AVR INTERRUPT PROGRAMMING IN ASSEMBLY AND C .

- A microcontroller can serve several devices using two methods: interrupts or polling.
- **Interrupt Method:**
- When a device needs service, it notifies the CPU by sending an interrupt signal.
- The CPU stops whatever it is doing and serves the device.
- This method allows the CPU to perform other tasks until an interrupt occurs, optimizing CPU time usage.
- **Polling Method:**
- The CPU continuously monitors the status of devices.
- When a specific condition is met, the CPU performs the required service.
- After servicing, the CPU moves on to monitor the next device.
- This method is inefficient because the CPU wastes time checking devices that do not need service.
- **Comparison and Example:**
- Polling can monitor multiple devices but at the cost of wasting CPU time.
- For example, when using timers, a bit test instruction ("SBIS TIFR0,TOV0") can be used to wait until the timer overflows. While waiting, the CPU cannot perform other tasks, which is an inefficient use of CPU time.
- Using the interrupt method, the CPU can perform other tasks and only gets interrupted when the timer signals that the time has elapsed, improving efficiency.

Interrupt service routine

- For every interrupt there must be a program associated with it.
- When an interrupt occurs this program is executed to perform certain services for the interrupt. This program is commonly referred to as an interrupt service routine (ISR).
- The interrupt service routine is also called the interrupt handler. When an interrupt occurs, the CPU runs the interrupt service routine. Now the question is how the ISR gets executed?

```
MAIN PROGRAM:
  Repeat the following forever
  {
     if UART received data
        Get the data and process it

     if time elapsed
        Do the task
  }
```

(a) Polling

```
MAIN PROGRAM:
 Do a task

On UART receive interrupt:
  Get the data and process it

On timer interrupt:
  Do the task
```

(b) Interrupt

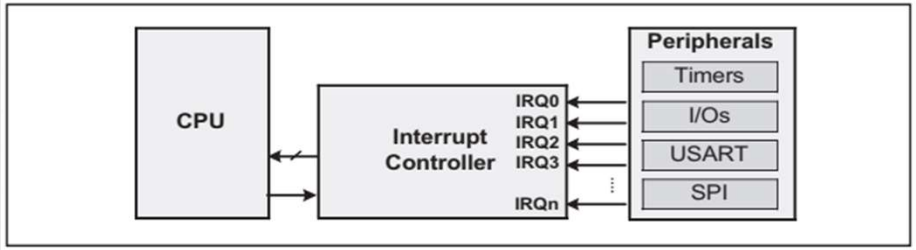
- In most CPUs, there are pins associated with interrupt controllers. These are input signals to the CPU. When these signals are triggered, the CPU pushes the Program Counter (PC) register onto the stack and loads the PC register with the address of the interrupt service routine (ISR). This causes the ISR to execute.

- In most microprocessors, each interrupt has a fixed location in memory that holds the address of its ISR. The group of memory locations set aside to hold the addresses of ISRs is called the interrupt vector table.

Table 10-1: Interrupt Vector Table for the ATmega328 AVR

Interrupt	ROM Location (Hex)
Reset	0000
External Interrupt Request 0	0002
External Interrupt Request 1	0004
Pin Change Interrupt Request 0	0006
Pin Change Interrupt Request 1	0008
Pin Change Interrupt Request 2	000A
Watchdog Time-out Interrupt	000C
Time/Counter2 Compare Match A	000E
Time/Counter2 Compare Match B	0010
Time/Counter2 Overflow	0012
Time/Counter1 Capture Event	0014
Time/Counter1 Compare Match A	0016
Time/Counter1 Compare Match B	0018
Time/Counter1 Overflow	001A
Time/Counter0 Compare Match A	001C
Time/Counter0 Compare Match B	001E
Time/Counter0 Overflow	0020
SPI Serial Transfer complete	0022
USART, Receive Complete	0024
USART, Data Register Empty	0026
USART, Transmit Complete	0028
ADC Conversion Complete	002A
EEPROM Ready	002C
Analog Comparator	002E
Two-wire Serial Interface (I2C)	0030
Store Program Memory Ready	0032

Note: In Atmega328, you can move the interrupt vector table to the beginning of boot Loader Section. For more information see the datasheet.



