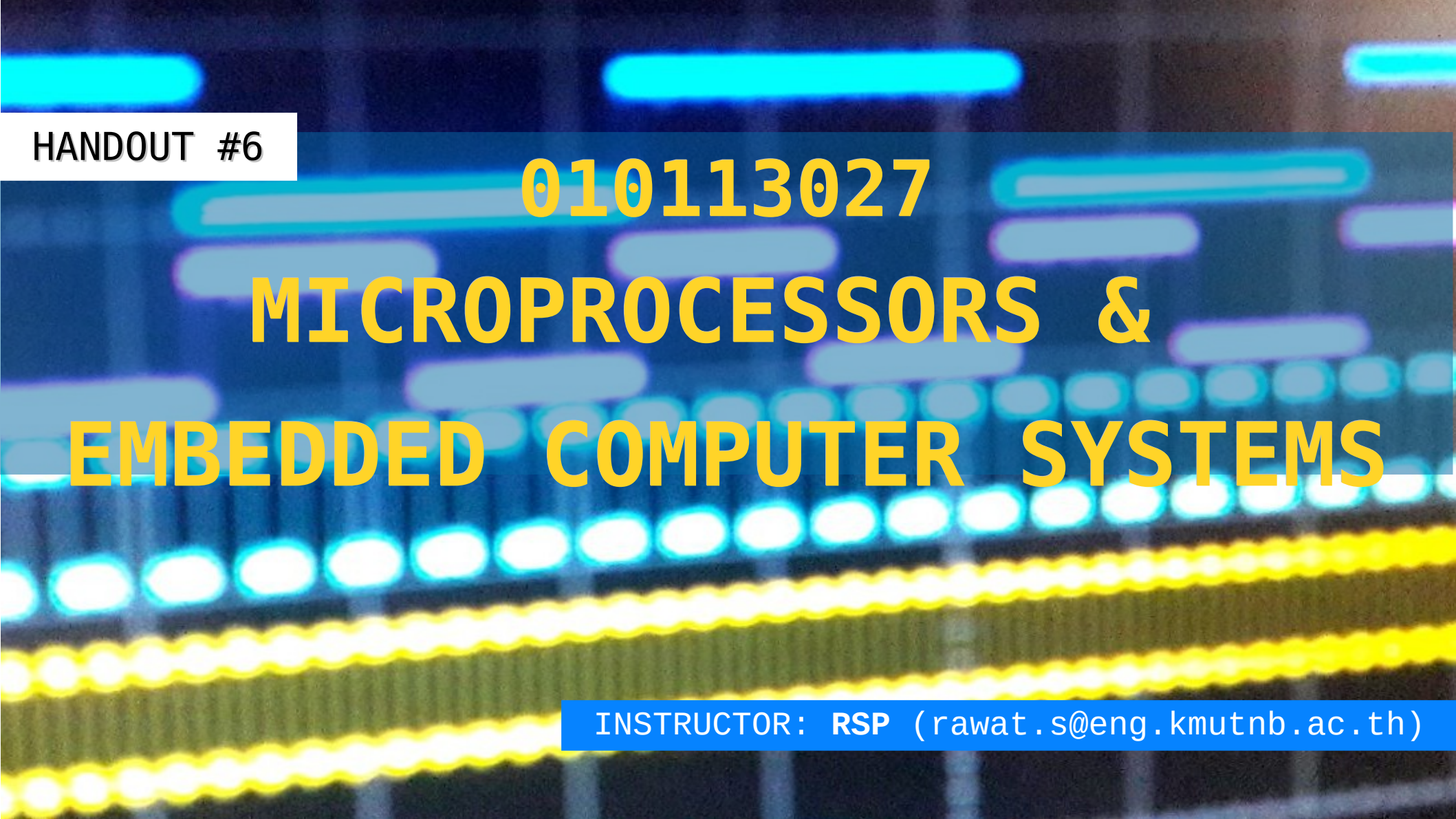


HANDOUT #6

010113027
MICROPROCESSORS & EMBEDDED COMPUTER SYSTEMS

INSTRUCTOR: **RSP** (rawat.s@eng.kmutnb.ac.th)

Terminology

- Interrupt
- Interrupt Service Routine (ISR)
- Interrupt Source
- Interrupt Vector
- Interrupt Vector Address
- Interrupt Vector Table
- Global interrupt enable / disable
- Maskable Interrupt
- Non-maskable Interrupt
- External vs. Internal Interrupt
- Interrupt Priority
- Interrupt Latency
- Interrupt Pending
- Interrupt Flag
- Nested Interrupt
- Context Switching

Interrupts

- An **interrupt** is a **hardware or software signal** that temporarily stops normal program execution and transfers control to a special function to handle an event.
- An **interrupt** is a special **event** or **exception** that occurs outside the normal execution flow of an MCU and requires immediate attention from the MCU, a process called **interrupt handling** or **servicing**.
- Interrupts typically originate from hardware, either **internal** (within the MCU) or **external** (from connected devices). They can be thought of as a request from hardware to the MCU to execute a predefined function, known as an **Interrupt Service Routine (ISR)**.
- When an interrupt occurs, the MCU temporarily pauses the current program execution, switches to execute the ISR until completion, and then resumes the interrupted program from where it left off.

Advantages of interrupts:

- Fast response to events (fast interrupt response)
- Reduced CPU idle time compared to polling

Terminology

ISR
- An ISR is a special function that executes automatically in response to an interrupt and handles the event.

Interrupt Source
- An interrupt source is the origin of an interrupt, meaning what caused the CPU to be interrupted.

Interrupt Vector
- An interrupt vector is an identifier that tells the CPU which ISR to execute for a given interrupt source.

Interrupt Vector Address
- The interrupt vector address is the memory address where execution jumps when an interrupt occurs.

Interrupt Vector Table
- The interrupt vector table is an array of interrupt vector addresses used by the CPU to dispatch ISRs.

Terminology

Global Interrupt Enable / Disable
- A global mechanism that enables or disables all maskable interrupts.

Maskable Interrupts
- Interrupts that can be disabled by software

Non-maskable Interrupts
- Interrupts that cannot be disabled

External Interrupts
- Interrupts that are triggered by external signals

Internal Interrupts
- Interrupts that are triggered by on-chip peripherals

Interrupt Priority
- Interrupt priority determines which interrupt is serviced first if multiple requests occur

Interrupt Pending
- An interrupt is pending if the event has occurred but has not yet been serviced

Terminology

Interrupt Flag

- An interrupt flag is a status bit that indicates an interrupt condition has occurred.

Interrupt Latency

- Interrupt latency is the time from an interrupt request to execution of the first ISR instruction.

Nested Interrupt

- A nested interrupt occurs when an interrupt is allowed to interrupt another ISR.

Context Switching for Interrupt

- Context switching is the process of saving and restoring CPU state when entering and leaving an interrupt.

Difference Between Polling and Interrupts

- **Polling** is a method where the CPU continuously loops to check for a specific event before proceeding. While waiting, the CPU cannot perform other tasks.
- **Example**:
  - Button press detection: With polling, the program repeatedly reads the input pin and checks whether the logic level is LOW (for an active-low buttons).
- **Interrupt-driven operation**, on the other hand, allows the MCU to wait for events using hardware mechanisms, rather than constantly using the CPU to check.
  - When the event occurs (an interrupt), the hardware notifies the CPU to respond.
  - If no interrupt occurs, the CPU can continue executing the main program normally.

Interrupt latency is the time delay between the occurrence of an **interrupt request (event)** and the start of execution of the corresponding **Interrupt Service Routine (ISR)**.

Interrupt Latency

Factors that determine **interrupt latency** in **AVR** (e.g., ATmega328P):

Current instruction execution

- The CPU must finish executing the current instruction before jumping to the ISR.
- Some instructions take multiple cycles, so longer instructions can increase latency.

Interrupt enable status

- If the global interrupt flag (the **I-bit in SREG**) is cleared (interrupts disabled), incoming interrupts are delayed until interrupts are enabled.

Nested interrupts

- If an ISR is already running and global interrupts are disabled, a new interrupt must wait, adding to latency.

Context saving overhead

- Before executing the ISR, the MCU pushes the program counter (PC) and other registers onto the stack.
- This saving and restoring adds several clock cycles to the latency.

Interrupt Latency of ATmega328P

- The **interrupt execution response** for all the enabled AVR interrupts is **four clock cycles minimum**. During this period, the **program counter (PC)** is **pushed** onto the **stack**.
- After four clock cycles the program vector address for the actual **interrupt handling routine (or ISR)** is executed. This jump takes **three clock cycles**.
- If an interrupt occurs during execution of a **multi-cycle instruction**, this instruction is completed before the interrupt is served.
- If an interrupt occurs when the MCU is in **sleep mode**, the interrupt execution response time is increased by **four clock cycles**. This increase comes in addition to the start-up time from the selected sleep mode.
- A **return from an interrupt handling routine** (executing the **RETI** instruction) takes **four clock cycles**. During this period, the program counter (two bytes) is popped back from the stack, the stack pointer is incremented by two, and the **I-bit in SREG** is set (**Global Interrupt Enable or GIE**).

Interrupt Source of AVR

AVR Interrupt Sources: An interrupt is generated by:

1) **Peripherals** (timer overflow, UART, ADC, etc.)

