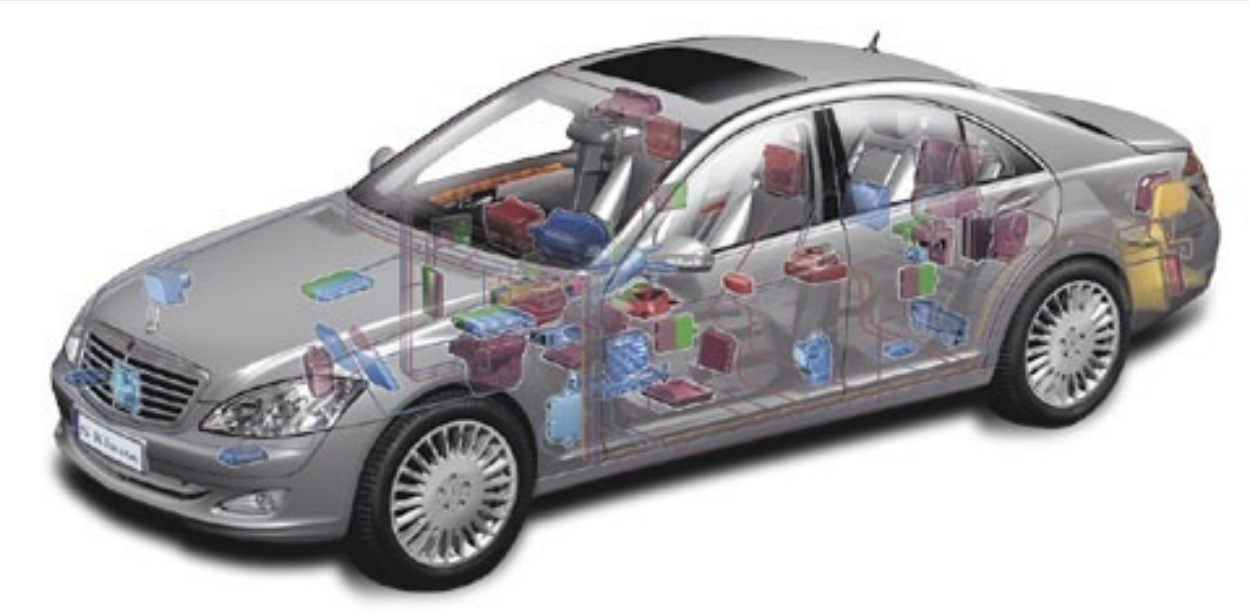
Embedded Systems and Software

Lecture 11 Interrupts

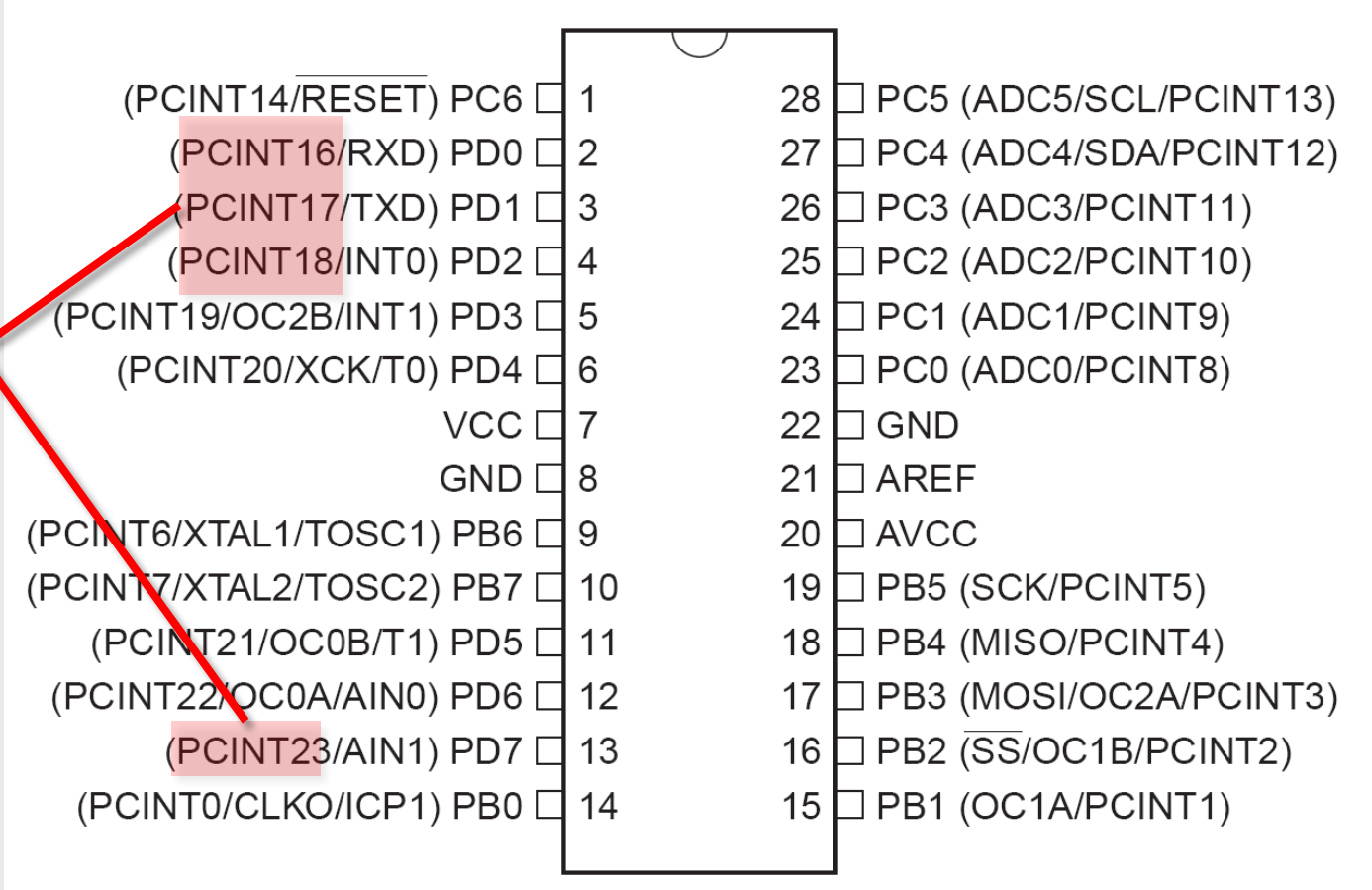


Interrupts

- **One way to think of interrupts is that they are hardware-generated functions calls**
- **Internal Hardware**
  - **When timer rolls over, then call a routine that blinks an LED**
  - **When built-in A/D converter is done converting, call a routine that manipulates the result**
- **External Hardware**
  - **When the voltage level on a I/O pin changes, call a routine that turns a motor**
  - **When the USART receives a bit, call a routine that stored the bit, etc.**
- **The routines that are called when an interrupt occurs are called *Interrupt Service Routine*s (ISRs)**

