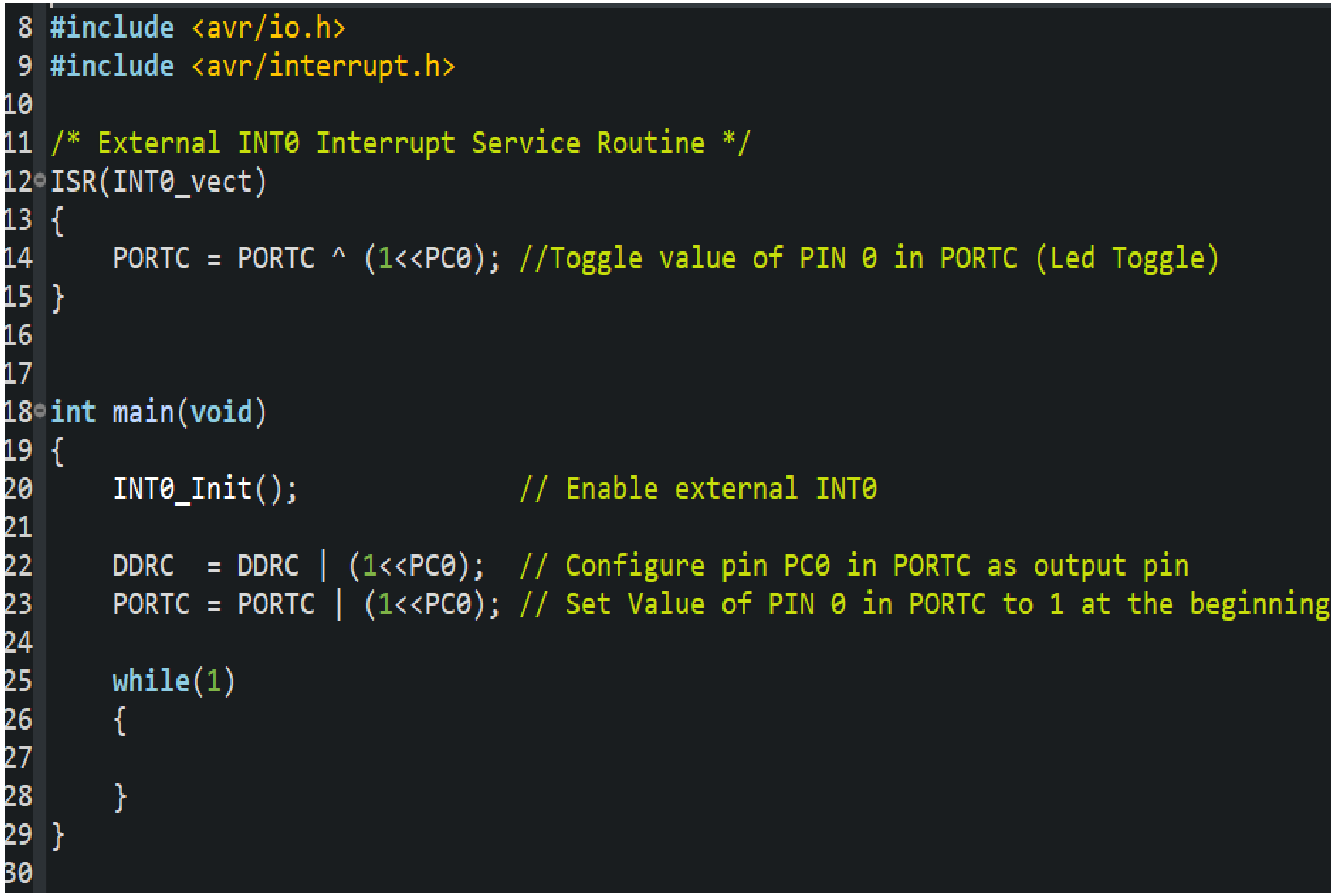
ISR : Interrupt Service Routine

Interrupt Overview

1.
2.
3.
4.

Interrupt Service Routine (ISR) or Interrupt Handler

- ❑ Piece of code that should be execute when an interrupt is triggered.
- ❑ For every interrupt there must be a program associated with it. When an interrupt occurs this program is executed to perform certain service for the interrupt.



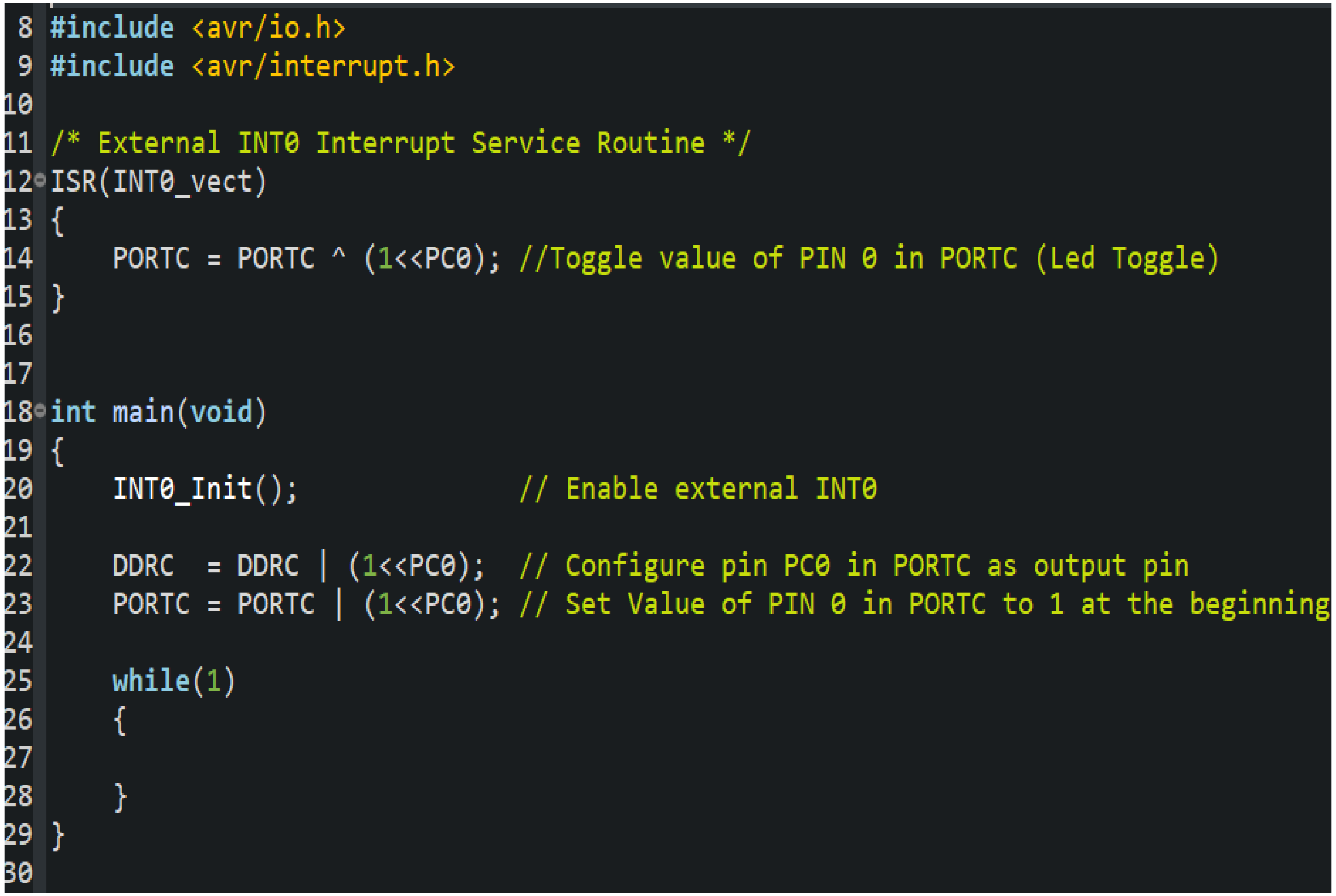
```
#include <avr/io.h>
#include <avr/interrupt.h>

/* External INT0 Interrupt Service Routine */
ISR(INT0_vect)
{
    PORTC = PORTC ^ (1<<PC0); //Toggle value of PIN 0 in PORTC (Led Toggle)
}


int main(void)
{
    INT0_Init();              // Enable external INT0

    DDRC  = DDRC | (1<<PC0);  // Configure pin PC0 in PORTC as output pin
    PORTC = PORTC | (1<<PC0); // Set Value of PIN 0 in PORTC to 1 at the beginning

    while(1)
    {

    }
}

```

Interrupt Overview

1.
2.
3.
4.

Interrupt Service Routine (ISR) or Interrupt Handler

- ❑ This program is commonly referred to as an Interrupt Service Routine (ISR) or Interrupt Handler When an interrupt occurs, the CPU runs the ISR. (No need to call this ISR)
- ❑ Usually each interrupt has its own ISR.
- ❑ Its address in ROM is usually saved in Interrupt Vector Table



```
#include <avr/io.h>
#include <avr/interrupt.h>

/* External INT0 Interrupt Service Routine */
ISR(INT0_vect)
{
    PORTC = PORTC ^ (1<<PC0); //Toggle value of PIN 0 in PORTC (Led Toggle)
}


int main(void)
{
    INT0_Init();               // Enable external INT0

    DDRC  = DDRC | (1<<PC0);  // Configure pin PC0 in PORTC as output pin
    PORTC = PORTC | (1<<PC0); // Set Value of PIN 0 in PORTC to 1 at the beginning

    while(1)
    {

    }
}
```

Interrupt Overview

D-BUGERZ

1.

2.

3.

4.

- ❑ ISR **should be deterministic, short and execute as fast as possible**. The interrupt should occur when it is time to perform a needed function, and the interrupt service routine should perform that function, and return right away. Placing backward (busy-wait loops, iterations) in the interrupt software should be avoided if possible it usually set a flag to a specific task indicating a certain event is happened, or save small data in buffer.

- ❑ ISR does not have a prototype.

- ❑ ISR **does not have any arguments or return value** as it is called automatically by hardware when its interrupt occur. (Void)

- ❑ ISR can use shared variables to communicate with the application

Interrupt Overview

Interrupt Vector Table(IVT) (Table in program memory)

- ❑ It is a constant table in ROM(Program Memory). In most cases it is exist at the beginning of the memory.
- ❑ Special addresses with respect to CPU.
- ❑ Each interrupt has specific location in the interrupt vector table for it's ISR
- ❑ This specific location should be programmed to have the address of the ISR for this interrupt.
- ❑ ATmega16/32 Microcontrollers Interrupt Vector Table has 21 different locations.

Interrupt Overview

1.