- External pin change
- Timer overflow
- UART receive or transmit complete
- ADC conversion complete

2) **External pins** (INTx, PCINT)

3) **Internal conditions** (Watchdog, System Reset, BOR)

In AVR, it is possible to **enable or disable all maskable interrupts** by the **I-bit in SREG**:

- `sei();` // set I-bit (enable interrupts0
- `cli();` // clear I-bit (disable interrupts)

ATmega328P: Interrupt Vector Table

- Since interrupts can originate from multiple sources, each **interrupt source** has a corresponding **interrupt vector**, which is the program address where its **ISR** begins.
- The **interrupt vectors** start at address **0x0000 (Program Address)** and continue sequentially, forming the **Interrupt Vector Table**.
  - **Address 0x0000**: Interrupt vector for **System Reset**, which has the **highest priority**.
  - **Address 0x0002**: Interrupt vector for **External Interrupt Request 0 (INT0)**
    Address 0x0004: Interrupt vector for **External Interrupt Request 1 (INT1)**
- The priority of interrupts decreases as the vector addresses increase.
- Since multiple interrupts can occur simultaneously, priority management is required to determine which interrupt the MCU services first.

ISR for AVR-GCC

- Each ISR is a C function with no arguments, no return value, defined with:

  `ISR(vector_name) {…}`

- The ISR function names for AVR-GCC is pre-defined. Examples of AVR interrupt names:
  - `INT0_vect`
  - `PCINT2_vect`
  - `TIMER0_OVF_vect`

- On AVR microcontrollers:
  - The **global interrupt enable (I-bit)** is automatically cleared on ISR entry.
  - No other interrupt can run unless interrupts are re-enabled manually using `sei()`.

```
ISR(TIMER0_OVF_vect) {
  // for timer 0 overflow
  // ...
}
```

ATmega328P: Interrupt Vector Table

Vector No.	Program Address	Source	Interrupt Definition
1	0x0000	RESET	External pin, power-on reset, brown-out reset and watchdog system reset
2	0x002	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/Counter2 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/Counter1 compare match B
14	0x001A	TIMER1 OVF	Timer/Counter1 overflow

ATmega328P: Interrupt Vector Table

Vector No.	Program Address	Source	Interrupt Definition
15	0x001C	TIMER0 COMPA	Timer/Counter0 compare match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 compare match B
17	0x0020	TIMER0 OVF	Timer/Counter0 overflow
18	0x0022	SPI, STC	SPI serial transfer complete
19	0x0024	USART, RX	USART Rx complete
20	0x0026	USART, UDRE	USART, data register empty
21	0x0028	USART, TX	USART, Tx complete
22	0x002A	ADC	ADC conversion complete
23	0x002C	EE READY	EEPROM ready
24	0x002E	ANALOG COMP	Analog comparator
25	0x0030	TWI	2-wire serial interface
26	0x0032	SPM READY	Store program memory ready

ATmega328P: Interrupt Vector Table

The most typical and general program setup for the reset and interrupt vector addresses in Atmel® ATmega328P is:

```
Address     Labels Code                 Comments
0x0000              jmp    RESET        ; Reset Handler
0x0002              jmp    EXT_INT0     ; IRQ0 Handler
0x0004              jmp    EXT_INT1     ; IRQ1 Handler
0x0006              jmp    PCINT0       ; PCINT0 Handler
0x0008              jmp    PCINT1       ; PCINT1 Handler
0x000A              jmp    PCINT2       ; PCINT2 Handler
0x000C              jmp    WDT          ; Watchdog Timer Handler
0x000E              jmp    TIM2_COMPA   ; Timer2 Compare A Handler
0x0010              jmp    TIM2_COMPB   ; Timer2 Compare B Handler
0x0012              jmp    TIM2_OVF     ; Timer2 Overflow Handler
0x0014              jmp    TIM1_CAPT    ; Timer1 Capture Handler
0x0016              jmp    TIM1_COMPA   ; Timer1 Compare A Handler
0x0018              jmp    TIM1_COMPB   ; Timer1 Compare B Handler
0x001A              jmp    TIM1_OVF     ; Timer1 Overflow Handler
0x001C              jmp    TIM0_COMPA   ; Timer0 Compare A Handler
0x001E              jmp    TIM0_COMPB   ; Timer0 Compare B Handler
0x0020              jmp    TIM0_OVF     ; Timer0 Overflow Handler
0x0022              jmp    SPI_STC      ; SPI Transfer Complete Handler
0x0024              jmp    USART_RXC    ; USART, RX Complete Handler
0x0026              jmp    USART_UDRE   ; USART, UDR Empty Handler
0x0028              jmp    USART_TXC    ; USART, TX Complete Handler
0x002A              jmp    ADC          ; ADC Conversion Complete Handler
0x002C              jmp    EE_RDY       ; EEPROM Ready Handler
0x002E              jmp    ANA_COMP     ; Analog Comparator Handler
0x0030              jmp    TWI          ; 2-wire Serial Interface Handler
0x0032              jmp    SPM_RDY      ; Store Program Memory Ready Handler
;
0x0033      RESET: ldi    r16, high(RAMEND); Main program start
0x0034              out    SPH,r16      ; Set Stack Pointer to top of RAM
0x0035              ldi    r16, low(RAMEND)
0x0036              out    SPL,r16
0x0037              sei                 ; Enable interrupts
0x0038      <instr> xxx
...    ...  ...    ...
```

JMP Instruction:

- Instruction Length:32 bits (2 words)
- Clock Cycles: 3

Program Counter

- 14 bits

Stack Pointer (SPH:SPL)

- 16 bits

ATmega328P: Reset Vector and Interrupt Vectors

BOOTRST	IVSEL	Reset Address	Interrupt Vectors Start Address
1	0	0x000	0x002
1	1	0x000	Boot reset address + 0x0002
0	0	Boot reset address	0x002
0	1	Boot reset address	Boot reset address + 0x0002

1. For the BOOTRST fuse "1" means unprogrammed while "0" means programmed.

MCUCR – MCU Control Register

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

```
BOOTRST  IVSEL  Reset jumps to                    Interrupt vectors at
-------------------------------------------------------------------------
1        0      Application start (0x0000)       0x0002
1        1      Application start (0x0000)       Boot section start + 0x0002
0        0      Boot section start               0x0002
0        1      Boot section start               Boot section start + 0x0002
```

Arduino Bootlader: BOOTRST=0 (programmed), Arduino sketches always run with IVSEL = 0.

ATmega328P: Reset Vector and Interrupt Vectors

The **ATmega328P** decides where to start executing code after a reset and where the interrupt vectors are located in memory, depending on two important bits: **BOOTRST** (**Boot Reset**) and **IVSEL** (**Interrupt Vector Select**).

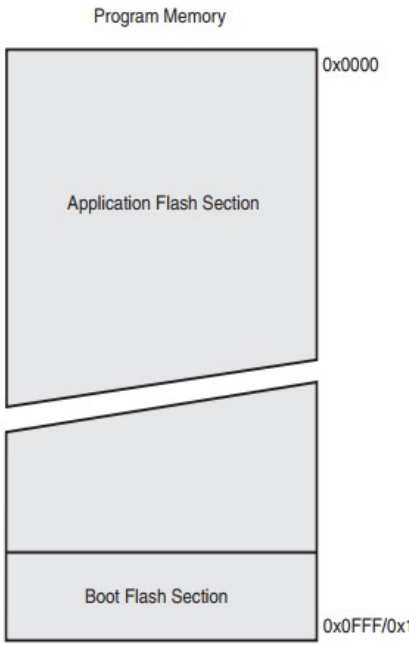
- The **Reset Vector** is the address the MCU jumps to when it starts up or is reset.
- **Interrupt Vectors** are addresses where the MCU jumps when a specific interrupt occurs.

The **BOOTRST fuse** determines where the **MCU** starts after reset.

- **BOOTRST = 0** (programmed): Reset goes to the **Boot Section** start address.
- **BOOTRST = 1** (unprogrammed): Reset goes to the **Application Section** start address (0x0000).

The **IVSEL bit** is in the **MCU Control Register** (**MCUCR**).

- **IVSEL = 0**: Interrupt vectors are at the start of the **Application Section** (0x0000 or 0x0000 + boot offset if **BOOTRST=0**)
- **IVSEL = 1**: Interrupt vectors are remapped to the **Boot Section**.



ATmega328P: Arduino Uno with Optiboot

- **Arduino Uno or Nano** always reserves 1024 bytes (512 words) for the boot area, even though the compiled **Optiboot** is only ~512 bytes.
- After reset, the MCU jumps to **Boot Reset Address (start of boot section)**.
- This is how **Optiboot** gains control immediately upon reset to check for a new program upload.
- When **Optiboot** is running, it may set the **IVSEL bit** in the **MCUCR** to 1. This relocates the **Interrupt Vector Table (IVT)** to the start of the **Bootloader Section** (word address = 0x3F00). This ensures that any interrupts that occur while the bootloader is running (e.g., Timer/UART interrupts) are handled by the Optiboot's own code.
- When **Optiboot** finishes its job and is about to jump to the main application sketch, it clears the **IVSEL bit** (setting it back to 0). This moves the **IVT** back to the start of the application space (0x0000), ensuring the application's interrupt vectors are active for the application program.

ATmega328P: Interrupt Handling

1) When an interrupt occurs, the CPU first completes the instruction currently being executed.
2) The **Global Interrupt Enable (GIE) bit** in the **SREG (Status Register)** is *cleared* to prevent nested interrupts.
3) The current value of the **Program Counter (PC)** -- the address of the next instruction -- is pushed onto the stack for later retrieval.
4) The address from the corresponding interrupt vector is then loaded into the PC, so the CPU begins executing instructions from that address.
5) The instruction at that vector address is typically a **JMP** (jump) to the first instruction of the **Interrupt Service Routine (ISR)**.
6) The CPU executes the ISR until it reaches the **RETI (Return from Interrupt) instruction**, signaling the end of the ISR.
7) Upon executing **RETI**, the CPU pops the saved PC value from the stack and resumes execution of the main program from where it was interrupted.
8) Finally, the **GIE bit** is *set* again, allowing the CPU to respond to further interrupts.