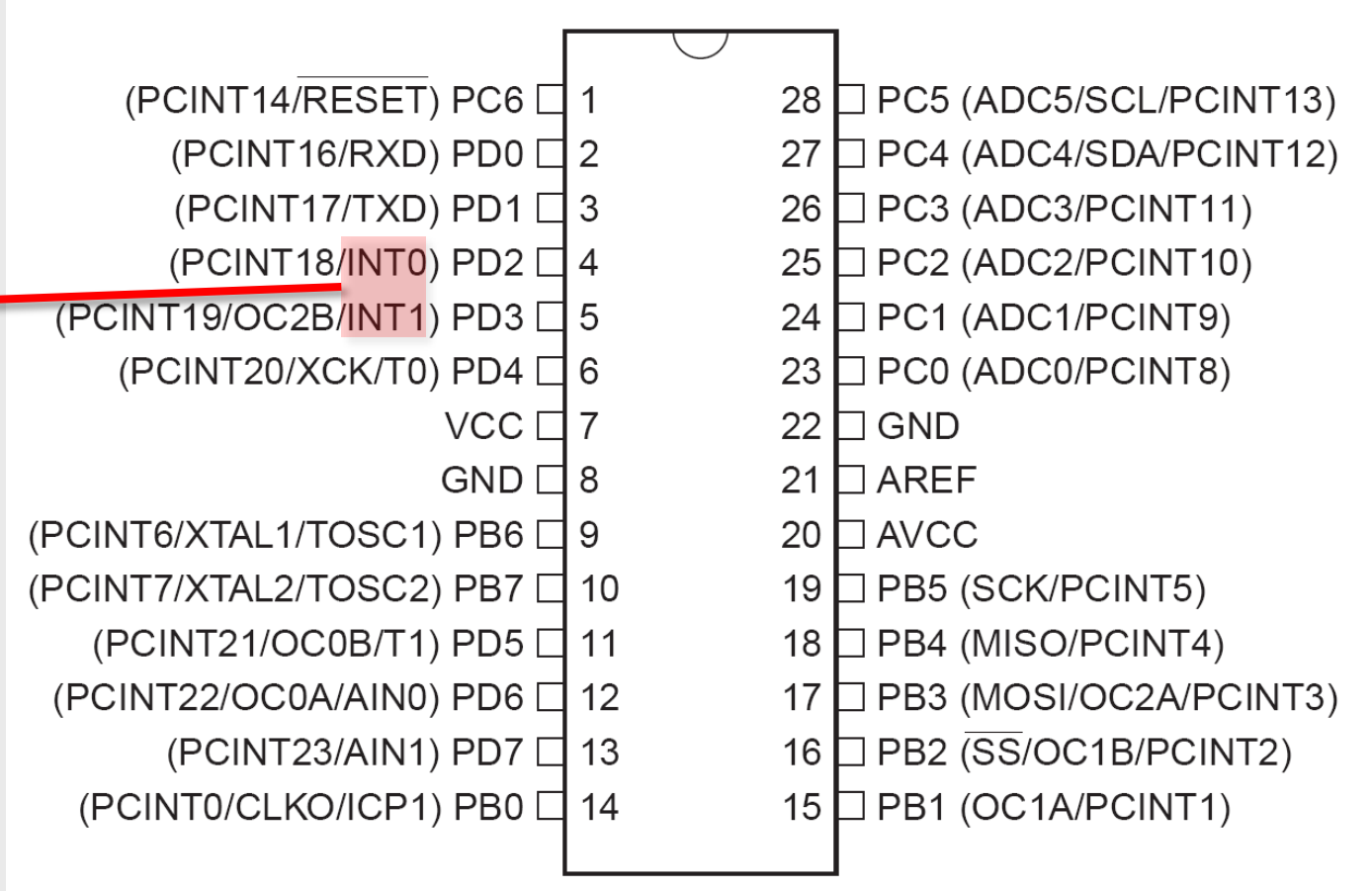
External Interrupts on ATmega88PA

"PCINT" → Pin Change Interrupt



External Interrupts on ATmega88PA

INT0 and INT1



INT0 and INT1 interrupts can be triggered by a falling or rising edge or a low level. This is set up in the External Interrupt Control Register A – EICRA. When the INT0 or INT1 interrupts are enabled and are configured as level triggered, the interrupts will trigger as long as the pin is held low. Low level interrupt on INT0 and INT1 is detected asynchronously. This implies that this interrupt can also be used for waking the part from sleep modes other than Idle mode.

Table 11-2. Reset and Interrupt Vectors in ATmega88PA

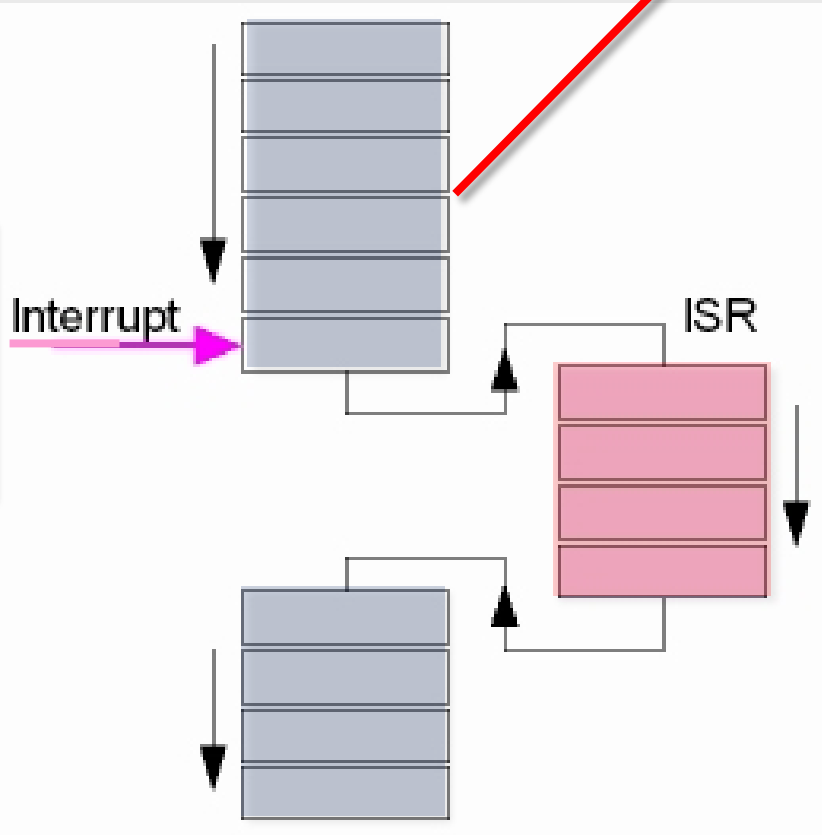
Vector No.	Program Address[(2)]	Source	Interrupt Definition
1	0x000[(1)]	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x001	INT0	External Interrupt Request 0
3	0x002	INT1	External Interrupt Request 1
4	0x003	PCINT0	Pin Change Interrupt Request 0
5	0x004	PCINT1	Pin Change Interrupt Request 1
6	0x005	PCINT2	Pin Change Interrupt Request 2
7	0x006	WDT	Watchdog Time-out Interrupt
8	0x007	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x008	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x009	TIMER2 OVF	Timer/Counter2 Overflow
11	0x00A	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x00B	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x00C	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x00D	TIMER1 OVF	Timer/Counter1 Overflow

