ECE3411

Lecture 4

External Interrupt

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


DING
DONG
amazon

Programmed I/O

- CPU has direct control over I/O

- Sensing status
- Read/write commands
- Transferring data

```
unsigned char counter;

while (1) {
   ...
   //button push of the switch connected to PINB7
   if (!(PINB & (1 << PINB7))) {
      counter++;
            _delay_ms(1000);
   ...
}
```

How to resolve the issue?

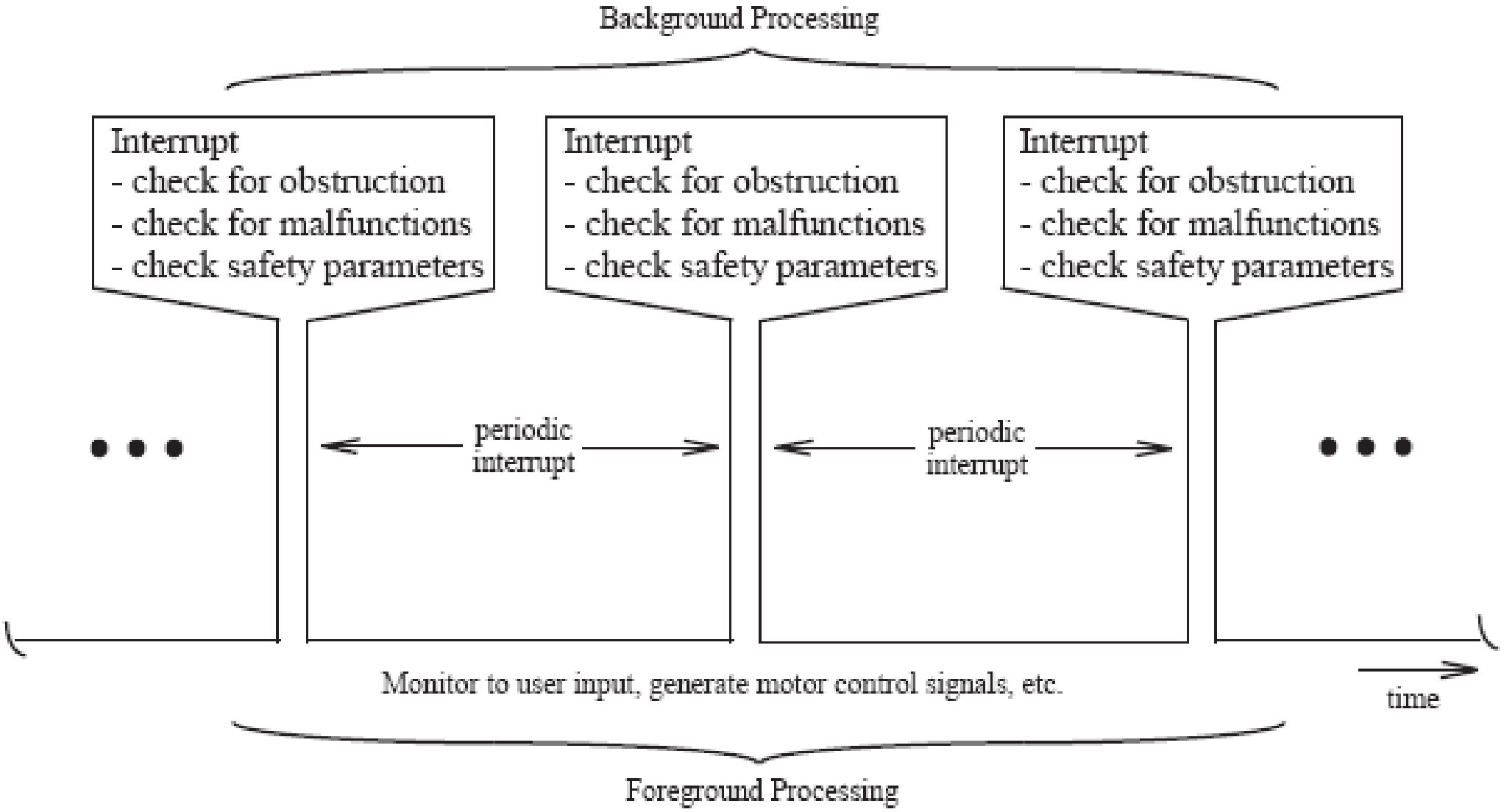
- CPU waits for I/O module to complete operation