ATmega328P: Interrupt Enable

Each interrupt source is associated with the following bits:

1) Interrupt Enable (IE) bit

- When set (1), it allows interrupts from that specific source to occur.

2) Interrupt Flag (IF) bit

- When the corresponding interrupt occurs, this bit is automatically set to 1.
- The flag is automatically cleared to 0 when the corresponding ISR is executed.

Example: External Interrupt Request 0 (ATmega328P)

- **INT0**: External Interrupt Request 0 Enable bit in the **EIMSK** (External Interrupt Mask Register)
- **INTF0**: External Interrupt Flag 0 bit in the **EIFR** (External Interrupt Flag Register)

An interrupt from any source can be serviced only if:

- The **Global Interrupt Enable (GIE) bit** in the **SREG** is set.
- The **Interrupt Enable bit** associated with that specific source is set.

ATmega328P: I/O Interrupts

AVR MCUs provide **two types of interrupts** generated from I/O pins:

External Interrupts

- Dedicated pins
- Each pin has its own interrupt vector.
- ATmega328P: **INT0** (PD2), **INT1** (PD3)
- Trigger modes
  - Low level: Interrupt while pin is low
  - Any change: Rising or falling
  - Falling edge (High -> LoW)
  - Rising edge (Low -> High)
- Related I/O register(s)
  - EICRA, EIMSK, EIFR
- Dedicated ISR per pin (for AVR-GCC)
  - `ISR(INT0_vect) { ... }`
  - `ISR(INT1_vect) { ... }`

Pin Change Interrupts

- ATmega328P: There are three groups of I/O pins (PCINT0–PCINT23)
  - **PCINT0**: PB0–PB7 pins
  - **PCINT1**: PC0–PC5 pins
  - **PCINT2**: PD0–PD7 pins
- Trigger mode: only any change
- Related registers
  - PCICR, PCIFR, PCMSK0/1/2
- One ISR per group, not per pin
  - `ISR(PCINT0_vect) { }`
  - `ISR(PCINT1_vect) { }`
  - `ISR(PCINT2_vect) { }`

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	

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.

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

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	

EIFR – External Interrupt Flag Register

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	

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	

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	

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	

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	

ATmega328P: C Code – Button LED Demo (Polling)

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

int main(void) {
    // Configure PB5 as output (LED pin)
    DDRB |= (1 << DDB5);
    // Configure PD2 as input (button pin)
    DDRD &= ~(1 << DDD2);
    // Enable internal pull-up on PD2
    PORTD |= (1 << PORTD2);

    while (1) {
        // Read the button (active-low)
        if ((PIND & (1 << PIND2)) == 0) {
            // if button pressed, turn LED on
            PORTB |= (1 << PORTB5);
        }
        else {
            // if button released, turn LED off
            PORTB &= ~(1 << PORTB5);
        }
    }
    return 0;
}
```

ATmega328P: Assembly – Button LED Demo (1)

```
.equ DDRB,  0x04
.equ PORTB, 0x05
.equ PIND,  0x09
.equ DDRD,  0x0A
.equ PORTD, 0x0B
.equ PB5,   5
.equ PD2,   2

.org 0x0000
    rjmp RESET

RESET:
    ; Configure PB5 (LED) as output
    sbi DDRB, PB5  ; DDRB5 = 1 (output pin)

    ; Configure PD2 as input
    cbi DDRD, PD2  ; DDRD2 = 0 (input pin)

    ; Enable internal pull-up on PD2
    ; PORTD2 = 1 (pull-up enabled)
    sbi PORTD, PD2
```

```
MAIN_LOOP:
    sbis PIND, PD2
    rjmp BTN_PRESSED

BTN_RELEASED:
    cbi PORTB, PB5 ; LED OFF
    rjmp MAIN_LOOP

BTN_PRESSED:
    sbi PORTB, PB5 ; LED ON
    rjmp MAIN_LOOP
```

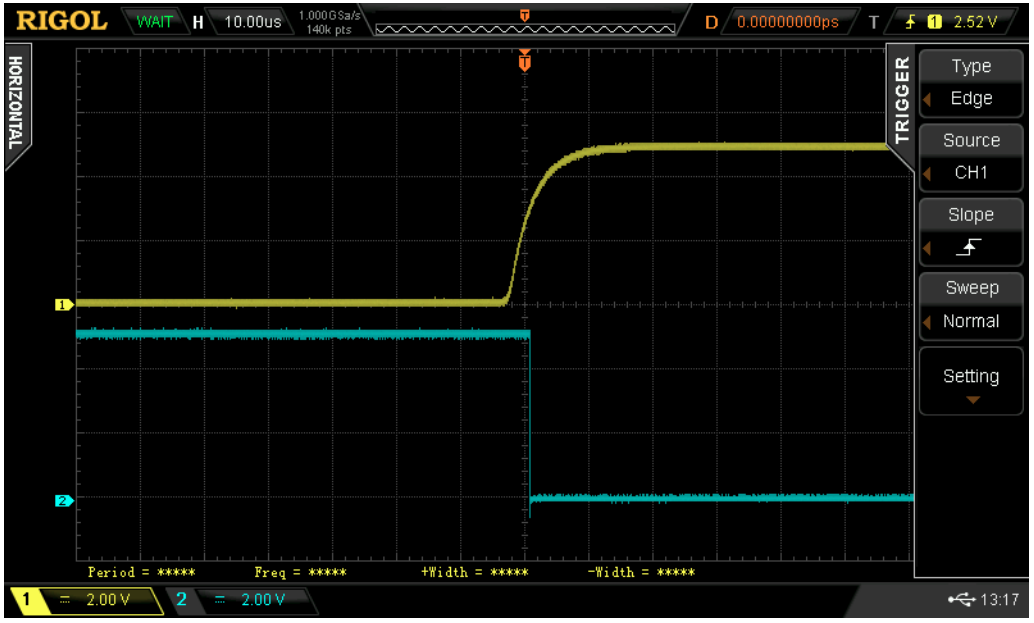
`sbis PIND, PD2`

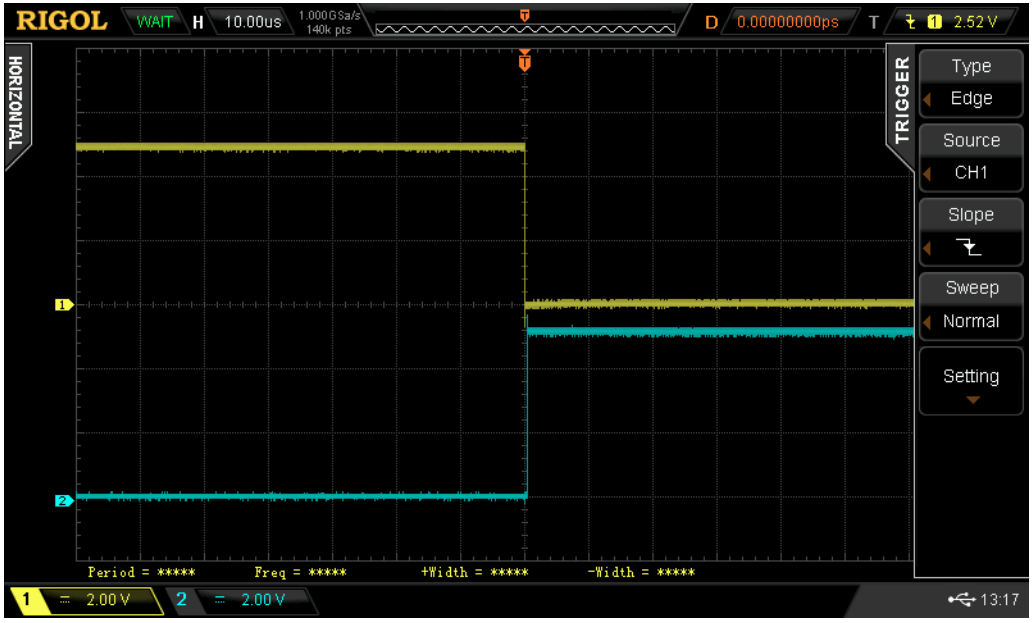
If bit **PD2** (bit 2) in the I/O register **PIND** is **logic** 1, skip the next instruction.

ATmega328P: Assembly – Button LED Demo (2)

```
.equ DDRB,  0x04
.equ PORTB, 0x05
.equ PIND,  0x09
.equ DDRD,  0x0A
.equ PORTD, 0x0B
.equ PB5,   5
.equ PD2,   2

.org 0x0000
    rjmp RESET

RESET:
    sbi DDRB, PB5       ; LED output
    cbi DDRD, PD2       ; Button input
    sbi PORTD, PD2      ; Enable pull-up