15	0x00E	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x00F	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x010	TIMER0 OVF	Timer/Counter0 Overflow
18	0x011	SPI, STC	SPI Serial Transfer Complete
19	0x012	USART, RX	USART Rx Complete
20	0x013	USART, UDRE	USART, Data Register Empty
21	0x014	USART, TX	USART, Tx Complete
22	0x015	ADC	ADC Conversion Complete
23	0x016	EE READY	EEPROM Ready
24	0x017	ANALOG COMP	Analog Comparator
25	0x018	TWI	2-wire Serial Interface
26	0x019	SPM READY	Store Program Memory Ready

Interrupts Concepts

Main program flow

User pushes a button and triggers a pin-change interrupt



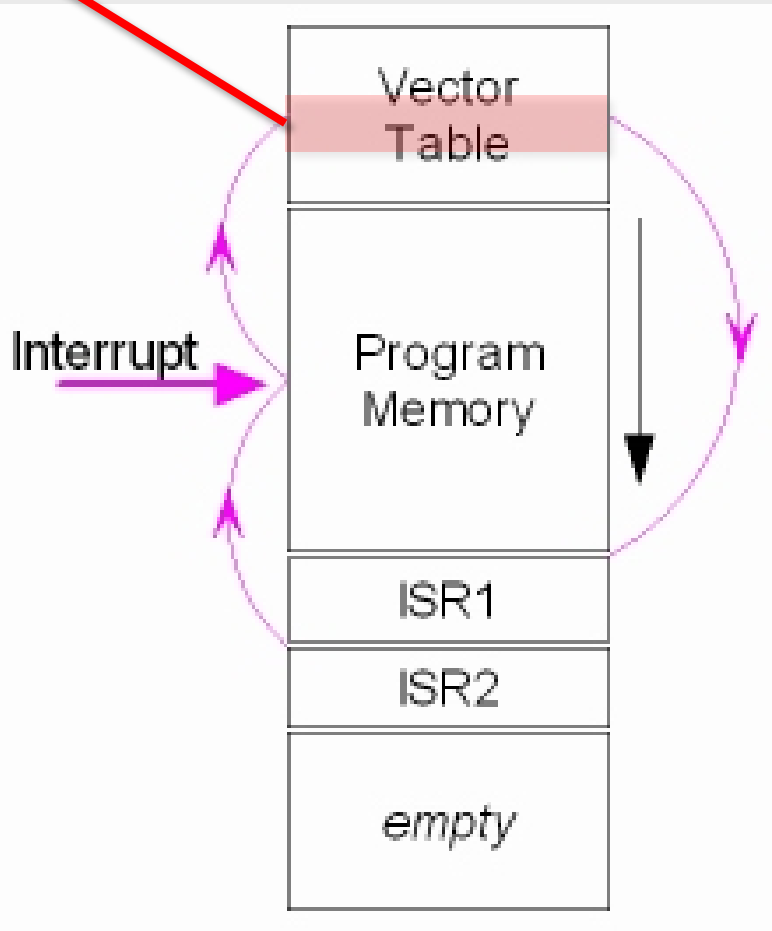
Generally, ISR should be fast – do one simple task as fast as possible.

ISRs should do their work transparently to the main program flow

See www.avrtutor.com/tutorial/interrupt/interrupts.php

Interrupts Concepts

One-instruction slot for ISR1



Starting at program address 0x00, there are predefined locations where different ISR start executing

For example, INT0's ISR starts execution at 0x0001

And the ISR for INT1 starts execution at 0x0002

Note: there is room for one instruction for every ISR.

Thus, this first instruction is an **rjmp** to the rest of the ISR code

Hence the name: ***vector tabl***e, or sometimes ***jump table.***

See www.avrtutor.com/tutorial/interrupt/interrupts.php

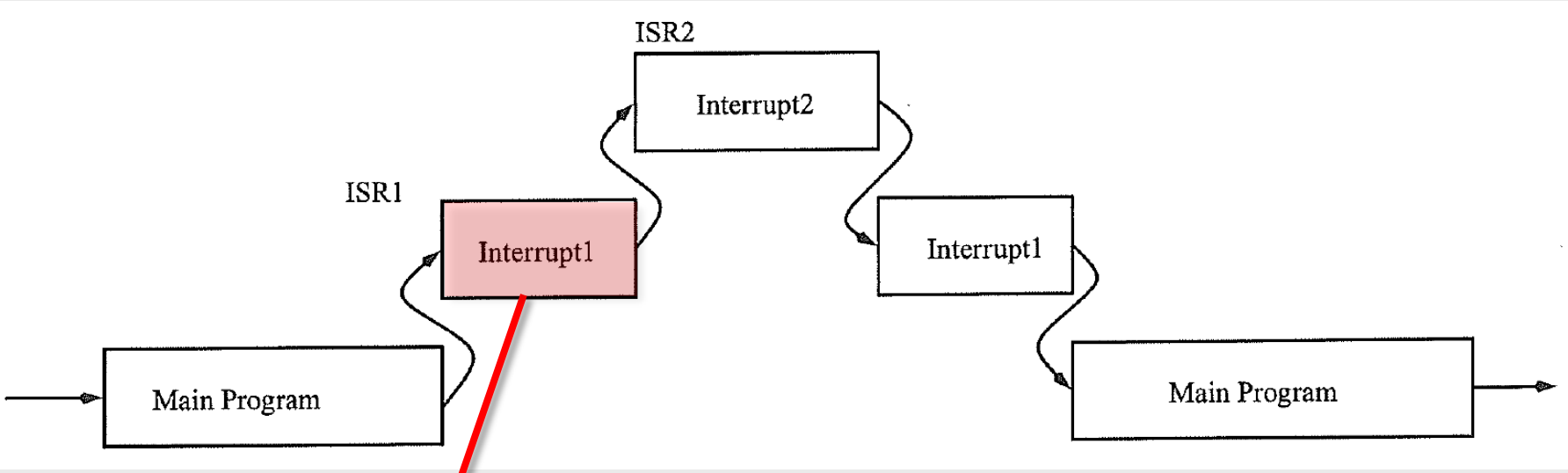
Nested Interrupts

When an ISR is invoked, the interrupts are turned off globally, i.e., there is an implied **cli.**

The **reti** instruction turn on interrupts globally.

