

به نام خدا

برنامه‌ریزی وقفه‌ها در AVR

آشنایی با وقفه‌ها

Dr. Aref Karimiafshar
A.karimiafshar@iut.ac.ir



Interrupts vs. Polling

- Two methods of receiving service from μC
 - Polling
 - The μC continuously monitors the status of a given device
 - When the status condition is met
 - » It performs the service
 - Interrupts
 - Whenever the device needs μC 's service
 - Notifies the μC by sending an interrupt signal
 - » Upon receiving the interrupt signal, the μC stops whatever it is doing and serves the device
 - The program associated with the interrupt is called
 - » Interrupt Service Routine (ISR) or Interrupt Handler

دو روش برای گرفتن سرویس از میکروکنترلر وجود دارد:
: Polling

در روش Polling میکروکنترلر مرتب میاد و وضعیت یک ابزاری رو که قراره بیاد و سرویس بگیره رو چک میکنه و وقتی که یکسری شرایط خاصی برقرار بود متوجه میشه کارمون آماده شده و حالا نیاز به سرویس داره یا باید سرویس بده بدین ترتیب میاد و اون سرویسی که باید انجام بشه رو اجرا میکنه پس در روش Polling خود پردازنده مرتب چک میکنه وزمانی که شرایط برقرار بود میاد به اون اون وسیله یا ابزاری که مدنظر هست سرویس میده

: Interrupts

توی این حالت وقتی که یک وسیله نیاز داره تا از پردازنده ما سرویس بگیره میاد و میکروکنترلر مارو با ارسال یک سیگنال Interrupts متوجه این قضیه می کنه و با دریافت این سیگنال Interrupts میکروکنترلر کار خودش رو متوقف می کنه و شروع میکنه به اون دیوایس سرویس دادن به اون برنامه ای که متناظر با این سرویس هستش و با فعال شدن این Interrupt قراره اجرا بشه بهش میگیم Interrupt Service Routine

Interrupts vs. Polling

- Polling
 - Can monitor the status of several devices
 - Serve each of them as certain conditions are met
 - Moves on to the next until each one is serviced
 - It is not an efficient use of μC
- Interrupts
 - μC can service many devices (not all at the same time, of course)
 - Each device can get the attention of the μC based on the priority assigned to it (polling cannot assign priority because it checks all devices in round-robin fashion)
 - The μC can ignore (mask) a device request (not possible in polling)

وقتی که پولینگ داریم میکروکنترلر ما میتونه وضعیت چندین دیوایس رو بیاد و چک بکنه و وقتی که یک شرایط خاصی رخ داد بیاد به اون ها سرویس بده و به این ترتیب میاد توی اون حلقه ای که داره مرتب این ها رو چک میکنه و وقتی به یکی سرویس داد میره سراغ بعدی ها تا همه اون ها رو سرویس بده --> این یک روش بهینه ای نیست

در مقابل ما Interrupts رو داریم که باز هم میکروکنترلر ما میتونه به تعداد زیادی از ابزارها سرویس بده منتها اینجا اون وسیله ای که میخواد سرویس بگیره میاد و میکروکنترلر رو با ارسال یک سیگنالی متوجه می کنه و این متوجه کردن براساس یکسری اولویت هایی انجام میشه همچنین امکانی در رابطه با پولینگ وجود نداره (ینی امکان اولویت بندی بین درخواست های صادره از وسایل مختلف وجود نداره)

یک امکان دیگه توی Interrupts : ما میتونیم یکسری درخواست هایی از بعضی از وسایل رو نادیده بگیریم اما همچین امکانی در حالت پولینگ وجود نداره

Interrupts vs. Polling

- Polling
 - Wastes much of μ C's time by polling devices that do not need services
- Interrupts
 - Avoid tying down the μ C

```
AGAIN:IN    R20,TIFR    ;read TIFR
          SBRS  R20,TOV0 ;if TOV0 is set skip next instruction
          RJMP  AGAIN
```

```
while ((TIFR&0x1)==0); //wait for TF0 to roll over
```

مهمترین ویژگی:

پولینگ به هر صورت میاد زمان اون پردازنده رو با چک کردن مدام وضعیت اون ابزارها تلف میکنه ولی Interrupts این کارو انجام نمیده و هیچ محدودیتی روی میکروکنترلر ایجاد نمیکنه پس ویژگی خوبی که Interrupts داره اینه که بدون اینکه بخواد زمان پردازنده ما رو اشغال بکنه و تلف بکنه می تونه اون سرویس خودش رو به ابزارهایی که نیاز داره به موقع برسونه ولی در حالت پولینگ اون پردازنده ما میاد مرتب وضعیت اینارو چک میکنه و وقت خودش رو تلف میکنه

این دو تا مثالی که زده شده اینجا داره میگه که تایم مرتبط با پردازنده ما عملاً داره هدر میره: زمانی که ست می کردیم که یک تایمر شروع بکنه به شمارش کردن و بعد می رفتیم و مرتب فلگ TOV0 چک می کردیم که ببینیم ست شده یا نه ینی به انتهای شمارش خودش رسیده یا نه عملاً توی یک همچین حلقه ای بودیم که چک می کرد ببینه اون فلگه فعال شده یا نه یا توی زبان سی هم همین رو داشتیم ینی توی یک حلقه ای مرتب پردازنده می چرخید تا زمانی که اون TF0 ست بشه

Interrupts

- For every Interrupt
 - There must be an Interrupt Service Routine (ISR) or Interrupt Handler
 - There is a fixed location in memory that holds the address of its ISR
 - The group of memory locations set aside to hold the ISR addresses is called
 - Interrupt Vector Table
- When an interrupt is invoked
 - The μ C runs the ISR

Interrupt:

چجوری از Interrupt استفاده بکنیم؟

برای هر Interrupt که توی Avr وجود داره ما یک ISR داریم که میاد عملیات خاصی رو به ازای رخداد یک Interrupt انجام میده

به ازای هر Interrupt که ما داریم یک محل خاصی تو حافظه وجود داره که ادرس این ISR را توی خودش ذخیره میکنه

به مجموعه همه این محل های حافظه که برای ذخیره سازی ادرس ISR های مرتبط با Interrupt مختلف کنار گذاشته شده ما میگیم Interrupt Vector Table

Interrupt Vector Table یک مجموعه ای از ادرس ها متناظر با Interrupts های متفاوت هستش که داره به ادرس اون محلی اشاره میکنه که ISR مرتبط با یک Interrupts یا

Interrupts های مختلف ذخیره شده

با فعال شدن یک Interrupts فعال میشه ISR اش و میکروکنترلر می ره و ISR مرتبط با اون رو اجرا میکنه

Steps in Executing an Interrupt

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

1. It finishes the instruction it 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.

گام های مرتبط با اجرا شدن یک وقفه:

بعد از اینکه یک وقفه فعال میشه میکروکنترلر این گام هایی که پایین گفته رو میاد و اجرا میکنه

1- اول اون دستوری که داره اجرا میشه رو تموم میکنه و بعد ادرس دستور بعدی که قرار بوده اجرا بشه ینی محتوای **program counter** رو روی استک ذخیره میکنه

2- بعد پرش میکنه به اون محلی از حافظه که **Interrupt Vector Table** هستش و این **Interrupt Vector Table** ما رو هدایت میکنه به اون محلی که **ISR** وجود داره

نکته: این فضایی که برای **Interrupt Vector Table** در نظر گرفته شده بسیار کوچیکه

3- بعد میکروکنترلر میاد و شروع میکنه اون سرویس روتین رو انجام دادن تا اینکه برسه به آخرین دستور اون که این دستور **RETI** هستش

4- با اجرا شدن دستور **RETI** میکروکنترلر بر میگردد به محلی که ازش وقفه خورده و این کار هم به این صورت انجام میشه که اون **PC** ما که توی استک ذخیره شده پاپ میشه و میاد و توی **PC** قرار میگیره و بعد میکروکنترلر می ره و از اون دستوری که الان **PC** داره بهش اشاره میکنه شروع می کنه به اجرا کردن

Interrupt Vector Table ATmega32

Interrupt Vector Table for the ATmega32 AVR

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

یک نمونه از جدول بردار وقفه:

این جدول مرتبط با ATmega32 هستش

نکته: مرتبط با هر کدوم از این وقفه ها ما یک ادرسی داریم که برای این وقفه اختصاص داده شده و این محل از حافظه ما رو ارجاع می ده به جایی که سرویس روتین مرتبط با این وقفه قرار داره

Interrupts in the AVR

Notice from Step 4 the critical role of the stack. For this reason, we must be careful in manipulating the stack contents in the ISR. Specifically, in the ISR, just as in any CALL subroutine, the number of pushes and pops must be equal.

Normally, the service routine for an interrupt is too long to fit into the memory space allocated. For that reason, a JMP instruction is placed in the vector table to point to the address of the ISR.