MAIN_LOOP:
    sbis PIND, PD2      ; skip next if PD2 is 1
    sbi PORTB, PB5      ; LED ON
    sbic PIND, PD2      ; skip next if PD2 is 0
    cbi PORTB, PB5      ; LED OFF
    rjmp MAIN_LOOP
```

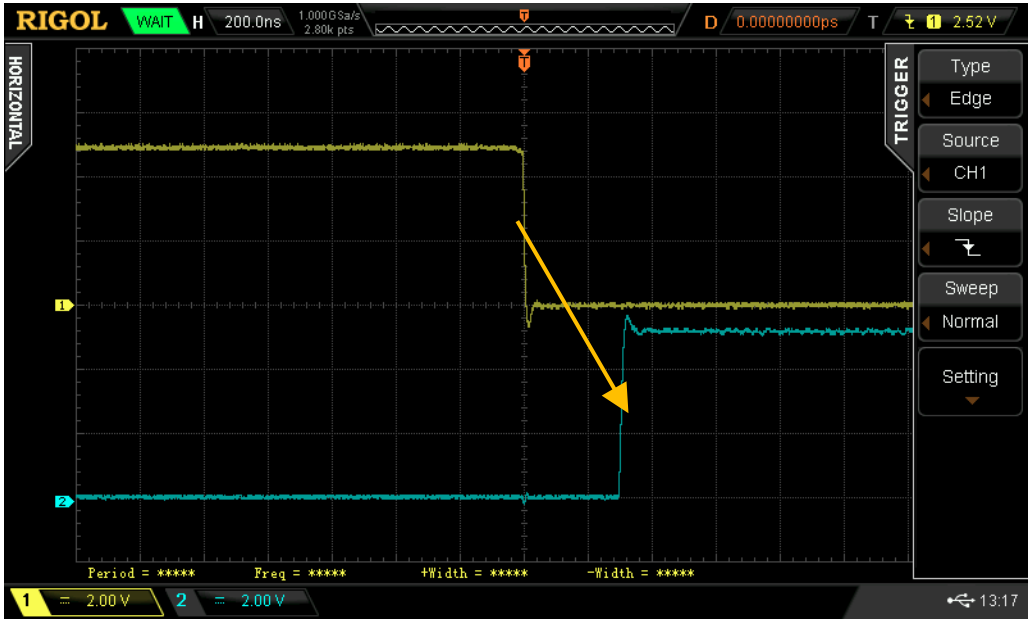
I/O Signal Measurement

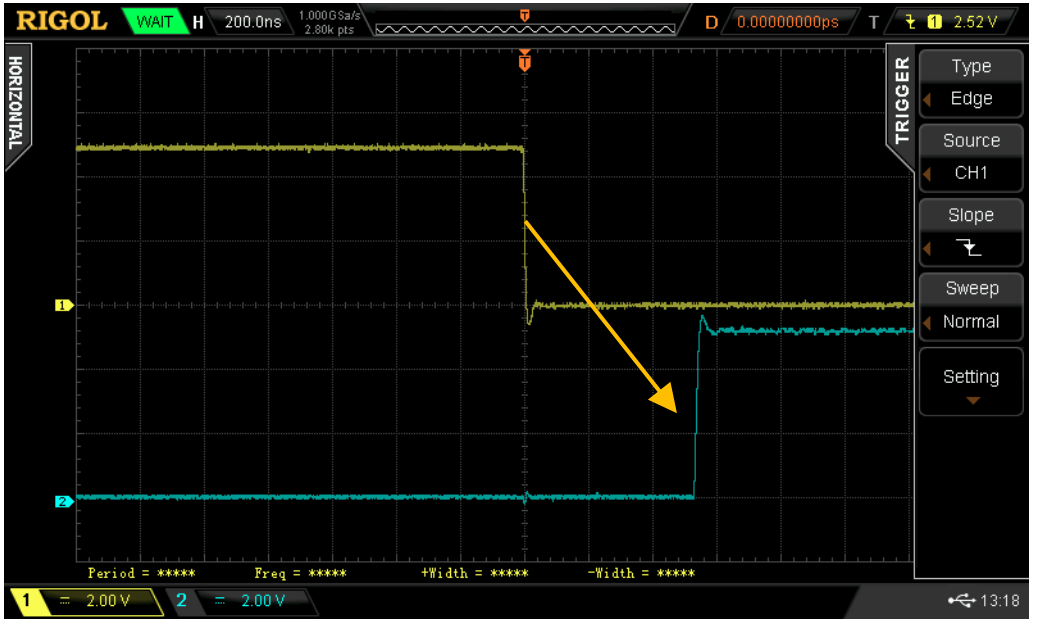




CH1: Input Signal (Push Button) on **PD2** pin
CH2: Output Signal (LED) on **PB5** pin

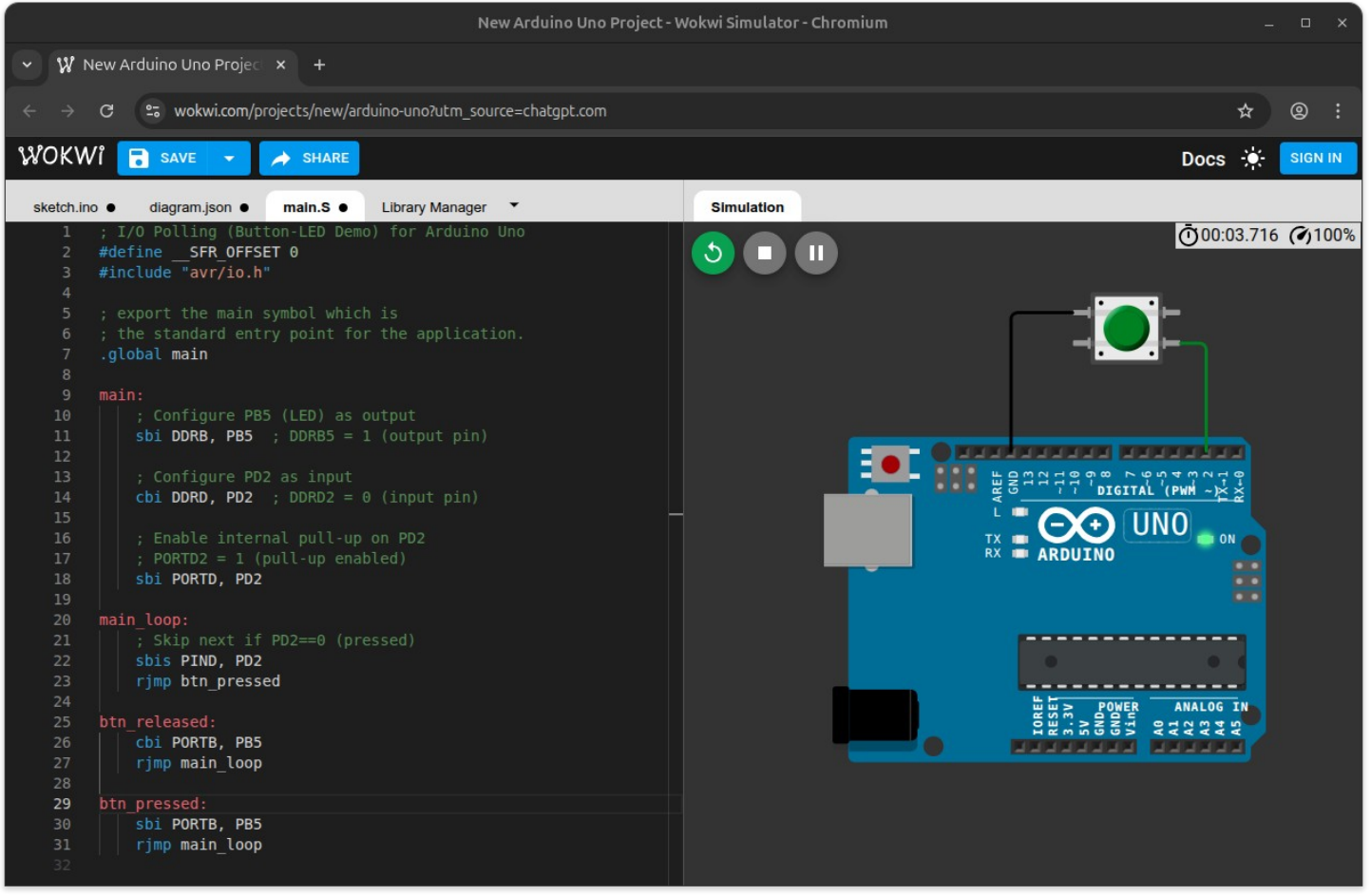
I/O Signal Measurement





Measurement of I/O response time

Simulation with Wokwi



[Experimental] Code modifications are necessary to make it compatible with Wokwi.

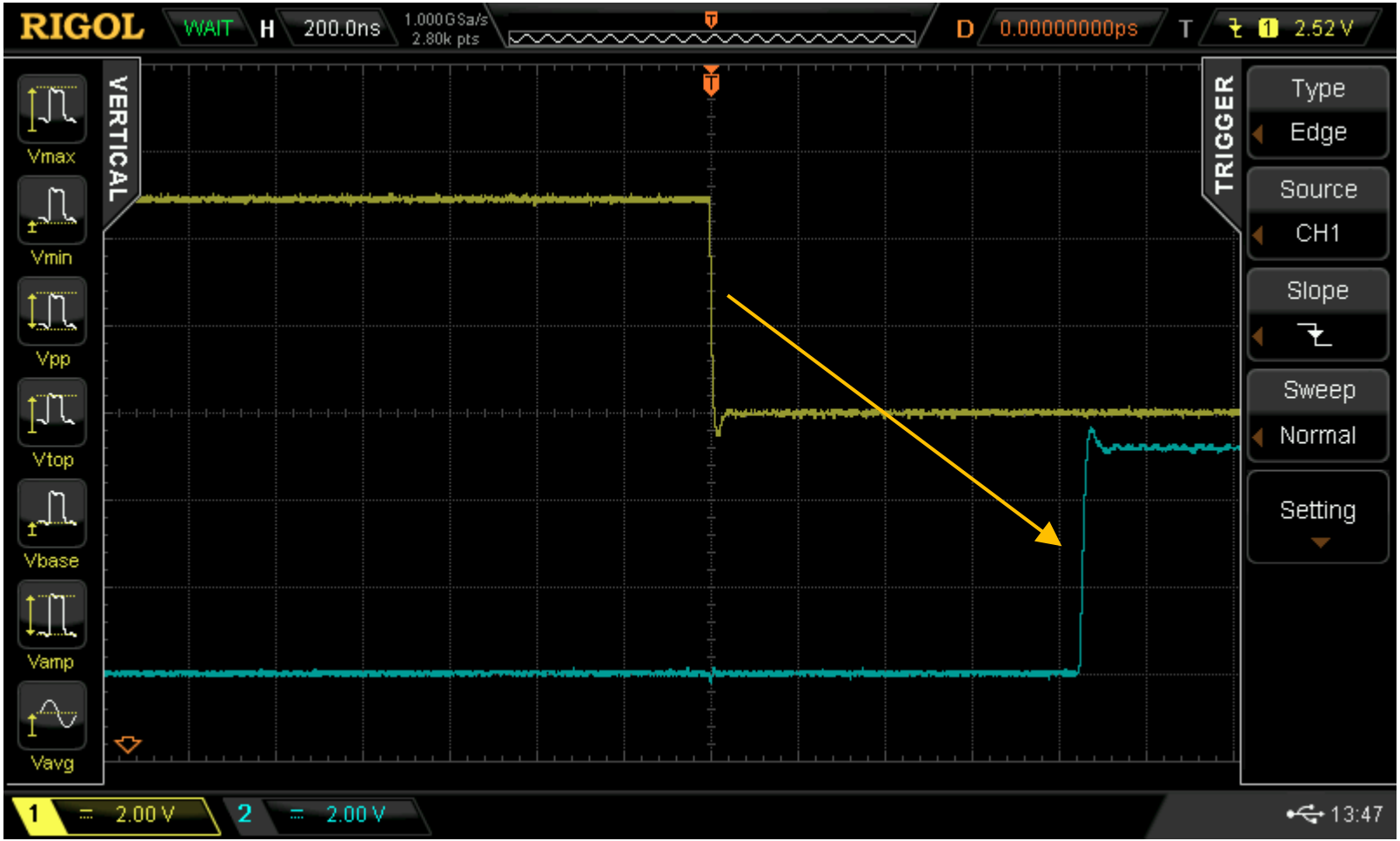
ATmega328P: C Code – Button LED Demo (Interrupt)

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

void init_gpio() {
    // Configure PB5 as output (LED)
    DDRB |= (1 << DDB5);
    // Configure PD2 as input (button)
    DDRD &= ~(1 << DDD2);
    // Enable internal pull-up on PD2
    PORTD |= (1 << PORTD2);
}

void init_ext_int0() {
   // Trigger INT0 on any logical change (both edges)
   EICRA |=  (1 << ISC00);   // ISC00 = 1
   EICRA &= ~(1 << ISC01);   // ISC01 = 0
   // Enable INT0 interrupt
   EIMSK |= (1 << INT0);
}
```