Thus, normally, an ISR will not be interrupted by other ISRs.

However:



If this ISR enables interrupts (**sei** instruction), then other ISRs can interrupt it.

Writing an ISR

Writing an ISR is similar to writing a normal function: create a label, save registers as needed, perform the task at hand (toggle bits, etc.).

When done, clean up and return.

When the hardware triggers the ISR, the return address is push onto the stack, the **PC** is loaded with the ISR address and execution continues.

ISRs should end with the RETI instruction. This pops the return address off the stack and load the PC, and execution continues with the next instruction

Example:

```
            ...
extint:     push    r0      ; Save r0 on the Stack
            ...
            pop     r0      ; Restore r0
            reti            ; Return and enable interrupts
```

RETI – Return from Interrupt

Returns from interrupt. The return address is loaded from the STACK and the Global Interrupt Flag is set.

Note that the Status Register (SREG) is not automatically stored when entering an interrupt routine, and it is not restored when returning from an interrupt routine. This must be handled by the application program. The Stack Pointer uses a pre-increment scheme during RETI.

Example:

```
            ...
extint:     push    r0      ; Save r0 on the Stack
            ...
            pop     r0      ; Restore r0
            reti            ; Return and enable interrupts
```

ISR Activation

Assume that the AVR had been configured so that Timer 0 ISR is activated when Timer 0 rolls over

Assume the main program is running and the counter in incrementing.

Main Program

```
…
                 …
                 sei   ; Enable interrupts
                 …
0x0102           ldi   r30,LOW(2*msg)
0x0103           ldi   r31,2*HIGH(msg)
            L20:
0x0105           lpm   r2,Z+
0x0106           tst   r2
0x0107           breq  done
…                …
0x011A           rjmp  L20
            done:
…                …
```

PC = 0x0106

Timer 0 Overflow ISR

```
0x0121 tim0_ovf:
0x0122 sbis PINC,1
0x0123 reti
```

Stack

Note: for simplicity the ISR does not show all the required house-keeping: saving SREG etc.

ISR Activation

Assume that the AVR had been configured so that Timer 0 ISR is activated when Timer 0 rolls over

Assume the main program is running and the counter in incrementing.

Main Program

```
…
                 …
                 sei   ; Enable interrupts
                 …
0x0102           ldi   r30,LOW(2*msg)
0x0103           ldi   r31,2*HIGH(msg)
            L20:
0x0105           lpm   r2,Z+
0x0106           tst   r2
0x0107           breq  done
…                …
0x011A           rjmp  L20
            done:
…                …
```

PC = 0x0107

Timer 0 Overflow ISR

```
0x0121 tim0_ovf:
0x0122 sbis PINC,1
0x0123 reti
```

Stack

Note: for simplicity the ISR does not show all the required house-keeping: saving SREG etc.

ISR Activation

Assume that the AVR had been configured so that Timer 0 ISR is activated when Timer 0 rolls over

Assume the main program is running and the counter in incrementing.

Main Program

```
…
                 …
                 sei   ; Enable interrupts
                 …
0x0102           ldi   r30,LOW(2*msg)
0x0103           ldi   r31,2*HIGH(msg)
            L20:
0x0105           lpm   r2,Z+
0x0106           tst   r2
0x0107           breq  done
…                …
0x011A           rjmp  L20
            done:
…                …
```

PC = 0x0107

Timer 0 Overflow ISR

```
0x0121 tim0_ovf:
0x0122 sbis PINC,1
0x0123 reti
```

Stack

Roll Over

Note: for simplicity the ISR does not show all the required house-keeping: saving SREG etc.

ISR Activation

Assume that the AVR had been configured so that Timer 0 ISR is activated when Timer 0 rolls over

Assume the main program is running and the counter in incrementing.

Main Program

```
…
                 …
                 sei   ; Enable interrupts
                 …
0x0102           ldi   r30,LOW(2*msg)
0x0103           ldi   r31,2*HIGH(msg)
            L20:
0x0105           lpm   r2,Z+
0x0106           tst   r2
0x0107           breq  done
…                …
0x011A           rjmp  L20
            done:
…                …
```

PC = 0x0107

Timer 0 Overflow ISR

```
0x0121 tim0_ovf:
0x0122 sbis PINC,1
0x0123 reti
```

Stack

```
0x01
0x07
```

Note: for simplicity the ISR does not show all the required house-keeping: saving SREG etc.

ISR Activation

Assume that the AVR had been configured so that Timer 0 ISR is activated when Timer 0 rolls over

Assume the main program is running and the counter in incrementing.

Main Program

```
…
                 …
                 sei   ; Enable interrupts
                 …
0x0102           ldi   r30,LOW(2*msg)
0x0103           ldi   r31,2*HIGH(msg)
                 L20:
0x0105           lpm   r2,Z+
0x0106           tst   r2
0x0107           breq  done
…                …
0x011A           rjmp  L20
                 done:
…                …
```

PC = 0x0122

Timer 0 Overflow ISR

```
0x0121 tim0_ovf:
0x0122 sbis PINC,1
0x0123 reti
```

Stack

```
0x01
0x07
```

Note: for simplicity the ISR does not show all the required house-keeping: saving SREG etc.

ISR Activation

Assume that the AVR had been configured so that Timer 0 ISR is activated when Timer 0 rolls over

Assume the main program is running and the counter in incrementing.

Main Program

```
…
                 …
                 sei   ; Enable interrupts
                 …
0x0102           ldi   r30,LOW(2*msg)
0x0103           ldi   r31,2*HIGH(msg)
                 L20:
0x0105           lpm   r2,Z+
0x0106           tst   r2
0x0107           breq  done
…                …
0x011A           rjmp  L20
                 done:
…                …
```

PC = 0x0123

Timer 0 Overflow ISR

```
0x0121 tim0_ovf:
0x0122 sbis PINC,1
0x0123 reti
```

Stack

```
0x01
0x07
```

Note: for simplicity the ISR does not show all the required house-keeping: saving SREG etc.

ISR Activation

Assume that the AVR had been configured so that Timer 0 ISR is activated when Timer 0 rolls over

Assume the main program is running and the counter in incrementing.

Main Program

```
…
                 …
                 sei   ; Enable interrupts
0x0102           …
0x0103           ldi   r30,LOW(2*msg)
                 ldi   r31,2*HIGH(msg)
                 L20:
0x0105           lpm   r2,Z+
0x0106           tst   r2
0x0107           breq  done
…                …
0x011A           rjmp  L20
                 done:
…                …
```

PC = 0x0123

Timer 0 Overflow ISR

```
0x0121 tim0_ovf:
0x0122 sbis PINC,1
0x0123 reti
```

Stack

```
0x01
0x07
```

Note: for simplicity the ISR does not show all the required house-keeping: saving SREG etc.