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

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

نکات:

توی گام 4 ام که گفتیم یک نقش اساسی از استک رو ما می بینیم: اون قسمتی که وقفه ما میاد و فعال میشه و ادرس اون جای بازگشت می ره روی استک قرار میگیره خیلی مهمه که اینجا این احتیاط رو داشته باشیم که به صورت صحیحی از استک استفاده بکنیم

نکته ای که اینجا وجود داره اینه که اگر می خوایم با استک کار بکنیم باید تعداد پاپ ها و پوش ها با هم برابر باشه به نحوی اون مقداری که قراره پاپ باشه ادرس بازگشت ما باشه

نکته: معمولا ساب روتین ها اونقدر بزرگ هستن برای هر وقفه که این ها توی اون فضایی که برای بردار وقفه وجود داره این ها جا نمیشن و اینجا ما یکسری JMP هایی داریم که به اون سرویس روتین مرتبط با اون وقفه اشاره میکنه

مثال:

مثلا برای وقفه ریست توی اون ادرسی که مرتبط با این وقفه جامپ هستش توی اون Vector Table ما این یک پرشی وجود داره که ما رو ارجاع میده به اون ساب روتینی که به صورت مفصل تر اونجا دستوراتی که قراره اجرا بشن رو داریم و لیست میشن

Sources of Interrupts in the AVR

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:

1. There are at least two interrupts set aside for each of the timers, one for overflow and another for compare match.
2. Three interrupts are set aside for external hardware interrupts. Pins PD2 (PORTD.2), PD3 (PORTD.3), and PB2 (PORTB.2) are for the external hardware interrupts INT0, INT1, and INT2, respectively.
3. Serial communication's USART has three interrupts, one for receive and two interrupts for transmit.
4. The SPI interrupts.
5. The ADC (analog-to-digital converter).

ما برای وقفه ها منابع متفاوتی می توانیم داشته باشیم که 5 دسته از این منابع هستند که خیلی پرکاربردن و مهمن:

معمولا به ازای تایمرهای متفاوتی که توی سیستم داریم برای هر کدومشون حداقل دو تا وقفه داریم: یکسری وقفه های خارجی داریم که می تونن از طریق پین شماره 2 و 3 توی پورت D و پین شماره 2 توی پورت B اعمال بشه و اینا رو تحت عنوان وقفه های INT0 , INT1 , INT2 می شناسیم

یکسری وقفه ها برای ارتباط سریال داریم: سه وقفه برای این ارتباط سریال وجود داره: یک وقفش برای دریافت و دو وقفه هم برای ارسال
یکسری از وقفه ها رو برای SPI داریم

و یکسری دیگه هم برای انالوگ به دیجیتال داریم ینی ADC

INT0 <--- همیشه D2

INT1 <--- همیشه D3

INT2 <--- همیشه B2

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.



چجوری وقفه ها رو می تونیم توی AVR فعال کنیم یا غیرفعال بکنیم؟
رجیستر وضعیت رو قبلا گفتیم

توی اولین گام برای اینکه بیایم وقفه ها رو فعال بکنیم باید بیایم و بیت IROST بکنیم : این یک گام پایه برای استفاده از وقفه ها هستش

Steps in Enabling an Interrupt

To enable any one of the interrupts, we take the following steps:

1. Bit D7 (I) of the SREG register must be set to HIGH to allow the interrupts to happen. This is done with the “SEI” (Set Interrupt) instruction.
2. If $I = 1$, each interrupt is enabled by setting to HIGH the interrupt enable (IE) flag bit for that interrupt. There are some I/O registers holding the interrupt enable bits.

Figure below shows that the TIMSK register has interrupt enable bits for Timer0, Timer1, and Timer2.



برای فعال کردن وقفه ها چه کاری باید انجام بدیم؟

- 1- بیت شماره 7 در رجیستر وضعیت رو ست میکنیم و ست کردن این هم از طریق یک دستوری تحت عنوان SEI هستش و این دستور میاد و بیت 1 رو توی رجیستر وضعیت ست میکنه
 - 2- بعد از این که این ست شد برای هر وقفه خاصی ما یکسری بیت های فعال سازی داریم که بسته به اینکه با کدوم یکی از این ها می خوایم کار بکنیم میایم و اون ها رو فعال میکنیم و این ها در یکسری رجیسترهای خاصی توی فضای I/O وجود داره
- یکی از این بیت ها که خیلی پرکاربرد هم هستش مرتبط با تایمرها هست: بیت های مرتبط با وقفه های این تایمرها در یک رجیستری تحت عنوان TIMSK وجود داره

Steps in Enabling an Interrupt

D7							D0
OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0

TOIE0	Timer0 overflow interrupt enable = 0 Disables Timer0 overflow interrupt = 1 Enables Timer0 overflow interrupt
OCIE0	Timer0 output compare match interrupt enable = 0 Disables Timer0 compare match interrupt = 1 Enables Timer0 compare match interrupt
TOIE1	Timer1 overflow interrupt enable = 0 Disables Timer1 overflow interrupt = 1 Enables Timer1 overflow interrupt
OCIE1B	Timer1 output compare B match interrupt enable = 0 Disables Timer1 compare B match interrupt = 1 Enables Timer1 compare B match interrupt
OCIE1A	Timer1 output compare A match interrupt enable = 0 Disables Timer1 compare A match interrupt = 1 Enables Timer1 compare A match interrupt
TICIE1	Timer1 input capture interrupt enable = 0 Disables Timer1 input capture interrupt = 1 Enables Timer1 input capture interrupt
TOIE2	Timer2 overflow interrupt enable = 0 Disables Timer2 overflow interrupt = 1 Enables Timer2 overflow interrupt
OCIE2	Timer2 output compare match interrupt enable = 0 Disables Timer2 compare match interrupt = 1 Enables Timer2 compare match interrupt

ساختار کلی رجیستر TIMSK:

اسلاید رو ببین!!

Example

Show the instructions to (a) enable (unmask) the Timer0 overflow interrupt and Timer2 compare match interrupt, and (b) disable (mask) the Timer0 overflow interrupt, then (c) show how to disable (mask) all the interrupts with a single instruction.

```
(a)  LDI R20, (1<<TOIE0) | (1<<OCIE2) ;TOIE0 = 1, OCIE2 = 1
      OUT TIMSK,R20 ;enable Timer0 overflow and Timer2 compare match
      SEI ;allow interrupts to come in

(b)  IN R20,TIMSK ;R20 = TIMSK
      ANDI R20,0xFF^(1<<TOIE0) ;TOIE0 = 0
      OUT TIMSK,R20 ;mask (disable) Timer0 interrupt
```

We can perform the above actions with the following instructions, as well:

```
IN R20,TIMSK ;R20 = TIMSK
CBR R20,1<<TOIE0 ;TOIE0 = 0
OUT TIMSK,R20 ;mask (disable) Timer0 interrupt

(c)  CLI ;mask all interrupts globally
```

Notice that in part (a) we can use “LDI, 0x81” in place of the following instruction:

“LDI R20, (1<<TOIE0) | (1<<OCIE2)”

مثال:

الف: وقفه های مرتبط با overflow در تایمر صفر و وقفه compare match رو برای تایمر دو فعال بکنیم

ب: میخوایم وقفه overflow رو برای تایمر صفر غیرفعال بکنیم <-- برای این قسمت دو تا روش گفت

ج: میخوایم همه وقفه ها رو غیرفعال بکنیم <-- برای این کار کافیه که اون بیت 1 که توی رجیستر وضعیت وجود داره رو بیایم و کلیر بکنیم

Programming Timer Interrupts

we showed how to monitor the timer flag TOV0, TOV1 and TOV2 with the instruction “SBRSC R20, TOV0”. In polling TOV0, we have to wait until TOV0 is raised. The problem with this method is that the microcontroller is tied down waiting for TOV0 to be raised, and cannot do anything else. Using interrupts avoids tying down the controller. If the timer interrupt in the interrupt register is enabled, TOV0 is raised whenever the timer rolls over and the microcontroller jumps to the interrupt vector table to service the ISR. In this way, the microcontroller can do other things until it is notified that the timer has rolled over. To use an interrupt in place of polling, first we must enable the interrupt because all the interrupts are masked upon reset. The TOIE_x bit enables the interrupt for a given timer. TOIE_x bits are held by the TIMSK register

Timer Interrupt Flag Bits and Associated Registers				
Interrupt	Overflow Flag Bit	Register	Enable Bit	Register
Timer0	TOV0	TIFR	TOIE0	TIMSK
Timer1	TOV1	TIFR	TOIE1	TIMSK
Timer2	TOV2	TIFR	TOIE2	TIMSK

در قسمت قبل که با تایمرها کار می کردیم مستقیما ست شدن فلگ های **overflow** رو چک می کردیم ینی یک دستوری داشتیم که مستقیما چک می کردیم که اگر این پرچم یک شده بود ینی دستور **SBRS R20,TOV0** دستور بعدی ما که جامپ بود رو عبور می کرد و از این حلقه چک کردن خارج میشد این روش باعث میشه که میکروکنترلر خودش رو مشغول این شمارش بکنه و عملا اون مزیت داشتن تایمر رو به این ترتیب از دست می دادیم ولی وقتی که از وقفه استفاده می کنیم وقفه این مشکل رو برای ما حل میکنه ینی هر زمان که این پرچم ست میشد می اومد میکروکنترلر رو از اون جریان عادی دستورات خارج میکرد و می برد سراغ وکتور تیبل و از اونجا به سمت سرویس روتین این وقفه و اون کاری که ما می خواستیم رو می اومدیم و از این طریق انجام میدادیم