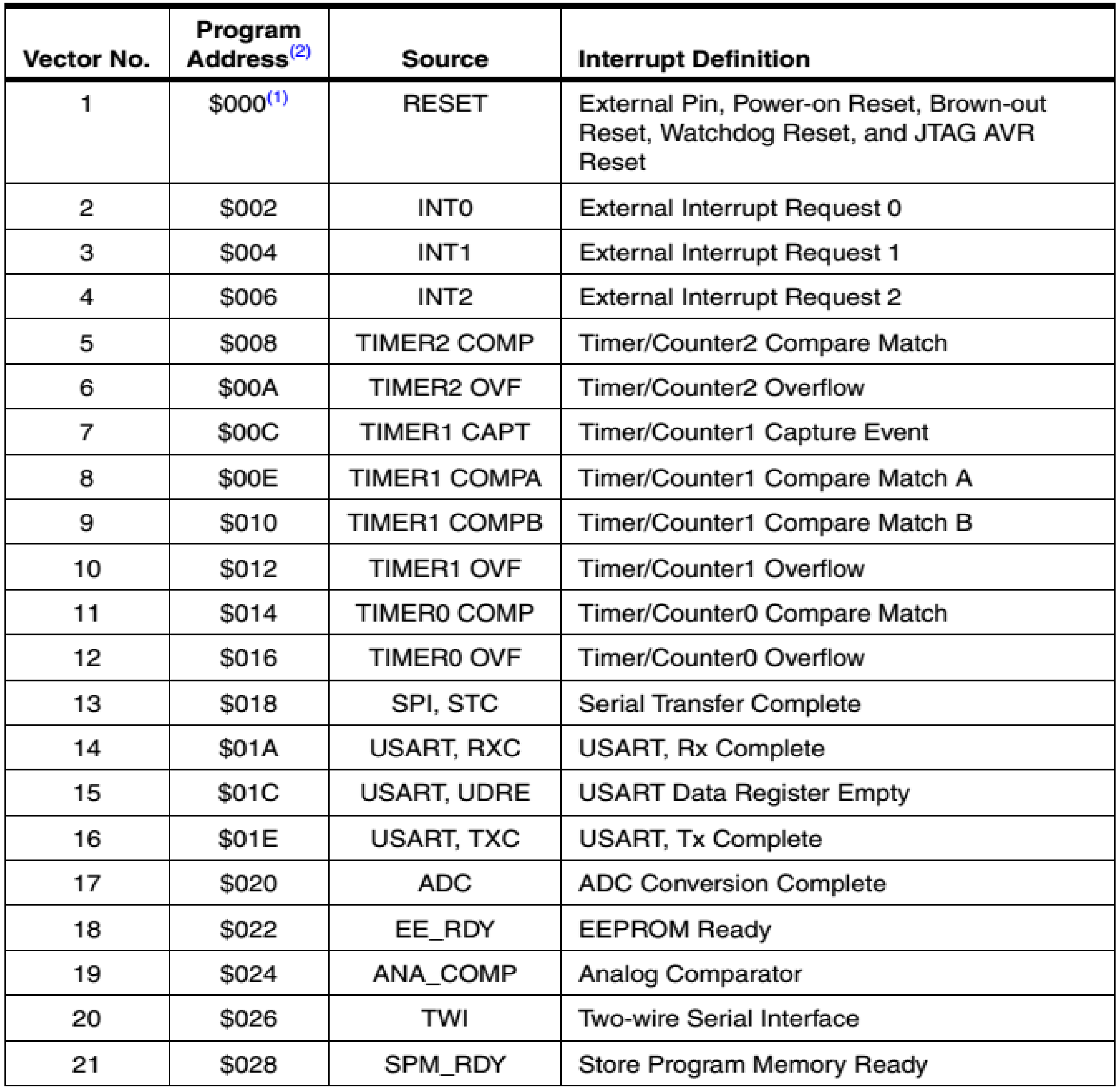
2.

3.

4.

- ❖ **Interrupt Vector Table for Atmega32**

→ Each interrupt has specific location in the interrupt vector table for it's ISR



Vector No.	Program Address[2]	Source	Interrupt Definition
1	$000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	INT2	External Interrupt Request 2
5	$008	TIMER2 COMP	Timer/Counter2 Compare Match
6	$00A	TIMER2 OVF	Timer/Counter2 Overflow
7	$00C	TIMER1 CAPT	Timer/Counter1 Capture Event
8	$00E	TIMER1 COMPA	Timer/Counter1 Compare Match A
9	$010	TIMER1 COMPB	Timer/Counter1 Compare Match B
10	$012	TIMER1 OVF	Timer/Counter1 Overflow
11	$014	TIMER0 COMP	Timer/Counter0 Compare Match
12	$016	TIMER0 OVF	Timer/Counter0 Overflow
13	$018	SPI, STC	Serial Transfer Complete
14	$01A	USART, RXC	USART, Rx Complete
15	$01C	USART, UDRE	USART Data Register Empty
16	$01E	USART, TXC	USART, Tx Complete
17	$020	ADC	ADC Conversion Complete
18	$022	EE_RDY	EEPROM Ready
19	$024	ANA_COMP	Analog Comparator
20	$026	TWI	Two-wire Serial Interface
21	$028	SPM_RDY	Store Program Memory Ready

1.

2.

3.

4.

Interrupt Overview

D-BUGERZ

1.
2.
3.
4.

❑ What will happen when an interrupt occurs?

1. The microcontroller CPU completes the execution of the current instruction and batch stores the address of the next instruction that should have been executed (the yshef contents of the PC), status register and the CPU registers are pushed onto the intrapt stack. This is called **a context switch.**
2. Microcontroller CPU jump to Interrupt Vector Table to get the ISR address of the triggered interrupt.
3. The interrupt vector of the triggered interrupt (ISR start address of this interrupt) is then loaded in the PC(program counter) from the Interrupt Vector Table and the CPU starts execution from that point up until reaches the end of the ISR.
4. The address that was stored on the stack in step 1 is reloaded in the PC, status register and all the CPU registers are popped from the stack(restore context).
5. The CPU then continue executing the main program

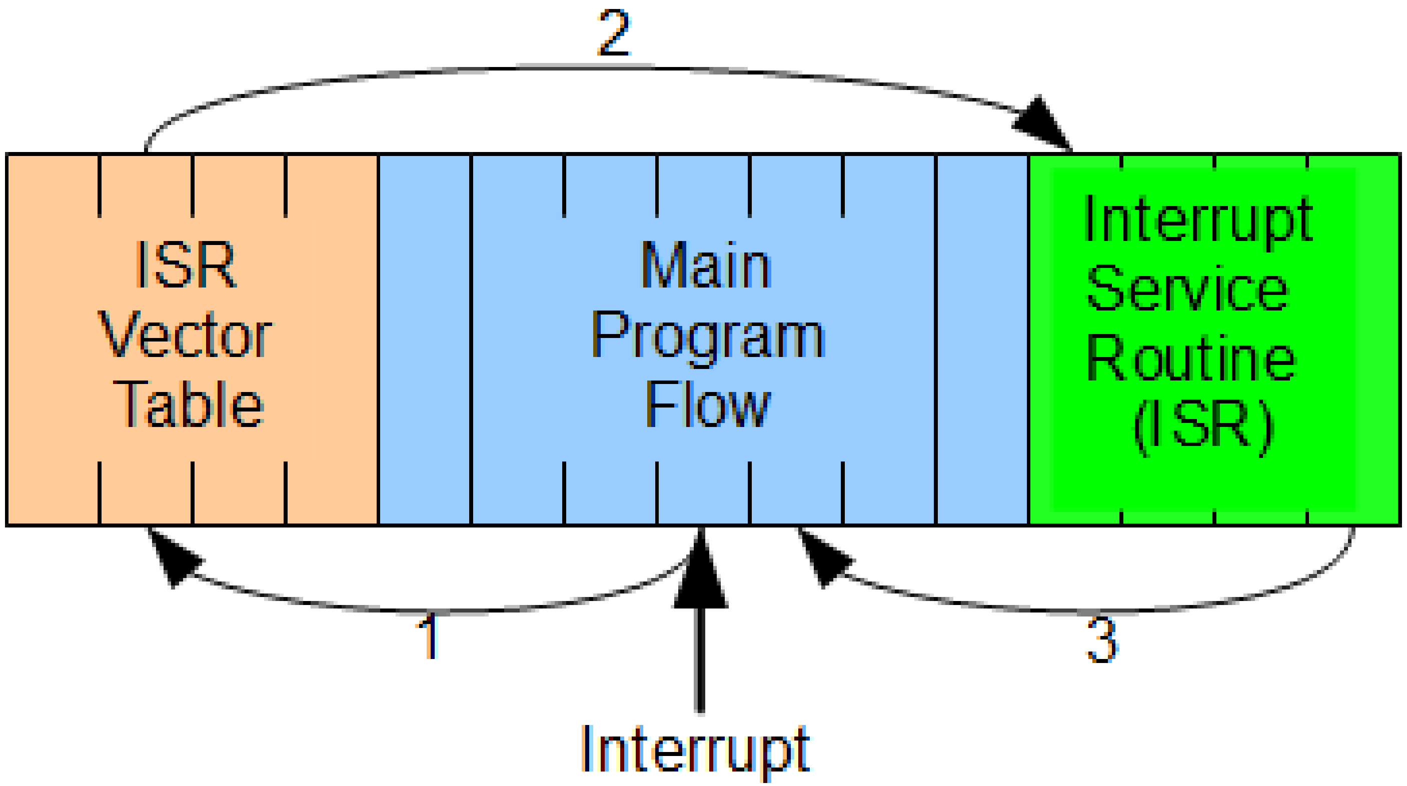
Interrupt Overview

1.

2.

3.

4.



Interrupt Sources

D-BUGERZ

1.

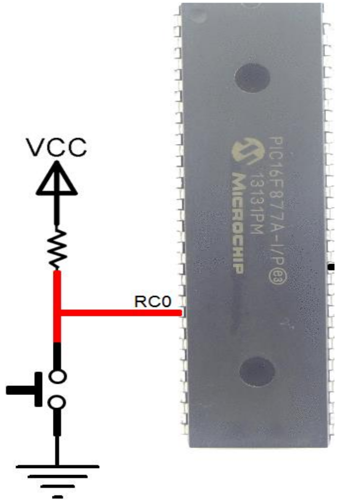
2.

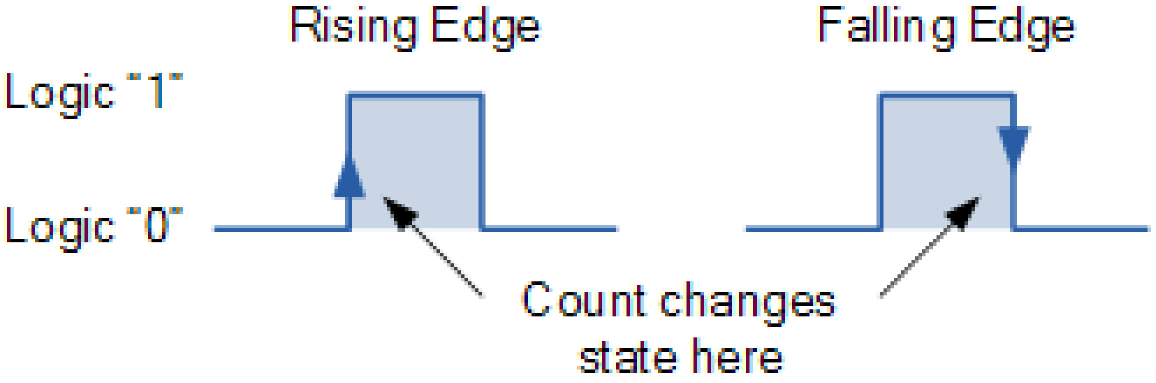
3.

