ECE3411 – Spring 2019

Lecture 7

Interrupts

Sung Yeul Park
Department of Electrical & Computer Engineering
University of Connecticut
Email: sung_yeul.park@uconn.edu

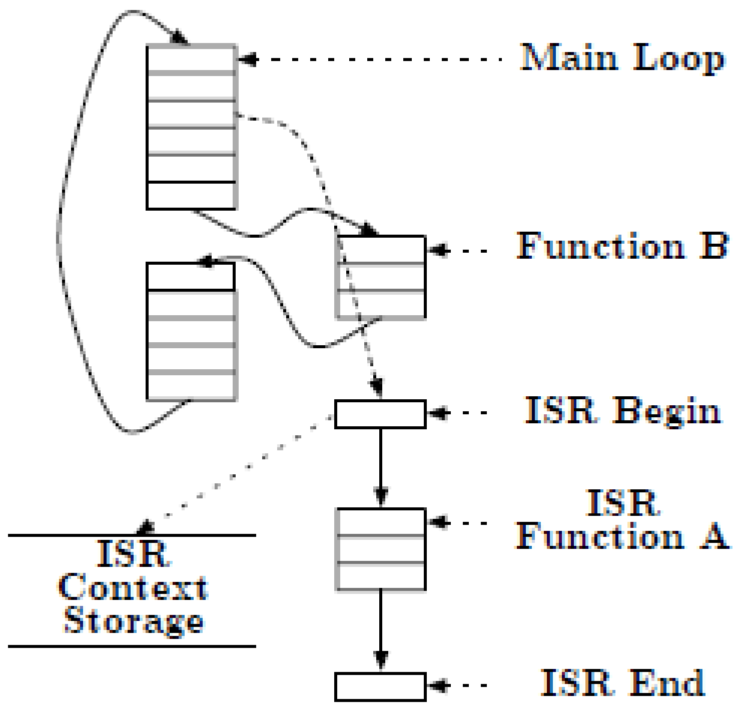
Copied from Lecture 2b, ECE3411 – Fall 2015, by
Marten van Dijk and Syed Kamran Haider

Based on the Atmega328P datasheet and material
from Bruce Land's video lectures at Cornel

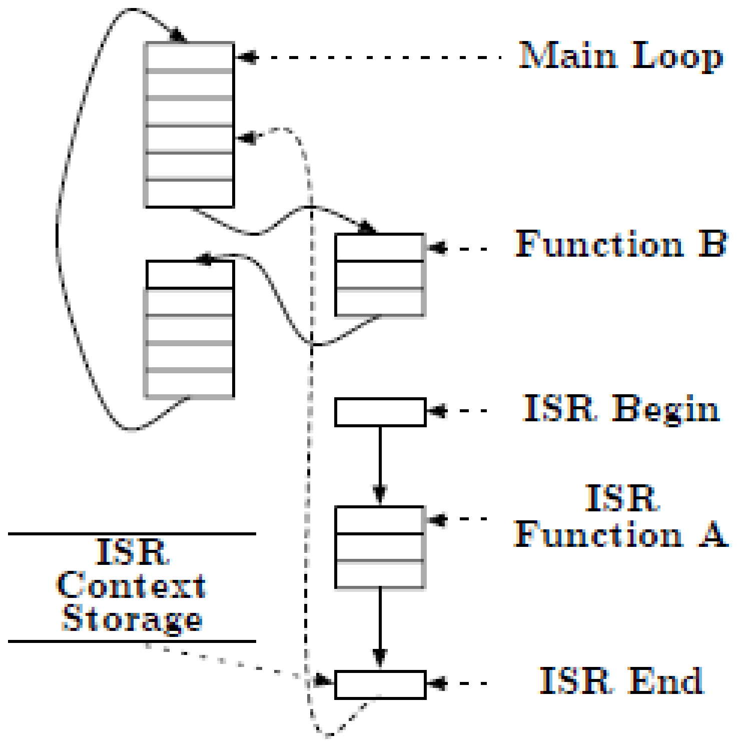
Polling vs Interrupt

	Polling	Interrupt
Operational Sequence	Check the pin status, then execute functions	enable interrupt service routine, then automatically execute functions
Applications	Loose events	Time critical events
Processing Location	Foreground	Background