ATmega328P: C Code – Button LED Demo (Interrupt)

```
int main(void) {
    init_gpio();
    init_ext_int0();
    sei();
    while (1) {}
}

// External Interrupt 0 Service Routine
ISR(INT0_vect) {
    // Active-low button: pressed = 0, released = 1
    if ((PIND & (1 << PD2)) == 0) {
        PORTB |= (1 << PB5);   // LED ON
    } else {
        PORTB &= ~(1 << PB5);  // LED OFF
    }
}
```

I/O Signal Measurement



ATmega328P: C Code – Button LED Demo (Interrupt)

File: main.s

```
.include "m328pdef.inc"

; Interrupt Vector Table
    jmp RESET           ; 0x0000 Reset vector
    jmp INT0_ISR        ; 0x0002 INT0 External Interrupt Request 0
    reti                ; 0x0004 INT1 External Interrupt Request 1
    ; ....

RESET:
    cli ; Disable interrupts during setup
    ; Configure stack pointer (RAMEND = 0x08FF)
    ldi r16, 0x08
    out SPH, r16
    ldi r16, 0xFF
    out SPL, r16
    ; Initialize some registers to zero
    clr r1
    ; Jump to main
    rjmp main

; Main code section
.section .text
.global main
```

```
main:
    ; Configure PB5 (LED) as output, initially OFF
    sbi DDRB, PB5       ; Set bit 5 in DDRB (output)
    cbi PORTB, PB5      ; Clear bit 5 in PORTB (LED OFF initially)

    ; Configure PD2 (Button) as input with pull-up
    cbi DDRD, PD2       ; Clear bit 2 in DDRD (input mode)
    sbi PORTD, PD2      ; Set bit 2 in PORTD (enable internal pull-up)

    ; Clear any pending INT0 interrupt flag
    ldi r16, (1 << INTF0)
    out EIFR, r16       ; Write 1 to INTF0 to clear it

    ; Configure INT0 for any logical change (both edges)
    ldi r16, 0x01       ; ISC01:ISC00 = 01 (any edge on INT0)
    sts EICRA, r16      ; Store to EICRA

    ; Enable INT0
    ldi r16, (1 << INT0)
    out EIMSK, r16

    ; Enable global interrupts
    sei
```

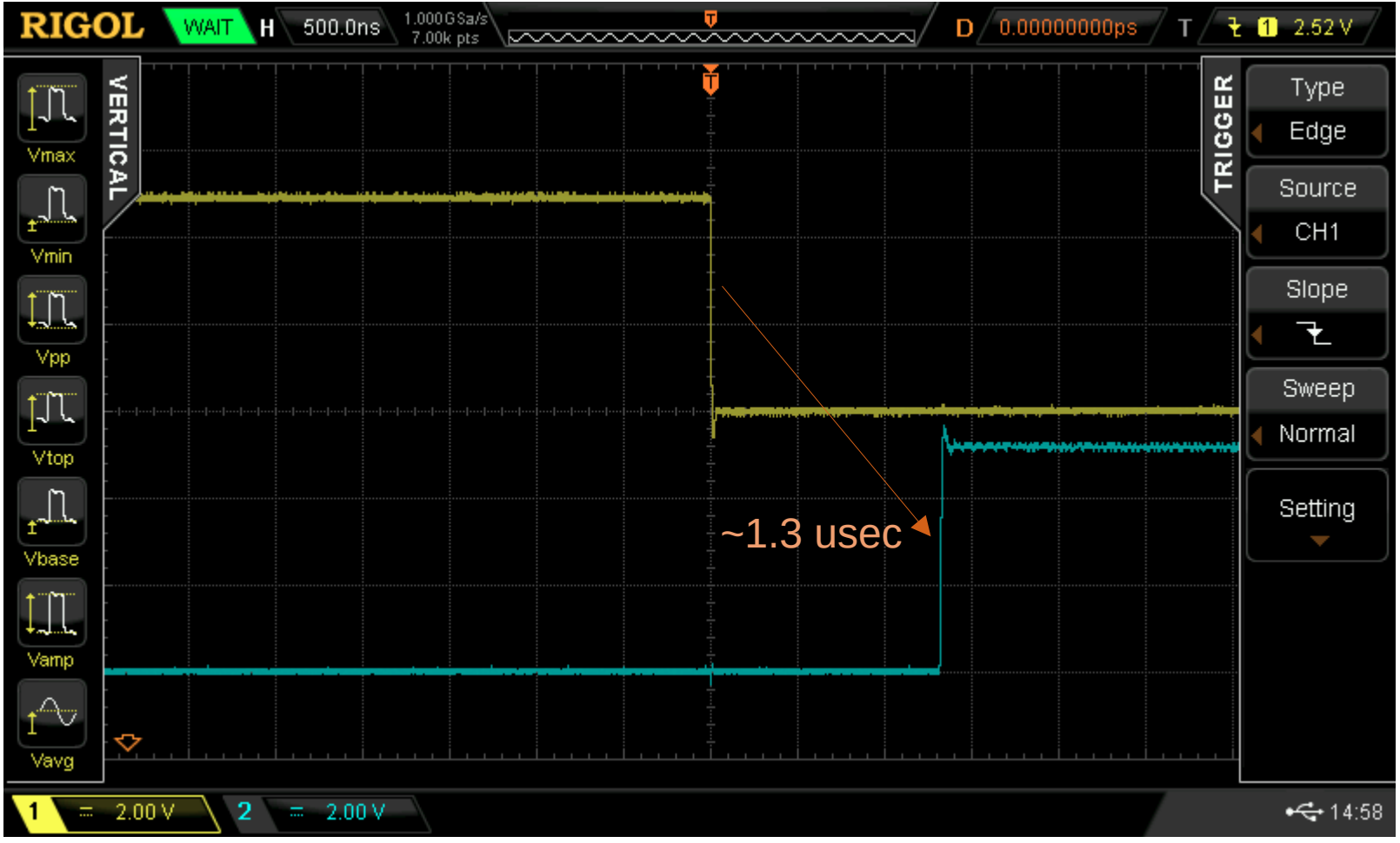
```
MAIN_LOOP:
    nop                 ; Do nothing
    rjmp MAIN_LOOP      ; Loop forever

; INT0 Interrupt Service Routine
.global INT0_ISR

INT0_ISR:
    push r16            ; Save register
    in r16, SREG        ; Save status register
    push r16

    ; Read PD2 and copy to PB5 (inverted logic: button low = LED high)
    in r16, PIND        ; Read button state
    sbrc r16, PD2       ; Skip if PD2=0 (button pressed)
    cbi PORTB, PB5      ; PD2=1 (released) => LED OFF
    sbrs r16, PD2       ; Skip if PD2=1 (button released)
    sbi PORTB, PB5      ; PD2=0 (pressed) => LED ON
    pop r16             ; Restore status register
    out SREG, r16
    pop r16             ; Restore register
    reti                ; Return from interrupt
```

