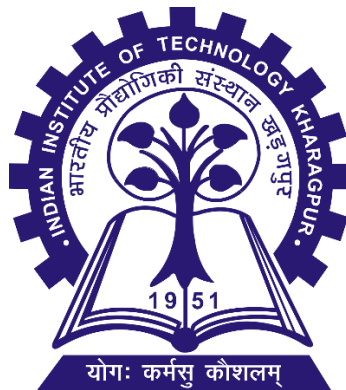


Embedded System on AVR Microcontroller (ATMEGA32)

Exp4: Hardware Interrupts in Microcontroller ATMEGA32

Submitted by
Ronit Dutta, MS in IOT and Signal Processing
Department of Electrical Engineering, IIT Kharagpur

Under the guidance of
Aurobinda Routray, Professor, Electrical Engineering



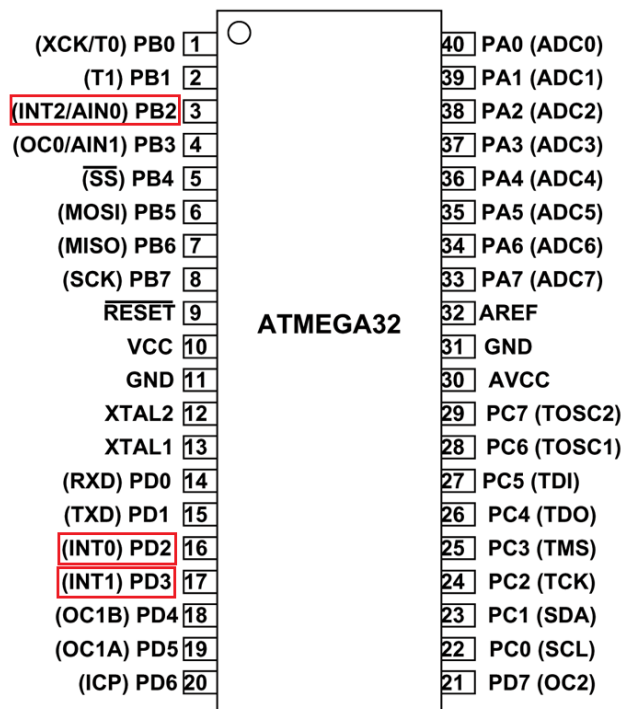
Department of Electrical Engineering
Indian Institute of Technology Kharagpur
January, 2024

• Interrupts in Microcontroller ATMEGA32

There are many sources of interrupts in the AVR, depending on which peripheral is incorporated into the chip. The following are some of the most widely used sources of interrupts in the AVR.

I. Hardware Interrupt:

AVR ATMEGA32 has three external hardware interrupts on pins PD2, PD3, and PB2 which are referred to as INT0, INT1, and INT2 respectively. Upon activation of these interrupts, the ATMEGA32 controller gets interrupted in whatever task it is doing and jumps to perform the interrupt service routine.



External interrupts can be level-triggered or edge-triggered. We can program this triggering. INT0 and INT1 can be level-triggered and edge-triggered whereas INT2 can be only edge-triggered.

- II. There are at least two interrupts set aside for each of the timers, one for overflow and another for compare match.
- III. Serial communication (USART) has three interrupts, one for receive and two interrupts for transmit.
- IV. The SPI (Serial Peripheral Interface) interrupts.
- V. The ADC (Analog to Digital Converter) interrupts.
- VI. The TWI (Two-Wire serial Interface) interrupts. It is also called as I2C or IIC (Inter Integrated Circuit).
- VII. EEPROM Ready interrupt.
- VIII. Store Program Memory Ready Interrupt.

A single microcontroller can serve several devices. There are two methods by which devices receive service from the microcontroller: **interrupts or polling**

• Interrupts Vs. Polling

In the interrupt method, whenever any device needs the microcontroller's service, the device notifies it by sending an interrupt signal. Upon receiving an interrupt signal, the microcontroller stops whatever it is doing and serves the device.

The program associated with the interrupt is called the **interrupt services routine (ISR)** or **interrupt handler**.

In **poling**, the microcontroller continuously monitors the status of a given device. When the status is met, it performs the service. After that, it moves on to monitor the next device until each one is serviced. Although polling can monitor the status of several devices and serve each of them as certain conditions are met, it is not an efficient use of the microcontroller.

- ✓ The advantages of interrupts is that the microcontroller can serve many devices (not all at the same time, of course); each device can get the attention of the microcontroller based on the priority assigned to it.
- ✓ The polling method cannot assign priority because it checks all devices in a round-robin fashion.
- ✓ More importantly, in the interrupt method the microcontroller can also ignore (mask) a device request for service.
- ✓ This also is not possible with the polling method.
- ✓ The most important reason that the interrupt method is preferable is that the polling method wastes much of the microcontroller's time by polling devices that do not need service.
- ✓ So interrupts are used to avoid tying down the microcontroller.