برای اینکه بیایم در شمارنده ها از وقفه استفاده بکنیم نیازه که ما علاوه بر اینکه اون بیت | در رجیستر وضعیت فعال میکنیم یکسری بیت های مرتبط با **timer overflow interrupt** ینی **TOIE** رو در رجیستر **TIMSK** فعال بکنیم : مثل داخل جدول : فقط کافیه متناسب با اون تایمری که داریم **TOIE** اش رو فعال بکنیم توی رجیستر **TIMSK**

Example

```
.INCLUDE "M32DEF.INC"
.ORG 0x0           ;location for reset
    JMP    MAIN
.ORG 0x16          ;location for Timer0 overflow
    JMP    T0_OV_ISR      ;jump to ISR for Timer0
;-main program for initialization and keeping CPU busy
.ORG 0x100
MAIN: LDI    R20,HIGH(RAMEND)
      OUT    SPH,R20
      LDI    R20,LOW(RAMEND)
      OUT    SPL,R20      ;initialize stack
      SBI    DDRB,5       ;PB5 as an output
      LDI    R20,(1<<TOIE0)
      OUT    TIMSK,R20    ;enable Timer0 overflow interrupt
      SEI                    ;set I (enable interrupts globally)
      LDI    R20,-32      ;timer value for 4  $\mu$ s
      OUT    TCNT0,R20    ;load Timer0 with -32
      LDI    R20,0x01
      OUT    TCCR0,R20    ;Normal, internal clock, no prescaler
      LDI    R20,0x00
      OUT    DDRC,R20     ;make PORTC input
      LDI    R20,0xFF
      OUT    DDRD,R20     ;make PORTD output
;------ Infinite loop
HERE: IN     R20,PINC      ;read from PORTC
      OUT    PORTD,R20    ;give it to PORTD
      JMP    HERE         ;keeping CPU busy waiting for interrupt 15
```

مثال:

ابتدا اون جدول وکتور مرتبط با وقفه ها رو داریم که از ادرس صفر شروع میشه و اولین دستوری هم که اینجا داریم مرتبط با اون وقفه ریست هستش و در زمانی که سیستم ما ریست میشه و شروع می کنه به کار کردن میاد این دستور جامپ رو اجرا میکنه توی ادرس صفر و این منتقل میکنه روند اجرا رو به یک سری برنامه اصلی

اون حلقه پایین صفحه هم برای این است که ما میخوایم مرتب از پورت C بخونیم و روی پورت D نمایش بدیم واسه همین این داخل حلقه است

همیشه جدول وکتور در ابتدا قرار میگیره

توی main:

ابتدا استک رو میایم آماده میکنیم و initialization میکنیم که از اون قسمت بالای حافظه رم بخواد رشد بکنه به سمت پایین که قبلا اینو گفتیم

Example cnt.

```
;-----ISR for Timer0 (it is executed every 4 µs)
.ORG 0x200
T0_OV_ISR:
    IN     R16,PORTB    ;read PORTB
    LDI    R17,0x20     ;00100000 for toggling PB5
    EOR    R16,R17
    OUT    PORTB,R16    ;toggle PB5
    LDI    R16,-32      ;timer value for 4 µs
    OUT    TCNT0,R16    ;load Timer0 with -32 (for next round)
    RETI               ;return from interrupt
```

Example: For this program, we assume that PORTC is connected to 8 switches and PORTD to 8 LEDs. This program uses Timer0 to generate a square wave on pin PORTB.5, while at the same time data is being transferred from PORTC to PORTD.

مثال: می خواهیم برنامه ای بنویسیم با فرض اینکه پورت C ما به یکسری سویچ متصل هستند ینی به 8 تا سویچ متصل اند قراره از این ها ورودی بگیریم و روی پورت D که به 8 تا LED متصل است نمایش بدیم و می خواهیم از تایمر صفر استفاده بکنیم و یک موج مربعی تولید بکنیم در عین حالی که داریم این موج رو تولید میکنیم روی پین شماره 5 پورت B می خواهیم به صورت مرتب هم داده هایی رو از پورت C بگیریم و روی پورت D نمایش بدیم

4 میلی ثانیه هم تاخیر داریم

Example cnt.

1. We must avoid using the memory space allocated to the interrupt vector table. Therefore, we place all the initialization codes in memory starting at an address such as \$100. The JMP instruction is the first instruction that the AVR executes when it is awakened at address 0000 upon reset. The JMP instruction at address 0000 redirects the controller away from the interrupt vector table.
2. In the MAIN program, we enable (unmask) the Timer0 interrupt with the following instructions:

```
LDI    R16,1<<TOV0
OUT    TIMSK,R16    ;enable Timer0 overflow interrupt
SEI                      ;set I (enable interrupts globally)
```

3. In the MAIN program, we initialize the Timer0 register and then enter into an infinite loop to keep the CPU busy. The loop could be replaced with a real-world application being executed by the CPU. In this case, the loop gets data from PORTC and sends it to PORTD. While the PORTC data is brought in and issued to PORTD continuously, the TOIE0 flag is raised as soon as Timer0 rolls over, and the microcontroller gets out of the loop and goes to \$0016 to execute the ISR associated with Timer0. At this point, the AVR clears the I bit (D7 of SREG) to indicate that it is currently serving an interrupt and cannot be interrupted again; in other words, no interrupt inside the interrupt.

نکته های برنامه قبلی:

1- ما وقفه ها رو اون جدول وکتوری اون قسمتی که ادرس های مرتبط با ساب روتین ها رو ذخیره می کردیم توی ادرس های ابتدایی ذخیره می کردیم و بعد برنامه رو از ادرس مثلا 100 که این 100 می تونه هر عدد دیگه هم باشه خودمون گذاشتیم 100 شروع میکنیم اینجا بحث اینه که اون قسمت اختصاص داده شده به جدول وکتور رو رها میکنیم و از یک ادرس جلوتر شروع میکنیم برنامه هامون رو قرار دادن

توی ادرس صفر ما یک جامپ داشتیم که این مرتبط بود با وقفه ریست و این فضای مرتبط با وکتور تیبل رو پرش میکرد و می رفت توی قسمت کد اصلی

2- وقتی که اولین دستور که مرتبط با وقفه ریست است رو اجرا می کنیم پرش میکنیم به یک تابع main که اونجا یکسری کارهایی رو قراره انجام بدیم مثلا توی این حالت خاص تایمر صفر رو میخوایم استفاده بکنیم پس باید بیایم و اون رو فعال بکنیم و فعال کردن این از طریق یک کردن اون بیت TOV0 هستش توی رجیستر TIMSK و بعد فعال کردن وقفه ها به صورت کلی

3- کار دیگه ای که در تابع main انجام دادیم این بود که تایمر رو آماده می کردیم و بعد اونو فعال می کردیم و بعد از اجرای اینا وارد یک حلقه می شدیم و توی این حلقه بی نهایت یکسری دستورات رو مرتبط پردازنده داره انجام میده و این ادامه پیدا میکنه تا اون تایمر overflow بکنه و پرچم وقفه مرتبط با اون فعال بشه وقتی که این اتفاق می افته میکروکنترلر از اون روال عادی خودش خارج میشه و مراجعه می کنه به ادرس 16 چرا 16؟ چون ادرس مرتبط با overflow کردن تایمر صفر

از ادرس 16 هستش و ما اینجا میایم ادرس ساب روتین مرتبط با این اتفاق رو قرار میدیم و بعد پردازنده می ره و شروع می کنه به اجرا کردن این ISR --> توی این زمان که قراره سوئیچ بشه و منتقل بشه اجرای دستورات ینی اجرای روند برنامه به ISR در اینجا AVR میاد و بیت مرتبط با وقفه ها ینی I رو در رجیستر وضعیت صفر میکنه و این باعث میشه اون وقفه ها به صورت موقت غیرفعال بشه و هیچ وقت وقفه دیگه ای رو پذیرش نکنه ینی وقفه ای رو درون یک وقفه نداشته باشیم البته این امکان وجود داره که بخوایم یک وقفه درون یک وقفه داشته باشیم منتها فعلا مدنظر ما نیست و توی این قسمت اونو غیر فعال میکنیم تا بتونیم به اون سرویس روتین مرتبط با این وقفه

برسیم

Example cnt.

4. The ISR for Timer0 is located starting at memory location \$200 because it is too large to fit into address space \$16–\$18, the address allocated to the Timer0 overflow interrupt in the interrupt vector table.
5. RETI must be the last instruction of the ISR. Upon execution of the RETI instruction, the AVR automatically enables the I bit (D7 of the SREG register) to indicate that it can accept new interrupts.
6. In the ISR for Timer0, notice that there is no need for clearing the TOV0 flag since the AVR clears the TOV0 flag internally upon jumping to the interrupt vector table.

4- ما سرویس روتین رو میایم و در ادرس 200 هگز قرار میدیم و فقط اون ادرس اشاره کننده به ابتدای این سرویس روتین رو در ادرس 16 تا 18 قرار میدیم ینی جایی که مرتبط با وقفه

overflow تایمر صفر هستش دلیل این کار هم این است که سرویس روتین ما بزرگه و توی این ادرس قرار نمیگیره و فقط ما اونجا یک جامپ می داریم که به ادرس 200 هگز مارو هدایت بکنه

5- آخرین دستوری که توی ساب روتین هستش RETI هستش و RETI هم میاد و اون پرچم I رو مجددا فعال میکنه

6- نیاز نیست که به صورت دستی TOV0 رو پاک بکنیم چرا که خود AVR به صورت اتوماتیک وقتی که پرش میکنه به اون ادرس که داخل وکتور تیبل ما هست این رو خودش به صورت اتوماتیک این کارو انجام میده

Example

What is the difference between the RET and RETI instructions? Explain why we cannot use RET instead of RETI as the last instruction of an ISR.