File: m328pdef.inc

```
;============================================================
; m328pdef.inc - ATmega328P Register Definitions

; I/O Port Registers (I/O addresses 0x00-0x3F)
.equ PINB  , 0x03
.equ DDRB  , 0x04
.equ PORTB , 0x05

.equ PINC  , 0x06
.equ DDRC  , 0x07
.equ PORTC , 0x08

.equ PIND  , 0x09
.equ DDRD  , 0x0A
.equ PORTD , 0x0B

; Interrupt Registers
.equ EIFR  , 0x3C    ; External Interrupt Flag Register (I/O)
.equ EIMSK , 0x1D    ; External Interrupt Mask Register (I/O)

; Status and Stack Registers
.equ SPL   , 0x3D    ; Stack Pointer Low
.equ SPH   , 0x3E    ; Stack Pointer High
.equ SREG  , 0x3F    ; Status Register

; Data Memory Registers (0x60-0xFF)
.equ EICRA , 0x69    ; External Interrupt Control Register A (Data)
```

```
; --------- Bit Positions ----------
.equ PB0, 0
.equ PB1, 1
.equ PB2, 2
.equ PB3, 3
.equ PB4, 4
.equ PB5, 5
.equ PB6, 6
.equ PB7, 7

.equ PD0, 0
.equ PD1, 1
.equ PD2, 2
.equ PD3, 3
.equ PD4, 4
.equ PD5, 5
.equ PD6, 6
.equ PD7, 7

.equ INT0, 0
.equ INT1, 1
.equ INTF0, 0
.equ INTF1, 1

.equ ISC00, 0
.equ ISC01, 1
.equ ISC10, 2
.equ ISC11, 3
; the rest is not shown (omitted)...
```

I/O Signal Measurement

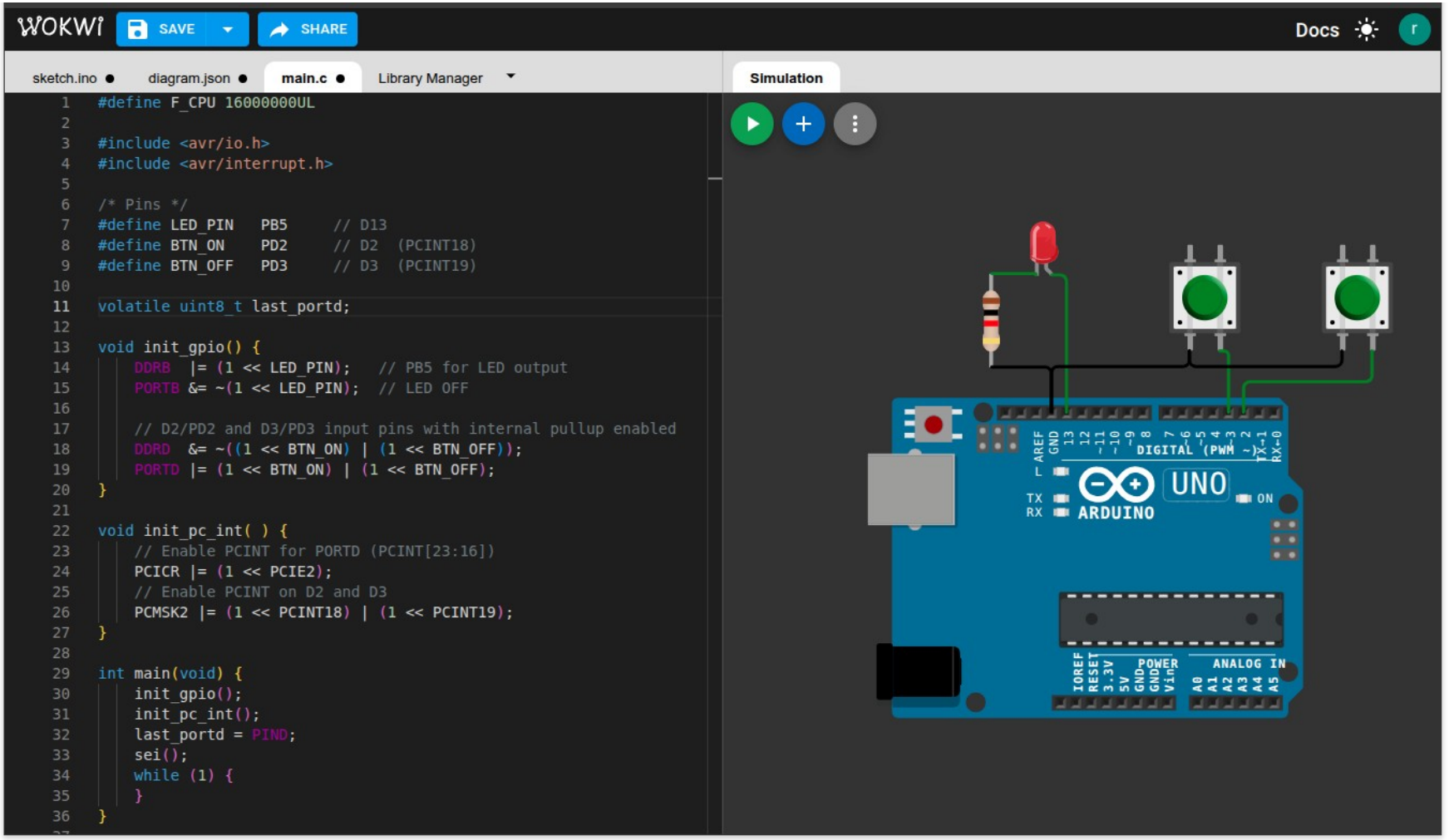


ATmega328P: C Code – Pin Change Interrupts

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

#define LED_PIN   PB5     // D13
#define BTN_ON    PD2     // D2  (PCINT18)
#define BTN_OFF   PD3     // D3  (PCINT19)

volatile uint8_t last_portd;

void init_gpio() {
    DDRB  |=  (1 << LED_PIN); // PB5 for LED output
    PORTB &= ~(1 << LED_PIN); // LED OFF

    // D2/PD2 and D3/PD3 input pins
    // with internal pullup enabled
    DDRD  &= ~((1 << BTN_ON) | (1 << BTN_OFF));
    PORTD |=   (1 << BTN_ON) | (1 << BTN_OFF);
}

void init_pc_int( ) {
    // Enable PCINT for PORTD (PCINT[23:16])
    PCICR  |= (1 << PCIE2);
    // Enable PCINT on D2 and D3
    PCMSK2 |= (1 << PCINT18) | (1 << PCINT19);
}
```

```
int main(void) {
    init_gpio();
    init_pc_int();
    last_portd = PIND;
    sei();
    while (1) {}
}

ISR(PCINT2_vect) {
    uint8_t current = PIND;
    uint8_t changed = current ^ last_portd;
    // if PD2 has changed from 1 to 0.
    if ((changed & (1 << BTN_OFF)) &&
       !(current & (1 << BTN_OFF))) {
        PORTB &= ~(1 << LED_PIN);
    }
    // if PD3 has changed from 1 to 0.
    else if ((changed & (1 << BTN_ON)) &&
            !(current & (1 << BTN_ON))) {
        PORTB |= (1 << LED_PIN);
    }
    last_portd = current;
}
```

Wokwi Uno Simulation



ATmega328P: Explain how this code works...

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <stdint.h>

#define LED_PIN  PB5  // PB5 / D13
#define BTN_PIN  PD2  // PD2 / D2 (active-low)

void init_gpio() {
    DDRB  |=  (1 << LED_PIN); // LED output
    PORTB &= ~(1 << LED_PIN); // LED OFF
    // Button input
    DDRD  &= ~(1 << BTN_PIN);
    // Enable internal pull-up
    PORTD |= (1 << BTN_PIN);
}
```

```
int main(void) {
    init_gpio();

    static uint16_t shift_reg = 0xFFFF;
    while (1) {
        // Sample PD2 (bit 2), active-low
        shift_reg = (shift_reg << 1)
                  | ((PIND >> BTN_PIN) & 0x01);
        // Detect stable LOW after HIGH
        if (shift_reg == 0xFF00) {
            // Toggle LED
            PORTB ^= (1 << LED_PIN);
        }
        _delay_ms(5);
    }
}
```

Given the following AVR bare-metal code that samples **PD2** every 5 ms using a 16-bit shift-register debounce filter, determine the **minimum time delay** between **PD2** transitioning from HIGH to LOW and **PB5** toggling state.

Interrupt Pitfall

An ISR takes a long time to execute or blocks.

- An ISR should respond to an event quickly and deterministically.
- When an ISR runs for a long time, or waits (blocks) for something to happen,

  it prevents the CPU from servicing other interrupts and normal code.
- While an ISR is running:
  - Other interrupts are disabled (on most MCUs, including AVR)
  - Interrupt latency increases
  - Time-critical events can be missed
  - The system can appear to “freeze” or behave unpredictably

Good Coding Practice: Deferred Interrupt Processing

```
// global shared variable
volatile uint8_t button_event; // event flag

ISR(INT1_vect) {
    button_event = 1; // Set flag
}

int main(void) {
    // ....
    sei();  // Enable interrupts once
    while (1) {
        if (button_event) {
            button_event = 0; // Clear flag
            sei();  // Re-enable interrupts
        }
    }
}
```

Good Coding Practice: Handle a 32-bit Shared Variable

```
volatile uint32_t counter = 0;

ISR(TIMER1_OVF_vect) { // Timer 1 overflow interrupt
    counter++;
}
```

```
// Single Atomic Block
cli(); // Disable all maskable interrupts
uint32_t value = counter; // Read the counter
if (value >= CNT_MAX) {
   value = 0; // Reset counter
}
count = value; // update the counter
sei(); // Re-enable maskable interrupts
```

Watchdog Timer (WDT)

- A **Watchdog Timer (WDT)** is a safety mechanism in embedded systems used to improve system reliability.
- When enabled, the WDT automatically resets or interrupts the MCU if the software becomes **unresponsive** due to EMI, noise, or software bugs.
- The WDT is commonly used for **fault recovery in embedded systems**.
- AVR microcontrollers (e.g., ATmega328P) include a built-in WDT.