- Wastes CPU time

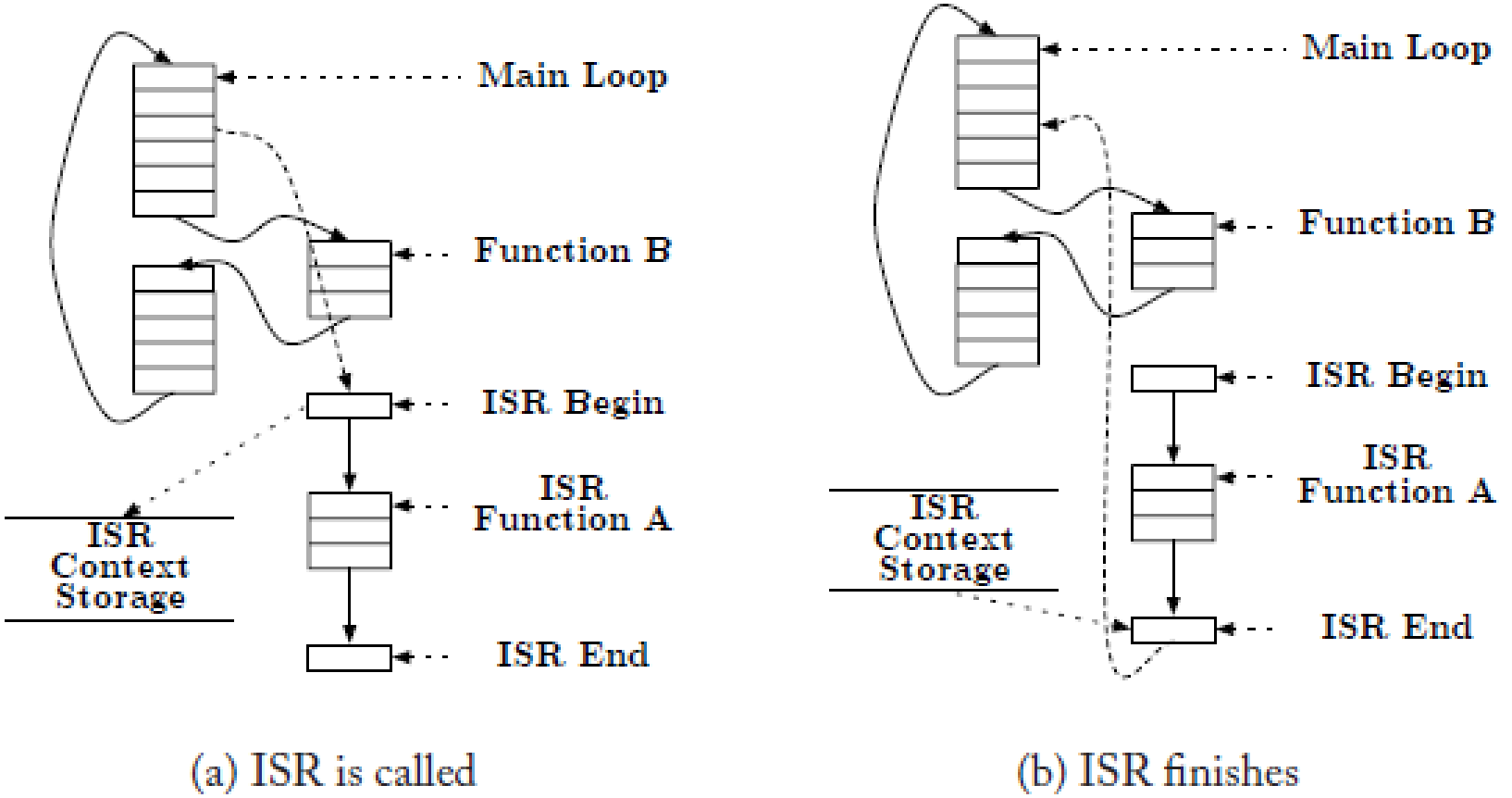
Polling vs Interrupt

	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

Foreground / Background processing



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.