Both perform the same actions of popping off the top bytes of the stack into the program counter, and making the AVR return to where it left off. However, RETI also performs the additional task of setting the I flag, indicating that the servicing of the interrupt is over and the AVR now can accept a new interrupt. If you use RET instead of RETI as the last instruction of the interrupt service routine, you simply block any new interrupt after the first interrupt, because the I would indicate that the interrupt is still being serviced.

مثال:

چه تفاوتی بین دستور RETI و RET هستش؟

در انتهای سرویس روتین های مرتبط با وقفه ها حتما باید از این RETI استفاده بکنیم ولی دلیل این است که وقتی که ما پرش میکنیم به سرویس روتین اون بیت | غیرفعال میشه و در بازگشت توسط دستور RETI این عملیات ست کردن پرچم | در رجیستر وضعیت به صورت اتوماتیک انجام میشه و اگر این کار صورت نگیره وقتی که برمی گردیم از اون ادرس و می خوایم به روال عادی برنامه برگردیم وقفه های ما غیر فعال خواهد بود ولی وقتی که از RETI استفاده میکنیم این در حقیقت میاد و دوباره رجیستر | رو فعال میکنه تا بتونیم وقفه های جدید تری رو دریافت بکنیم و به اون ها مجدد سرویس بدیم

Example: Program below uses Timer0 and Timer1 interrupts simultaneously, to generate square waves on pins PB1 and PB7 respectively, while data is being transferred from PORTC to PORTD.

```
.INCLUDE "M32DEF.INC"
.ORG 0x0 ;location for reset
JMP MAIN ;bypass interrupt vector table
.ORG 0x12 ;ISR location for Timer1 overflow
JMP T1_OV_ISR ;go to an address with more space
.ORG 0x16 ;ISR location for Timer0 overflow
JMP T0_OV_ISR ;go to an address with more space
;----main program for initialization and keeping CPU busy
.ORG 0x100
MAIN: LDI R20,HIGH(RAMEND)
OUT SPH,R20
LDI R20,LOW(RAMEND)
OUT SPL,R20 ;initialize stack point
SBI DDRB,1 ;PB1 as an output
SBI DDRB,7 ;PB7 as an output
LDI R20,(1<<TOIE0)|(1<<TOIE1)
OUT TIMSK,R20 ;enable Timer0 overflow interrupt
SEI ;set I (enable interrupts globally)
LDI R20,-160 ;value for 20  $\mu$ s +
OUT TCNT0,R20 ;load Timer0 with -160
LDI R20,0x01
OUT TCCR0,R20 ;Normal mode, int clk, no prescaler
LDI R20,HIGH(-640) ;the high byte
OUT TCNT1H,R20 ;load Timer1 high byte
```

مثال:

می خواهیم از تایمر صفر و یک استفاده بکنیم به صورت همزمان و یکسری موج های مربعی روی پین شماره 1 و 7 پورت B داشته باشیم و در حین اینکه داریم موج های مربعی رو داریم تولید میکنیم اون قضیه خوندن از پورت C و نمایش روی پورت D هم داشته باشیم

چون میخوایم از تایمر صفر و یک به صورت همزمان استفاده بکنیم میایم و قسمت وکتور تیبل های اینا رو به صورت موثر بالا کد تنظیم میکنیم

توی main:

ابتدا استک رو مقدار دهی میکنیم

و بعد ادامه کد...

Example cnt.

```
LDI    R20,LOW(-640)    ;the low byte
OUT     TCNT1L,R20      ;load Timer1 low byte
LDI     R20,0x00
OUT     TCCR1A,R20      ;Normal mode
LDI     R20,0x01
OUT     TCCR1B,R20      ;internal clk, no prescaler
LDI     R20,0x00
OUT     DDRC,R20        ;make PORTC input
LDI     R20,0xFF
OUT     DDRD,R20        ;make PORTD output
;----- Infinite loop
HERE: IN    R20,PINC      ;read from PORTC
      OUT   PORTD,R20     ;and give it to PORTD
      JMP   HERE          ;keeping CPU busy waiting for interrupt
;-----ISR for Timer0 (It comes here after elapse of 20 µs time)
.ORG 0x200
T0_OV_ISR:
      LDI   R16,-160      ;value for 20 µs
      OUT   TCNT0,R16     ;load Timer0 with -160 (for next round)
      IN    R16,PORTB     ;read PORTB
      LDI   R17,0x02      ;00000010 for toggling PB1
      EOR   R16,R17
      OUT   PORTB,R16     ;toggle PB1
      RETI                ;return from interrupt
;-----ISR for Timer1 (It comes here after elapse of 80 µs time)
.ORG 0x300
T1_OV_ISR:
      LDI   R18,HIGH(-640)
      OUT   TCNT1H,R18    ;load Timer1 high byte
      LDI   R18,LOW(-640)
      OUT   TCNT1L,R18    ;load Timer1 low byte (for next round)
      IN    R18,PORTB     ;read PORTB
      LDI   R19,0x80      ;10000000 for toggling PB7
      EOR   R18,R19
      OUT   PORTB,R18     ;toggle PB7
      RETI                ;return from interrupt
```

تا قبل از حلقه بی نهایت همه کدها یکبار انجام میشه

Compare Match Timer Flag and Interrupt

Using Timer0, write a program that toggles pin PORTB.5 every 40 μ s, while at the same time transferring data from PORTC to PORTD. Assume XTAL = 1 MHz.

1/1 MHz = 1 μ s and 40 μ s/1 μ s = 40. That means we must have OCR0 = 40 - 1 = 39

```
.INCLUDE "M32DEF.INC"
.ORG 0x0    ;location for reset
    JMP     MAIN
.ORG 0x14    ;ISR location for Timer0 compare match
    JMP     TO_CM_ISR
;main program for initialization and keeping CPU busy
.ORG 0x100
MAIN: LDI     R20,HIGH(RAMEND)
      OUT     SPH,R20
      LDI     R20,LOW(RAMEND)
      OUT     SPL,R20    ;set up stack
      SBI     DDRB,5     ;PB5 as an output
      LDI     R20,(1<<OCIE0)
      OUT     TIMSK,R20  ;enable Timer0 compare match interrupt
      SEI                     ;set I (enable interrupts globally)
      LDI     R20,39
      OUT     OCR0,R20   ;load Timer0 with 39
      LDI     R20,0x09
      OUT     TCCR0,R20  ;start Timer0, CTC mode, int clk, no prescaler
      LDI     R20,0x00
      OUT     DDRC,R20   ;make PORTC input
      LDI     R20,0xFF
      OUT     DDRD,R20   ;make PORTD output
;----- Infinite loop
HERE: IN      R20,PINC    ;read from PORTC
      OUT     PORTD,R20  ;and send it to PORTD
      JMP     HERE       ;keeping CPU busy waiting for interrupt
;-----ISR for Timer0 (it is executed every 40  $\mu$ s)
TO_CM_ISR:
      IN      R16,PORTB  ;read PORTB
      LDI     R17,0x20    ;00100000 for toggling PB5
      EOR     R16,R17
      OUT     PORTB,R16  ;toggle PB5
      RETI                ;return from interrupt
```

مثال:

می‌خواهیم برنامه‌ای بنویسیم که روی پورت B پایه شماره 5 مرتب صفر و یک بکنیم با یک تاخیر 40 میکروثانیه و همزمان از پورت C ورودی بگیریم و روی پورت D نمایش بدیم و فرض میکنیم فرکانس کاری ما هم 1 مگاهرتز است

Programming External Hardware Interrupts

The number of external hardware interrupt interrupts varies in different AVR. The ATmega32 has three external hardware interrupts: pins PD2 (PORTD.2), PD3 (PORTD.3), and PB2 (PORTB.2), designated as INT0, INT1, and INT2, respectively. Upon activation of these pins, the AVR is interrupted in whatever it is doing and jumps to the vector table to perform the interrupt service routine.

the interrupt vector table locations \$2, \$4, and \$6 are set aside for INT0, INT1, and INT2, respectively. The hardware interrupts must be enabled before they can take effect. This is done using the INTx bit located in the GICR register.

