Module 6 - Interrupt

1. Introduction to Interrupt

An interrupt is a mechanism used in microcontroller programming to pause the execution of the current program and call a specific routine or function when a particular event occurs. These events, defined by the programmer, can range from a specific condition on an input pin to a timer overflow or other hardware-defined triggers. Once the routine or function finishes executing, the program resumes from the exact point where it was interrupted.

The Arduino Uno (ATMega328P) provides two primary types of interrupts: External Interrupts and Timer Interrupts. While their functions and triggers differ, they share the same goal: interrupting the main program flow to handle specific events immediately.

External Interrupt

An External Interrupt is triggered by a change in the voltage level on specific input pins of the microcontroller. On the Arduino Uno, there are two pins dedicated to external interrupts: Pin 2 and Pin 3. These are the most commonly used pins when implementing interrupts via the Arduino programming language.

External Interrupts: INT0 and INT1

While the Arduino IDE refers to these simply as Pin 2 and Pin 3, the ATMega328P datasheet identifies them as **INT0** and **INT1**. These are hardware-level designations that correspond to specific physical pins.

- **INT0 (Digital Pin 2):** This is the first external interrupt. It has a higher priority in the Interrupt Vector Table than INT1, meaning if both occur at the exact same time, the microcontroller will handle INT0 first.
- **INT1 (Digital Pin 3):** This is the second external interrupt.

Trigger Modes

Both INT0 and INT1 can be configured to trigger the Interrupt Service Routine (ISR) based on four specific signal states:

1. **LOW:** Triggered whenever the pin is at a logic low level.
2. **CHANGE:** Triggered whenever the pin changes value (High to Low or Low to High).
3. **RISING:** Triggered specifically when the pin goes from Low to High.
4. **FALLING:** Triggered specifically when the pin goes from High to Low.

Internal Interrupt

An Internal Interrupt is triggered by modules located inside the microcontroller itself. In the ATMega328P, these are generated by timers and are referred to as Timer Interrupts. A Timer Interrupt is specifically triggered by a timer overflow. The Arduino Uno features three internal timers: Timer0, Timer1, and Timer2. (For more details on how timers operate, please refer to the previous module).

To expand on the specific hardware components of the ATMega328P (the heart of the Arduino Uno), we need to look at how the microcontroller labels and manages these specific interrupt sources.

Internal Interrupts: Timer0, Timer1, and Timer2

The Arduino Uno has three hardware timers, each capable of generating interrupts. These are essential for tasks that require precise timing without blocking the `void loop()`.

Name	Size	Possible Interrupts	Uses in Arduino
TIMER0	8 bits (0 - 255)	• Compare Match • Overflow	• delay(), millis(), micros() • analogWrite() pins 5, 6
TIMER1	16 bits (0 - 65,535)	• Compare Match • Overflow • Input Capture	• Servo functions • analogWrite() pins 9, 10
TIMER2	8 bits (0 - 255)	• Compare Match • Overflow	• tone() • analogWrite() pins 3, 11

1. Timer0 (8-bit)

- **Role:** This timer is used by the Arduino internal functions like `delay()`, `millis()`, and `micros()`.
- **Interrupt Potential:** It can trigger an **Overflow Interrupt** (when the counter hits 255 and resets to 0) or a **Compare Match Interrupt**.
- **Note:** Modifying Timer0 registers directly is generally discouraged because it will break the Arduino's built-in time-keeping functions.

2. Timer1 (16-bit)

- **Role:** Because it is a 16-bit timer, it can count up to 65,535. This allows for much longer and more precise timing intervals than Timer0 or Timer2.
- **Use Case:** It is frequently used by the `Servo` library.

- **Interrupt Potential:** Like Timer0, it supports Overflow and Compare Match interrupts, but with much higher resolution.

