AVR Interrupts in C

Implementation of interrupts are not explicitly addressed by the C language. It is dependent on the compiler.

We are using avr gcc so we will refer to:

http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html

Macros for writing interrupt handler functions

#define ISR(vector, attributes)
#define SIGNAL(vector)

#define EMPTY_INTERRUPT(vector)

#define ISR_ALIAS(vector, target_vector)

#define reti()

#define BADISR_vect **Catch-all interrupt vector**

Commands to Enable and disable interrupts:

#define sei()
#define cli()

ISR attributes

#define ISR_BLOCK

#define ISR_NOBLOCK

#define ISR_NAKED

#define ISR_ALIASOF(target_vector)

Sample Interrupt Definition:

```
#include <avr/interrupt.h>
ISR(ADC_vect)  //vector names provided by compiler
{
    // user code here
}
```

Interrupts and Vector Names available for AVR

Available interrupt vector names can be found in compiler documentation:
http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html

BADISR_vect

- A very common error students had when working on projects had been calling of interrupts that were not defined.
  - Debugging interrupt-driven code is often difficult
    - External interrupts can't don't work on the timescale of a debugger.
    - The instruction pointer seemingly jumps around when stepping through code.
    - Worse is when an interrupt vector is not defined and program seemly resets itself for no reason.
    - Creating a default ISR using BADISR_vect and printing a message or turning on an LED helps detect this!

Allowing Nested Interrupt Calls (ISR_NOBLOCK)

Nested interrupts require interrupts to be enabled during interrupt service routine execution.

```
ISR(XXX_vect, ISR_NOBLOCK)
{
  ...
}
```

ISR_NOBLOCK attribute takes care of calling sei() ASAP.
sei is called BEFORE the prolog (which includes commands to save state), so it calls it sooner than if sei() is just added to default

ISRBLOCK version:

```
ISR(XXX_vect, ISR_BLOCK)
{
   sei();  //enables interrupts AFTER PROLOGE
  ...
}
```

Empty interrupt service routines

In rare circumstances, in interrupt vector does not need any code to be implemented at all. The vector must be declared anyway, so when the interrupt triggers it won't execute the BADISR_vect code (which by default restarts the application).
This could for example be the case for interrupts that are solely enabled for the purpose of getting the controller out of sleep_mode().
A handler for such an interrupt vector can be declared using the EMPTY_INTERRUPT() macro, which has no body

Example:

```
EMPTY_INTERRUPT(ADC_vect);  //no body
```

Pasted from <http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html>

Manually defined ISRs (NAKED ISR)

In some circumstances, the compiler-generated prologue and epilogue of the ISR (responsible for saving and restoring states) might not be optimal for the job. Perhaps the register states do not need to be saved and restored by the ISR. A manually defined ISR could be considered, particularly to speedup the interrupt handling.

This can be done with inline assembly or by creating a naked ISR:

```
ISR(TIMER1_OVF_vect, ISR_NAKED)
{
  PORTB |= _BV(0);  // results in SBI which
                    //    does not affect SREG
  reti();
}
```

Pasted and modified from <http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html>

Shared ISRs
(Two vectors sharing ISR code)

In some circumstances, the actions to be taken upon two different interrupts might be completely identical so a single implementation for the ISR would suffice. For example, pin-change interrupts arriving from two different ports could logically signal an event that is independent from the actual port (and thus interrupt vector) where it happened. Sharing interrupt vector code can be accomplished using the ISR_ALIASOF() attribute to the ISR macro:

```
ISR(PCINT0_vect)
{
   ...
   // Code to handle the event.
}
ISR(PCINT1_vect, ISR_ALIASOF(PCINT0_vect));
```