Interrupt Subsystem



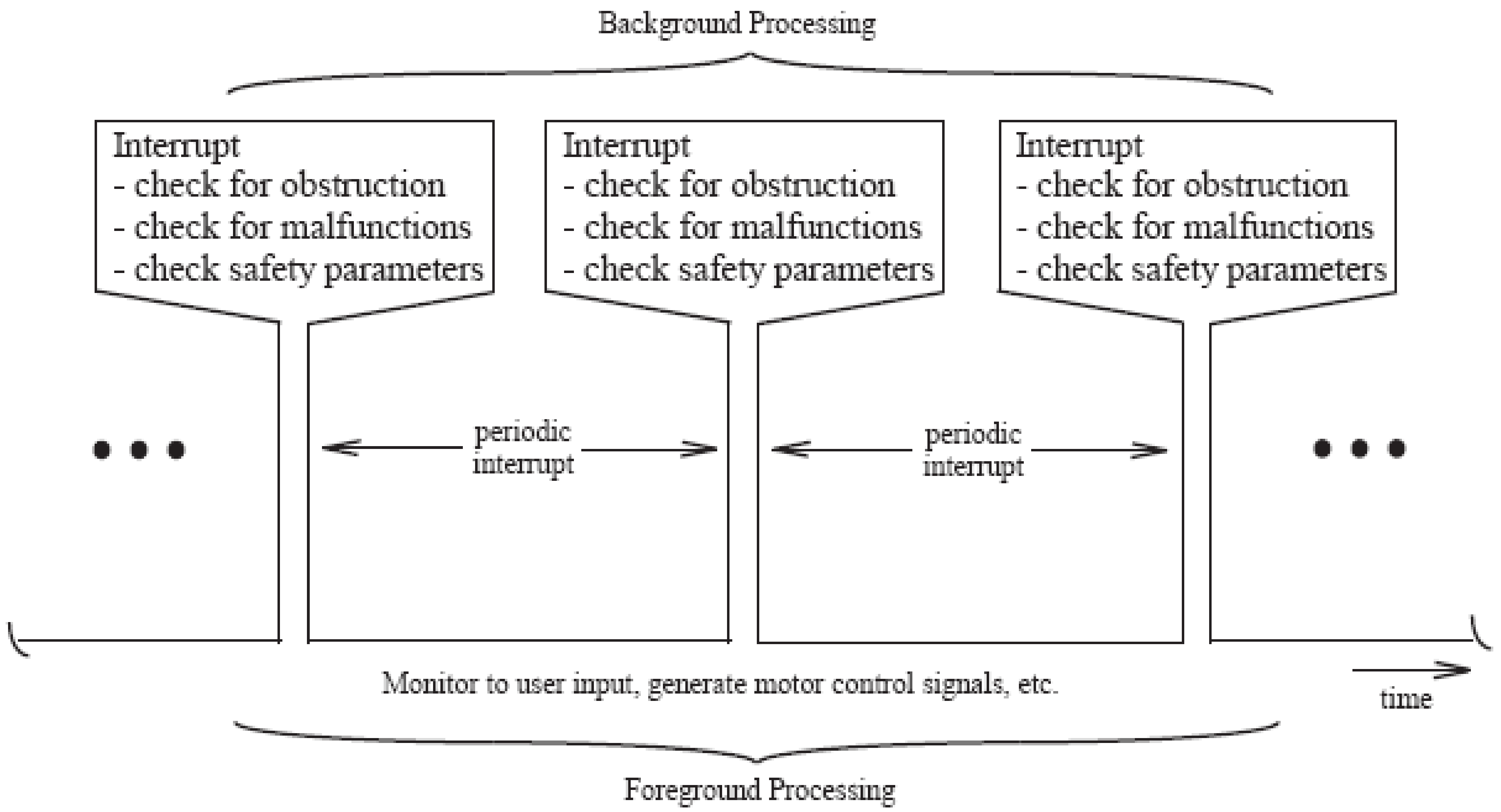
(a) ISR is called



(b) ISR finishes

- Microcontroller handles unscheduled (although planned) higher priority events.
- The program is transitioned into a specific task, organized into a function called an interrupt service routine (ISR).
- Once the ISR is compete, the microcontroller will resume processing where it is left off before the interrupt event occurred.