ISR Activation

Assume that the AVR had been configured so that Timer 0 ISR is activated when Timer 0 rolls over

Assume the main program is running and the counter in incrementing.

Main Program

```
…
                 …
                 sei   ; Enable interrupts
                 …
0x0102           ldi   r30,LOW(2*msg)
0x0103           ldi   r31,2*HIGH(msg)
                 L20:
0x0105           lpm   r2,Z+
0x0106           tst   r2
0x0107           breq  done
…                …
0x011A           rjmp  L20
                 done:
…                …
```

PC = 0x0107

Timer 0 Overflow ISR

```
0x0121 tim0_ovf:
0x0122 sbis PINC,1
0x0123 reti
```

Stack

Note: for simplicity the ISR does not show all the required house-keeping: saving SREG etc.

ISR Activation

Assume that the AVR had been configured so that Timer 0 ISR is activated when Timer 0 rolls over

Assume the main program is running and the counter in incrementing.

Main Program

```
…
                 …
                 sei   ; Enable interrupts
                 …
0x0102           ldi   r30,LOW(2*msg)
0x0103           ldi   r31,2*HIGH(msg)
            L20:
0x0105           lpm   r2,Z+
0x0106           tst   r2
0x0107           breq  done
…                …
0x011A           rjmp  L20
            done:
…                …
```

PC = 0x0107

Timer 0 Overflow ISR

```
0x0121 tim0_ovf:
0x0122 sbis PINC,1
0x0123 reti
```

Stack

Note: for simplicity the ISR does not show all the required house-keeping: saving SREG etc.

Interrupt-Driven Programming

In some programs, the main program configures the controller and interrupts, and then enters the main loop that does nothing. The real work is done by the ISR(s) when it/they is/are triggered

```
Interrupt [EXT_INT0] void ext_int_isr(void)
{
   PORTC = PORTC ^ 0x01;   // Toggle LSB of PORT C
}

void main(void)
{
   DDRC = 0x01;   // Port C LSB is output
   GICR = 0x40;   // Enable INT0
   MCUCR = 0x02;  // INT0 on falling edge
   #asm("sei");   // Enable interrupts
   while(1)
       ;          // Loop forever
}
```

Note: for simplicity the ISR does not show all the required house-keeping: saving **SREG** etc.

Important Considerations I

ISRs should return with an **RETI** and not **RET** instruction…

The **sei** and **cli** instructions globally enable/disable all interrupts by setting/clearing the Global Interrupt Flag (I) in **SREG** (Status Register).

Unless disabled by **cli**, interrupts can in principle occur any time, so one has to structure program to account for this.

Related to previous point – generally speaking, ISRs should save and restore resources they use → do their work as transparently as possible

Generally, ISR should be fast – do one simple task as fast as possible. Be careful about implied delays.

```
Interrupt [TMR0] void tmr0_int_isr(void)
{
    printf("%d",n);  // Print something
}
```

The ISR is one line, but the C printf function is hugely complex and quite long, so this ISR does NOT meet the fast criteria…

Important Considerations II

Once enabled, ISRs can occur at any time in the program. Programmers should structure the main program flow accordingly and plan for interrupts at any time.

Since ISRs can occur at any time in the program, the ISR should guard against side effects and save and restore registers they use.

Protect sections of code that must not be interrupted using **cli** and **sei**:

```
…
cli    ; Turn off all interrupts
       ; Code that should not be interrupted
       ; goes here
sei    ; Enable interrupts
…
```

Code that should not be interrupted is called a **critical section**.

What constitutes a critical section? This depends on the specific application. An embedded system may generate precise pulse (PWM for servo control) and have a button for user input. A critical section in this case may be a section of code where the PWM hardware or timers are reloaded, and one does not want to interrupt because a user presses the button to perform some non-critical task such as turning on the backlight on an LCD display.

Interrupt-Driven Programming

```
Interrupt [EXT_INT0] void ext_int_isr(void)
{
   // Get RPG State
}

Interrupt [EXT_INT1] void ext_int_isr(void)
{
   // Process push button
}

Interrupt [TMR0] void tmr0_int_isr(void)
{
   PORTC = PORTC ^ 0x01;   // Toggle LSB of PORT C
   TCNT0 = 125;            // reload timer
}

void main(void)
{
   DDRC = 0x01;   // Port C LSB is output
   GICR = 0x40;   // Enable INT0
   MCUCR = 0x02;  // INT0 on falling edge
   #asm("sei");   // Enable interrupts
   while(1)
       ;          // Loop forever
}
```

Triggered when RPG changes the pin it is connected to.

Note: for simplicity the ISRs does not show all the required house-keeping: saving **SREG** etc.

Interrupt-Driven Programming

```
Interrupt [EXT_INT0] void ext_int_isr(void)
{
   // Get RPG State
}

Interrupt [EXT_INT1] void ext_int_isr(void)
{
   // Process push button
}

Interrupt [TMR0] void tmr0_int_isr(void)
{
   PORTC = PORTC ^ 0x01;   // Toggle LSB of PORT C
   TCNT0 = 125;            // reload timer
}

void main(void)
{
   DDRC = 0x01;   // Port C LSB is output
   GICR = 0x40;   // Enable INT0
   MCUCR = 0x02;  // INT0 on falling edge
   #asm("sei");   // Enable interrupts
   while(1)
         ;           // Loop forever
}
```

Triggered when user presses button

Note: for simplicity the ISRs does not show all the required house-keeping: saving **SREG** etc.

Interrupt-Driven Programming

```
Interrupt [EXT_INT0] void ext_int_isr(void)
{
   // Get RPG State
}

Interrupt [EXT_INT1] void ext_int_isr(void)
{
   // Process push button
}

Interrupt [TMR0] void tmr0_int_isr(void)
{
   PORTC = PORTC ^ 0x01;   // Toggle LSB of PORT C
   TCNT0 = 125;            // reload timer
}

void main(void)
{
   DDRC = 0x01;   // Port C LSB is output
   GICR = 0x40;   // Enable INT0
   MCUCR = 0x02;  // INT0 on falling edge
   #asm("sei");   // Enable interrupts
   while(1)
         ;           // Loop forever
}
```

Triggered when timer that generates square wave rolls over

Note: for simplicity the ISRs does not show all the required house-keeping: saving **SREG** etc.