Steps in executing an interrupt

Here's the detailed process of what happens when an interrupt is activated in the AVR microcontroller:
1.Completion of Current Instruction: The microcontroller finishes executing the current instruction and saves the address of the next instruction (the program counter, PC) onto the stack.
2.Jump to Interrupt Vector Table: It then jumps to a fixed location in memory known as the interrupt vector table. This table directs the microcontroller to the address of the interrupt service routine (ISR).
3.Execution of ISR: The microcontroller starts executing the ISR until it reaches the last instruction of the subroutine, which is RETI (return from interrupt).
4.Return to Interrupted Location: Upon executing the RETI instruction, the microcontroller returns to the point where it was interrupted. It retrieves the PC address from the stack by popping the top bytes of the stack into the PC and starts executing from that address.
Critical Role of the Stack: From step 4, it's clear that the stack plays a crucial role in handling interrupts. Therefore, it's essential to handle the stack contents carefully within the ISR. Specifically, in the ISR, just as in any CALL subroutine, the number of pushes and pops must be balanced.
Visual Reference: The uploaded image likely contains additional information or diagrams that illustrate these steps or the interrupt vector table. Let me know if you need further details or explanations based on the image content

Sources of Interrupts in the AVR

- There are various sources of interrupts in the AVR microcontroller, depending on which peripherals are included in the chip. Some of the most widely used sources of interrupts in the AVR are:

1. **Timers**: There are at least two interrupts set aside for each timer, one for overflow and another for compare match.
2. **External Hardware Interrupts**: In the ATmega328, two interrupts are set aside for external hardware interrupts. Pins PD2 (PORTD.2) and PD3 (PORTD.3) are for the external hardware interrupts INT0 and INT1, respectively.

1. **USART (Serial Communication)**: There are three interrupts for the USART, one for receive and two for transmit.
2. **SPI Interrupts**:
3. **ADC (Analog-to-Digital Converter)**

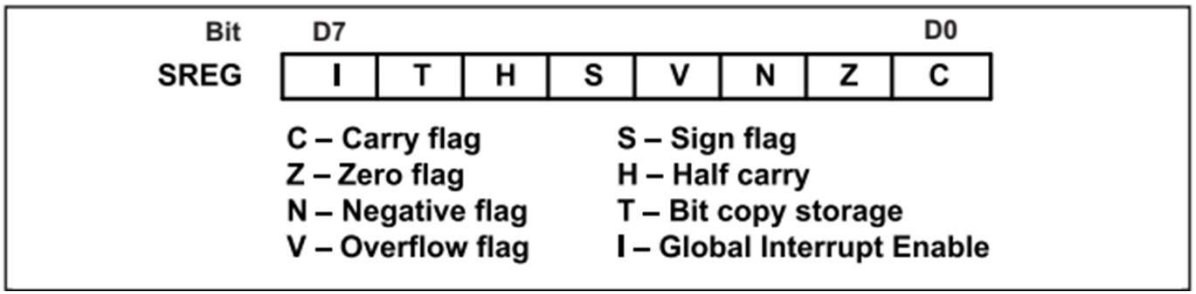
- These sources allow the microcontroller to handle various events and data transfers efficiently, leveraging the interrupt mechanism to optimize CPU usage and responsiveness.

```
            .ORG  0      ;wake-up ROM reset location
            JMP   MAIN   ;bypass interrupt vector table

;---- the wake-up program
            .ORG  $100
MAIN:       ....         ;enable interrupt flags
            ....
```

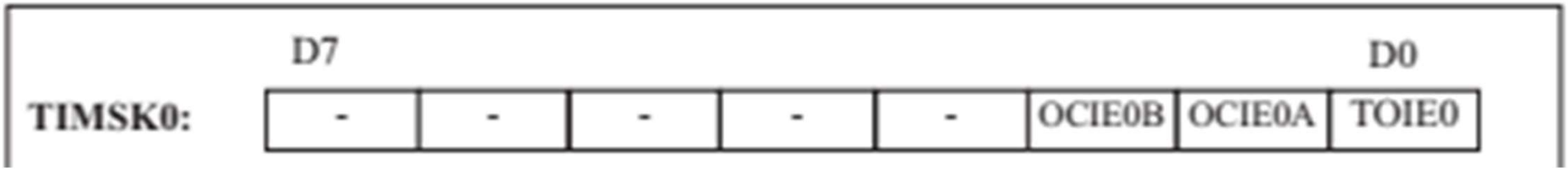
Enabling and disabling an interrupt

- Upon reset, all interrupts are disabled (masked), meaning that none will be responded to by the microcontroller if they are activated.
- The interrupts must be enabled (unmasked) by software in order for the microcontroller to respond to them.
- The D7 bit of the SREG (Status Register) register is responsible for enabling and disabling the interrupts globally.
- Figure shows the SREG register. The I bit makes the job of disabling all the interrupts easy.
- With a single instruction "CLI" (Clear Interrupt), we can make I = 0 during the operation of a critical task.