Foreground / Background processing



Program Layout

- `main()` executes slow code forever → you never exit main in a MCU
  - Interrupt driven tasks are asynchronously called from main, for example:
    - a HW timer may cause a HW event every 1000 cycles, upon which in the corresponding ISR a SW counter is incremented;
    - upon reaching an a-priori defined maximum value, the background code calls a corresponding procedure which executes some task, and upon returning the SW counter is reset to 0
- ISRs have no parameters, no return value, they save CPU state (and C does this for you); they are called by HW events:
  - E.g., the bit value for position RXC0 in UCSR0A goes to high when receiving a character is completed
  - If the mask bit in UCSR0B for position RXCIE0 is set [meaning that an interrupt is enabled for the flag (UCSR0A & (1<<RXC0))], then the MCU will jump to the address of the ISR as indicated by the interrupt vector table for source USART, RX.
  - It takes about 75 cycles to go in and out of an ISR; another 32 cycles to safe state of the MCU (32 registers); another 7/8 cycles overhead.

Program Layout

- Initialization procedure:
  - Set up tables,
  - Initialize timers,
  - Do bookkeeping before you can put on interrupts
  - Turn on the master interrupt bit: This is the I-bit in register SREG, the C-macro `sei()` does this for you

Bit	7	6	5	4	3	2	1	0
	I	T	H	S	V	N	Z	C
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