Register for Interrupt Enable

- Initialization procedure:
  - Set up tables,
  - Initialize external interrupt, Pin Change interrupt, or Timer interrupt
  - 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. Turn off the mast interrupt bit: the C-macro cli();

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

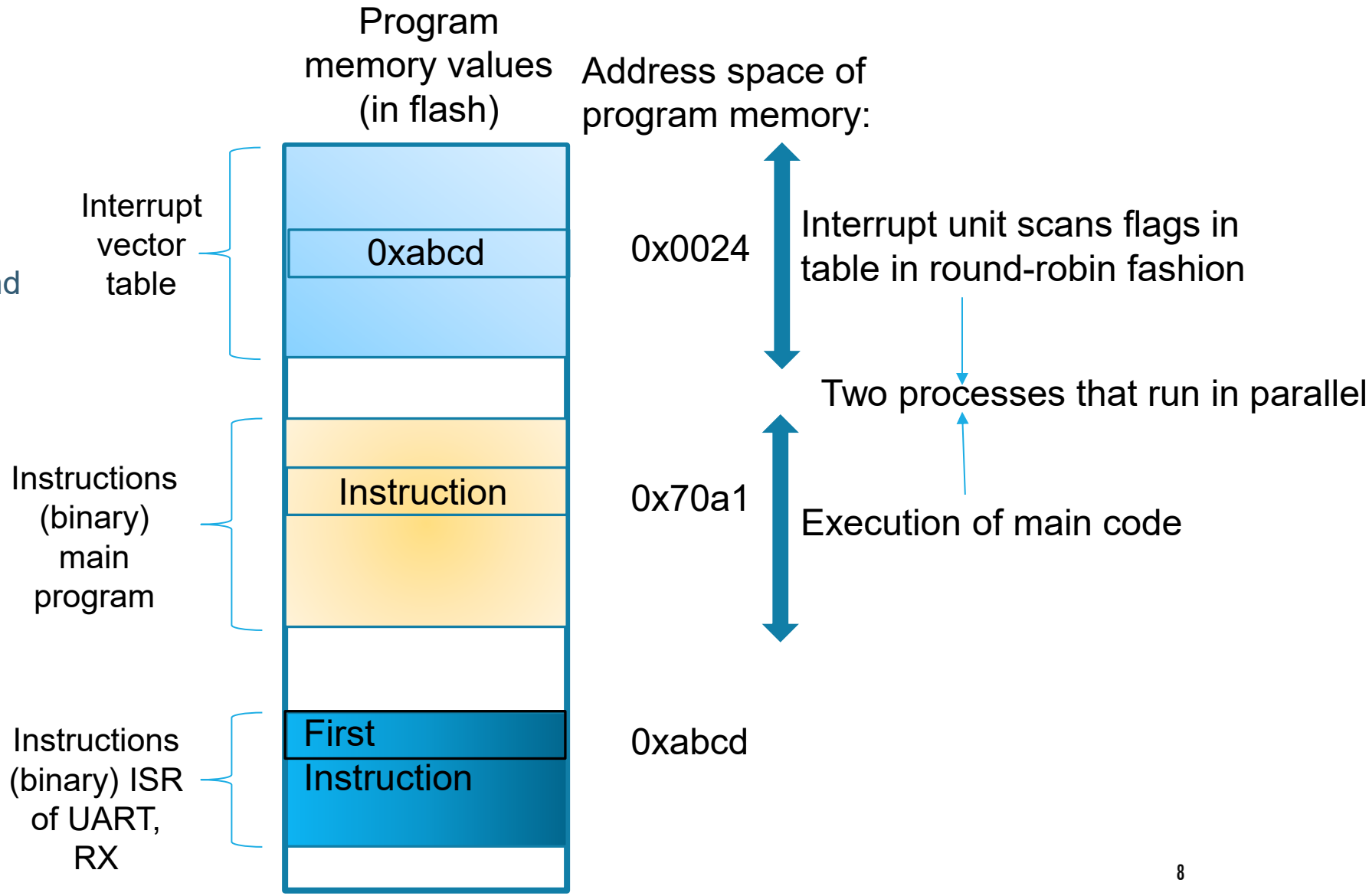
Bit 7 – I Global Interrupt Enable

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. HW Interrupt event which makes to be enabled, then the interrupt unit sees flag when checking for the HW event
→ Interrupt unit looks at the Interrupt vector table at position 0x0024 , 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 is executed
→ Upon return from interrupt, the PC stack pops the value 0x70a1
→ PC gets the next address and main program continues its execution



Interrupt Vector Table

Table 10-3. Interrupt Vector Mapping

Vector number	Program Address (word)	Peripheral Source (name)	Description
0	0x00	RESET	
1	0x02	NMI	Non-Maskable Interrupt available for: • CRCSCAN • CFD
2	0x04	BOD_VLM	Voltage Level Monitor Interrupt
3	0x06	CLKCTRL_CFD	External crystal oscillator/clock source failure Interrupt
4	0x08	MVIO_MVIO	Multi-Voltage Input/Output Interrupt
5	0x0A	RTC_CNT	Real-Time Counter Overflow or Compare Match Interrupt
6	0x0C	RTC_PIT	Real-Time Counter Periodic Interrupt
7	0x0E	CCL_CCL	Configurable Custom Logic Interrupt
8	0x10	PORTA_PORT	PORT A External Interrupt
9	0x12	TCA0_OVF TCA0_LUNF	Normal: Timer/Counter Type A Overflow Interrupt Split: Timer/Counter Type A Low Underflow Interrupt
10	0x14	TCA0_HUNF	Normal: Unused Split: Timer/Counter Type A High Underflow Interrupt
11	0x16	TCA0_CMP0 TCA0_LCMP0	Normal: Timer/Counter Type A Compare 0 Interrupt Split: Timer/Counter Type A Low Compare 0 Interrupt