4.

Hardware Interrupts

- ❑ A hardware interrupt is a signal which can tell the CPU that something happen in hardware device, and should be immediately responded. Hardware interrupts are triggered by peripheral devices outside the microprocessor. For example:
- ➢ Signal on external PIN like reset interrupt or external interrupt.
- ➢ Timer Overflow Interrupt.
- ➢ ADC Conversion Complete Interrupt.
- ➢ Serial Port Send/Receive Complete Interrupt.





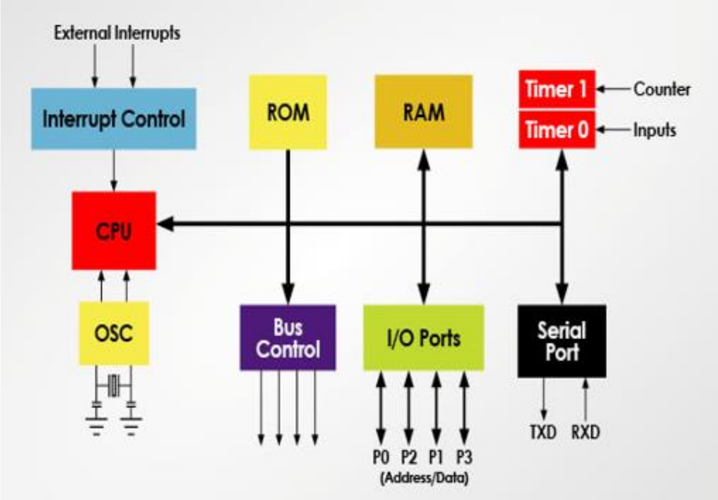
Interrupt Types

1. Maskable Interrupts

- ❑ It **could be disabled or enabled** depends on the Global Interrupt Enable bit inside the processor.
- ❑ In AVR Processor this bit is called I-bit and it is located in the Processor Status Register(SREG).

For example:

- ➢ External Interrupt event on I/O Pin.
- ➢ IRQs from peripherals in the microcontroller.



2. Non-Maskable Interrupts

- ❑ Interrupts **can not be disabled** and are always enabled.
- ❑ NMI is always assigned with highest interrupt priority level. That's why it is commonly used in real time applications.
- ❑ AVR processor only support one NMI which is **reset interrupt**.

D-BUGERZ

1.

2.

3.

4.

How To Trigger Interrupt Request

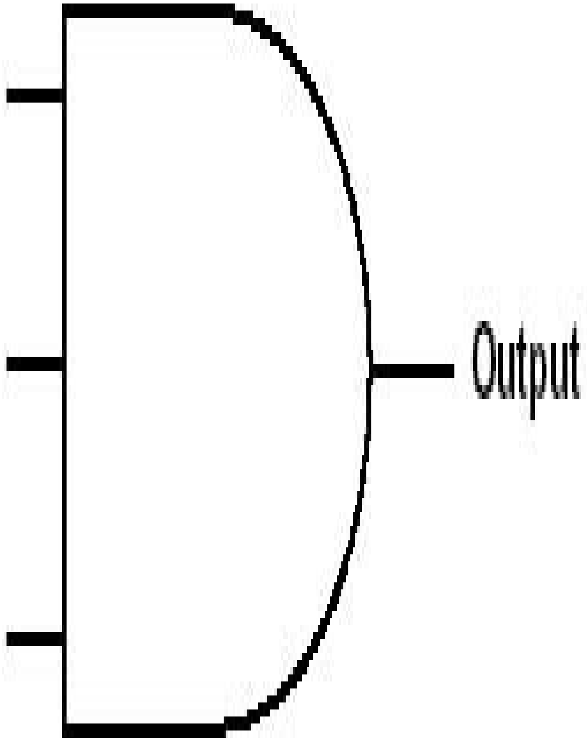
1.
2.
3.
4.

❑ Each peripheral device has a group of special function registers that must be used to access the device for configuration. For a given peripheral interrupt to take effect, the interrupt for that peripheral must be enabled. The special function registers for that device provide the way to enable the interrupts.

❑ In case disable all the interrupts is required just **clear** the **Global Interrupt Enable (GIE).** All Maskable interrupts are disabled whatever the MIE value was 1 or 0.

❑ Each Maskable interrupt in the MC has a specific bit called **Module Interrupt Enable (MIE).** It is used to enable or disable the interrupt for this module or peripheral.

❑ Also each Maskable interrupt in the microcontroller has a specific bit **called Module Interrupt flag (MIF).** This bit is set whenever the interrupt event occur. Whether or not the interrupt is enabled. This flag is also called Automatic flag as it is set automatically by hardware.



How To Trigger Interrupt Request

1.
2.
3.
4.

- ❑ Each peripheral device has a group of special function registers that must be used to access the device for configuration. For a given peripheral interrupt to take effect, the interrupt for that peripheral must be enabled. The special function registers for that device provide the way to enable the interrupts.

- ❑ Each Maskable interrupt in the MC has a specific bit called **Module Interrupt Enable (MIE).** It is used to enable or disable the interrupt for this module or peripheral.

- ❑ Also each Maskable interrupt in the microcontroller has a specific bit **called Module Interrupt flag (MIF).** This bit is set whenever the interrupt event occur. Whether or not the interrupt is enabled. This flag is also called Automatic flag as it is set automatically by hardware.

How To Trigger Interrupt Request

- ❑ In case disable all the interrupts is required just **clear** the **Global Interrupt Enable (GIE).** All Maskable interrupts are disabled whatever the MIE value was 1 or 0.

- ❑ The Non-Maskable interrupts have no MIE and the GIE doesn't affect them. These interrupts when happen, they must be served!

- ❑ **Important Note:** The Module Interrupt Flag (MIF) must be cleared after executing the ISR code and before the next interrupt. To make sure that it will be set again when the next interrupt event occurs. Some architectures cleared it by default once the ISR is executed like AVR and others you must clear it by yourself in the code.

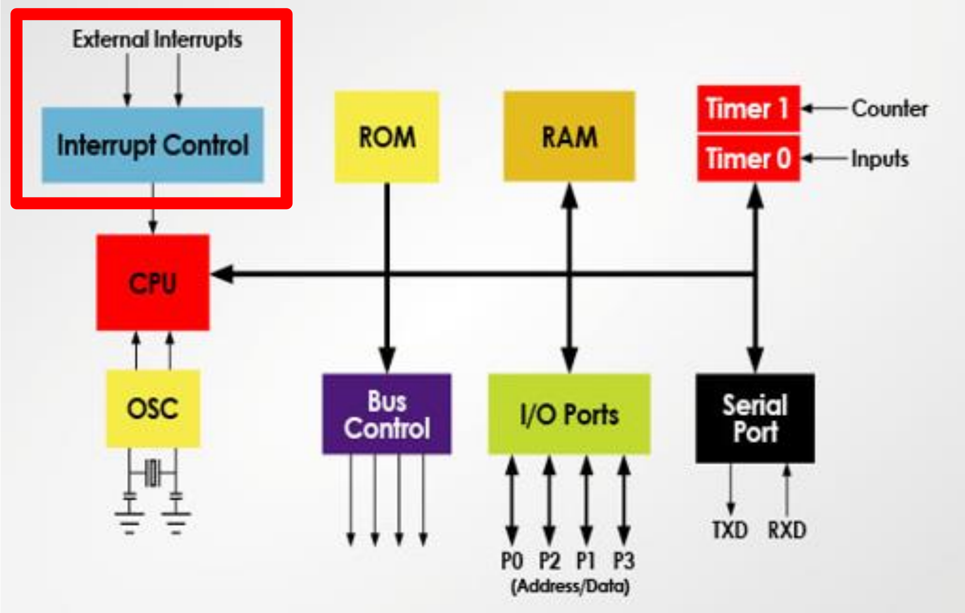
Interrupt Controller Unit

D-BUGERZ

1.
2.
3.
4.

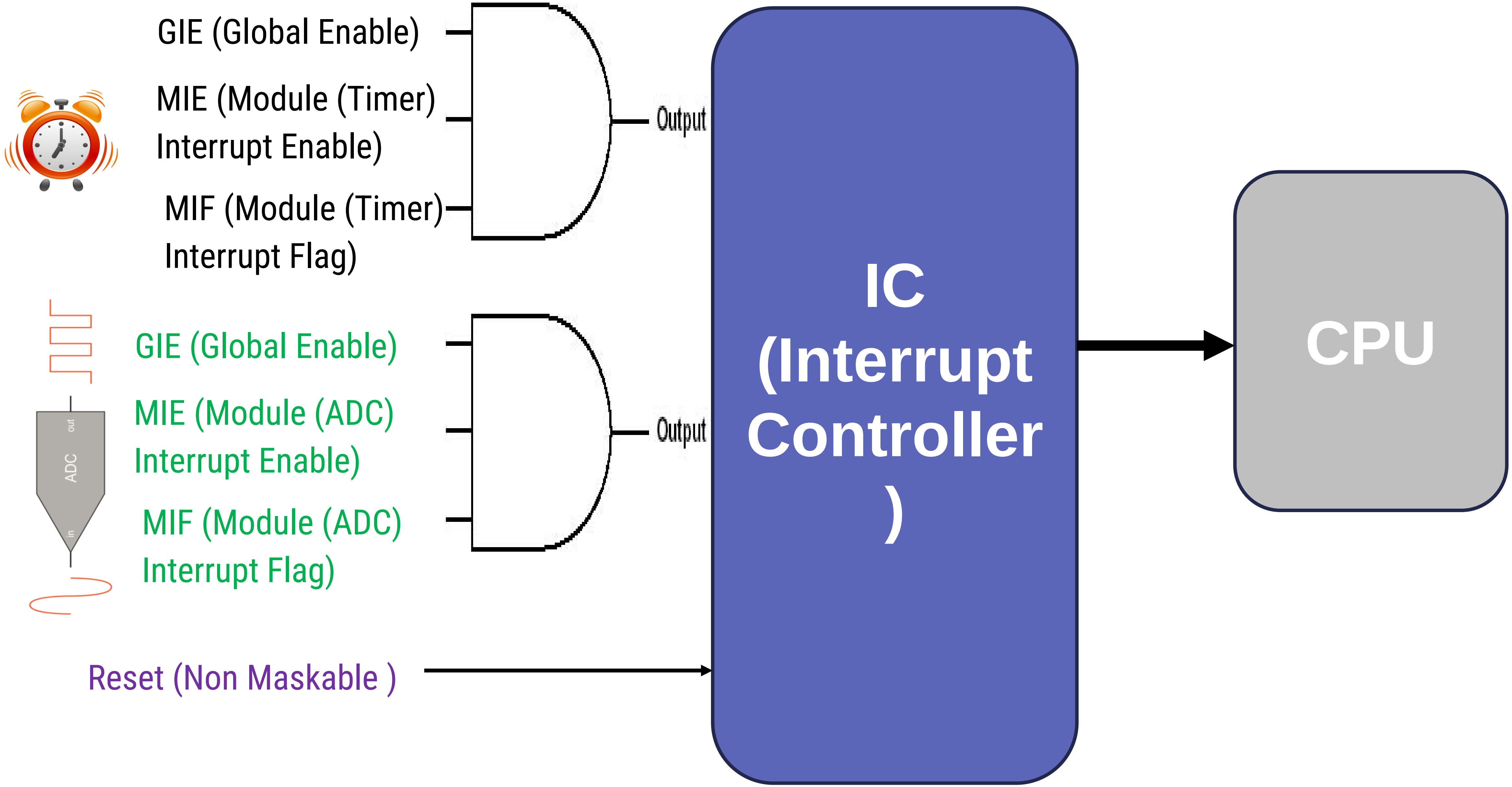
- ❑ The Interrupt Controller (IC) is a hardware unit that handles the interrupt serving and priorities. All interrupts are connected to the IC, when any interrupt happens, the IC receives the interrupt request and generate a signal to the processor on its INT pin. Then the IC tells the processor the ID of the interrupt happened through a special data bus.