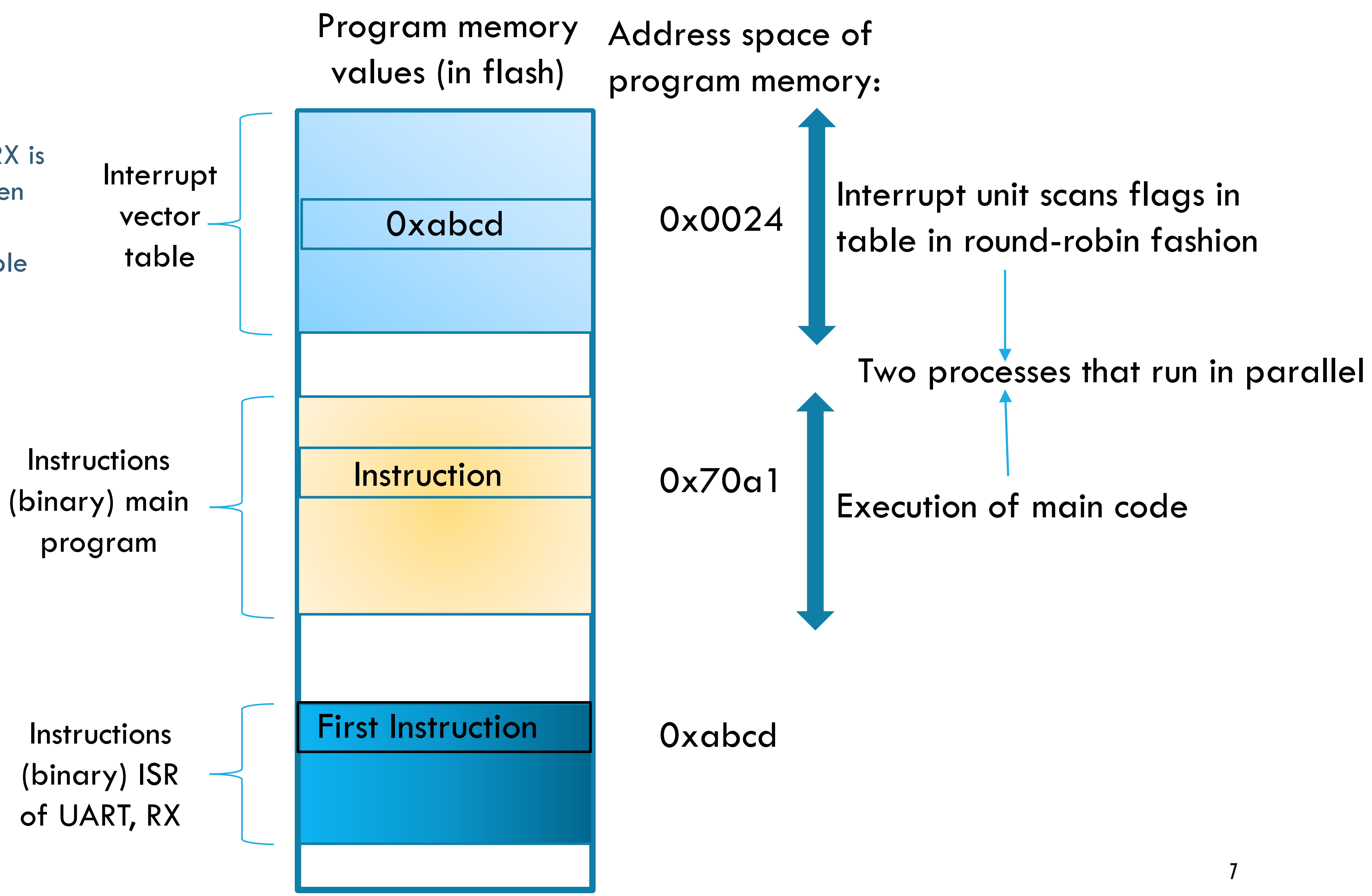
Bit 7 – I Global Interrupt Enable

The global interrupt enable bit must be set for the interrupts to be enabled. The individual interrupt enable control is then performed in separate control registers. If the Global Interrupt Enable register is cleared, none of the interrupts are enabled independent of the individual interrupt enable settings. The I-bit is cleared by hardware after an interrupt has occurred, and is set by the RETI instruction to enable subsequent interrupts. The I-bit can also be set and cleared by the application with the SEI and CLI instructions, as described in the instruction set reference.

Execution of an ISR

1. Character receive complete
→ UART RX HW event which makes flag UCSR0A & (1<<RXC0) non-zero
→ If UCSR0B |= (1<<RXCIE0), i.e., ISR UART RX is enabled, then the interrupt unit sees flag when checking for the UART RX HW event
→ Interrupt unit looks at the Interrupt vector table at position 0x0024 for UART RX, and reads address 0xabcd

2. Program counter (PC) points at 0x70a1, corresponding instruction is executed to completion
→ 0x70a1 is pushed on to the PC stack
→ PC becomes 0xabcd
→ ISR UART RX is executed
→ Upon return from interrupt, the PC stack pops the value 0x70a1
→ PC gets the next address and main program continues its execution



Program memory values (in flash)
Address space of program memory:
Interrupt vector table
0xabcd
0x0024
Interrupt unit scans flags in table in round-robin fashion
Two processes that run in parallel
Instructions (binary) main program
Instruction
0x70a1
Execution of main code
Instructions (binary) ISR of UART, RX
First Instruction
0xabcd

Execution of an ISR

1. Some HW event sets a flag in some register, e.g., (UCSR0A & (1<<RXC0)) goes to high → If the corresponding interrupt is enabled, e.g., by initially programming UCSR0B |= (1<<RXCIE0), then this flag is detected by the Interrupt Unit of the MCU which scans the flags which correspond to the vector table in round robin fashion
2. If the CPU is executing an ISR, then finish the ISR, else finish current instruction
3. Push the Program Counter (PC) on the stack
4. Clear the I-bit in SREG (after this, none of the interrupts are enabled)
5. The CPU jumps to the vector table and clears the corresponding flag: I.e., clear flag in register UCSR0A as in UCSR0A &= ~(1<<RXC0)
6. The CPU jumps to the ISR indicated by the address in the vector table
7. The compiler created a binary which saves state, executes your ISR code, restores state, and returns: return from interrupt (RETI)
8. RETI enables the I-bit in SREG and re-checks the interrupt flag registers in the vector table (since other HW events may have occurred in the meantime)