• Interrupt Vectors in ATmega32

For every interrupt, there must be an interrupt service routine (ISR), or interrupt handler. When an interrupt is invoked, the microcontroller runs the interrupt service routine. Generally, in most microcontrollers, for every interrupt there is a fixed location in memory that holds the address of its ISR. The group of memory locations set aside to hold the address of ISRs is called the interrupt vector table, as shown in Table below.

Table: Interrupt Vector Table

| 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 |

• Steps in executing an interrupt

Upon activation of an interrupt, the microcontroller goes through the following steps:

1. It finishes the instruction which is currently executing and saves the address of the next instruction (program counter) on the stack.
2. It jumps to a fixed location in memory called the interrupt vector table. The interrupt vector table directs the microcontroller to the address of the interrupt service routine (ISR).
3. The microcontroller starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine, which is **RETI (RETurn from Interrupt)**.
4. Upon executing the RETI instruction, the microcontroller returns to the place where it was interrupted. First, it gets the program counter (PC) address from

the stack by popping the top bytes of the stack into the PC. Then it starts to execute from that address.

• Enabling and Disabling an Interrupt through Status Register

The Status Register contains information about the result of the most recently executed arithmetic instruction. This information can be used for altering program flow in order to perform conditional operations. Note that the Status Register is updated after all ALU operations, as specified in the Instruction Set Reference. This will in many cases remove the need for using the dedicated compare instructions, resulting in faster and more compact code.

The Status Register is not automatically stored when entering an interrupt routine and restored when returning from an interrupt. This must be handled by software.

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 | |

| | | | | | | | | |
|---------------------|--|--|--------------|----------------|---|---|--|---|
| SEI | | | Set SREG I | SREG-Bit I ← 1 | - | 1 | | 1 |
| CLI | | | Clear SREG I | SREG-Bit I ← 0 | - | 1 | | 1 |

In assembly, the CLI and SEI instructions clear and set the I-bit of the SREG register respectively. In C, the cli() and sei() macros do the same task.

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 Associated with External Interrupts

I. GICR (General Interrupt Control Register)

| Bit | ^{PB2} 7 | ^{PD2} 6 | ^{PD3} 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 | |

INT0: When this bit is '1' and global interrupt bit in SREG is '1' (i.e. I bit), then INT0 (External Interrupt 0) is enabled.

INT1: When this bit is '1' and global interrupt bit in SREG is '1' (i.e. I bit), then INT1 (External Interrupt 1) is enabled.

INT2: When this bit is '1' and global interrupt bit in SREG is '1' (i.e. I bit), then INT2 (External Interrupt 2) is enabled.

There are two type of activation for the external hardware interrupts.

- Level Triggered
- Edge Triggered

INT0 and INT1 can be edge or level triggered, but INT2 can be edge triggered only.

The MCUCR and MCUCSR registers decide the triggering options of the external hardware interrupts INT0, INT1 and INT2.





II. MCUCR (MCU Control Register)

| 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 | |

This register decides the triggering options of the external hardware interrupts INT0 and INT1.

ISC01 and ISC00 (Interrupt Sense Control bits):





These bits define the level or edge that triggers the INT0 pin.

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

ISC11 and ISC10 (Interrupt Sense Control bits):

These bits define the level or edge that triggers the INT1 pin.

LAFR

| ISC11 | ISC10 | | Description |
|-------|-------|---|--|
| 0 | 0 |  | The low level on the INT1 pin generates an interrupt request. |
| 0 | 1 |  | Any logical change on the INT1 pin generates an interrupt request. |
| 1 | 0 |  | The falling edge on the INT1 pin generates an interrupt request. |
| 1 | 1 |  | The rising edge on the INT1 pin generates an interrupt request. |



III. MCUCSR (MCU Control and Status Register)

| 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 |

This register decides the triggering options of the external hardware interrupt INT2.

ISC2 (Interrupt Sense Control bit):

ISC2 bit defines the INT2 interrupt edge triggering.

| ISC2 | | Description |
|------|---|--|
| 0 |  | The falling edge on the INT2 pin generates an interrupt request. |
| 1 |  | The rising edge on the INT2 pin generates an interrupt request. |

FR

IV. GIFR (General Interrupt Flag Register)

| 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 | |

INTF1: External Interrupt Flag 1

When an edge or logic change on the INT1 pin triggers an interrupt request, INTF1 becomes set(one). If the I-bit in SREG and the INT1 bit in GICR are set (one), the MCU will jump to the corresponding Interrupt Vector. The flag is cleared when the interrupt routine is executed. This flag is always cleared when INT1 is configured as a level interrupt.