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 interrupt vector table is at the beginning of program memory and each location contains a jump instruction to 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).

Registers related to External Interrupts

•PORTx.DIRCLR: Set pin as input.

•PORTx.PINnCTRL: Configure interrupt sense control (ISC bits).

•PORTx.INTFLAGS: Clear interrupt flags.

•PORTx.INTCTRL: Set interrupt level (low, medium, high).

•ISR(PORTx_PORT_vect): Interrupt service routine for the port.

•Configuration overview:

- Set pin as input.

- Configure interrupt sense control using PORTx.PINnCTRL register.

- Enable interrupt level using PORTx.INTCTRL.

- Write an ISR for the corresponding PORTx_PORT_vect.

Input Sense Configuration (ISC)

18.5.12 Multi-Pin Configuration

Bit	7	6	5	4	3	2	1	0
	INVEN	INLVL			PULLUPEN	ISC[2:0]		
Access	R/W	R/W			R/W	R/W	R/W	R/W
Reset	0	0			0	0	0	0

Bits 2:0 – ISC[2:0] Input/Sense Configuration

This bit field controls the input and sense configuration of pin n. The sense configuration determines how to trigger a port interrupt.

Value	Name	Description
0x0	INTDISABLE	Interrupt disabled but digital input buffer enabled
0x1	BOTHEDGES	Interrupt enabled with sense on both edges
0x2	RISING	Interrupt enabled with sense on rising edge
0x3	FALLING	Interrupt enabled with sense on falling edge
0x4	INPUT_DISABLE	Interrupt and digital input buffer disabled(1)
0x5	LEVEL	Interrupt enabled with sense on low level(2)
other	—	Reserved

Notes:

1. If the digital input buffer for pin n is disabled, bit n in the Input Value (PORTx.IN) register will not be updated.
2. The LEVEL interrupt will keep triggering continuously as long as the pin stays low.