<http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html>

Example C code Timer and Interrupts

https://www.mainframe.cx/~ckuethe/avr-c-tutorial/lesson10.c

Pasted from <https://www.mainframe.cx/~ckuethe/avr-c-tutorial/lesson10.c>

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

volatile uint8_t intrs;  //global var to count interrupt events

ISR(TIMER0_OVF_vect) {
    /* this ISR is called when TIMER0 overflows */
    intrs++;
    /* strobe PORTB5*/
    if (intrs >= 61){   //LED is toggled every 62 times this is called
        PORTB ^= _BV(5); //_BV is a macro _BV(5) is same as  (1<<5)
        intrs = 0;
    }
}
```

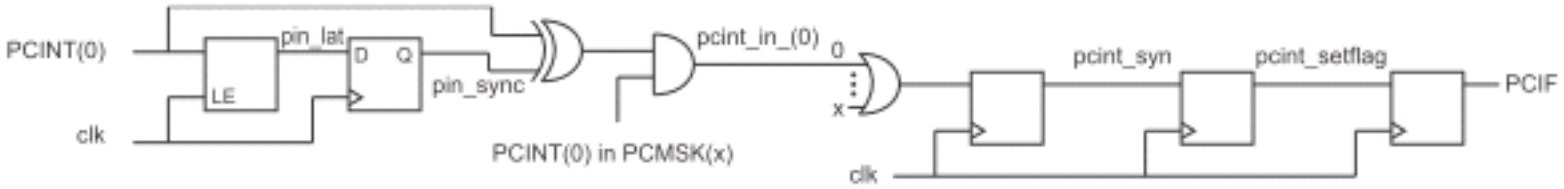
Pasted from <https://www.mainframe.cx/~ckuethe/avr-c-tutorial/lesson10.c>