INTF0: External Interrupt Flag 0

When an edge or logic change on the INTO pin triggers an interrupt request, INTF0 becomes set(one). If the I-bit in SREG and the INTO bit in GICR are set (one), the MCU will jump to the corresponding interrupt vector. The flag is

cleared when the interrupt routine is executed. This flag is always cleared when INT0 is configured as a level interrupt.

INTF2: External Interrupt Flag 2

When an event on the INT2 pin triggers an interrupt request, INTF2 becomes set (one). If the I-bit in SREG and the INT2 bit in GICR are set (one), the MCU will jump to the corresponding Interrupt Vector. The flag is cleared when the interrupt routine is executed.

● Interrupts Priority:

If two interrupts are activated at the same time, the interrupt with the higher priority is served first. The priority of each interrupt is related to the address of that interrupt in the interrupt vector.

The interrupt that has a lower address, has a higher priority.

For example, the address of INT0 is 0x0002, the address of INT1 is 0x0004 and the address of INT2 is 0x0006. Thus, the INT0 has a higher priority, then INT1 and then INT2. If all of these interrupts are activated at the same time, then INT0 is served first.

● Example in C-Language

A Push button is connected at INT0 to sense Edge Trigger for turning On and Off the LED at PORTC PIN0(That is complement of the previous state for an Edge Trigger) and continuously one LED at PORTA PINA0 is blinking.

Note: For Simulation on SimulIDE 1.0.0 always use Edge Trigger. The SimulIDE 1.0.0 does not support Level Trigger simulation. The Level Trigger simulation is possible on SimulIDE 0.4.15.

```
// Interrupt at INT0 Edge Triggered
```

```
#include <avr/io.h>
#define F_CPU 1000000L
#include <util/delay.h>
#include <avr/interrupt.h>
```

```
int main(void)
{
    DDRD=0x00;
    PORTD=0x04;
    DDRA=DDRA|0x01;
    DDRC=DDRC|0x01;
    PORTC=0x00;
    GICR=0x40;
    MCUCR=0x03;
    sei();
    while (1)
    {
```



```

        PORTA=PORTA|0x01;
        _delay_ms(500);
        PORTA=PORTA&0xFE;
        _delay_ms(500);
    }

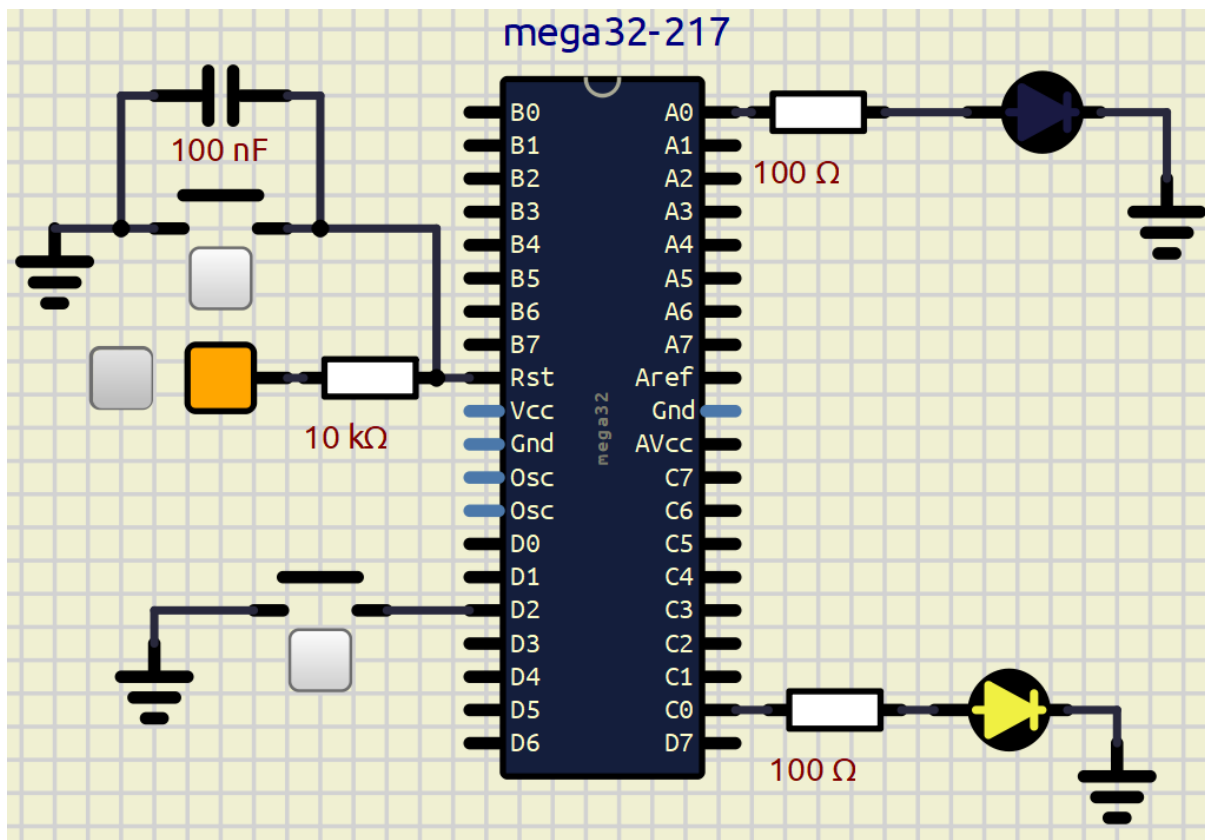
ISR(INT0_vect)
{
    PORTC=!PORTC;
    _delay_ms(1000);
}