- ❑ When the processor receives an interrupt request from the IC and gets its ID, it jumps to the corresponding location in the IVT to find the address of its ISR.



Interrupt Controller Unit

1.
2.
3.
4.



GIE (Global Enable)

MIE (Module (Timer) Interrupt Enable)

MIF (Module (Timer) Interrupt Flag)

GIE (Global Enable)

MIE (Module (ADC) Interrupt Enable)

MIF (Module (ADC) Interrupt Flag)

Reset (Non Maskable)

IC (Interrupt Controller)

CPU

Interrupt Controller Unit

❑ The IC handles the Interrupt Priorities, if two interrupts happened at the same time, the higher priority interrupt will run first. The IC tells the processor about the higher priority interrupt first, then tells the processor about the lower one.

❑ IC in AVR Processor assigns a default priority level for each interrupt by its location in Interrupt Vector Table (Static Priorities according to IVT).

❑ Normally maskable interrupts have priority levels. but Non-Maskable Interrupts (NMIS) have fixed priority levels. NMI is always assigned with the highest priority level.

❑ Priority determines the order of service when two or more requests are made simultaneously.

Interrupt Nesting

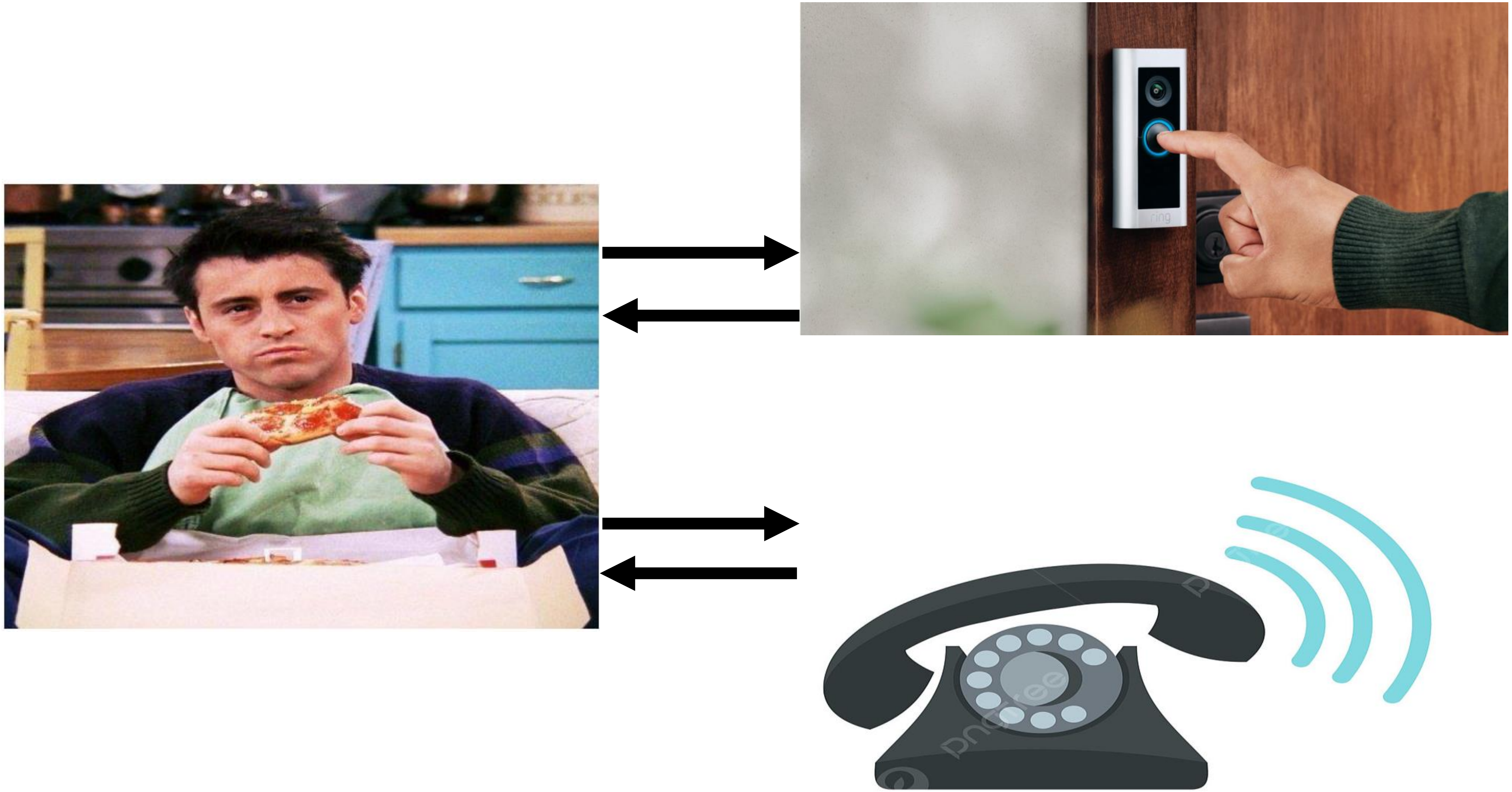
D-BUGERZ

1.

2.

3.

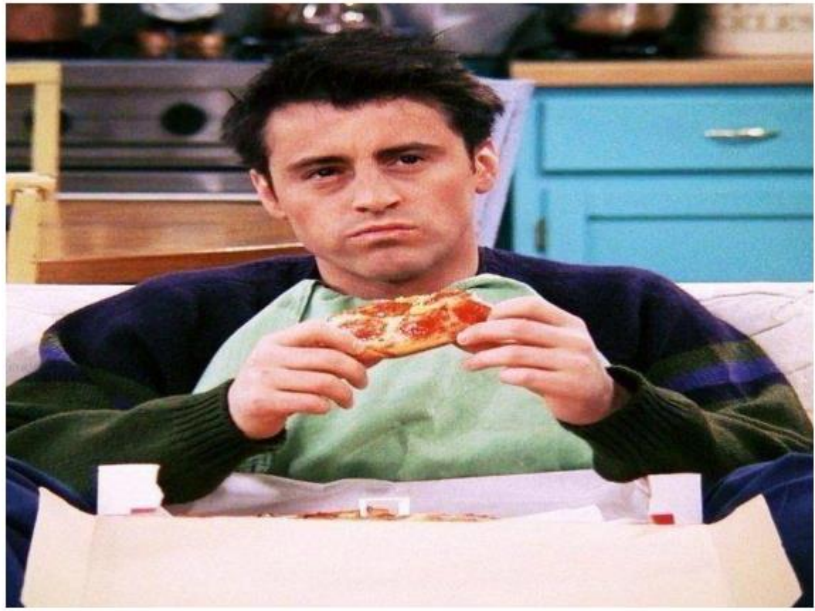
4.

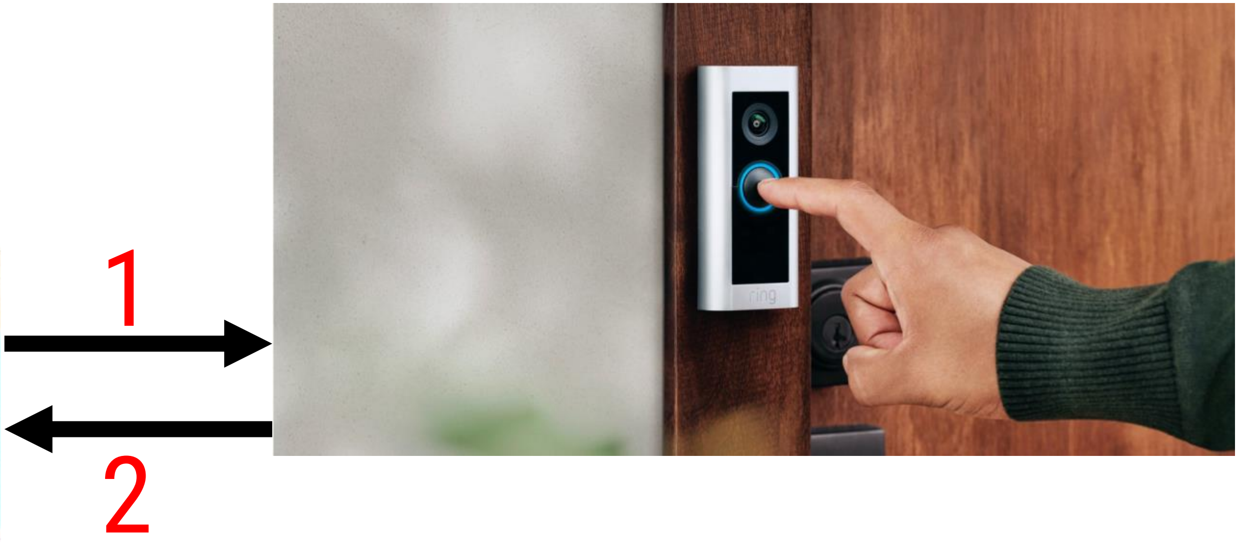


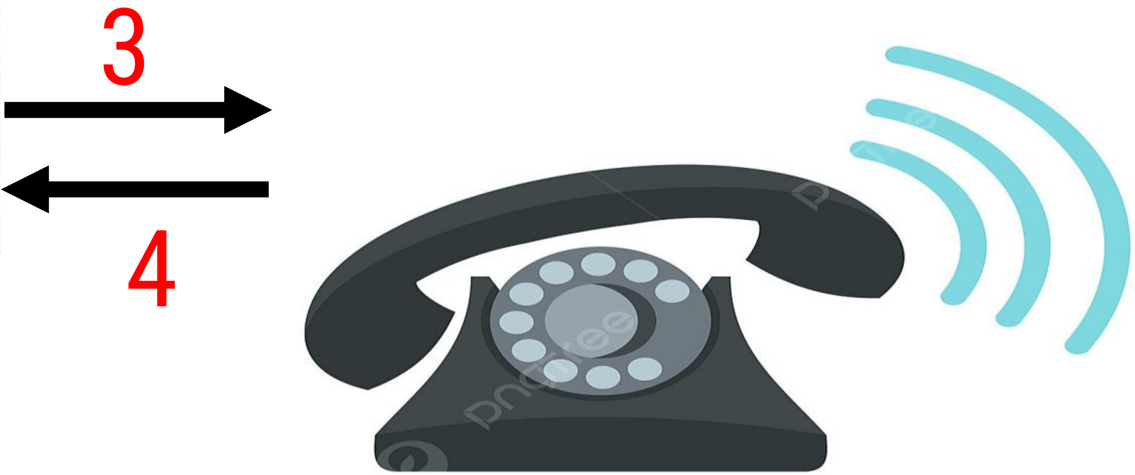
1.
2.
3.
4.