Interrupt-Driven Programming

```
Interrupt [EXT_INT0] void ext_int_isr(void)
{
   // Get RPG State
}

Interrupt [EXT_INT1] void ext_int_isr(void)
{
   // Process push button
}

Interrupt [TMR0] void tmr0_int_isr(void)
{
   PORTC = PORTC ^ 0x01;   // Toggle LSB of PORT C
   TCNT0 = 125;            // reload timer
}

void main(void)
{
   DDRC = 0x01;   // Port C LSB is output
   GICR = 0x40;   // Enable INT0
   MCUCR = 0x02;  // INT0 on falling edge
   #asm("sei");   // Enable interrupts
   while(1)
         ;           // Loop forever
}
```

Configure

Note: for simplicity the ISRs does not show all the required house-keeping: saving **SREG** etc.

Interrupt-Driven Programming

```
Interrupt [EXT_INT0] void ext_int_isr(void)
{
   // Get RPG State
}

Interrupt [EXT_INT1] void ext_int_isr(void)
{
   // Process push button
}

Interrupt [TMR0] void tmr0_int_isr(void)
{
   PORTC = PORTC ^ 0x01;   // Toggle LSB of PORT C
   TCNT0 = 125;            // reload timer
}

void main(void)
{
   DDRC = 0x01;   // Port C LSB is output
   GICR = 0x40;   // Enable INT0
   MCUCR = 0x02;  // INT0 on falling edge
   #asm("sei");   // Enable interrupts
   while(1)
         ;           // Loop forever
}
```

Note: for simplicity the ISRs does not show all the required house-keeping: saving **SREG** etc.

Wait for interrupts

Using a Timer Overflow Interrupt to Generate Square Waves

Assume we want to use timer 0 to generate a square wave on PC5, using interrupts.

- * Set timer up to generate an interrupt when it overflows
- * In the ISR toggle output pin PC5
- * Reload the timer 0 with the required start value
- * Set up ISR jump table/vector
- * Global enable interrupts

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Timer 0 Overflow interrupt ISR
;;
tim0_ovf:
   push  r25
   in    r25,sreg
   push  r25
   sbi   PINC,5      ; Toggle PORTC,5
   ldi   r25,201     ; Reload counter
   out   TCNT0,r25
   pop   r25
   out   sreg,r25
   pop   r25
   reti
```

Can name this anything, but it makes sense to use same naming conventions as in AVR documentation

Note: This ISRs does the required house-keeping: save **SREG** and **R25**

Assume we want to use timer 0 to generate a square wave on PC5, using interrupts.

- Set timer up to generate an interrupt when it overflows
- In the ISR toggle output pin PC5
- Reload the timer 0 with the required start value
- Set up ISR jump table/vector
- Global enable interrupts

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Timer 0 Overflow interrupt ISR
;;
tim0_ovf:
   push  r25
   in    r25,sreg
   push  r25
   sbi   PINC,5      ; Toggle PORTC,5
   ldi   r25,201     ; Reload counter
   out   TCNT0,r25
   pop   r25
   out   sreg,r25
   pop   r25
   reti
```

Save **SREG** and **R25**

Note: This ISRs does the required house-keeping: save **SREG** and **R25**

Assume we want to use timer 0 to generate a square wave on PC5, using interrupts.

- * Set timer up to generate an interrupt when it overflows
- * In the ISR toggle output pin PC5
- * Reload the timer 0 with the required start value
- * Set up ISR jump table/vector
- * Global enable interrupts

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Timer 0 Overflow interrupt ISR
;;
tim0_ovf:
   push  r25
   in    r25,sreg
   push  r25
   sbi   PINC,5            ; Toggle PORTC,5
   ldi   r25,201           ; Reload counter
   out   TCNT0,r25
   pop   r25
   out   sreg,r25
   pop   r25
   reti
```

Note how we toggle PC5 by writing 1 to PINC,5

Note: This ISRs does the required house-keeping: save **SREG** and **R25**

Assume we want to use timer 0 to generate a square wave on PC5, using interrupts.

- Set timer up to generate an interrupt when it overflows
- In the ISR toggle output pin PC5
- Reload the timer 0 with the required start value
- Set up ISR jump table/vector
- Global enable interrupts

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Timer 0 Overflow interrupt ISR
;;
tim0_ovf:
   push  r25
   in    r25,sreg
   push  r25
   sbi   PINC,5      ; Toggle PORTC,5
   ldi   r25,201     ; Reload counter
   out   TCNT0,r25
   pop   r25
   out   sreg,r25
   pop   r25
   reti
```

Reload counter.

Note: This ISRs does the required house-keeping: save SREG and r25

Assume we want to use timer 0 to generate a square wave on PC5, using interrupts.

- Set timer up to generate an interrupt when it overflows
- In the ISR toggle output pin PC5
- Reload the timer 0 with the required start value
- Set up ISR jump table/vector
- Global enable interrupts

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Timer 0 Overflow interrupt ISR
;;
tim0_ovf:
   push  r25
   in    r25,sreg
   push  r25
   sbi   PINC,5      ; Toggle PORTC,5
   ldi   r25,201     ; Reload counter
   out   TCNT0,r25
   pop   r25
   out   sreg,r25
   pop   r25
   reti
```

Restore **SREG** and **R25**

Note: This ISRs does the required house-keeping: save **SREG** and **R25**

Assume we want to use timer 0 to generate a square wave on PC5, using interrupts.

- Set timer up to generate an interrupt when it overflows
- In the ISR toggle output pin PC5
- Reload the timer 0 with the required start value
- Set up ISR jump table/vector
- Global enable interrupts

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Timer 0 Overflow interrupt ISR
;;
tim0_ovf:
   push  r25
   in    r25,sreg
   push  r25
   sbi   PINC,5       ; Toggle PORTC,5
   ldi   r25,201      ; Reload counter
   out   TCNT0,r25
   pop   r25
   pop   r25
   out   sreg,r25
   reti
```

Return with **reti**

Note: This ISRs does the required house-keeping: save **SREG** and **R25**

```
;; Shows how to use timer 0 overflow interrupt.

.include "m88padef.inc"
.cseg
.org 0x00          ; PC points here after power up,
   rjmp  reset     ;   hardware reset, WDT timeout

.org 0x010         ; PC points here on timer 0
   rjmp  tim0_ovf  ;   over flow interrupt

.org  0x1a         ; Just past the last ISR vector
reset:             ;  on ATmega88PA (see docs)

; Configure PC5 as output, used for LED.
   sbi   DDRC,5

; Enable overflow interrupt on 8-bit timer 0.
   lds   r24,TIMSK0
   ori   r24,0x01       ; Overflow interrupt enable
   sts   TIMSK0,r24

; Turn timer 0 on, use system clock, prescaled with
; 256 => increment at (10 MHz)/256 => 25.6 us/tick
   ldi   r24,0x04
   out   TCCR0B,r24

; Enable interrupts and enter main loop.
   sei
main:
    rjmp  main
```