For example, the following instructions enable INT0:

```
LDI    R20, 0x40
OUT     GICR, R20
```



وقفه های سخت افزاری خارجی:

به صورت خاص ATmega32 سه وقفه خارجی در اختیار ما قرار میدهد که این ها از طریق پین شماره 2 و 3 روی پورت D و پین شماره 2 روی پورت B در اختیار ما قرار میدهد و این ها رو تحت وقفه های INT0 , INT1 , INT2 می شناسیم و با فعال شدن این ها میاد و یک سری سرویس روتین های خاصی شروع میکنه به اجرا شدن

ادرس مرتبط با این وقفه ها برای INT0 , INT1 , INT2 به ترتیب برابر با 6 , 4 , 2 است با توجه به فضایی که اینجا در اختیار داریم فقط ما ادرس اون ساب روتین ها رو توی این فضا قرار میدیم و با فعال شدن این وقفه ها می ریم سراغ اون ادرس ها و از اون ها هدایت می شیم به سمت اون سرویس روتین هایی که قراره به صورت مفصل یکسری کارهایی رو انجام بدیم

برای استفاده از این وقفه ها باید بیایم و اون هارو فعال بکنیم بیت های فعال ساز این وقفه ها در یک رجیستری به نام GICR قرار داره اینجا یکسری INT داریم برای هر تایمر متناسب با اون تایمر و برای اینکه بیایم از این وقفه ها استفاده بکنیم باید بیایم این هارو توی این رجیستر فعال بکنیم یک نمونه دستوراتی که می تونیم این کارو انجام بدیم اینجا آورده شده: مثلا میخوایم INT0 رو فعال بکنیم برای این کار کافیه که 40 هگز روی توی این رجیستر بریزیم این کار باعث میشه که این بیت ما فعال بشه و وقفه صفر رو بتونیم استفاده بکنیم

General Interrupt Control Register (GICR)

D7			D0				
INT1	INT0	INT2	-	-	-	IVSEL	IVCE

- INT0** External Interrupt Request 0 Enable
 = 0 Disables external interrupt 0
 = 1 Enables external interrupt 0
- INT1** External Interrupt Request 1 Enable
 = 0 Disables external interrupt 1
 = 1 Enables external interrupt 1
- INT2** External Interrupt Request 2 Enable
 = 0 Disables external interrupt 2
 = 1 Enables external interrupt 2

These bits, along with the I bit, must be set high for an interrupt to be responded to.

The INT0 is a low-level-triggered interrupt by default, which means, when a low signal is applied to pin PD2 (PORTD.2), the controller will be interrupted and jump to location \$0002 in the vector table to service the ISR.

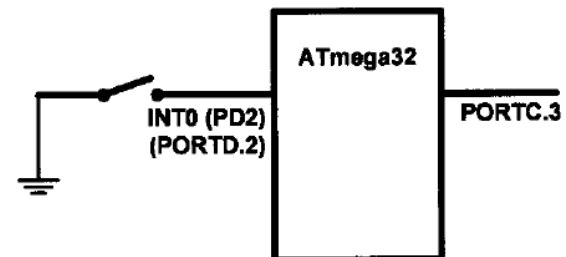
نکته: این بیت ها رو باید در کنار بیت | استفاده بکنیم و زمانی این بیت ها تاثیرگذار خواهند بود که اون بیت | فعال باشه

به صورت پیش فرض زمانی که میخوایم از این وقفه ها استفاده بکنیم و یا میکرو رو برای بار اول می خوایم استفاده بکنیم و هیچ تنظیمی برای اون انجام ندادم اینا به صورت فعال به سطح هستند به صورت پیش فرض ینی وقتی که داریم برای تایمر صفر صحبت میکنیم این به صورت پیش فرض حساس به سطح پایین هستش ینی اگر اون سطح سیگنال ما توی قسمت low باشه این فعال میشه ینی اگر منبع وقفه به پورت D پایه شماره 2 وصل باشه این زمانی فعال میشه که low باشه و وقتی این در قسمت low قرار میگیره اینو فعال میکنه و پرشی صورت میگیره به اون وکتور تیبیل به اون قسمتی که مرتبط با این وقفه هستش و از اونجا هدایت می شیم به سمت ISR مرتبط با این وقفه

Example

Assume that the INT0 pin is connected to a switch that is normally high. Write a program that toggles PORTC.3 whenever the INT0 pin goes low.

```
.INCLUDE "M32DEF.INC"
.ORG 0                      ;location for reset
    JMP     MAIN
.ORG 0x02                   ;vector location for external interrupt 0
    JMP     EX0_ISR
MAIN: LDI     R20,HIGH(RAMEND)
    OUT     SPH,R20
    LDI     R20,LOW(RAMEND)
    OUT     SPL,R20          ;initialize stack
    SBI     DDRC,3           ;PORTC.3 = output
    SBI     PORTD,2          ;pull-up activated
    LDI     R20,1<<INT0     ;enable INT0
    OUT     GICR,R20
    SEI                     ;enable interrupts
HERE: JMP     HERE           ;stay here forever
EX0_ISR:
    IN      R21,PINC         ;read PINC
    LDI     R22,0x08         ;00001000
    EOR     R21,R22
    OUT     PORTC,R21
    RETI
```



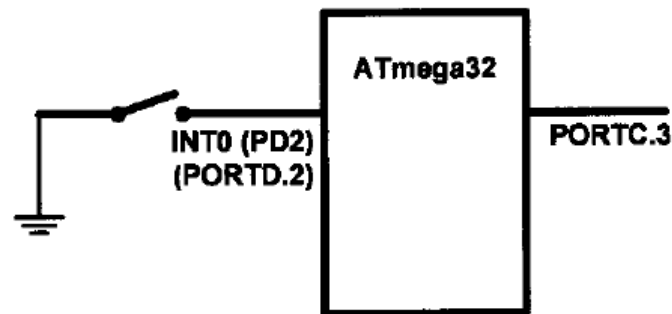
مثال:

اینجا به پورت D پایه شماره 2 یک منبعی رو با یک سویچی وصل کردیم به این وقفه و می خوایم برنامه ای بنویسیم که از این پین شماره 2 به عنوان یک منبع خارجی وقفه استفاده بکنیم

اینجا حلقه بی نهایت ما به صورت مختصر به صورت JMP است و این حلقه بی نهایت مرتب اجرا میشه تا اینکه یک وقفه خارجی به این پین وارد بشه ینی سطح سیگنال روی این پایه در یک وضعیت low قرار بگیره

Example _{cnt.}

the microcontroller is looping continuously in the HERE loop. Whenever the switch on INT0 (pin PD2) is activated, the microcontroller gets out of the loop and jumps to vector location \$0002. The ISR for INT0 toggles the PC0. If, by the time it executes the RETI instruction, the INT0 pin is still low, the microcontroller initiates the interrupt again. Therefore, if we want the ISR to be executed once, the INT0 pin must be brought back to high before RETI is executed, or we should make the interrupt edge-triggered.



مثال:

توی مثال صفحه قبل میکروکنترلر میاد توی یک حلقه که به صورت بی نهایت داره تکرار میشه تا یک وقفه ای از سمت این منبع خارجی صادر بشه و با فعال شدن این سویچ این کار انجام میشه و با فعال شدن این سویچ روند عادی از این حلقه خارج میشه و پرش می کنیم به اون ادرسی که ادرس سرویس روتین هستش ینی ادرس 2 هگز و بعد اونجا به سرویس روتین و بعد ادامه کار..

در زمانی که سرویس روتین انجام میشه و میخواد برگرده و دستور RETI رو انجام بدیم اگر همچنان پین 2 پورت D توی وضعیت low قرار داشته باشه دوباره میکروکنترلر میاد و وقفه رو از اول شروع میکنه به اجرا کردن بنابراین برای این که فقط یکبار این سرویس روتین مرتبط با این وقفه انجام بشه باید دقت داشته باشیم که این وضعیت روی وقفه صفر رو به حالت نرمال برگردونیم تا دوباره مرتب اون سرویس روتین مرتبط با اون وقفه دوباره اجرا نشه ینی قبل از اینکه بخوایم برگردیم به اون دستوری که از اون پرش کردیم این INTO به اون حالت نرمال خودش باید برگرده تا دوباره اون دستور ما اجرا نشه این یک راه حل است یا اینکه از این راه بریم:

اینکه ما بیایم اجرای این وقفه رو حساس بکنیم به لبه سیگنال ینی حساس به لبه باشه

Edge-Triggered vs. Level-Triggered Interrupts

There are two types of activation for the external hardware interrupts: (1) level triggered, and (2) edge triggered. INT2 is only edge triggered, while INT0 and INT1 can be level or edge triggered.

As stated before, upon reset INT0 and INT1 are low-level-triggered interrupts. The bits of the MCUCR register indicate the trigger options of INT0 and INT1.



(Interrupt Sense Control bits) These bits define the level or edge on the external interrupt.

چجوری یک وقفه می تونه فعال بشه؟





معمولا وقفه هایی که به عنوان وقفه خارجی داریم می تونن حساس به سطح باشن یا حساس به لبه که البته برای وقفه خارجی شماره دو اون فقط میتونه حساس به لبه باشه ولی وقفه های صفر و یک این ها می تونن هم حساس به سطح باشن و هم حساس به لبه

برای اینکه بیایم و تنظیم بکنیم که اون وقفه ما حساس به سطح باشه یا حساس به لبه یک رجیستری داریم تحت عنوان MCUCR توی این یکسری بیت های کنترلی وجود داره که توی این رجیستر دوتا مرتبط با وقفه صفر داره و دوتا مرتبط با وقفه یک





با تنظیم کردن این بیت ها می تونیم وقفه هایی داشته باشیم بسته به مقداری که اینا می گیرین می تونن حساس به سطح باشن یا حساس به لبه

MCU Control Register (MCUCR)

ISC01, ISC00 (Interrupt Sense Control bits) These bits define the level or edge on the external INT0 pin that activates the interrupt, as shown in the following table:

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

ISC11, ISC10 These bits define the level or edge that activates the INT1 pin.