Interrupt Nesting

- ❑ What if both done at same time???







- ❖ So you must put priority to your interrupts

Interrupt Nesting

D-BUGERZ

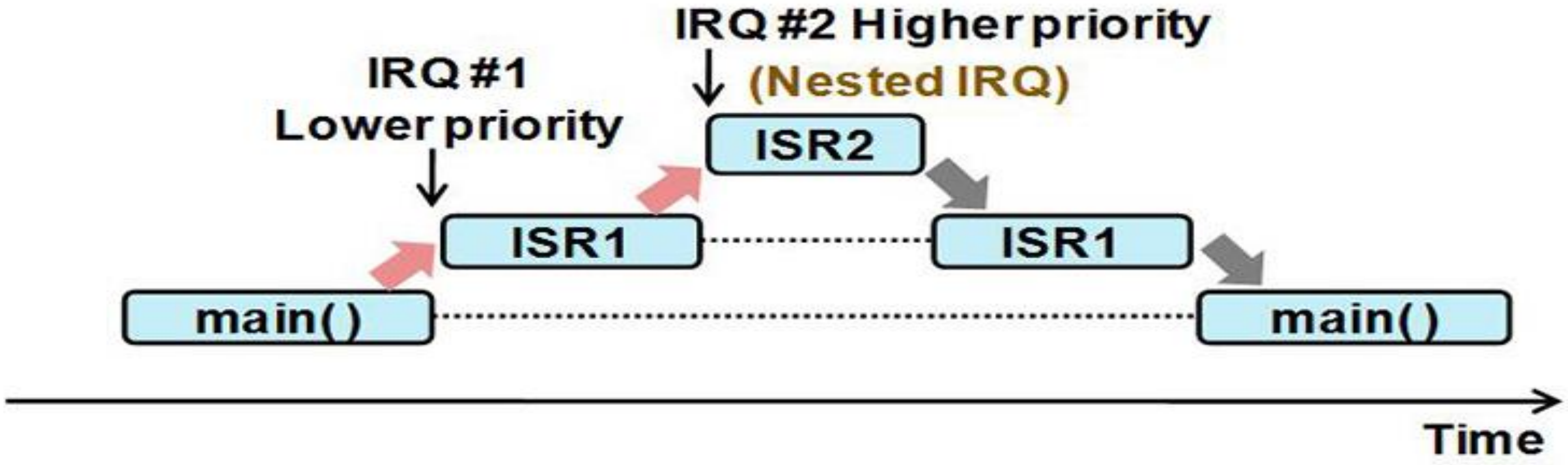
1.

2.

3.

4.

- ❑ Interrupts Nesting is the ability to suspend the execution of the current interrupt and serve a new higher priority interrupt, Then get back to the old interrupt after serving the new one.
- ❑ It is usually happened if Global Interrupt is enabled and the new interrupt has higher priority.
- ❑ Priority allows a higher priority request to suspend a lower priority request currently being processed.



Interrupt Nesting

1.
2.
3.
4.

❑ When an interrupt occurs, the IC performs a comparison between the priority level of current interrupt and the priority level of the new coming interrupt. The current running interrupt will be suspended and the control will be transferred to the service routine of the new coming interrupt if the priority level of the new coming interrupt is higher.

❑ If an interrupt request of equal or lower priority is generated while executing ISR of a higher priority interrupt, that request is postponed until the current ISR is completed.

External Interrupt

1.
2.
3.
4.

Interrupt sources provided with the AVR Microcontrollers

- ❑ The AVR microcontroller provide both internal and external interrupt sources.
- ❑ The internal interrupts are associated with the microcontroller's peripherals. That is the Timer/Counter, Analog, Analog Digital Converter, USART, etc.
- ❑ The external interrupts are triggered via external pins. for ATmega16/32 microcontrollers there are four (4) external interrupts:
    1. RESET interrupt: Triggered from pin 9.
    2. External Interrupt 0 (INTO): Triggered from pin 16(PD2).
    3. External Interrupt 1 (INT1): Triggered from pin 17(PD3).
    4. External Interrupt 2 (INT2): Triggered from pin 3(PB2).

1.

2.

3.

4.