Interrupts

Lower range of program storage in flash:

Table 15-1. Reset and Interrupt Vectors in ATmega328PB

Vector No	Program Address	Source	Interrupts definition
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2_COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2_COMPB	Timer/Coutner2 Compare Match B
10	0x0012	TIMER2_OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1_CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1_COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1_COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1_OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0_COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0_COMPB	Timer/Coutner0 Compare Match B
17	0x0020	TIMER0_OVF	Timer/Counter0 Overflow
18	0x0022	SPI0 STC	SPI1 Serial Transfer Complete
19	0x0024	USART0_RX	USART0 Rx Complete
20	0x0026	USART0_UDRE	USART0, Data Register Empty
21	0x0028	USART0_TX	USART0, Tx Complete

If you want to set the mask bit of an interrupt, i.e., you enable a certain interrupt, then you ***must*** write a corresponding ISR (interrupt service routine).

The table contains the address of the ISR that you write (upon the HW event that will cause the interrupt, the program counter will jump to the address indicated by the table to execute the programmed ISR).

Program memory has 2^16 registers
→ an address has 16 bits, e.g., 0xabcd
→ 0xabcd is stored in two 8-bit registers
→ Interrupt vector table associates interrupt vectors to addresses 0x0000, 0x0002, 0x0004 etc. (by increments of 2)

Problems

- Example: An ISR with print statement calls the print procedure, which buffers the characters to be printed in HW since printing is slow.
- Now, the HW executes the printing statement in parallel with the rest of the ISR.
- The ISR finishes.
- Before the print statement is finished the ISR is triggered again
- Not even a single character may be printed!!

Problems

- In your ISR you may enable the master interrupt bit → this creates a nested interrupt → not recommended

- Memory of one event deep: e.g.,
  - MCU handles a first flag of (UCSR0A & (1<<RXC0))
  - After clearing this flag, the same HW event happens again which will again set the interrupt flag vector for (UCSR0A & (1<<RXC0)) (which will be handled after the current interrupt)
  - But more interrupts for (UCSR0A & (1<<RXC0)) are forgotten while handling the current interrupt (first flag)!!
  - You need to write ***efficient*** ISR code to avoid missing HW events, which may cause your application to be unreliable.

ISR(USART_RX_vect)

- fscanf uses:

```
int uart_getchar(FILE *stream)
{
    …
    while ( !(UCSR0A & (1<<RXC0)) ) ;
    c = UDR0;
    …
    uart_putchar(c, stream);
    …
}
```

- During the while loop other tasks need to wait → fscanf's implementation is blocking
- Need non-blocking code: write a ISR which waits until the character is there

External Interrupts

- Chapter 15 datasheet
- INT0 & INT1
  - Can be triggered by a falling or rising edge or a low level → EICRA (External Interrupt Control Register A)
  - Low level interrupt is detected asynchronously → can be used to wake from idle mode as well as sleep modes (will see one such example in a forthcoming lecture)
    - If 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. (Start-up time defined by SUT and CKSEL fuses, chapter 8)
- PCINT23..0
  - The pin change interrupt PCI0 will trigger if any enabled PCINT7..0 pin toggles
  - The pin change interrupt PCI1 will trigger if any enabled PCINT14..8 pin toggles
  - The pin change interrupt PCI2 will trigger if any enabled PCINT23..16 pin toggles