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

برای تایمر صفر ما ISC00 , ISC01 رو داریم که 4 تا حالت می تونه بگیره که توی جدول نوشته:

- اگر 00 باشه حساس به صفر است
- اگر 01 داشته باشیم با تغییر از یک سطح به سطح دیگه این وقفه فعال میشه
- 10 --> حساس به لبه پایین رونده است
- 11 --> حساس به بالا رونده است

برای تایمر یک ما ISC10 , ISC11 رو داریم که 4 حالت داره:

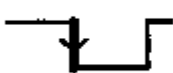

- 00 --> حساس به سطح پایین
- 01 --> هر گونه تغییر از سطح بالا به پایین
- 10 --> حساس به لبه پایین رونده
- 11 --> حساس به لبه بالا رونده

MCU Control and Status Register (MCUCSR)

The ISC2 bit of the MCUCSR register defines whether INT2 activates in the falling edge or the rising edge, edge-triggered. Upon reset ISC2 is 0, meaning that the external hardware interrupt of INT2 is falling edge triggered.



ISC2 This bit defines whether the INT2 interrupt activates on the falling edge or the rising edge.

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

وقفه شماره 2:

یک رجیستری داریم تحت عنوان MCUCSR و یک بیتی داریم تحت عنوان ISC2 که این مشخص می‌کند که الان این وقفه خارجی ما حساس به لبه پایین رونده باشد یا حساس به لبه بالا رونده باشد

Example

Show the instructions to (a) make INT0 falling edge triggered, (b) make INT1 triggered on any change, and (c) make INT2 rising edge triggered.

```
(a)  LDI    R20, 0x02
      OUT    MCUCR, R20

(b)  LDI    R20, 1<<ISC10      ;R20 = 0x04
      OUT    MCUCR, R20

(c)  LDI    R20, 1<<ISC2       ;R20 = 0x40
      OUT    MCUCSR, R20
```

مثال:

الف: وقفه صفر ما حساس به لبه پایین رونده باشد

ب: وقفه یک رو به گونه ای تنظیم بکنیم که حساس به هر گونه تغییری از سطح بالا به پایین یا برعکس باشد

ج: وقفه دو حساس به لبه بالا رونده باشد

Example

Assume that the INT0 pin is connected to a switch that is normally high. Write a program so that whenever INT0 goes low, it toggles PORTC.3 only once.

```
.INCLUDE "M32DEF.INC"
.ORG 0                                ;location for reset
    JMP    MAIN
.ORG 0x02                             ;location for external interrupt 0
    JMP    EX0_ISR
MAIN: LDI    R20,HIGH(RAMEND)
    OUT    SPH,R20
    LDI    R20,LOW(RAMEND)
    OUT    SPL,R20                    ;initialize stack
    LDI    R20,0x2                    ;make INT0 falling edge triggered
    OUT    MCUCR,R20
    SBI    DDRC,3                     ;PORTC.3 = output
    SBI    PORTD,2                    ;pull-up activated
    LDI    R20,1<<INT0               ;enable INT0
    OUT    GICR,R20
    SEI                                ;enable interrupts
HERE: JMP    HERE
EX0_ISR:
    IN     R21,PORTC
    LDI    R22,0x08                   ;00001000 for toggling PC3
    EOR    R21,R22
    OUT    PORTC,R21
    RETI
```

مثال:

توی این مثال میخوایم از وقفه خارجی شماره صفر استفاده بکنیم و زمانی که اون سگینال که روی پین مرتبط با این وقفه از حالت بالا به پایین میاد ینی لبه پایین رونده باشه و روی پین شماره 3 پورت C اونو تغییر وضعیت بدیم

Sampling the Edge-Triggered and the Level-Triggered Interrupts

The edge interrupt (the falling edge, the rising edge, or the change level) is latched by the AVR and is held by the INTFx bits of the GIFR register. This means that when an external interrupt is in an edge-triggered mode (falling edge, rising edge, or change level), upon triggering an interrupt request, the related INTFx flag becomes set. If the interrupt is active (the INTx bit is set and the I-bit in SREG is one), the AVR will jump to the corresponding interrupt vector location and the INTFx flag will be cleared automatically, otherwise, the flag remains set. The flag can be cleared by writing a one to it. For example, the INTF1 flag can be cleared using the following instructions:

```
LDI    R20, (1<<INTF1)    ;R20 = 0x80
OUT    GIER, R20           ;clear the INTF1 flag
```



توی AVR وقفه ما می تونه حساس به سطح یا حساس به لبه باشه
و زمانی ک حساس به لبه باشه AVR میاد latch میکنه مقدار این فعال شدن این وقفه ها رو و این
به معنی که زمانی که یک وقفه خارجی رخ میده و حساس به لبه هستش با فعال شدن اون وقفه یک
پرچمی داریم تحت عنوان INTF که این میاد و ست میشه متناسب با هر کدوم از اون وقفه های
خارجی و اگر INT فعال باشه ینی وقفه مرتبط با وقفه خارجی فعال باشه و بیت I هم فعال بشه
AVR پرش میکنه به اون ادرسی که توی وکتورتیبل وجود داره و از اونجا به سرویس روتین و
توی این زمان بیت INTF ما به صورت خودکار پاک میشه و اون سرویس روتین انجام میشه و
کارهای دیگه ولی اگر شرایط برقرار نباشه این همین جوری ست باقی می مونه و ما می تونیم به
صورت نرم افزاری برنامه بنویسیم که اینو پاک کنیم مثالش هم پایین نوشته

Sampling the Edge-Triggered and the Level-Triggered Interrupts

Notice that in edge-triggered interrupts (falling edge, rising edge, and change level interrupts), the pulse must last at least 1 instruction cycle to ensure that the transition is seen by the microcontroller. This means that pulses shorter than 1 machine cycle are not guaranteed to generate an interrupt.

When an external interrupt is in level-triggered mode, the interrupt is not latched, meaning that the INTFx flag remains unchanged when an interrupt occurs, and the state of the pin is read directly. As a result, when an interrupt is in level-triggered mode, the pin must be held low for a minimum time of 5 machine cycles to be recognized.

زمانی که به صورت حساس به لبه می‌خواهیم از وقفه‌ها استفاده بکنیم نیاز داریم که حداقل یک سیکل ساعت اون سیگنال ما فعال بشه که بتونیم اون وقفه رو فعال حساب بکنیم اما زمانی که حساس به سطح داریم از وقفه‌ها استفاده میکنیم با توجه به این که این latch همیشه و به صورت مستقیم منتقل میشه به اون قسمتی که وقفه‌ها رو فعال میکنه ما اینجا باید حداقل اینو برای 5 سیکل ماشین توی اون وضعیت نگه داریم تا اون وقفه فعال بشه

Interrupt Priority in the AVR

- What happens when two interrupts are activated at the same time?
 - Priority
 - 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 lower address has a higher priority

چه اتفاقی می افتد وقتی که دوتا وقفه با هم در یک سیستم فعال بشن؟
AVR برای کنترل این همزمانی در وقفه ها از یک مکانیزمی تحت عنوان اولویت استفاده میکنه و نحوی کار به این صورت است که اگر چند وقفه همزمان رخ بده اون وقفه ای اولویت بالاتری داره ابتدا سرویس داده میشه

نحوی اولویت بندی هم براساس اون محلی هستش که این وقفه ها در اون وکتور تیبل قرار داره و هر چه اون وقفه ای که ما میخوایم ازش استفاده بکنیم در ادرس های پایین تری قرار داشته باشه اولویت بالاتری داره مثلا وقفه ریست اولویت بالاتری نسبت به وقفه های تایمرها داره

Interrupt inside an interrupt

What happens if the AVR is executing an ISR belonging to an interrupt and another interrupt is activated? When the AVR begins to execute an ISR, it disables the I bit of the SREG register, causing all the interrupts to be disabled, and no other interrupt occurs while serving the interrupt. When the RETI instruction is executed, the AVR enables the I bit, causing the other interrupts to be served. If you want another interrupt (with any priority) to be served while the current interrupt is being served you can set the I bit using the SEI instruction. But do it with care. For example, in a low-level-triggered external interrupt, enabling the I bit while the pin is still active will cause the ISR to be reentered infinitely, causing the stack to overflow with unpredictable consequences.

نکته:

به صورت اتوماتیک خود AVR زمانی که می خواست بره و سرویس روتین مرتبط با اون وقفه رو اجرا بکنه می اومد بیت I به صورت موقت کلیر می کرد تا وقفه ها غیرفعال بشن و زمانی که کارش تموم شد توی اون سرویس روتین توسط دستور RETI این بیت مجدداً یک میشد تا بتونه وقفه های جدیدی رو دریافت بکنه

اگر ما بخوایم در خود سرویس روتین هم پذیرای یک وقفه دیگر هم باشیم : به صورت دستی باید بیایم بیت I ست بکنیم توی این حالت وقفه ها فعال میشن و می تونیم وقفه های جدیدی رو دریافت بکنیم --> اشکالاتی که به صورت نمونه ممکنه پیش بیاد اینه که مثلاً اگر از وقفه ها به صورت سطح یا low استفاده بکنیم اینجا اگر بیتی رو به صورت دستی فعال بکنیم ینی وقفه ها در خود سرویس روتین فعال باشه و زمانی که میخواد اینو شروع بکنه و با توجه به اینکه ممکنه اون منبع وقفه ما همچنان در سطح پایین وجود داشته باشه دوباره وقفه رو تکرار میکنه و این ممکنه در یک حلقه بی نهایتی گیر بکنه پس اگر قراره درون یک وقفه ای یک وقفه ای داشته باشیم باید اینو به صورت دستی فعال بکنیم و خودمون مراقب کنترل هایی در این زمینه باشیم که اتفاق های ناخواسته ای در این زمینه رخ نده

Context Saving in Task Switching

In multitasking systems, such as multitasking real-time operating systems (RTOS), the CPU serves one task (job or process) at a time and then moves to the next one. In simple systems, the tasks can be organized as the interrupt service routine. For example, the program does two different tasks:

- (1) copying the contents of PORTC to PORTD,
- (2) toggling PORTC.2 every 5 μ s

While writing a program for a multitasking system, we should manage the resources carefully so that the tasks do not conflict with each other.