BREAK 5 MIN

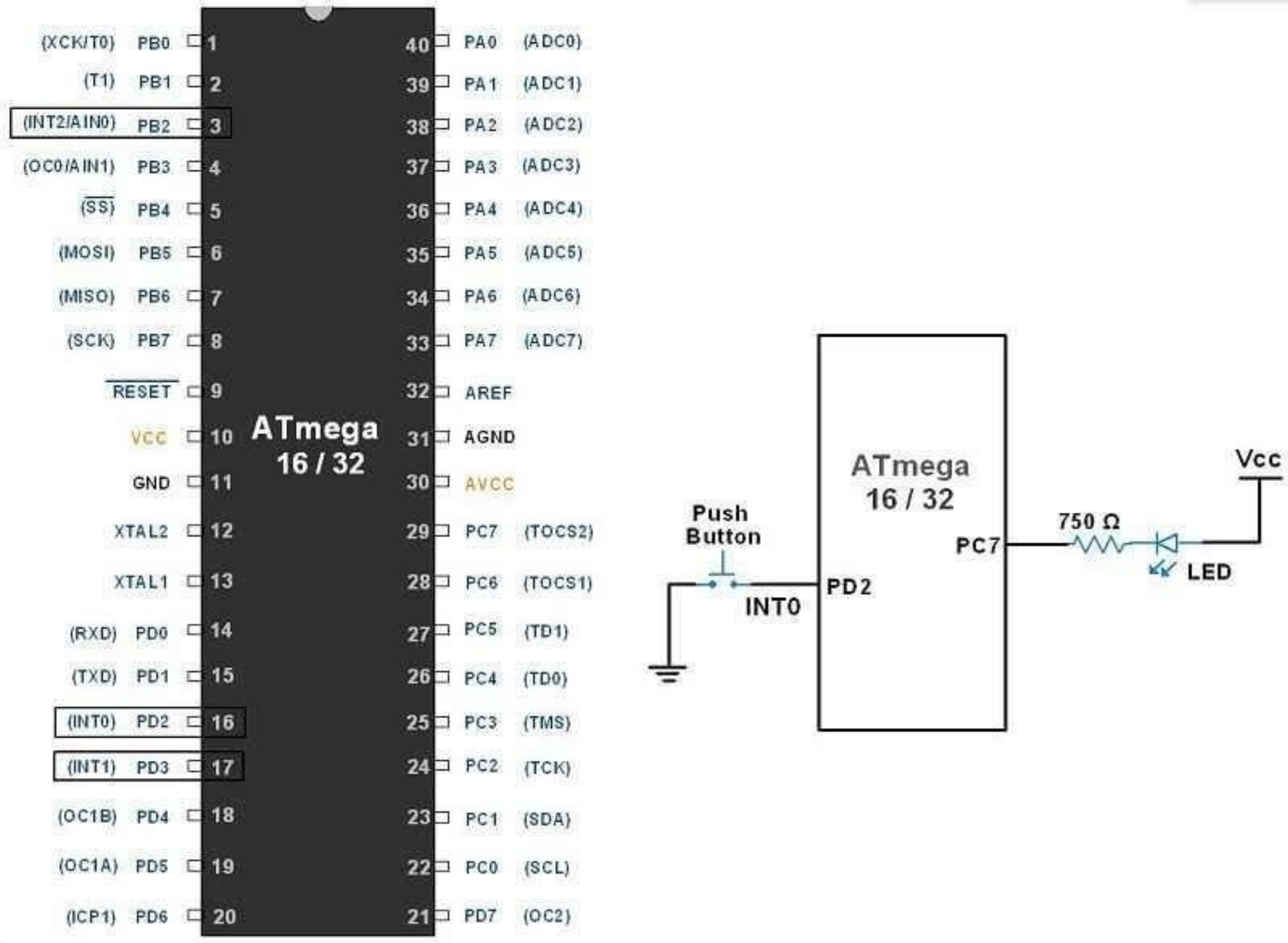
Recall Last session

Interfacing with 7-Segment

Introduction to Interrupt

Start Coding with External Interrupt

External Interrupt Pins in ATmega16/32



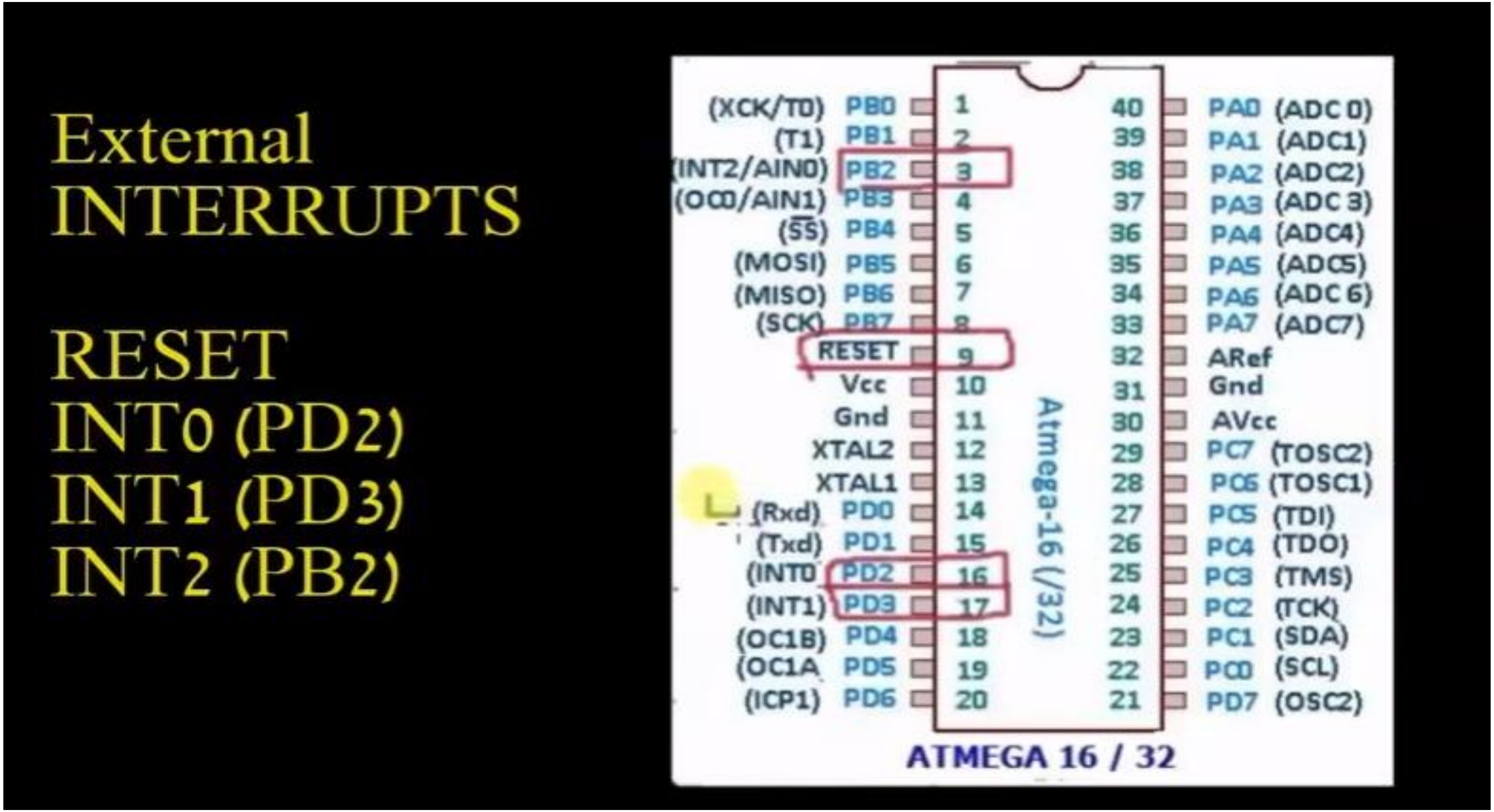
Registers of External Interrupt

1.

2.

3.

4.



External INTERRUPTS

RESET

INT0 (PD2)

INT1 (PD3)

INT2 (PB2)

How To Trigger Interrupt Request

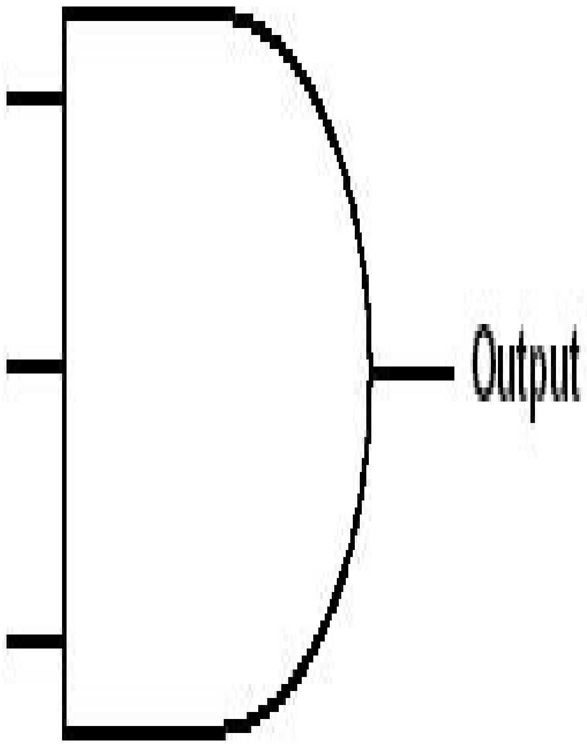
1.
2.
3.
4.

- ❑ Each peripheral device has a group of special function registers that must be used to access the device for configuration. For a given peripheral interrupt to take effect, the interrupt for that peripheral must be enabled. The special function registers for that device provide the way to enable the interrupts.

- ❑ In case disable all the interrupts is required just **clear** the **Global Interrupt Enable (GIE).** All Maskable interrupts are disabled whatever the MIE value was 1 or 0.

- ❑ Each Maskable interrupt in the MC has a specific bit called **Module Interrupt Enable (MIE).** It is used to enable or disable the interrupt for this module or peripheral.

- ❑ Also each Maskable interrupt in the microcontroller has a specific bit **called Module Interrupt flag (MIF).** This bit is set whenever the interrupt event occur. Whether or not the interrupt is enabled. This flag is also called Automatic flag as it is set automatically by hardware.



Registers of External Interrupt

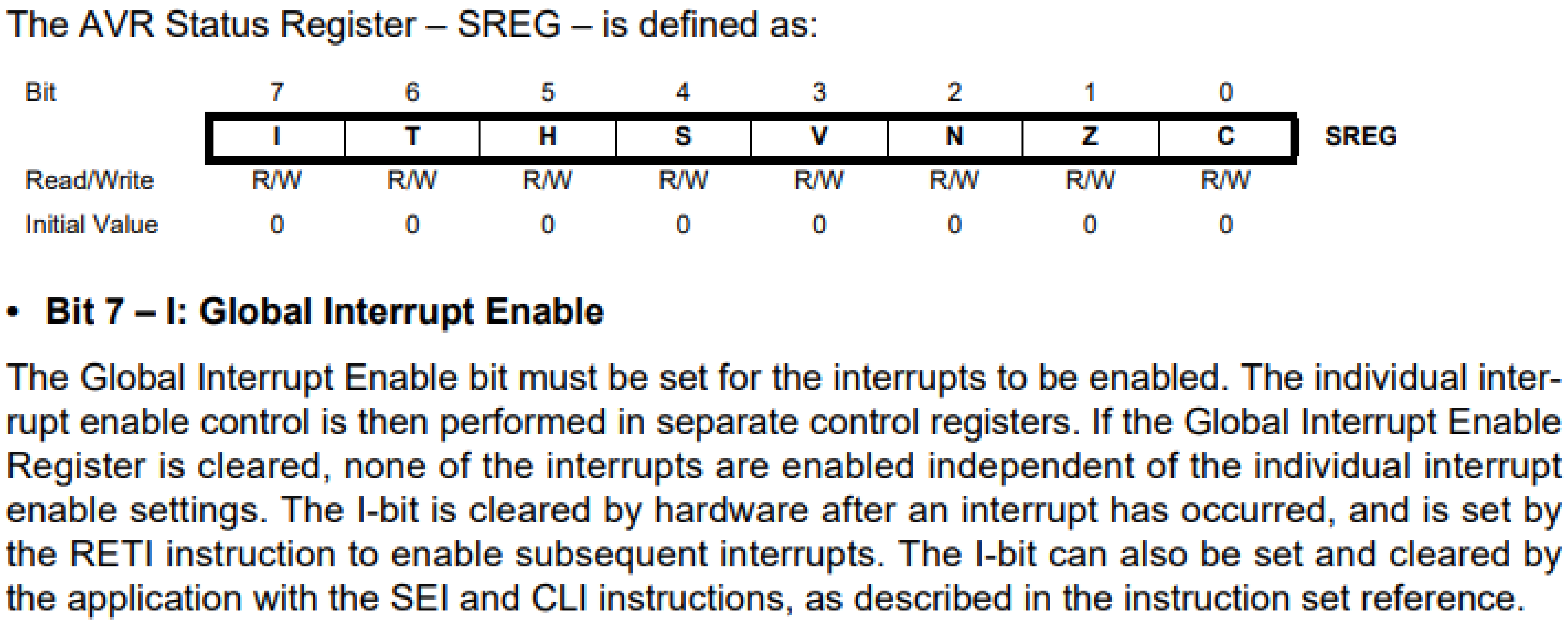
1.

2.

3.

4.

- ❑ Page 10 in Datasheet



The AVR Status Register – SREG – is defined as:

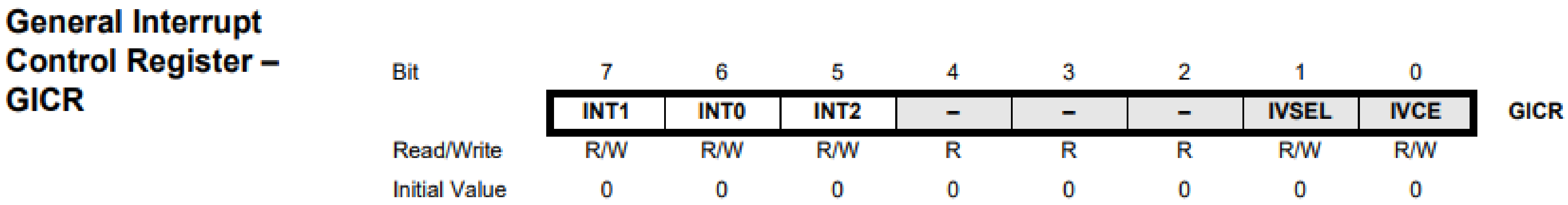
Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – I: Global Interrupt Enable**

The Global Interrupt Enable bit must be set for the interrupts to be enabled. The individual interrupt enable control is then performed in separate control registers. If the Global Interrupt Enable Register is cleared, none of the interrupts are enabled independent of the individual interrupt enable settings. The I-bit is cleared by hardware after an interrupt has occurred, and is set by the RETI instruction to enable subsequent interrupts. The I-bit can also be set and cleared by the application with the SEI and CLI instructions, as described in the instruction set reference.

Registers of External Interrupt

❑ Page 67 in Datasheet



General Interrupt Control Register – GICR

Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	INT2	–	–	–	IVSEL	IVCE	GICR
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit 7 - INT1: External Interrupt Request 1 Enable

- ➢ 0: Disable external interrupt 1
- ➢ 1: Enable external interrupt 1

Bit 6 - INTO: External Interrupt Request 0 Enable

- ➢ 0: Disable external interrupt 0
- ➢ 1: Enable external interrupt 0

Bit 5 - INT2: External Interrupt Request 2 Enable

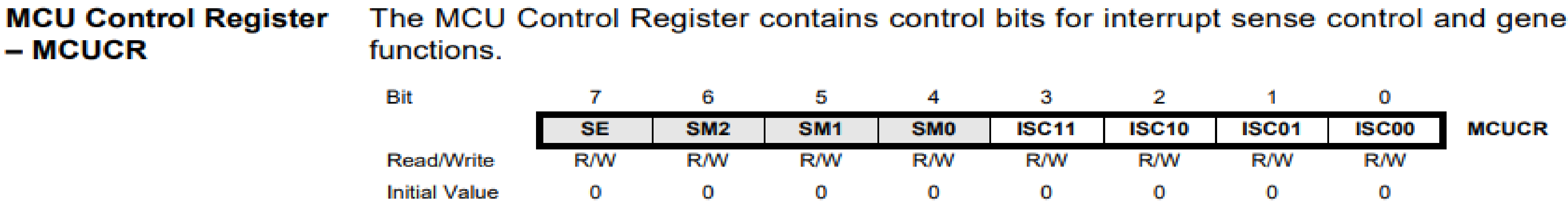
- ➢ 0: Disable external interrupt 2
- ➢ 1: Enable external interrupt 2

- **Apart from enabling a specific interrupt, Global Interrupts Enable bit (I-bit) MUST be enabled for the microcontroller to react to the interrupt event.**

Registers of External Interrupt

1.
2.
3.
4.