```

• Make the below Circuit on SimulIDE to simulate

Components Required:

1. ATmega32
2. Push Switches
3. 10K Ohm Resistors
4. 100 Ohm Resistor
5. Fixed Voltage
6. 100nF Capacitor
7. LEDs



After uploading the HEX file, verify the simulation by TA and Show the MCU Monitor Status for GICR, MCUCR, GIFR etc.



• Above Example in Assembly Level Language

```
// Interrupt Assembly
.INCLUDE "M32DEF.INC"
.ORG 0x0000
    JMP MAIN
.ORG 0x0002
    JMP EX0_ISR
MAIN:  LDI R16,HIGH(RAMEND)
        OUT SPH,R16
        LDI R16,LOW(RAMEND)
        OUT SPL,R16

        SBI DDRA,PINA0
        SBI DDRC,PINC0
        CBI PORTC,PINC0
        CBI DDRD,PIND2
        SBI PORTD,PIND2 Enable pull-up on PD2

        SEI
        LDI R16,0x40
        OUT GICR,R16
        LDI R16,0x03
        OUT MCUCR,R16 LAFR

        LDI R16,0x01;

LOOP:  SBI PORTA,PINA0
        CALL Delay
        CBI PORTA,PINA0
        CALL Delay
        JMP LOOP

EX0_ISR:
        COM R16
        OUT PORTC,R16
        RETI

Delay:  LDI R17,0xFF //Loop 1
L1:    LDI R18,0xFF // Loop 2
L2:    LDI R19,0x04 // Loop 3
L3:    NOP
        DEC R19
        BRNE L3 //Loop 3 End
        DEC R18
        BRNE L2 //Loop 2 End
        DEC R17
        BRNE L1 // Loop 1 End
        RET
```

Make the above circuit and then upload the HEX file, verify the simulation by TA and Show the MCU Monitor Status for GICR, MCUCR, GIFR etc.

- Class Assignment 1:** Write a C and assembly code for ATMEGA32 as shown in below circuit to accept Edge Trigger to increment and decrement the digit on seven segment display. The LED on PORTC0 is continuously blinking.

PUSH Button at PORTD2: For Increment the digit

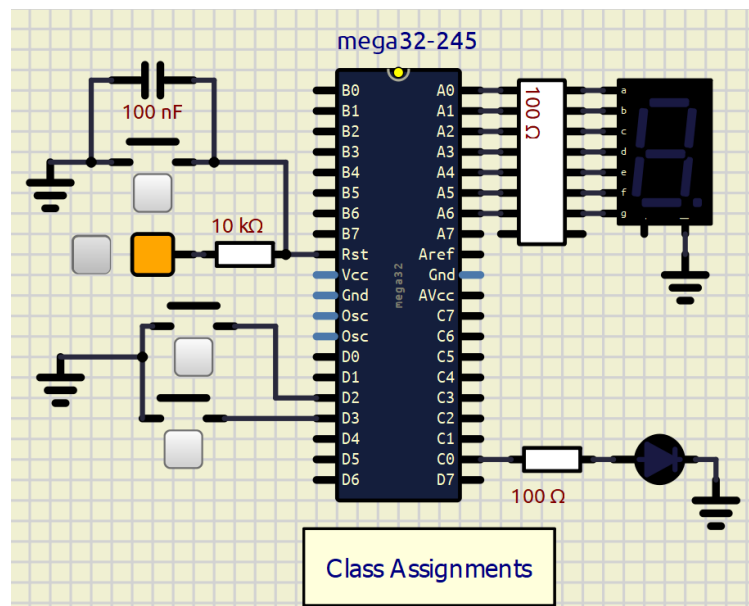
PUSH Button at PORTD3: For Decrement the digit

After 9 the digit should be 0 when incrementing.

After 0 the digit should be 9 when decrementing.

Use PORTA for Common Cathode Seven Segment Display.

Simulate the code on SimulIDE and make the hardware.



- Class Assignment 2:** Write a C and assembly code for ATMEGA32 as shown in below circuit to accept Edge Trigger for toggling between free running UP Counter and free running DOWN Counter.