اگر یک سیستمی داریم که قراره چند وظیفه رو با هم انجام بده این ها معمولا مالتی تسکینگ دارن یکی از مکانیزیم هایی که میشه این چند وظیفه ای رو پیاده سازی کرد استفاده از سرویس روتین ها به روش های مختلفی هستش

توی یک سیستم خیلی ساده که ما چند تسک داریم این کارو میتونیم در خود اون سرویس روتین هایی که داریم بیایم پیاده سازی بکنیم مثال:

فرض کنید یک برنامه ای داریم که قراره دوتا کار انجام بده یکی از پورت C بخونه و روی پورت D نمایش بده و کار دوم هم اینه که پورت C پایه شماره 2 رو بیاد و با یک فرکانسی تغییر وضعیت بده

توی برنامه ای که میخوایم چند تسک داشته باشیم باید از منابع به نحوی استفاده بکنیم که با هم تداخلی نداشته باشند --> نکته

Example

consider a system that should perform the following tasks: (1) increasing the contents of PORTC continuously, and (2) increasing the content of PORTD once every 5 μ s. Read the following program. Does it work?

```
.INCLUDE "M32DEF.INC"
.ORG 0x0    ;location for reset
    JMP     MAIN
.ORG 0x14   ;location for Timer0 compare match
    JMP     T0_CM_ISR
;-main program for initialization and keeping CPU busy
.ORG 0x100
MAIN: LDI    R20,HIGH(RAMEND)
      OUT    SPH,R20
      LDI    R20,LOW(RAMEND)
      OUT    SPL,R20    ;set up stack
      SBI    DDRB,5     ;PB5 as an output
      LDI    R20,(1<<OCIE0)
      OUT    TIMSK,R20  ;enable Timer0 compare match interrupt
      SEI                     ;set I (enable interrupts globally)
      LDI    R20,160
      OUT    OCR0,R20   ;load Timer0 with 160
      LDI    R20,0x09
      OUT    TCCR0,R20  ;CTC mode, int clk, no prescaler
      LDI    R20,0xFF
      OUT    DDRC,R20   ;make PORTC output
      OUT    DDRD,R20   ;make PORTD output
      LDI    R20, 0
HERE: OUT    PORTC,R20   ;PORTC = R20
      INC    R20
      JMP     HERE      ;keeping CPU busy waiting for interrupt
;-----ISR for Timer0
T0_CM_ISR:
      IN     R20,PIND
      INC    R20
      OUT    PORTD,R20  ;PORTD = R20
      RETI             ;return from interrupt
```

مثال:

توی این سیستم این دوتا کار انجام بشه:
مقدار پورت C مرتبا یک واحد افزایش بده
و پورت D هم بخونه و با یک دوره خاصی بیاد و اون رو هم افزایش بده

این برنامه عملاً داره دوتا کار انجام میده ولی یک اشکالی توی این برنامه وجود داره که اون همزمانی که ما میخوایم رو در اختیار قرار نمیده: از R20 در تابع اصلی و در سرویس روتین استفاده شده

Example cnt.

consider a system that should perform the following tasks: (1) increasing the contents of PORTC continuously, and (2) increasing the content of PORTD once every 5 μ s. Read the following program. Does it work?

The tasks do not work properly, since they have resource conflict and they interfere with each other. R20 is used and changed by both tasks, which causes the program not to work properly. For example, consider the following scenario: The content of R20 increases in the main program, at first becoming 0, then 1, and so on. When the timer interrupt occurs, R20 is 95, and PORTC is 95 as well. In the ISR, the R20 is loaded with the content of PORTD, which is 0. So, when it goes back to the main program, the content of R20 is 1 and PORTC will be loaded by 2. But if the program worked properly, PORTC would be loaded with 96.

We can solve such problems in the following two ways:

(1) Using different registers for different tasks. In the program discussed above, if we use different registers in the main program and in the ISR, the program will work properly.

با توجه به اینکه میاد و محتوای پورت C رو در R20 قرار میدی و اونو مرتب افزایش میدی: اگر اولش R20 صفر بوده توی اون حلقه R20 مرتب افزایش پیدا میکنه و زمانی که این R20 داره افزایش پیدا میکنه و این حلقه اجرا میشه ممکنه تایمر ما وقفه خودش رو صادر بکنه و بره سراغ این ساب روتینی که برای این تایمر داشته

توی این ساب روتین اولین کاری که انجام میشه اینه که محتوای پورت D رو میخونه و داخل R20 می ریزه و بعد افزایش میدی و این باعث میشه اون مقدارهای قبلی که توی R20 داشتیم عملاً نابود بشه و اون چیزی که میخواستیم رو برای ما انجام نده
برای رفع این مشکل دوتا راه حل وجود داره:

1- توی تابع اصلی و توی ساب روتین از رجیسترهای متفاوتی استفاده بکنیم

2- Context اونو بیایم و سیو بکنیم:

ما یک قسمت اصلی برنامه داریم و یک قسمت ساب روتین
توی قسمت اصلی برنامه ما یه حلقه یکسری اتفاق هایی داره می افته و وسط این اتفاقات یک دفعه یک تایمر وقفه صادر می کنه و میخواد بره توی ساب روتین و برای اینکه این اتفاقاتی که تا اینجا افتاده رو داشته باشیم و بعد بتونیم اونو ادامه بدیم میایم و اون ها رو در استک ذخیره می کنه: خب پس توی ساب روتین قبل از اینکه بخوایم از R20 استفاده بکنیم میایم و محتوای قبلی اون رو روی استک ذخیره میکنیم و بعد با اون کار میکنیم و زمانی که خواستیم از این ساب روتین خارج بشیم دوباره اونو پاپ میکنیم در اون رجیستر تا همه چیز مثل قبل بشه

توی مالتی تسکینگ می تونیم از این روش بریم

Example cnt.

```
.INCLUDE "M32DEF.INC"
.ORG 0x0                ;location for reset
    JMP    MAIN

.ORG 0x14                ;location for Timer0 compare match
    JMP    T0_CM_ISR

;-----main program for initialization and keeping CPU busy
.ORG 0x100
MAIN: LDI    R20,HIGH(RAMEND)
      OUT    SPH,R20
      LDI    R20,LOW(RAMEND)
      OUT    SPL,R20    ;set up stack
      SBI    DDRB,5     ;PB5 as an output
      LDI    R20,(1<<OCIE0)
      OUT    TIMSK,R20  ;enable Timer0 compare match interrupt
      SEI                    ;set I (enable interrupts globally)
      LDI    R20,160
      OUT    OCR0,R20   ;load Timer0 with 160
      LDI    R20,0x09
      OUT    TCCR0,R20  ;start timer,CTC mode,int clk,no prescaler
      LDI    R20,0xFF
      OUT    DDRC,R20   ;make PORTC output
      OUT    DDRD,R20   ;make PORTD output
      LDI    R20, 0
HERE: OUT    PORTC,R20   ;PORTC = R20
      INC    R20
      JMP    HERE       ;keeping CPU busy waiting for int.

;-----ISR for Timer0
T0_CM_ISR:
    IN      R21,PIND
    INC     R21
    OUT     PORTD,R21    ;toggle PB5
    RETI                ;return from interrupt
```

Example cnt. (Context Saving)

(2) Context saving. In big programs we might not have enough registers to use separate registers for different tasks. In these cases, we can save the contents of registers on the stack before execution of each task, and reload the registers at the end of the task. This saving of the CPU contents before switching to a new task is called *context saving* (or *context switching*). See the following program:

```
.INCLUDE "M32DEF.INC"
.ORG 0x0    ;location for reset
    JMP    MAIN
.ORG 0x14    ;location for Timer0 compare match
    JMP    T0_CM_ISR
;main program for initialization and keeping CPU busy
.ORG 0x100
MAIN: LDI    R20,HIGH(RAMEND)
      OUT    SPH,R20
      LDI    R20,LOW(RAMEND)
      OUT    SPL,R20    ;set up stack
      SBI    DDRB,5      ;PB5 as an output
      LDI    R20,(1<<OCIE0)
      OUT    TIMSK,R20   ;enable Timer0 compare match interrupt
      SEI                      ;set I (enable interrupts globally)
      LDI    R20,160
      OUT    OCR0,R20    ;load Timer0 with 160
      LDI    R20,0x09
      OUT    TCCR0,R20   ;CTC mode, int clk, no prescaler
      LDI    R20,0xFF
      OUT    DDRC,R20    ;make PORTC output
      OUT    DDRD,R20    ;make PORTD output
      LDI    R20, 0
      HERE: OUT    PORTC,R20 ;PORTC = R20
            INC    R20
            JMP    HERE    ;keeping CPU busy waiting for interrupt
;-----ISR for Timer0
T0_CM_ISR:
      PUSH    R20        ;save R20 on stack
      IN      R20,PIND
      INC     R20
      OUT     PORTD,R20   ;toggle PB5
      POP     R20        ;restore value for R20
      RETI                ;return from interrupt
```