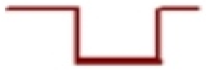
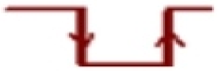
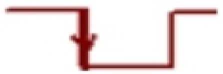
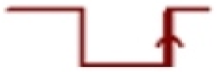
❑ Page 67 in Datasheet



MCU Control Register – MCUCR

The MCU Control Register contains control bits for interrupt sense control and general MCU functions.

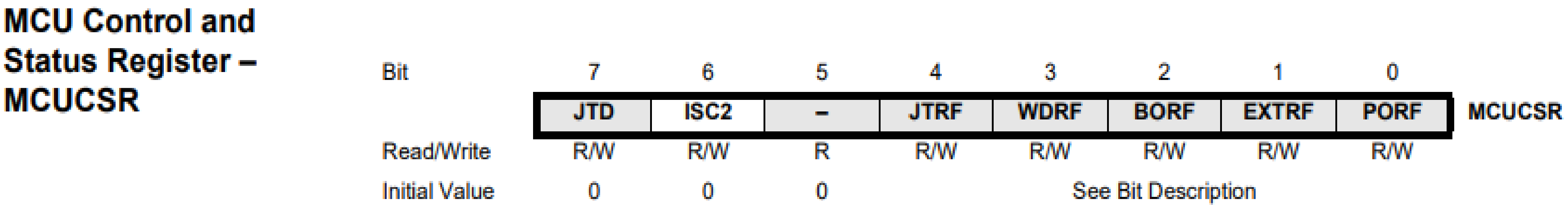
Bit	7	6	5	4	3	2	1	0	
	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

ISC01	ISC00		Description
0	0		The low level on INT0 pin generates an interrupt request.
0	1		Any logical change on INT0 pin generates an interrupt request.
1	0		The falling edge on INT0 pin generates an interrupt request.
1	1		The rising edge on INT0 pin generates an interrupt request.

Registers of External Interrupt

1.
2.
3.
4.

- ❑ Page 67 in Datasheet



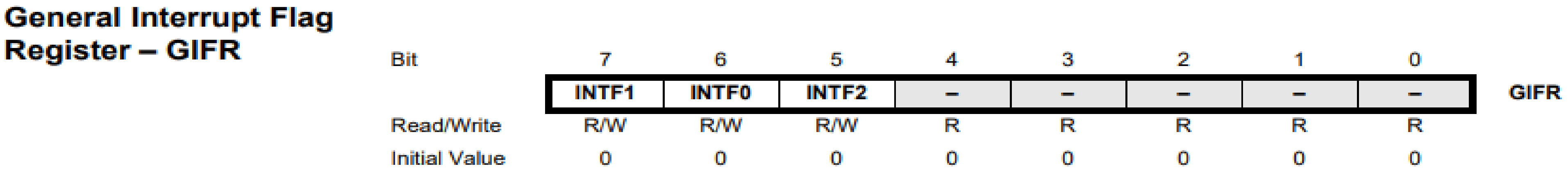
MCU Control and Status Register – MCUCSR

Bit	7	6	5	4	3	2	1	0	
	JTD	ISC2	–	JTRF	WDRF	BORF	EXTRF	PORF	MCUCSR
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	See Bit Description					

ISC2	Description
0	The falling edge of INT2 generate an interrupt request
1	The rising edge of INT2 generate an interrupt request

Registers of External Interrupt

- ❑ Page 68 in Datasheet



General Interrupt Flag Register – GIFR

Bit	7	6	5	4	3	2	1	0	
	INTF1	INTF0	INTF2	–	–	–	–	–	GIFR
Read/Write	R/W	R/W	R/W	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

- ❑ The GIFR register contains the interrupt flags for all the ATmega16/32 external interrupts. Each of these bit are set individually to logic 1 when the interrupt event for the specified interrupt occurs.

- ❑ Note that the flag bit of an interrupt is set, whether or not the interrupt is enabled, once the interrupt event occurs.

- ❑ If the interrupt is enabled, it's corresponding interrupt flag bit will be cleared after executing it's ISR code. Alternatively, the flag can be cleared by writing a logical one to it

How To Trigger Interrupt Request

D-BUGERZ

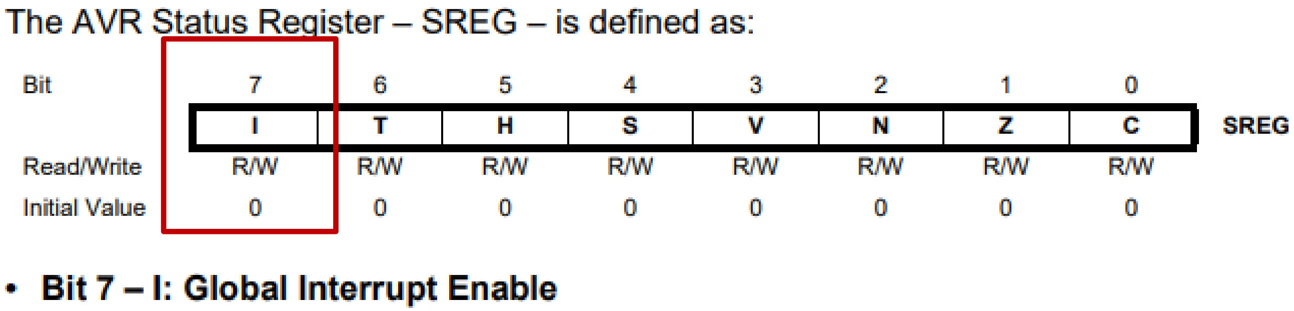
1.

2.

3.

4.

❑ Global Interrupt Enable (GIE).

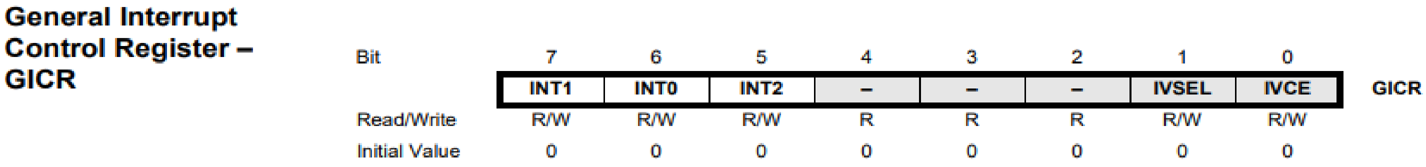


The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
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	

- **Bit 7 – I: Global Interrupt Enable**

❑ Module Interrupt Enable (MIE)

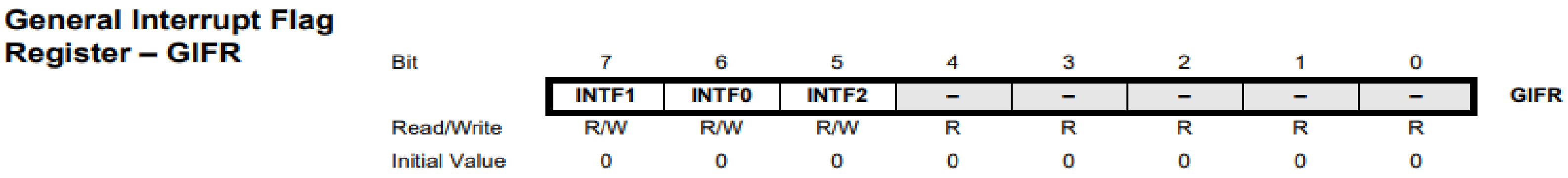


General Interrupt Control Register – GICR

Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	INT2	–	–	–	IVSEL	IVCE	GICR
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

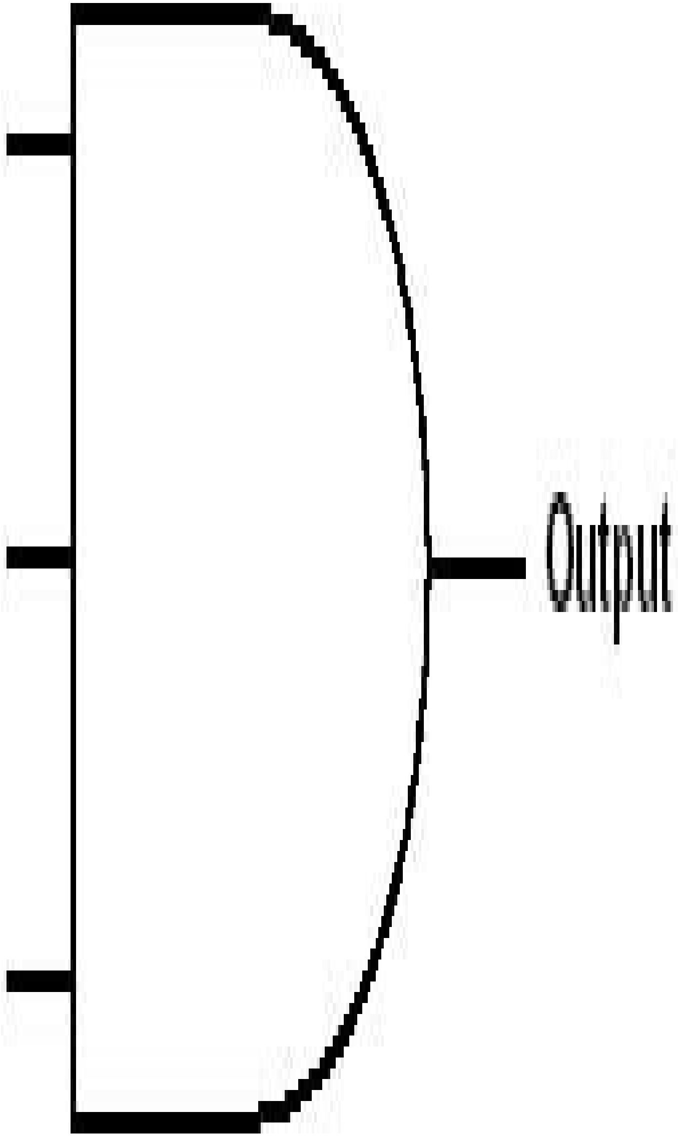
→ **There are 2nd step here to configure when you want to receive interrupt**

❑ Module Interrupt flag (MIF).