```
int main(void) {
    /*
     * set up cpu clock divider. the TIMER0 overflow ISR toggles the
     * output port after enough interrupts have happened.
     * 16MHz (FCPU) / 1024 (CS0 = 5) -> 15625 incr/sec
     * 15625 / 256 (number of values in TCNT0) -> 61 overflows/sec
     */
    TCCR0B |= _BV(CS02) | _BV(CS00);
/* Enable Timer Overflow Interrupts */
    TIMSK0 |= _BV(TOIE0);
/* other set up */
    DDRB = 0xff;  //set pin direction
    TCNT0 = 0;    //set timer
    intrs = 0;    //set overflow counter
/* Enable Interrupts */
    sei();
    while (1); /* empty loop */
}
```

Pasted from <https://www.mainframe.cx/~ckuethe/avr-c-tutorial/lesson10.c>

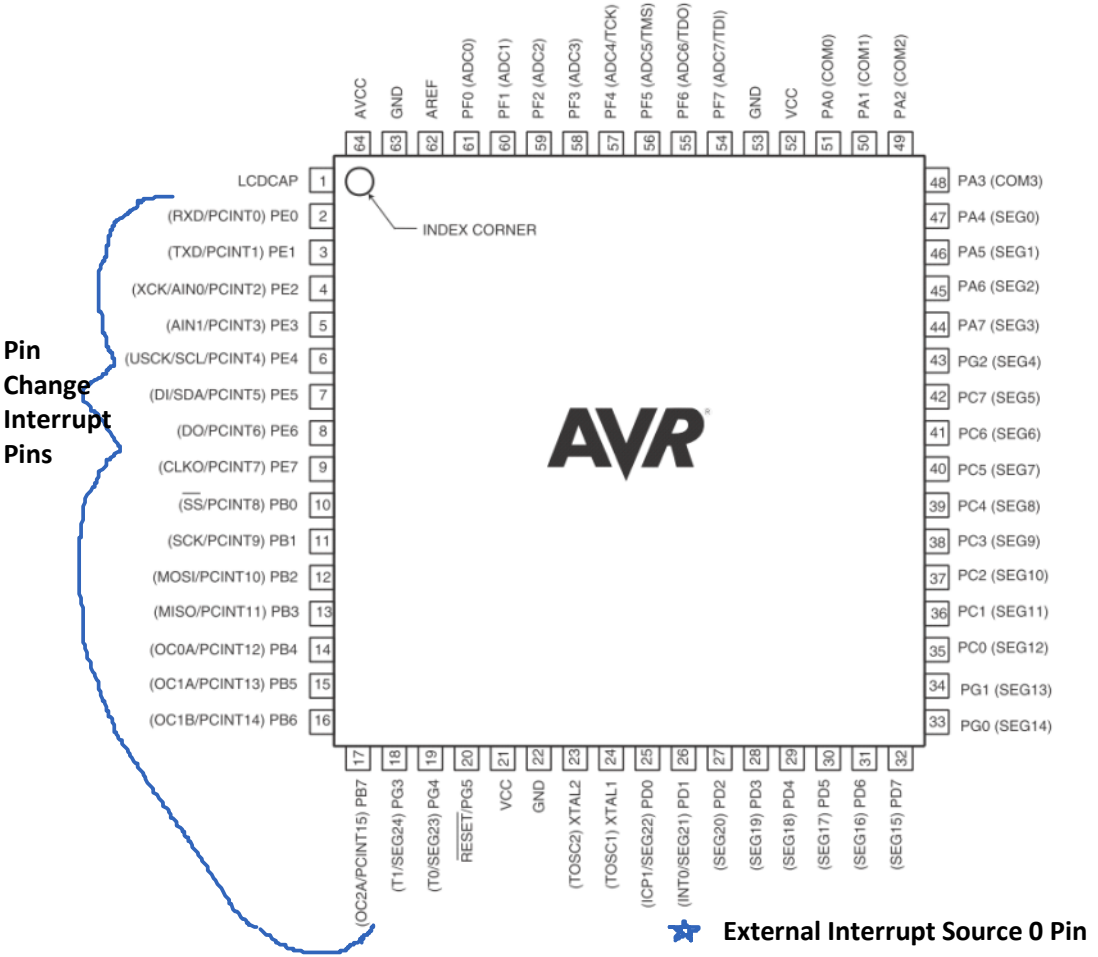
Pin Change Interrupts

Figure 12-1. Pin Change Interrupt



The External Interrupts are triggered by the INT0 pin or any of the PCINT15..0 pins. Observe that, if enabled, the interrupts will trigger even if the INT0 or PCINT15..0 pins are configured as outputs. This feature provides a way of generating a software interrupt. The pin change interrupt PCI1 will trigger if any enabled PCINT15..8 pin toggles. Pin change interrupts PCI0 will trigger if any enabled PCINT7..0 pin toggles. The PCMSK1 and PCMSK0 Registers control which pins contribute to the pin change interrupts. Pin change interrupts on PCINT15..0 are detected asynchronously. This implies that these interrupts can be used for waking the part also from sleep modes other than Idle mode.

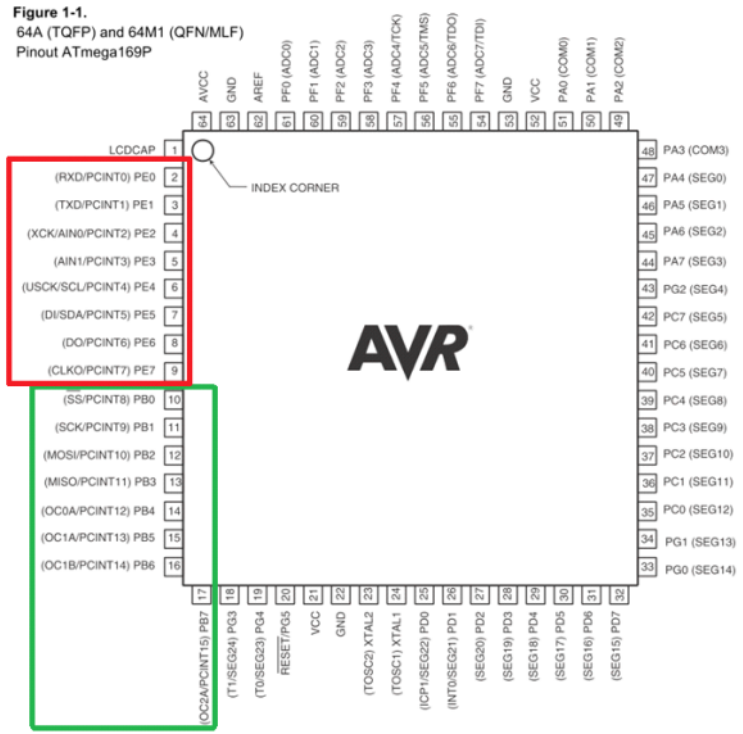
Figure 1-1. 64A (TQFP) and 64M1 (QFN/MLF) Pinout ATmega169P



Pin Change Interrupt Registers

Each pin change interrupt enable is controlled at THREE levels rather than two by three bits

- **the global interrupt enable**
- **EIMSK**
- **PCMSK1 or PCMSK0**



EICRA – External Interrupt Control Register A
controls trigger for INT0 pin, low-level, any change, falling edge, or rising edge

EIMSK – External Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	PCIE1	PCIE0	–	–	–	–	–	INT0	EIMSK
Read/Write	R/W	R/W	R	R	R	R	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Controls enables for interrupts PCIE1 PCIE0 and INT0

EIFR – External Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x1C (0x3C)	PCIF1	PCIF0	–	–	–	–	–	INTF0	EIFR
Read/Write	R/W	R/W	R	R	R	R	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Flag bits for interrupts
PCIF1 are cleared when ISR is executed or when a 1 (yes a 1) is written to it
INTF0 is cleared when ISR is executed, when a 1 is written to it, or when INT0 is configured as level-interrupt

PCMSK1,PCMSK0
Have bits to enable individual pins to trigger interrupts on their change
1 enables and 0 is disable

PCMSK1 – Pin Change Mask Register 1

Bit	7	6	5	4	3	2	1	0	
(0x6C)	PCINT15	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8	PCMSK1
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	

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	

Buttons and Interrupts

Be aware that when using mechanical sources for interrupts they may trigger multiple ISR calls depending on the situation. Our buttons seem to be debounced.

Note: As an experiment, you can try to count the number of transitions by using an interrupt-based software counter. Later we will use a hardware timer trigger of a pin that can do this even more precisely.

Optional Reading Material: Location of Interrupts

11.2 Moving Interrupts Between Application and Boot Space

Friday, April 01, 2011 10:32 AM

Two possible locations for interrupt vector table boot and application

Special sequence to change interupt vector table selection to prevent accidental change, involves a lock bit.

See 11.2 Moving Interrupts Between Application and Boot Space for more details.