Steps in Enabling an Interrupt

To enable any interrupt in the AVR microcontroller, follow these steps:
1.Set Global Interrupt Enable: Bit D7 (I) of the SREG register must be set to HIGH to allow interrupts to happen. This is done using the SEI (Set Interrupt) instruction.
2.Enable Specific Interrupts: Once the global interrupt enable bit (I) is set to 1, each specific interrupt is enabled by setting the interrupt enable (IE) flag bit for that interrupt to HIGH. There are I/O registers that hold the interrupt enable bits.
For example, the TIMSKn registers have interrupt enable bits for Timer0, Timer1, and Timer2. As each peripheral is studied, the corresponding registers holding the interrupt enable bits will be examined. It is important to note that if the global interrupt enable bit (I) is 0, no interrupt will be responded to, even if the specific interrupt enable bit is set to HIGH.
Visual Reference: Figure shows the TIMSKn registers with interrupt enable bits for the timers, providing a visual guide for this process.



TOIE0 (Timer0 Overflow Interrupt Enable)

•= 0: Disables Timer0 overflow interrupt

•= 1: Enables Timer0 overflow interrupt

OCIE0A (Timer0 Output Compare A Match Interrupt Enable)

•= 0: Disables Timer0 compare A match interrupt

•= 1: Enables Timer0 compare A match interrupt

OCIE0B (Timer0 Output Compare B Match Interrupt Enable)

•= 0: Disables Timer0 compare B match interrupt

•= 1: Enables Timer0 compare B match interrupt

These bits are used to enable or disable specific interrupts for Timer0 in the AVR microcontroller. Setting the corresponding bit to 1 will enable the interrupt, allowing the microcontroller to respond to the specified event, while setting it to 0 will disable the interrupt, preventing the microcontroller from responding to that event.

Table 10-2: Timer Interrupt Flag Bits and Associated Registers				
Interrupt	**Overflow Flag Bit**	**Register**	**Interrupt Enable Bit**	**Register**
Timer0	TOV0	TIFR0	TOIE0	TIMSK0
Timer1	TOV1	TIFR1	TOIE1	TIMSK1
Timer2	TOV2	TIFR2	TOIE2	TIMSK2

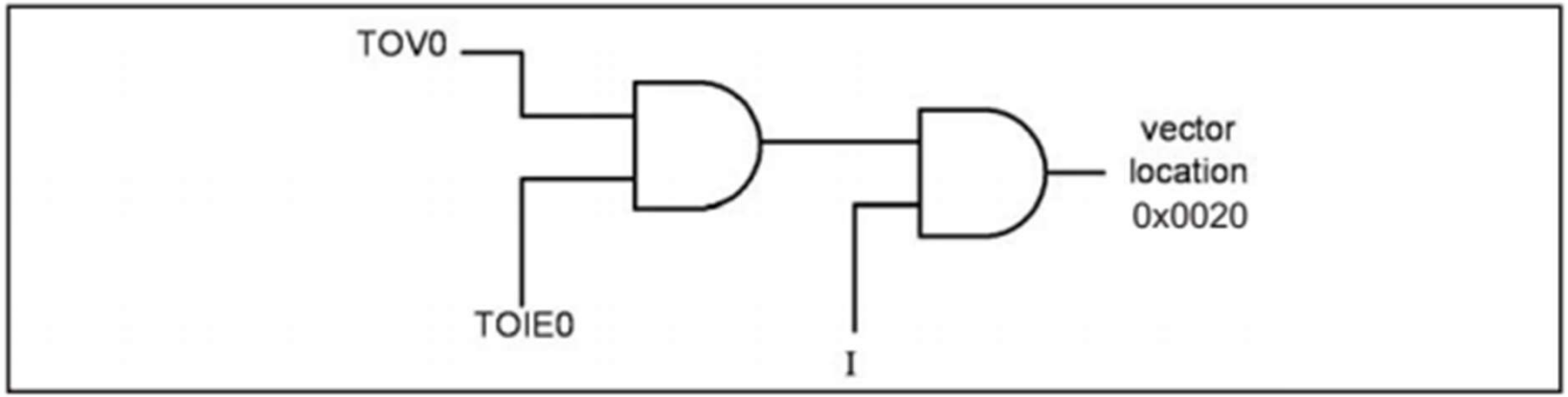


Figure 10-6. The Role of Timer Overflow Interrupt Enable (TOIE0)

Algorithm for Counting Pulses Using an External Interrupt on AVR

1.Initialize variables:

- Create a volatile uint16_t variable pulse_counter to store the pulse count.

2.Configure external interrupt:

- Set pin PD2 as an input.
- Enable the internal pull-up resistor on pin PD2.
- Configure the external interrupt on INT0 to detect a rising edge.
- Enable the external interrupt INT0.
- Enable global interrupts.

3.Define the Interrupt Service Routine (ISR):

- Increment the pulse_counter variable each time a rising edge is detected on pin INT0.

4.Configure port B for output:

- Set all pins on port B as output.

5.Main program loop:

- In the main loop, update port B with the value of the pulse_counter to display it on the LEDs connected to port B