WDT inside the AVR MCU

- The WDT has an **independent on-chip oscillator** and can run even if the main system clock fails.
- The timeout period is programmable.
- The WDT can operate in **multiple modes**:

  1) System Reset mode: the MCU performs a system reset.

  2) Interrupt mode: the MCU executes the WDT ISR.

  3) Interrupt + System Reset mode.

- The WDT can be enabled in two ways:

  1) By software configuration of the WDT control registers (e.g., `WDTCSR`).

  2) By programming fuse bits (watchdog always-on `WDTON=0`).

- When enabled, the WDT counts down and can be reset (“kicked”) using:
  - The `WDR` instruction (assembly).
  - The `wdt_reset()` function.

WDTCSR – Watchdog Timer Control Register

Bit	7	6	5	4	3	2	1	0	
(0x60)	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0	WDTCSR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	X	0	0	0	

- **Bit 7 - WDIF: Watchdog Interrupt Flag**

This bit is set when a time-out occurs in the watchdog timer and the watchdog timer is configured for interrupt. WDIF is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, WDIF is cleared by writing a logic one to the flag. When the I-bit in SREG and WDIE are set, the watchdog time-out interrupt is executed.

- **Bit 6 - WDIE: Watchdog Interrupt Enable**

When this bit is written to one and the I-bit in the status register is set, the watchdog interrupt is enabled. If WDE is cleared in combination with this setting, the watchdog timer is in interrupt mode, and the corresponding interrupt is executed if time-out in the watchdog timer occurs. If WDE is set, the watchdog timer is in interrupt and system reset mode. The first time-out in the watchdog timer will set WDIF.

Executing the corresponding interrupt vector will clear WDIE and WDIF automatically by hardware (the watchdog goes to system reset mode). This is useful for keeping the watchdog timer security while using the interrupt. To stay in interrupt and system reset mode, WDIE must be set after each interrupt. This should however not be done within the interrupt service routine itself, as this might compromise the safety-function of the watchdog system reset mode. If the interrupt is not executed before the next time-out, a system reset will be applied.

- **Bit 4 - WDCE: Watchdog Change Enable**

This bit is used in timed sequences for changing WDE and prescaler bits. To clear the WDE bit, and/or change the prescaler bits, WDCE must be set.

Once written to one, hardware will clear WDCE after four clock cycles.

- **Bit 3 - WDE: Watchdog System Reset Enable**

WDE is overridden by WDRF in MCUSR. This means that WDE is always set when WDRF is set. To clear WDE, WDRF must be cleared first. This feature ensures multiple resets during conditions causing failure, and a safe start-up after the failure.

- **Bit 5, 2..0 - WDP3..0: Watchdog Timer Prescaler 3, 2, 1 and 0**

The WDP3..0 bits determine the watchdog timer prescaling when the watchdog timer is running. The different prescaling values and their corresponding time-out periods are shown in Table 10-3.

MCUSR – MCU Status Register

The MCU status register provides information on which reset source caused an MCU reset.

Bit	7	6	5	4	3	2	1	0	
0x35 (0x55)	–	–	–	–	WDRF	BORF	EXTRF	PORF	MCUSR
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	See Bit Description				

- **Bit 7..4: Res: Reserved Bits**

These bits are unused bits in the Atmel® ATmega328P, and will always read as zero.

- **Bit 3 – WDRF: Watchdog System Reset Flag**

This bit is set if a watchdog system reset occurs. The bit is reset by a power-on reset, or by writing a logic zero to the flag.

- **Bit 2 – BORF: Brown-out Reset Flag**

This bit is set if a brown-out reset occurs. The bit is reset by a power-on reset, or by writing a logic zero to the flag.

- **Bit 1 – EXTRF: External Reset Flag**

This bit is set if an external reset occurs. The bit is reset by a power-on reset, or by writing a logic zero to the flag.

- **Bit 0 – PORF: Power-on Reset Flag**

This bit is set if a power-on reset occurs. The bit is reset only by writing a logic zero to the flag.

To make use of the reset flags to identify a reset condition, the user should read and then reset the MCUSR as early as possible in the program. If the register is cleared before another reset occurs, the source of the reset can be found by examining the reset flags.

Table 10-2. Watchdog Timer Configuration

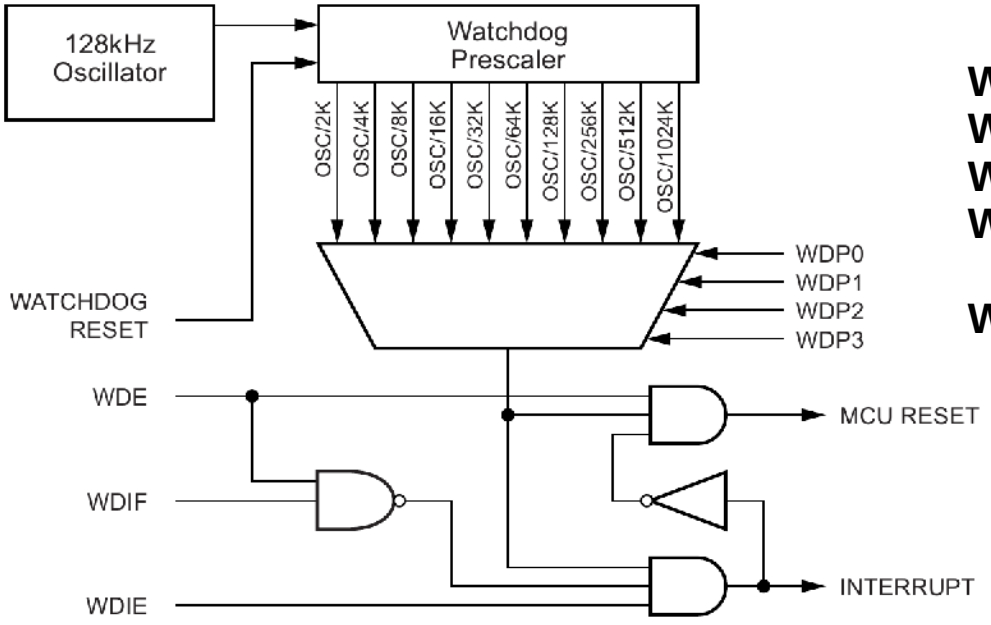
WDTON[1]	WDE	WDIE	Mode	Action on Time-out
1	0	0	Stopped	None
1	0	1	Interrupt mode	Interrupt
1	1	0	System reset mode	Reset
1	1	1	Interrupt and system reset mode	Interrupt, then go to system reset mode
0	x	x	System reset mode	Reset

Note: 1. WDTON fuse set to “0” means programmed and “1” means unprogrammed.

Table 10-3. Watchdog Timer Prescale Select

WDP3	WDP2	WDP1	WDP0	Number of WDT Oscillator Cycles	Typical Time-out at V_{CC} = 5.0V
0	0	0	0	2K (2048) cycles	16ms
0	0	0	1	4K (4096) cycles	32ms
0	0	1	0	8K (8192) cycles	64ms
0	0	1	1	16K (16384) cycles	0.125s
0	1	0	0	32K (32768) cycles	0.25s
0	1	0	1	64K (65536) cycles	0.5s
0	1	1	0	128K (131072) cycles	1.0s
0	1	1	1	256K (262144) cycles	2.0s
1	0	0	0	512K (524288) cycles	4.0s
1	0	0	1	1024K (1048576) cycles	8.0s
1	0	1	0	Reserved	
1	0	1	1		
1	1	0	0		
1	1	0	1		
1	1	1	0		
1	1	1	1		

WDT inside the AVR MCU



WDIF: Watchdog Interrupt Flag
WDIE: Watchdog Interrupt Enable
WDCE: Watchdog Change Enable
WDE: Watchdog System Reset Enable
- always set when the WDRF bit in MCUSR is set.

WDRF: Watchdog System Reset Flag
- reset flag by writing a logic zero to the flag.

Before changing any WDT configuration, the watchdog timer should be reset before any change of the WDP bits. This can be done only by the `WDR` instruction.

To reset the WDT in AVR GCC code:

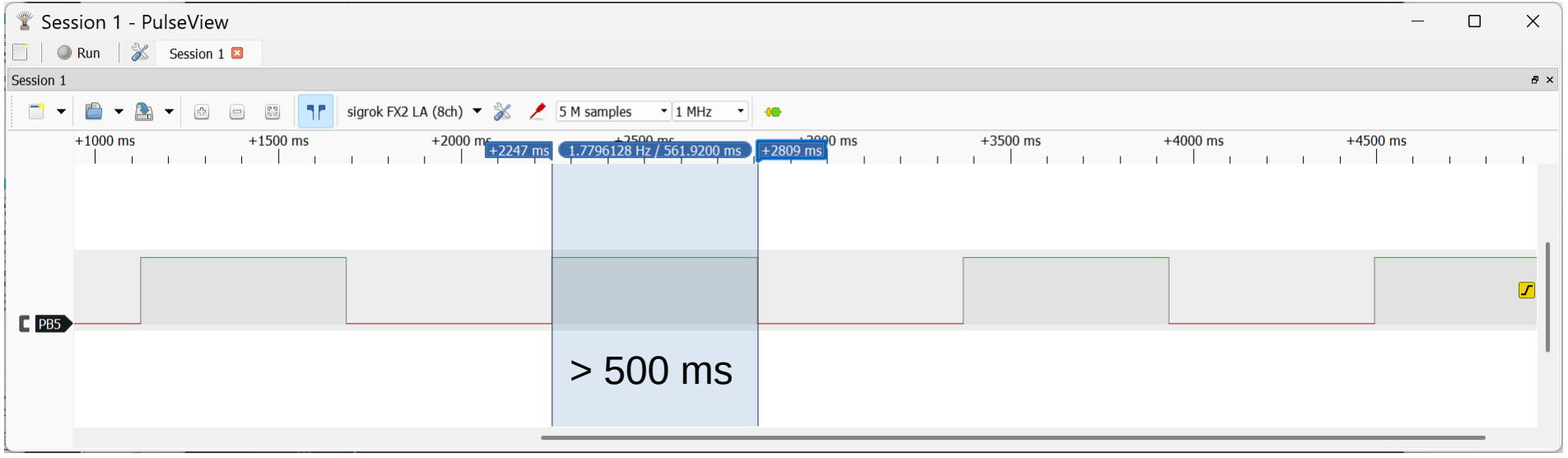
- `wdt_reset();  // #include <avr/wdt.h>`
- `__asm__ __volatile__("wdr");`