```
Move_interrupts:
; Get MCUCR
in r16, MCUCR
mov r17, r16
; Enable change of Interrupt Vectors
ori r16, (1<<IVCE)
out MCUCR, r16
; Move interrupts to Boot Flash section
ori r17, (1<<IVSEL)
out MCUCR, r17
ret
```

```
void Move_interrupts(void)
{
uchar temp;
/* Get MCUCR*/
temp = MCUCR;
/* Enable change of Interrupt Vectors */
MCUCR = temp | (1<<IVCE);
/* Move interrupts to Boot Flash section
*/
MCUCR = temp | (1<<IVSEL);
}
```

11.2.1 MCUCR – MCU Control Register

Bit	7	6	5	4	3	2	1	0	
0x35 (0x55)	JTD	-	-	PUD	–	–	IVSEL	IVCE	MCUCR
Read/Write	R/W	R	R	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit 1 – IVSEL: Interrupt Vector Select
When the IVSEL bit is cleared (zero), the Interrupt Vectors are placed at the start of the Flash memory. When this bit is set (one), the Interrupt Vectors are moved to the beginning of the Boot
Loader section of the Flash. The actual address of the start of the Boot Flash Section is determined by the BOOTSZ Fuses. Refer to the section ”Boot Loader Support – Read-While-Write Self-Programming” on page 280 for details.

- Bit 0 – IVCE: Interrupt Vector Change Enable

The IVCE bit must be written to logic one to enable change of the IVSEL bit. IVCE is cleared by
hardware four cycles after it is written or when IVSEL is written. Setting the IVCE bit will disable
interrupts, as explained in the description in ”Moving Interrupts Between Application and Boot Space” on page 59. See Code Example.