Interrupt Vector Table

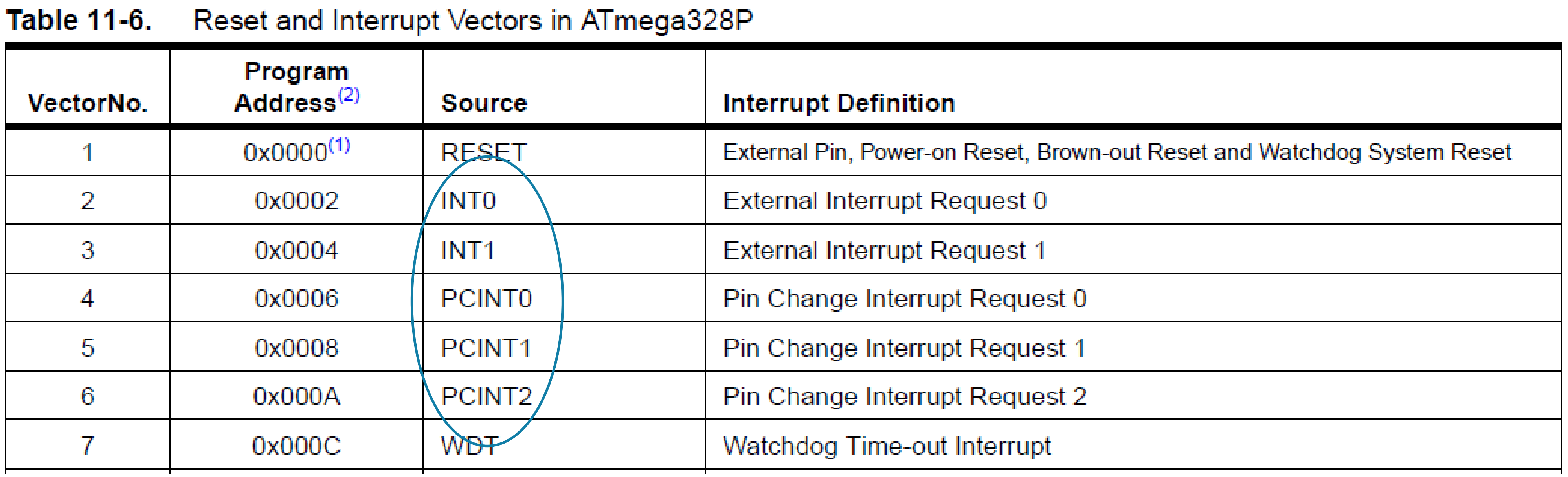


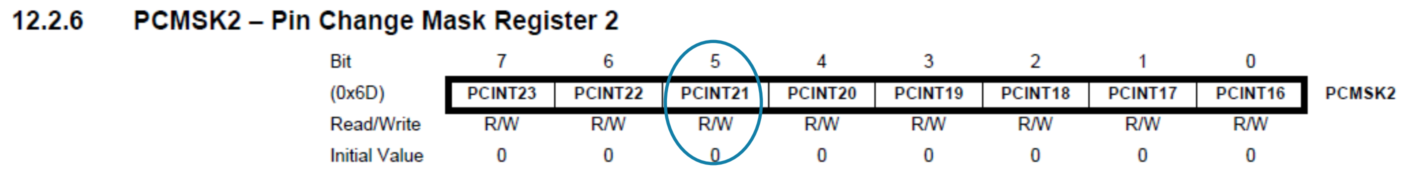
Table 11-6. Reset and Interrupt Vectors in ATmega328P

VectorNo.	Program Address[2]	Source	Interrupt Definition
1	0x0000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt

- Notice that the external interrupts and pin interrupt are at the top of the table
- They will be the first to be checked after an ISR finishes → They have priority
- Usage: Program a SW interrupt for executing an *atomic* piece of code
  - A pin is set as an output
  - Main code toggles the pin
  - This creates a PCINT HW event and sets a corresponding flag
  - Interrupt unit will scan this flag first and prioritizes the corresponding PCINT ISR (i.e., if during toggling another ISR is called due to some other HW event, then once this ISR is finished the PCINT ISR will be called next)
  - The PCINT ISR will be fully executed without interruption → an atomic execution