External Interrupt Flag

Interrupt Flags

Bit	7	6	5	4	3	2	1	0
	INT[7:0]							
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

Bits 7:0 – INT[7:0] Pin Interrupt Flag

Pin Interrupt Flag n is cleared by writing a '1' to it.

Pin Interrupt Flag n is set when the change or state of pin n (Pxn) matches the pin's Input/Sense Configuration (ISC) in PORTx.PINnCTRL.

Writing a '0' to bit n in this bit field has no effect.

Writing a '1' to bit n in this bit field will clear Pin Interrupt Flag n.

<Interrupt.h>

```
// interrupt.h - AVR Interrupt Handling

#ifndef _AVR_INTERRUPT_H_

#define _AVR_INTERRUPT_H_

#include <avr/io.h>


#ifdef __cplusplus

extern "C" {

#endif

// Enable global interrupts

#define sei()  __asm__ __volatile__ ("sei" ::: "memory")

// Disable global interrupts

#define cli()  __asm__ __volatile__ ("cli" ::: "memory")

// Macro to define an ISR

#define ISR(vector, ...)            \

    void vector (void) __attribute__ ((signal, used,
externally_visible)) __VA_ARGS__; \

    void vector (void)

// Interrupt Vector Definitions (AVR128DB48-specific)

#define PORTA_PORT_vect    _VECTOR(1)  // Pin Change
Interrupt for PORTA

#define PORTB_PORT_vect    _VECTOR(2)  // Pin Change
Interrupt for PORTB

#define PORTC_PORT_vect    _VECTOR(3)  // Pin Change
Interrupt for PORTC

#define INT0_vect          _VECTOR(4)  // External Interrupt 0

#define INT1_vect          _VECTOR(5)  // External Interrupt 1

#ifdef __cplusplus

}

#endif

#endif /* _AVR_INTERRUPT_H_ */
```

External Interrupt Trigger Modes

Each external interrupt pin can e configured to trigger on different conditions:

Interrupt Trigger Mode	Description	PORT_ISC Setting
Low Level	Triggers when the pin stays LOW.	PORT_ISC_LEVEL_gc
Rising Edge	Triggers when the pin changes from LOW to HIGH.	PORT_ISC_RISING_gc
Falling Edge	Triggers when the pin changes from HIGH to LOW.	PORT_ISC_FALLING_gc
Both Edges	Triggers on both rising and falling edges.	PORT_ISC_BOTHEDGES_gc
Input Buffer Disabled	Disables the input buffer (No interrupt).	PORT_ISC_INPUT_DISABLE_gc

External Interrupt Example

Here is an example **C program** for the **AVR128DB48,** demonstrating how to use a External Interrupt. This example configures a **pin change interrupt on PORTA pin 2 (PA2),** so when a button is pressed, an LED on **PORTB pin 5 (PB5)** toggles.

Features:

- Uses **PA2** as the interrupt pin.
- Configures **PB5** as an LED output.
- Triggers an interrupt when PA2 changes state (**rising or falling edge**).
- The **ISR toggles the LED** on PB5.
- **Interrupt flag is cleared** in the ISR.

How It Works

- **PA2 is configured** as an input with a pin change interrupt on **both edges (rising & falling).**
- **PB5 is set** as an output for an LED.
- When PA2 **changes state** (HIGH to LOW or LOW to HIGH), an interrupt is triggered.
- The **ISR toggles the LED** state on PB5.
- The **interrupt flag is cleared** inside the ISR.