Saving Flags of the SREG Register

The flags of SREG are important especially when there are conditional jumps in our program. We should save the SREG register if the flags are changed in a task.

```
Sample_ISR:
    PUSH    R20
    IN      R20,SREG
    PUSH    R20
    ...
    POP     R20
    OUT     SREG,R20
    POP     R20
    RETI
```

یکی از پیچیدگی‌هایی که این روش Context داره اینه که توی خیلی از برنامه‌ها محتوای رجیسترهای متفاوت روی اجرا برنامه تاثیرگذار خواهد بود و مجبوریم اون هارو هم ذخیره بکنیم و توی خیلی از این برنامه‌ها ممکنه این رجیستر وضعیت محتوایی رو داشته باشه که ما در ادامه اجرای برنامه به اون نیاز داریم بنابراین مجبوریم که این رجیسترها رو هم ذخیره بکنیم تا بتونیم زمانی که برنامه رو از سر میگیریم محتوایی رو که در اختیار داریم به گونه‌ای صحیح استفاده بکنیم

مثال: یک تغییری در خطوط قبلی برنامه در رجیستر وضعیت دادیم که اگر اینو سیو نکنیم ادامه برنامه ممکنه روال متفاوت تری نسبت به اون چیزی که قبل از پرش به ساب روتین داشتیم ادامه بده

پس این موارد برای برنامه‌های مالتی تسکینگ باید رعایت بشه

Interrupt programming in C

In C language there is no instruction to manage the interrupts. So, in WinAVR the following have been added to manage the interrupts:

1. **Interrupt include file:** We should include the interrupt header file if we want to use interrupts in our program. Use the following instruction:
`#include <avr\interrupt.h>`
2. **cli() and sei():** In Assembly, the CLI and SEI instructions clear and set the I bit of the SREG register, respectively. In WinAVR, the `cli()` and `sei()` macros do the same tasks.
3. **Defining ISR:** To write an ISR (interrupt service routine) for an interrupt we use the following structure:

```
ISR(interrupt vector name)  
{  
    //our program  
}
```

وقفه ها در زبان سی:

سه امکان کلی وجود داره برای اینکه بتونیم وقفه ها رو در زبان سی استفاده بکنیم کافیه فایل هدر مرتبط با وقفه ها رو اضافه بکنیم به اون فایل برنامه اینجا دستوراتی مثل `sei` , `cli` داریم

قالب کلی وقفه ینی نوشتن ساب روتین به صورت شکل شماره 3 است

Interrupt programming in C

Interrupt Vector Name for the ATmega32/ATmega16 in WinAVR

Interrupt	Vector Name in WinAVR
External Interrupt request 0	INT0_vect
External Interrupt request 1	INT1_vect
External Interrupt request 2	INT2_vect
Time/Counter2 Compare Match	TIMER2_COMP_vect
Time/Counter2 Overflow	TIMER2_OVF_vect
Time/Counter1 Capture Event	TIMER1_CAPT_vect
Time/Counter1 Compare Match A	TIMER1_COMPA_vect
Time/Counter1 Compare Match B	TIMER1_COMPB_vect
Time/Counter1 Overflow	TIMER1_OVF_vect
Time/Counter0 Compare Match	TIMER0_COMP_vect
Time/Counter0 Overflow	TIMER0_OVF_vect
SPI Transfer complete	SPI_STC_vect
USART, Receive complete	USART0_RX_vect
USART, Data Register Empty	USART0_UDRE_vect
USART, Transmit Complete	USART0_TX_vect
ADC Conversion complete	ADC_vect
EEPROM ready	EE_RDY_vect
Analog Comparator	ANALOG_COMP_vect
Two-wire Serial Interface	TWI_vect
Store Program Memory Ready	SPM_RDY_vect

اون نامی که توی قالب شماره سه انتخاب میکنیم از این جدول بردار وقفه است

Example

Using Timer0 generate a square wave on pin PORTB.5, while at the same time transferring data from PORTC to PORTD.

```
#include "avr/io.h"
#include "avr/interrupt.h"

int main ()
{
    DDRB |= 0x20;           //DDRB.5 = output

    TCNT0 = -32;            //timer value for 4  $\mu$ s
    TCCR0 = 0x01;           //Normal mode, int clk, no prescaler

    TIMSK = (1<<TOIE0);    //enable Timer0 overflow interrupt
    sei ();                 //enable interrupts

    DDRC = 0x00;            //make PORTC input
    DDRD = 0xFF;            //make PORTD output

    while (1)               //wait here
        PORTD = PINC;

    ISR (TIMER0_OVF_vect)   //ISR for Timer0 overflow
    {
        TCNT0 = -32;
        PORTB ^= 0x20;      //toggle PORTB.5
    }
}
```

میخوایم از تایمر صفر برای تولید یک موج مربعی روی پین شماره 5 پورت B استفاده بکنیم و همزمان هم داده هایی که از پورت C دریافت می کنیم روی پورت D نمایش بدیم

Example

Using Timer0 and Timer1 interrupts, generate square waves on pins PB1 and PB7 respectively, while transferring data from PORTC to PORTD.

```
#include "avr/io.h"
#include "avr/interrupt.h"

int main ( )
{
    DDRB |= 0x82;           //make DDRB.1 and DDRB.7 output
    DDRC = 0x00;           //make PORTC input
    DDRD = 0xFF;           //make PORTD output

    TCNT0 = -160;
    TCCR0 = 0x01;           //Normal mode, int clk, no prescaler

    TCNT1H = (-640)>>8;     //the high byte
    TCNT1L = (-640);        //the low byte
    TCCR1A = 0x00;
    TCCR1B = 0x01;
    TIMSK = (1<<TOIE0)|(1<<TOIE1); //enable Timers 0 and 1 int.
    sei ();                 //enable interrupts

    while (1)               //wait here
        PORTD = PINC;

    ISR (TIMER0_OVF_vect)    //ISR for Timer0 overflow
    {
        TCNT0 = -160;       //TCNT0 = -160 (reload for next round)
        PORTB ^= 0x02;      //toggle PORTB.1
    }

    ISR (TIMER1_OVF_vect)    //ISR for Timer0 overflow
    {
        TCNT1H = (-640)>>8;
        TCNT1L = (-640);    //TCNT1 = -640 (reload for next round)

        PORTB ^= 0x80;      //toggle PORTB.7
    }
}
```

مثال:

با استفاده از تایمرهای شماره صفر و یک یک موج مربعی روی پین های شماره 1 و 7 پورت B ایجاد بکنیم و همزمان از پورت C بخونیم و روی پورت D نمایش بدیم

Example

Using Timer1, write a program that toggles pin PORTB.5 every second, while at the same time transferring data from PORTC to PORTD. Assume XTAL = 8 MHz.

```
#include "avr/io.h"
#include "avr/interrupt.h"

int main ()
{
    DDRB |= 0x20;           //make DDRB.5 output

    OCR0 = 40;
    TCCR0 = 0x09;           //CTC mode, internal clk, no prescaler

    TIMSK = (1<<OCIE0);    //enable Timer0 compare match int.
    sei ();                 //enable interrupts

    DDRC = 0x00;           //make PORTC input
    DDRD = 0xFF;           //make PORTD output

    while (1)              //wait here
        PORTD = PINC;

}

ISR (TIMER0_COMP_vect)    //ISR for Timer0 compare match
{
    PORTB ^= 0x20;         //toggle PORTB.5
}
```

می‌خواهیم از تایمر یک استفاده کنیم و هر ثانیه روی پین شماره 5 پورت B یک موجی تولید کنیم و همزمان از پورت C بخونیم و روی پورت D بنویسیم

Example

Assume that the INT0 pin is connected to a switch that is normally high. Write a program that toggles PORTC.3, whenever INT0 pin goes low. Use the external interrupt in level-triggered mode.

```
#include "avr/io.h"
#include "avr/interrupt.h"

int main ()
{
    DDRC = 1<<3;           //PC3 as an output
    PORTD = 1<<2;           //pull-up activated
    GICR = (1<<INT0);       //enable external interrupt 0
    sei ();                 //enable interrupts

    while (1);              //wait here
}

ISR (INT0_vect)             //ISR for external interrupt 0
{
    PORTC ^= (1<<3);        //toggle PORTC.3
}
```

از وقفه سخت افزاری شماره صفر استفاده بکنیم فرض میکنیم به یک سوئیچی متصل است که در حالت عادی ینی در وضعیت high قرار داره و یک برنامه ای به زبان سی می خوایم بنویسیم که بین شماره 3 پورت C هر زمانی که این وقفه خارجی صادر شد ینی در وضعیت low قرار گرفت تغییر وضعیت بده

پایان

موفق و پیروز باشید