Example PCINT21 = PD5

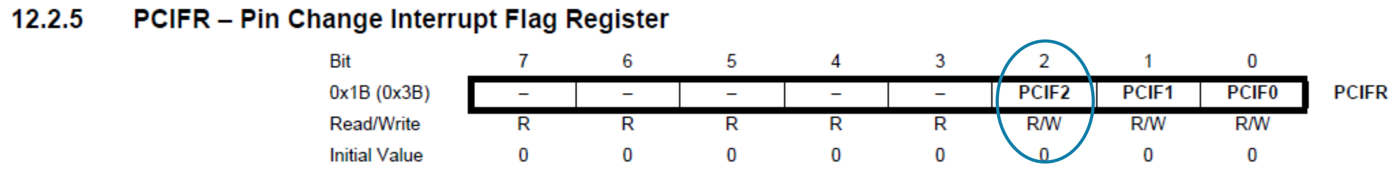
```
DDRD |= (1<<DDD5); //PD5=PCINT21 is output
```



12.2.6 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	

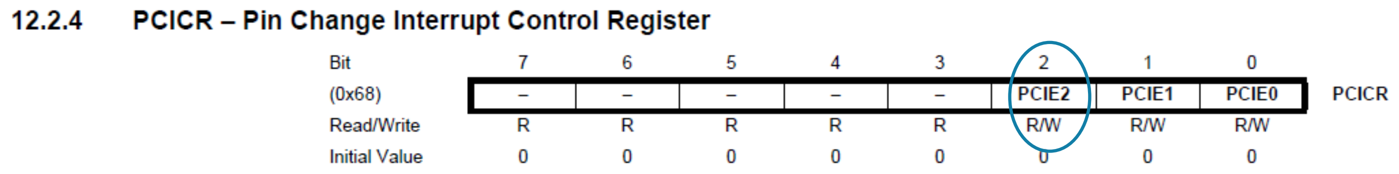
```
PCMSK2 = (1<<PCINT21); //toggling PD5 sets flag
```



12.2.5 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	

When PD5 toggles, flag PCIFR & (1<<PCIF2) changes from 0 (0 as an integer represents 0x00) to (1<<PCIF2) (which, represented as an integer, equals 4)



12.2.4 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	

```
PCICR |= (1<<PCIE2); //Enable interrupt for
                     //PCIFR.PCIF2
```

Write
- ISR(PCINT2_vect){ Atomic code;}
- If the atomic code needs to be executed in the main program, just toggle PORTD ^= (1<<PORTD5);

Sequence of Events

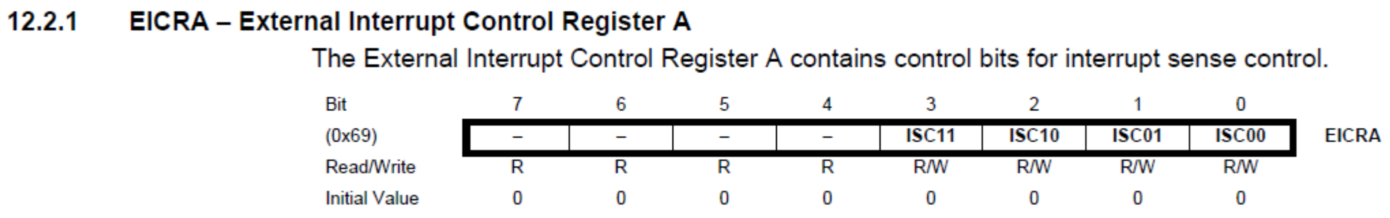
1. In main program toggle PORTD ^= (1<<PORTD5);
2. PCIFR.PCIF2 is set to 1
3. PCICR |= (1<<PCIE2); → Interrupt unit checks PCIFR.PCIF2
4. If currently an ISR is executing, finish its execution and start the next instruction in the main program
5. As soon as the current instruction in the main program is finished, the interrupt unit checks for flags with enabled interrupts
6. The interrupt unit does this in round robin fashion but starts at the top of the interrupt vector table after an ISR is finished → prioritizes RESET over external interrupts over pin interrupts over the rest
7. Looks up address corresponding to ISR(PCINT2_vect), saves register state, puts PC on stack, etc.
8. Execute without any interruption ISR(PCINT2_vect){ Atomic code;}
9. During RETI state is restored, flag PCIFR.PCIF2 is cleared, and PC points to the next instruction in the main program