PA2	Button Input (with pull-up)
PB5	LED Output

External Interrupt

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

// Interrupt Service Routine for PORTA (Pin Ch Interrupt)
ISR(PORTA_PORT_vect)
{
    if (PORTA.INTFLAGS & PIN2_bm) {
// Check if PA2 caused the interrupt
        PORTB.OUTTGL = PIN5_bm;
// Toggle PB5 (LED)
        PORTA.INTFLAGS = PIN2_bm;
// Clear interrupt flag
    }
}

void Ext_Int_Init()
{
    PORTA.DIRCLR = PIN2_bm;          // Set PA2 as input
    PORTA.PIN2CTRL = PORT_ISC_BOTHEDGES_gc;
// Interrupt on both rising & falling edges

    sei();  // Enable global interrupts
}
```

```
void LED_Init()
{
    PORTB.DIRSET = PIN5_bm;          // Set PB5 as output
    PORTB.OUTCLR = PIN5_bm;          // Start with LED off
}

int main(void)
{
    LED_Init();
    Ext_Int_Init();

    while (1) {
        // Main loop does nothing, ISR handles LED toggling
    }
}
```

Variable for Interrupt Problems

- Compiler optimizations

```
char flag = 0;
int main() {
    while (1) {
        ...
        if (flag == 1) {
            flag = 0;
            ...
        }
    }
}

ISR(PORTA_PORT_vect){
    flag = 1; // set flag to let main know
}
```

- Common use pattern is that the `ISR` sets a flag that tells main to do something
- When the compiler sees this code, it assumes that `flag` will only change in the `main` function and it puts the `flag` value in a register
- The `ISR` changes the `flag` variable in memory, and since the `main` function is looking at the `flag` variable in a register, it doesn't see the change

Use volatile for interrupt

- Compiler optimizations

```
volatile char flag = 0;
int main() {
    while (1) {
        ...
        if (flag == 1) {
            flag = 0;
            ...
        }
    }
}

ISR(PORTA_PORT_vect) {
    flag = 1; // set flag to let main know
}
```

- `volatile` keyword tells compiler to always keep the variable in memory
- Good practice to use the `volatile` keyword for any variables that are shared between `ISR` and `main` code

Problem: Race Conditions

- Race Conditions
  - Interrupts can happen at any time including in the middle of an operation

```
int main() {
    if (flag == 0) {
        flag = 1;
        // do something that should be done once
    }
}
```

  - What if ISR has similar code?

Problems

```
char flag = 0;
int main() {
    if (flag == 0) {
        flag = 1;
        // one-time code
    }
}

ISR() {
    if (flag == 0) {
        flag = 1;
        // one-time code
    }
}
```

1. `main` function runs and sees that `flag` is 0
2. `main` moves into `if` but is interrupted before `flag` is set to 1
3. `ISR` runs and sees that `flag` is 0
4. `ISR` moves into `if` and sets `flag` to 1
5. `ISR` runs ”one-time-code”
6. `ISR` finishes and control returns to `main`
7. `main` sets `flag` to 1
8. `main` runs “one-time-code”

Problems

- More Race Conditions – “test and set”

```
char flag = 0;
int main() {
    while (1)
        if (flag == 1) {
            flag = 0;
            // code
        }
    }
}

ISR() {
    flag = 1;
}
```

1. `ISR` runs and sets `flag` to 1
2. `main` function runs and sees that `flag` is 1
3. `main` moves into `if` but is interrupted before `flag` is set to 0
4. `ISR` runs again and sets `flag` to 1
5. `ISR` finishes and control returns to `main`
6. `main` sets `flag` to 0
7. `main` misses the fact that `flag` was set again

Problems

- Solution – turn off interrupts while testing and setting

```
char flag = 0;
int main() {
    while (1)
       if (flag == 1) {
          cli(); //Disable global interrupts
          if (flag == 1) { flag = 0; }
          sei(); // Enable global interrupts
          // code
       }
    }
}

ISR() {
    flag = 1;
}
```

Clears the global interrupt enable bit (I-bit) in the Status Register (SREG).

Set the global interrupt enable bit (I-bit) in the Status Register (SREG).

Lab Practice #3: External Interrupt and GPIO

- Two buttons will be used. One is the on-board button1 (PB2) and the other is the external button2 connect to PB5 and set up both as input.
- Set up PB2 and PB5 for External Interrupt.
- Connect 8 LEDs to PD 0 ~ 7 and set up as output.
- Blink all 8 LEDs one after another with the frequency of 4Hz.
- When Button 0 is pressed, LED blinks at 8Hz.
- When Button 1 is pressed, LED blinks at 1Hz.