ATmega328P: C Code – WDT Interrupt Mode

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

ISR(WDT_vect) {
  PORTB ^= (1 << PB5);   // Toggle PB5 / D13 (LED)
}

static void init_gpio() {
  DDRB |= (1 << PB5);   // Configure PB5 as output
  PORTB &= ~(1 << PB5); // LED off initially
}

static void init_wdt() {
  cli(); // Disable global interrupts
  __asm__ __volatile__("wdr");
  MCUSR &= ~(1 << WDRF); // Clear WDT reset flag
  // Start a sequence to change WDT configuration
  WDTCSR |= (1 << WDCE) | (1 << WDE);
  // Configure WDT: Interrupt mode, ~0.5s timeout
  WDTCSR = (1 << WDIE) | // Enable WDT interrupt
           (1 << WDP2) | // Prescaler bits: 0b0101
           (1 << WDP0);
}
```

```c
int main(void) {
   init_gpio();
   init_wdt();
   // Enable global interrupts
   sei();
   while (1) {}
}
```

Logic Analyzer: Pulse Width Measurement

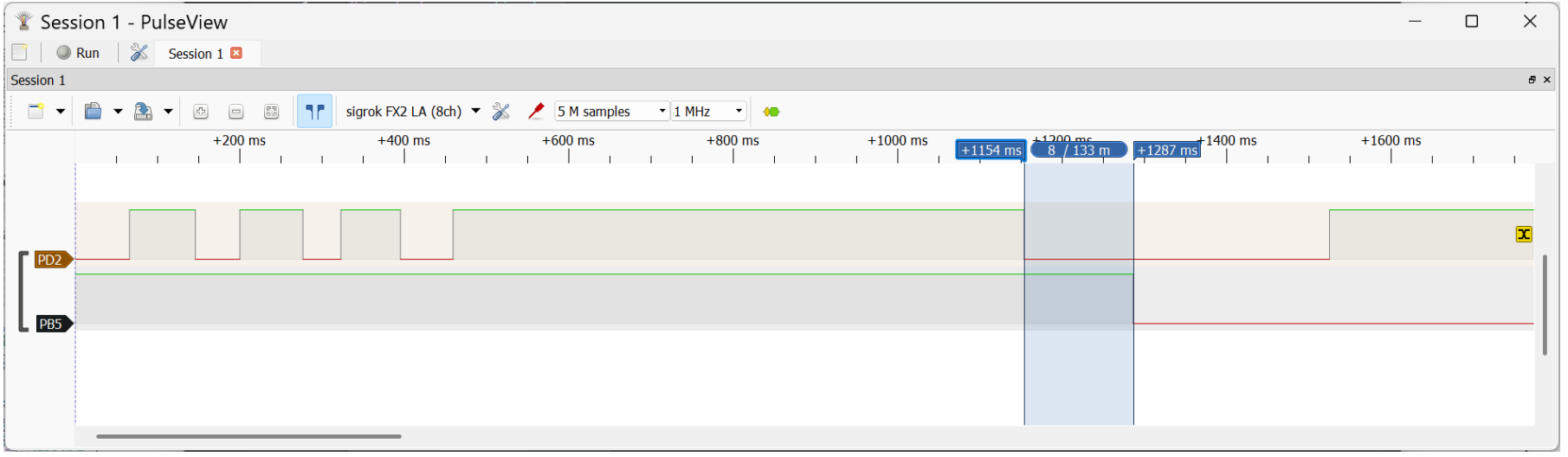


ATmega328P: C Code – WDT ISR & Button Debounce

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

#define LED_PIN PB5 // Arduino Uno LED
#define BTN_PIN PD2 // Push button

// 16-bit shift register for debouncing
static volatile uint16_t shift_reg = 0xFFFF;

ISR(WDT_vect) {
  // Sample the PD2 pin
  shift_reg = (shift_reg << 1)
            | ((PIND >> BTN_PIN) & 0x01);
}

static void init_gpio(void) {
  // LED as output
  DDRB  |=  (1 << LED_PIN);
  PORTB &= ~(1 << LED_PIN);
  // Button as input with pull-up
  DDRD  &= ~(1 << BTN_PIN);
  PORTD |=  (1 << BTN_PIN);
}
```

```c
static void init_wdt() {
  cli();
  __asm__ __volatile__("wdr");
  MCUSR &= ~(1 << WDRF); // Clear WDT reset flag
  // Enable WDT configuration
  WDTCSR |= (1 << WDCE) | (1 << WDE);
  // WDT interrupt mode, ~16 ms timeout
  // WDP3:0 = 0b0000
  WDTCSR = (1 << WDIE);
}
int main(void) {
    init_gpio();
    init_wdt();
    sei();
    while (1)  {
      // Detect button press (stable LOW)
      cli();
      if (shift_reg == 0xFF00) {
        shift_reg = 0x0000;
        PORTB ^= (1<<LED_PIN); // Toggle LED
      }
      sei();
    }
}
```


Session 1 - PulseView
Run
Session 1
Session 1
sigrok FX2 LA (8ch)
5 M samples
1 MHz
+500 ms
+1000 ms
+1500 ms
+2000 ms
+2500 ms
+3000 ms
+3500 ms
+4000 ms
Button
LED
PD2
PB5


Session 1 - PulseView
Run
Session 1
Session 1
sigrok FX2 LA (8ch)
5 M samples
1 MHz
+200 ms
+400 ms
+600 ms
+800 ms
+1000 ms
+1154 ms
8 / 133 m
+1287 ms
+1400 ms
+1600 ms
PD2
PB5

System Clock Prescaling

- The **ATmega328P** has a system clock prescaler, and the system clock can be divided by setting the "**CLKPR – Clock Prescale Register**".
- This feature can be used to decrease the system clock frequency and the power consumption when the requirement for processing power is low.
- This can be used with all clock source options.

To avoid unintentional changes of clock frequency, a special write procedure must be followed to change the **CLKPS (Clock Prescaler Select)** bits:

1. Write the clock prescaler change enable (**CLKPCE: Clock Prescaler Change Enable**) bit to one and all other bits in **CLKPR** to zero.
2. Within four cycles, write the desired value to **CLKPS** while writing a zero to **CLKPCE**.

Interrupts must be disabled when changing prescaler setting to make sure the write procedure is not interrupted.

CLKPR – Clock Prescale Register

Bit	7	6	5	4	3	2	1	0	
(0x61)	CLKPCE	–	–	–	CLKPS3	CLKPS2	CLKPS1	CLKPS0	CLKPR
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	See Bit Description				

Table 8-16. Clock Prescaler Select

CLKPS3	CLKPS2	CLKPS1	CLKPS0	Clock Division Factor
0	0	0	0	1
0	0	0	1	2
0	0	1	0	4
0	0	1	1	8
0	1	0	0	16
0	1	0	1	32
0	1	1	0	64
0	1	1	1	128
1	0	0	0	256
1	0	0	1	Reserved
1	0	1	0	Reserved
1	0	1	1	Reserved
1	1	0	0	Reserved
1	1	0	1	Reserved
1	1	1	0	Reserved
1	1	1	1	Reserved

ATmega328P: C Code – Clock Speed Switching

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

#define LED PB5
#define BTN PD2

static inline
void switch_clk(uint8_t prescale) {
    uint8_t s = SREG; // Save SREG
    cli();
    CLKPR = (1<<CLKPCE);
    CLKPR = prescale;
    SREG = s; // Restore SREG
}

static void init_gpio() {
  DDRB |= (1<<LED);
  DDRD &= ~(1<<BTN);
  PORTD |= (1<<BTN);
}
```

```
int main(void) {
  uint8_t slow = 0, prev_value = 1;

  init_gpio();

  while (1) {
    PORTB ^= (1<<LED);
    _delay_ms(50);
    uint8_t btn_value = (PIND & (1<<BTN))!=0;
    if (prev_value && !btn_value) {
        slow ^= 1;
        switch_clk(slow ? 0x03 : 0x00);
        _delay_ms(50);
    }
     prev_value = btn_value;
  }
}
```

Note: Changing the clock speed directly affects `_delay_ms()`.

ATmega328P: SREG Saving and Restoring

```
// Disable interrupts
cli();
// Do something that must not be interrupted
// …
// Enable interrupts
sei();
```

If interrupts were already disabled, `sei()` incorrectly enables them. This can cause unexpected behavior and hard-to-find bugs.

```
// Save full CPU status (including I bit)
uint8_t s = SREG;
// Disable interrupts
cli();
// Do something that must not be interrupted
// ..
SREG = s; // Restore original interrupt state
```

After restoring the original interrupt state:
- If interrupts were ON, they turn back ON
- If interrupts were OFF, they stay OFF

Logic Analyzer: Pulse Width Measurement