NOTE: Instead of PORTD ^= (1<<PORTD5); the main code can also directly set PCIFR |= (1<<PCIF2);

INT1

- Programming external interrupt INT1 = PD3 on falling edge
  - Switch connected to PD3 (set to PD3 to input): DDRD &= ~(1<<DDD3);
  - #define SW_PRESSED !(PIND & (1<<PIND3))
  - If SW_PRESSED {…} checks whether PIND & (1<<PIND3) == 0
  - PD3 low means pressed and PD3 high means not pressed: Want to detect falling edge

- EICRA |= (1<<ISC11);



12.2.1 EICRA – External Interrupt Control Register A

The External Interrupt Control Register A 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	

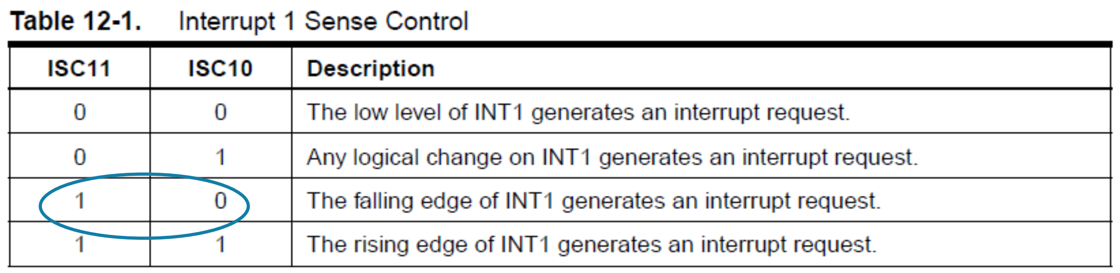
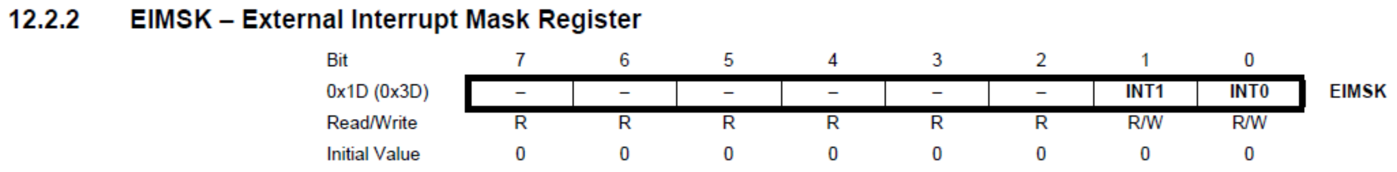


Table 12-1. Interrupt 1 Sense Control

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.

- EIMSK |= (1<<INT1);

Need to write ISR and implement a debounce state machine …



12.2.2 EIMSK – External Interrupt Mask Register

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	

INT1

```
#define SW_PRESSED !(PIND & (1<<PIND3))

void Initialize(void)
{
   DDRD &= ~(1<<DDD3);
   EICRA |= (1<<ISC11);
   EIMSK |= (1<<INT1);
   …. Timer 0 ….
   poll_time = POLLING_DELAY;
   DebounceFlag = 0;
}

void ISR(TIMER0_COMPA_vect)
{
   if ((poll_time>0) && (DebounceFlag==1)) --poll_time;
   …
}

ISR(INT1_vect)
{
   EIMSK &= ~(1<<INT1); // Disable interrupt
   … record this event …
   DebounceFlag = 1;
}
```

```
void PollButton(void)
{
  if SW_PRESSED { … latest recorded event is for a button push …}
  DebounceFlag = 0;
  poll_time = POLLING_DELAY;
  EIMSK |= (1<<INT1);
}

int main(void)
{
   Initialize();
   sei();

   while(1)
   {
      if (poll_time == 0) {PollButton();}
      …
    }
}
```