PC points here at reset. Jump to main loop

```
;; Shows how to use timer 0 overflow interrupt.

.include "m88padef.inc"
.cseg
.org 0x00          ; PC points here after power up,
   rjmp  reset     ;   hardware reset, WDT timeout

.org 0x010         ; PC points here on timer 0
   rjmp  tim0_ovf  ;   over flow interrupt

.org  0x1a         ; Just past the last ISR vector
reset:             ;  on ATmega88PA (see docs)

; Configure PC5 as output, used for LED.
   sbi   DDRC,5

; Enable overflow interrupt on 8-bit timer 0.
   lds   r24,TIMSK0
   ori   r24,0x01       ; Overflow interrupt enable
   sts   TIMSK0,r24

; Turn timer 0 on, use system clock, prescaled with
; 256 => increment at (10 MHz)/256 => 25.6 us/tick
   ldi   r24,0x04
   out   TCCR0B,r24

; Enable interrupts and enter main loop.
   sei
main:
    rjmp  main
```

Timer 0's slot in the ISR vector table. Place an **rjmp** there to the rest of the ISR code

```
;; Shows how to use timer 0 overflow interrupt.

.include "m88padef.inc"
.cseg
.org 0x00          ; PC points here after power up,
   rjmp  reset     ;   hardware reset, WDT timeout

.org 0x010         ; PC points here on timer 0
   rjmp  tim0_ovf  ;   over flow interrupt

.org  0x1a         ; Just past the last ISR vector
reset:             ;  on ATmega88PA (see docs)

; Configure PC5 as output, used for LED.
   sbi   DDRC,5

; Enable overflow interrupt on 8-bit timer 0.
   lds   r24,TIMSK0
   ori   r24,0x01       ; Overflow interrupt enable
   sts   TIMSK0,r24

; Turn timer 0 on, use system clock, prescaled with
; 256 => increment at (10 MHz)/256 => 25.6 us/tick
   ldi   r24,0x04
   out   TCCR0B,r24

; Enable interrupts and enter main loop.
   sei
main:
    rjmp  main
```

0x1a is the address just beyond the last ISR vector on the ATmega88PA

```
;; Shows how to use timer 0 overflow interrupt.

.include "m88padef.inc"
.cseg
.org 0x00          ; PC points here after power up,
   rjmp  reset     ;   hardware reset, WDT timeout

.org 0x010         ; PC points here on timer 0
   rjmp  tim0_ovf  ;   over flow interrupt

.org  0x1a         ; Just past the last ISR vector
reset:             ;  on ATmega88PA (see docs)

; Configure PC5 as output, used for LED.
   sbi   DDRC,5

; Enable overflow interrupt on 8-bit timer 0.
   lds   r24,TIMSK0
   ori   r24,0x01       ; Overflow interrupt enable
   sts   TIMSK0,r24

; Turn timer 0 on, use system clock, prescaled with
; 256 => increment at (10 MHz)/256 => 25.6 us/tick
   ldi   r24,0x04
   out   TCCR0B,r24

; Enable interrupts and enter main loop.
   sei
main:
    rjmp  main
```

0x1a is the address just beyond the last ISR vector on the ATmega88PA

```
;; Shows how to use timer 0 overflow interrupt.

.include "m88padef.inc"
.cseg
.org 0x00          ; PC points here after power up,
   rjmp  reset     ;   hardware reset, WDT timeout

.org 0x010         ; PC points here on timer 0
   rjmp  tim0_ovf  ;   over flow interrupt

.org  0x1a         ; Just past the last ISR vector
reset:             ;  on ATmega88PA (see docs)

; Configure PC5 as output, used for LED.
   sbi   DDRC,5

; Enable overflow interrupt on 8-bit timer 0.
   lds   r24,TIMSK0
   ori   r24,0x01        ; Overflow interrupt enable
   sts   TIMSK0,r24

; Turn timer 0 on, use system clock, prescaled with
; 256 => increment at (10 MHz)/256 => 25.6 us/tick
   ldi   r24,0x04
   out   TCCR0B,r24

; Enable interrupts and enter main loop.
   sei
main:
    rjmp  main
```

Note the use of **lds** and **sts** rather than **in/out**. This is because the address of **TMSK0** is outside the 0…63 address range that **in/out** can handle.

```
;; Shows how to use timer 0 overflow interrupt.

.include "m88padef.inc"
.cseg
.org 0x00          ; PC points here after power up,
   rjmp  reset     ;   hardware reset, WDT timeout

.org 0x010         ; PC points here on timer 0
   rjmp  tim0_ovf  ;   over flow interrupt

.org  0x1a         ; Just past the last ISR vector
reset:             ;  on ATmega88PA (see docs)

; Configure PC5 as output, used for LED.
   sbi   DDRC,5

; Enable overflow interrupt on 8-bit timer 0.
   lds   r24,TIMSK0
   ori   r24,0x01       ; Overflow interrupt enable
   sts   TIMSK0,r24

; Turn timer 0 on, use system clock, prescaled with
; 256 => increment at (10 MHz)/256 => 25.6 us/tick
   ldi   r24,0x04
   out   TCCR0B,r24

; Enable interrupts and enter main loop.
   sei
main:
    rjmp  main
```

Configure timer

```
;; Shows how to use timer 0 overflow interrupt.

.include "m88padef.inc"
.cseg
.org 0x00          ; PC points here after power up,
   rjmp  reset     ;   hardware reset, WDT timeout

.org 0x010         ; PC points here on timer 0
   rjmp  tim0_ovf  ;   over flow interrupt

.org  0x1a         ; Just past the last ISR vector
reset:             ;  on ATmega88PA (see docs)

; Configure PC5 as output, used for LED.
   sbi   DDRC,5

; Enable overflow interrupt on 8-bit timer 0.
   lds   r24,TIMSK0
   ori   r24,0x01       ; Overflow interrupt enable
   sts   TIMSK0,r24

; Turn timer 0 on, use system clock, prescaled with
; 256 => increment at (10 MHz)/256 => 25.6 us/tick
   ldi   r24,0x04
   out   TCCR0B,r24

; Enable interrupts and enter main loop.
   sei
main:
    rjmp  main
```

Turn on interrupts and enter main loop

EICRA – External Interrupt Control Register A

The External Interrupt Control Register A **EIRCA** contains control bits for interrupt sense control.

Bit	7	6	5	4	3	2	1	0	
(0x69)	–	–	–	–	ISC11	ISC10	ISC01	ISC00	EICRA
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Reserved

INT1 Sense Control

INT0 Sense Control

Table 12-1. Interrupt 1 Sense Control

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.

Table 12-2. Interrupt 0 Sense Control

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.

EIMSK – External Interrupt Mask Register

When the **INT1** bit is set (one) and the **I**-bit in the Status Register (**SREG**) is set (one), the external pin interrupt is enabled.

Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	–	–	–	–	–	–	INT1	INT0	EIMSK
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Reserved

When the **INT**0 bit is set (one) and the **I**-bit in the Status Register (**SREG**) is set (one), the external pin interrupt is enabled.

EIFR – External Interrupt Flag Register

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 **EIMSK** are set (one), the MCU will jump to the corresponding Interrupt Vector. The flag is cleared when the interrupt routine is executed.

Alternatively, the flag can be cleared by writing a logical one to it. This flag is always cleared when **INT1** is configured as a level interrupt.

Bit	7	6	5	4	3	2	1	0	
0x1C (0x3C)	–	–	–	–	–	–	INTF1	INTF0	EIFR
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Reserved

EIFR – External Interrupt Flag Register

When an edge or logic change on the **INT0** pin triggers an interrupt request, **INTF0** becomes set (one). If the **I**-bit in **SREG** and the **INT0** bit in **EIMSK** are set (one), the MCU will jump to the corresponding Interrupt Vector. The flag is cleared when the interrupt routine is executed.

Alternatively, the flag can be cleared by writing a logical one to it. This flag is always cleared when **INT0** is configured as a level interrupt.

Bit	7	6	5	4	3	2	1	0	
0x1C (0x3C)	–	–	–	–	–	–	INTF1	INTF0	EIFR
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Reserved

PIN Change Registers

PCICR – Pin Change Interrupt Control Register

Bit	7	6	5	4	3	2	1	0	
(0x68)	–	–	–	–	–	PCIE2	PCIE1	PCIE0	PCICR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

PCIFR – Pin Change Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x1B (0x3B)	–	–	–	–	–	PCIF2	PCIF1	PCIF0	PCIFR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

PCMSK2 – Pin Change Mask Register 2

Bit	7	6	5	4	3	2	1	0	
(0x6D)	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16	PCMSK2
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	

PCMSK1 – Pin Change Mask Register 1

Bit	7	6	5	4	3	2	1	0	
(0x6C)	–	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8	PCMSK1
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

PCMSK0 – Pin Change Mask Register 0

Bit	7	6	5	4	3	2	1	0	
(0x6B)	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	PCMSK0
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	

Some I/O Locations are Problematic

The **IN/OUT** instructions access location in I/O space. However, some configuration are out of reach of the **IN/OUT** (0…63) range, so one has to use **LDS** and **STS** instructions:

Overflow Interrupt Enable

Bit	7	6	5	4	3	2	1	0	
(0x6E)	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	TIMSK0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

SRAM Address

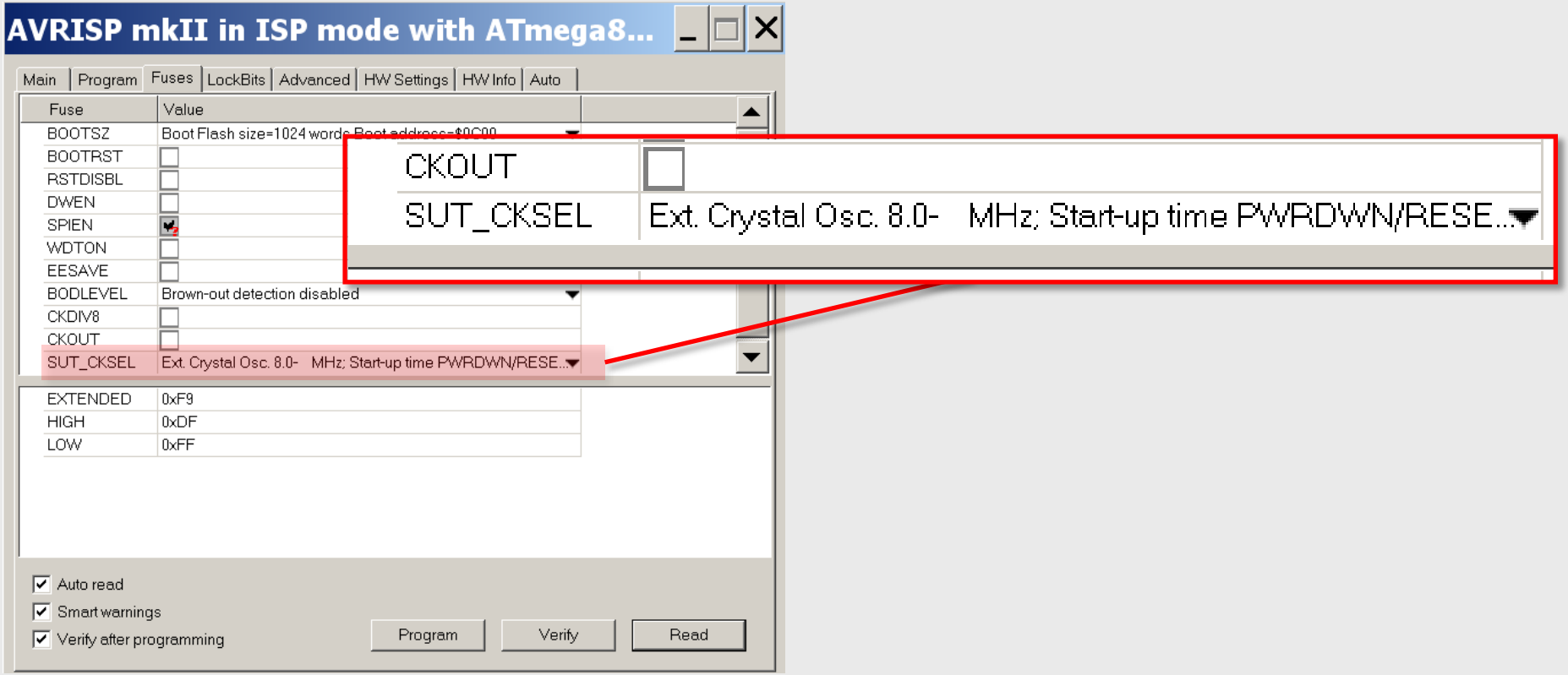
```
; Enable overflow interrupt on 8-bit timer 0.
; Use LDS and STS instructions, since IN/OUT can
; only access addresses in range 0...63, and TIMSK0
; on ATmega88 is at 0x6E.
   lds    r24,TIMSK0      ; Grab the 8-bit timer's interrupt mask
   ori    r24,0x01        ; Enable overflow interrupt
   sts    TIMSK0,r24      ; Update the interrupt mask
```

Waking Up Via External Interrupts

Note that if a level triggered interrupt is used for wake-up from Power-down, the required level must be held long enough for the MCU to complete the wake-up to trigger the level interrupt.

If the level disappears before the end of the Start-up Time, the MCU will still wake up, but no interrupt will be generated.

The start-up time is defined by the **SUT** and **CKSEL** Fuses




"That's all Folks!"