General Interrupt Flag Register – GIFR

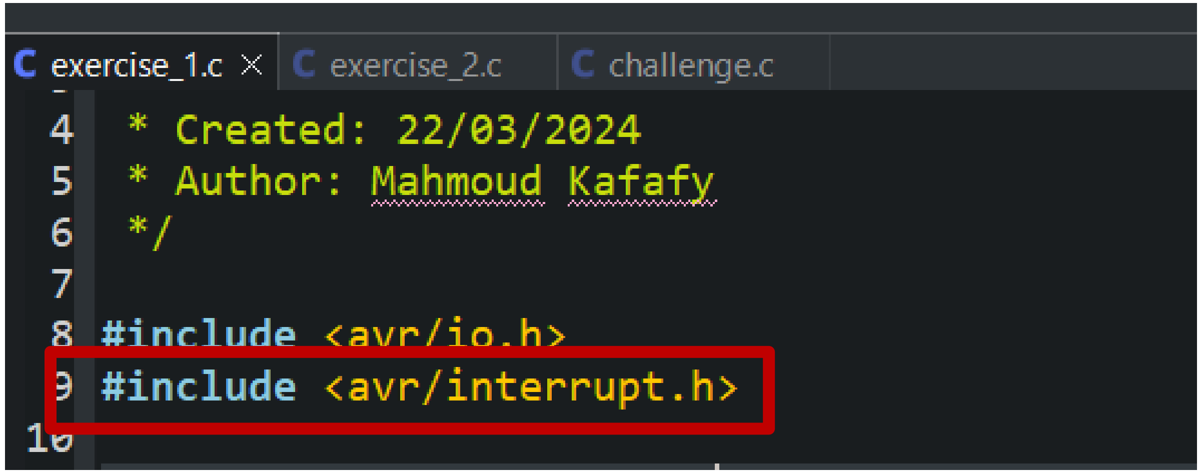
Bit	7	6	5	4	3	2	1	0	
	INTF1	INTF0	INTF2	–	–	–	–	–	GIFR
Read/Write	R/W	R/W	R/W	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	



Interrupt Coding Notes

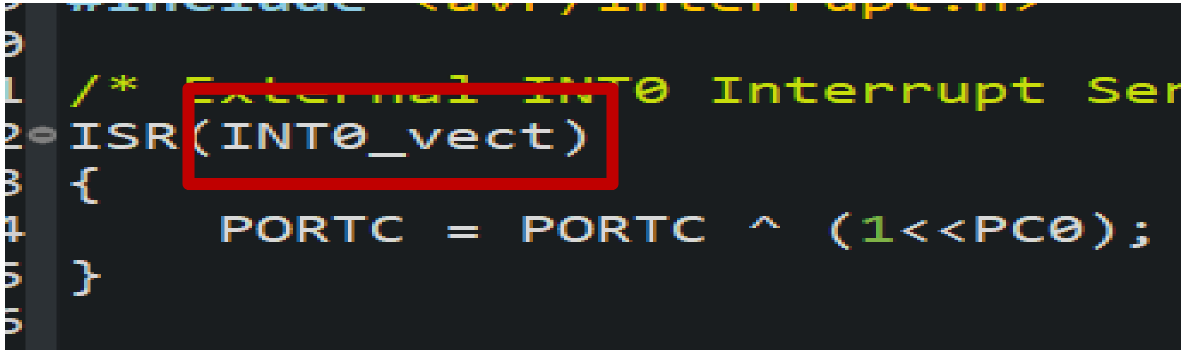
1.
2.
3.
4.

- ❑ AVR-GCC compiler needs a library to understand ISR definition



```
 * Created: 22/03/2024
 * Author: Mahmoud Kafafy
 */

#include <avr/io.h>
#include <avr/interrupt.h>
```

- ❑ Write ISR for a specific interrupt using AVR library:



```
#include <avr/interrupt.h>

/* External INT0 Interrupt Ser
ISR(INT0_vect)
{
    PORTC = PORTC ^ (1<<PC0);
}
```

Toggle pin

D-BUGERZ

1.

2.

3.

4.

- First Time:

Initial Value = 0000 0000

^

1<<1 = 0000 0010

0000 0010

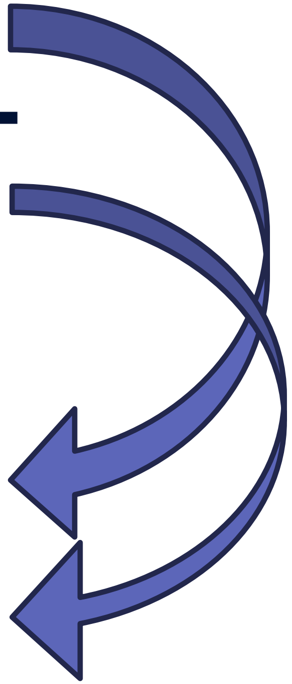
- Second Time:

1<<1 = 0000 0010

^

0000 0010

0000 0000



Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Exercise 1

1.
2.
3.
4.

Write Embedded C code using ATmega16/32 µC to control led by external interrupt INTO

Requirements:

1- Configure the PC clock with internal 1Mhz Clock.
2- The led is connected to pin 0 in PORTC.
3- Connect the Led using Negative Logic configuration.
4- Connect a push button with Pull Down configurations at INTO pin (PD2).
5-When the INTO is triggered just toggle the led.

Interrupt Coding Notes

1.

2.

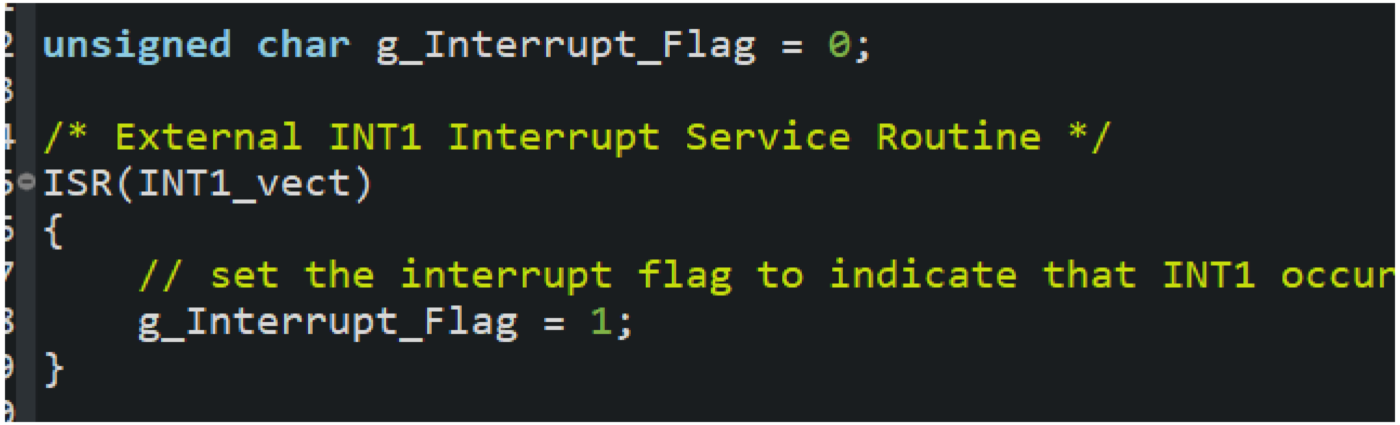
3.

4.

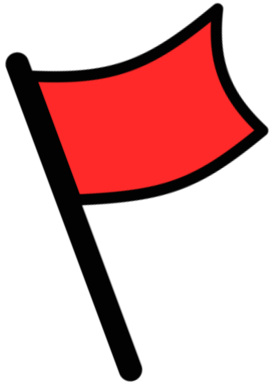
❑ 0xF8 = 1111 1000

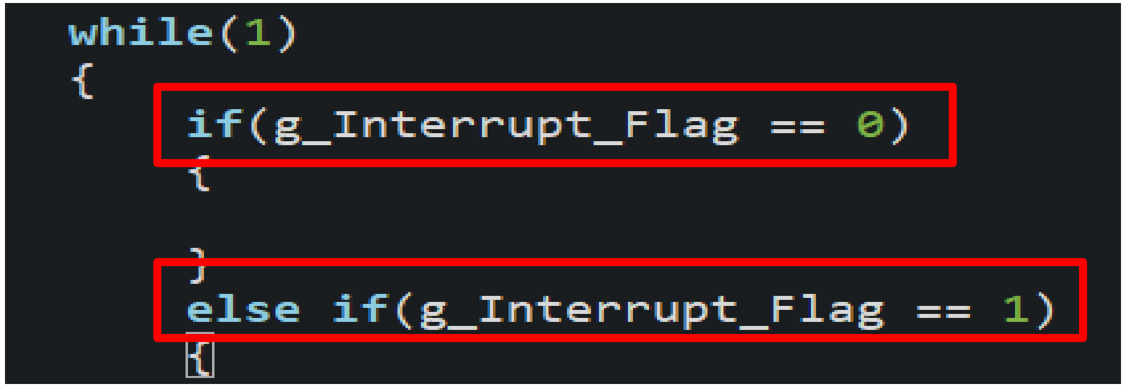
❑ 0x07 = 0000 0111

❑ We don’t need to write long code in ISR so what we do?



```
unsigned char g_Interrupt_Flag = 0;

/* External INT1 Interrupt Service Routine */
ISR(INT1_vect)
{
    // set the interrupt flag to indicate that INT1 occur
    g_Interrupt_Flag = 1;
}
```





```
while(1)
{
    if(g_Interrupt_Flag == 0)
    {

    }
    else if(g_Interrupt_Flag == 1)
    {
```

Exercise 2

1.

2.

3.

4.

Write Embedded C code using ATmega16/32 µC to control 3-LEDs by external interrupt INT1

<u>Requirements:</u>

1- Configure the µC clock with 16Mhz Crystal Oscillator.

2- Use the 3 LEDs at PC0, PC1 and PC2.

3- LEDs are connected using Positive Logic configuration.

4- A roll action is perform using the LEDs each led for half second. The first LED is lit and roll down to the last LED then back to the first LED. This operation is done continuously.

5- When the INT1 is triggered all three LEDs flash five times for five seconds.

Challenge 4

D-BUGERZ

Write Embedded C code using ATmega16/32 µC to control a 7-segment using a INT2

Requirements:

- ✓ Configure the µC clock with internal 1Mhz Clock.
- ✓ The 7-segment is connected to first 4-pins of PORTC.
- ✓ The 7-Segment type is common anode.
- ✓ Connect the switch using Pull down configuration on INT2/PB2 pin.
- ✓ When the INT2 is triggered just increase the number appeared in the 7-Segment display, and if the 7-segment reaches the maximum number (9) overflow occurs.

Do you have any questions?

hello@mail.com
555-111-222
mydomain.com

D-BUGERZ