3. Timer2 (8-bit)

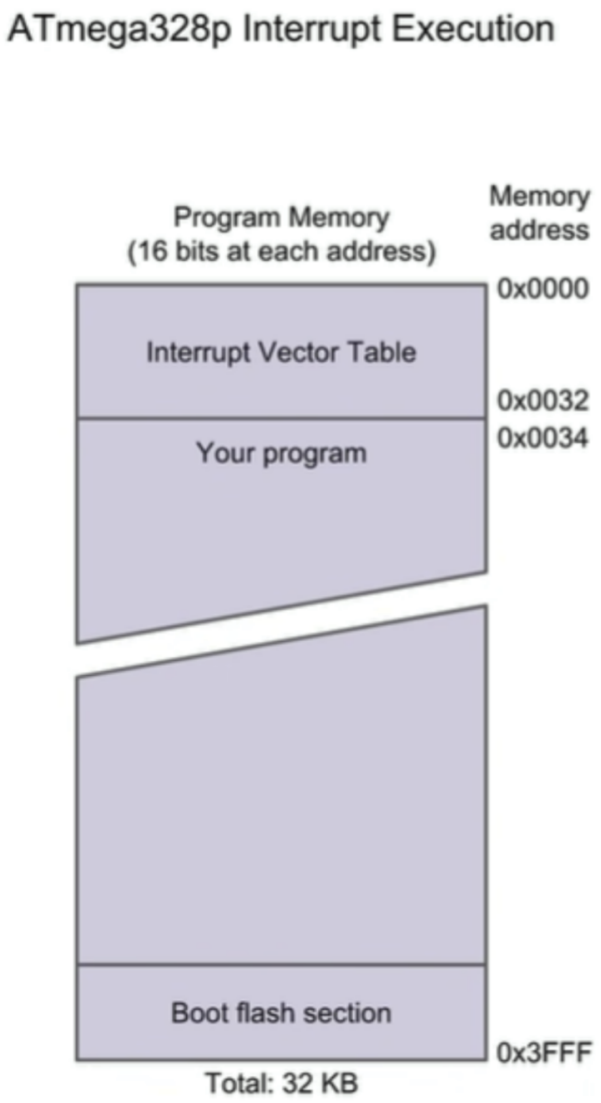
- **Role:** This is another 8-bit timer, similar to Timer0, but it is "independent" of the main time-keeping functions.
- **Use Case:** It is commonly used by the `tone()` library for generating audio frequencies.
- **Interrupt Potential:** It is an excellent choice for a custom periodic interrupt (e.g., checking a sensor every 1ms) without interfering with `delay()`.

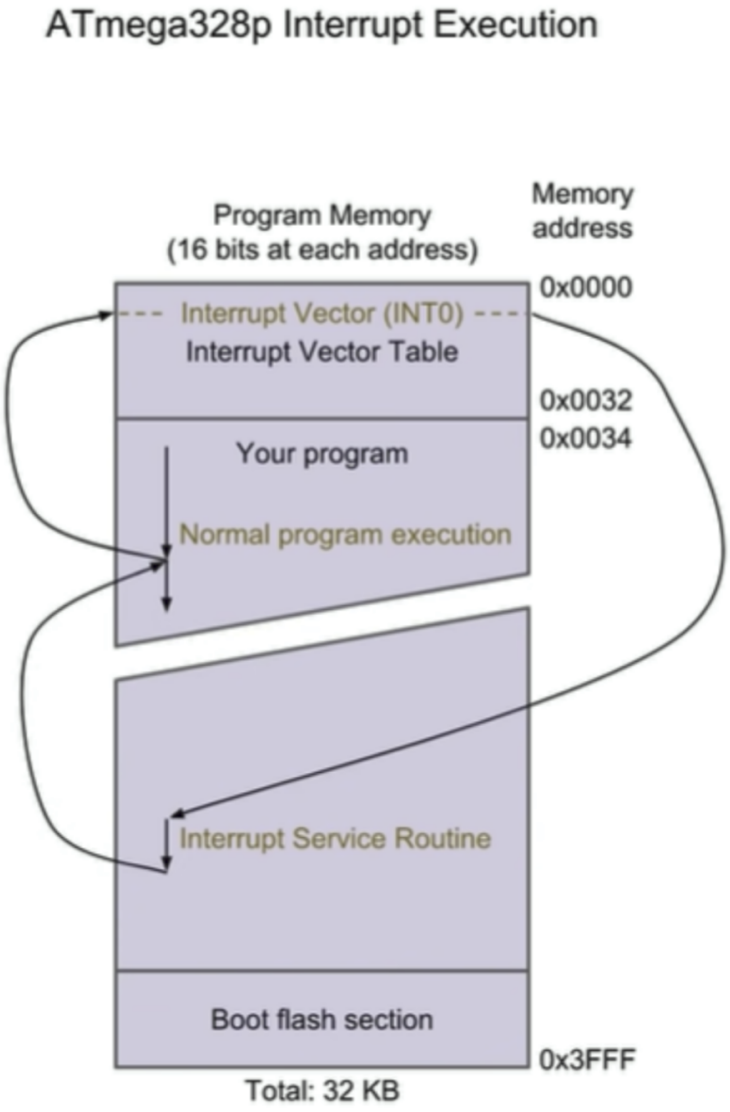
2. Interrupt Handler

On the ATMega328P microcontroller, there are three essential requirements that must met to enable an interrupt:

1. **Global Interrupt Enabled**: Interrupts must be allowed to occur globally across any part of the program. While the Arduino environment typically enables this by default, it is important to verify this when using different microcontrollers or during troubleshooting.
2. **Individual Interrupt Enabled**: Each specific interrupt must be permitted to occur via dedicated interrupt control registers.
3. **Interrupt Condition Met**: The microcontroller must receive a signal (either internal or external) that matches the predefined trigger criteria.

To enable a microcontroller to process interrupts, several steps must be followed. This module organizes these steps according to their placement in the code for easier implementation. The first step involves preparing the **Interrupt Handler** the specific code that will run when the interrupt is triggered.





Microcontrollers use a specialized memory block called the **Interrupt Vector Table**. This table stores the memory addresses of the programs to be executed for each specific type of interrupt.

Vector No.	Program Address	Source	Interrupt Definition
1	0x0000	RESET	External pin, Power-on Reset, Brown-Out Reset, 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

Vector No.	Program Address	Source	Interrupt Definition
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 A
14	0x001A	TIMER1_OVF	Timer/Counter1 Overflow
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

These addresses are fixed and cannot be changed. When an interrupt occurs, the microcontroller automatically jumps to the corresponding address in the Interrupt Vector Table to execute the associated program.

For example if an external interrupt (INT0) is triggered, the microcontroller will jump to address 0x0002 to execute the code defined in that location.

See the program address between INT0 and INT1, which are 0x0002 and 0x0004 respectively. The gap between these addresses is only 2 bytes, which is the size of a single instruction in AVR assembly language. This means that the interrupt handler for INT0 must be concise and fit within this limited space, often requiring the use of a jump instruction to redirect to a larger block of code

if necessary.

Example:

```
#define __SFR_OFFSET 0x00
#include "avr/io.h"

.org 0x0002 ; external interrupt 0 vector
    rjmp toggle_led

.global main

main:
 ; your main program here

toggle_led:
    ; main interrupt logic here
    in r16, PORTB
    eor r16, (1<<PB5)
    out PORTB, r16
    reti ; return from interrupt
```

From the above example, we have initialized the handler for the interrupt, and then created a routine named `toggle_led`, which contains the code that will be executed when the interrupt is triggered. When an INT0 occurs (for example, when a button connected to the INT0 pin is pressed), the microcontroller will jump to the `toggle_led` routine. After performing the necessary actions, we need to inform the microcontroller that the interrupt has been successfully handled and to continue with the main program. We can use the keyword `RETI`, which stands for Return from Interrupt, to achieve this.

> An interrupt will take priority over the main program, meaning that when an interrupt occurs, the microcontroller will temporarily halt the execution of the main program to execute the interrupt handler. Programmer must be cautious when writing interrupt handlers, as they should be efficient and not contain long-running code, to avoid delaying the main program for too long.

Using Vector Names

Instead of using raw addresses for interrupts, AVR assembly language provides predefined vector names that can be used to improve code readability. For example, instead of using `.org 0x0002` for the INT0 interrupt, we can use the predefined name `INT0_vect` as follows:

```
#define __SFR_OFFSET 0x00
#include "avr/io.h"

.global INT0_vect ; declare the interrupt vector as global
.global main

main:
 ; your main program here

INT0_vect: ; Toggle LED
    in r16, PORTB
    ldi r17, (1 << 5)
    eor r16, r17 ; Toggle PB5
    out PORTB, r16
    reti
```

> Pro tip: Just always use Vector Names instead of raw addresses :)

3. External Interrupt Registers

> For detailed information about the registers, please refer to the Atmega32p datasheet. here

From the previous section, we have set up the interrupt handler for external interrupt 0. Now, we will set up the registers to setup/initialize this interrupt. Let's say we want to toggle an LED on pin PB5 with falling edge trigger. We can write the following code in our setup section:

```
#define __SFR_OFFSET 0x00
#include "avr/io.h"

.global main
.global INT0_vect

main:
    ; initialize external interrupt 0
    ldi r16, (1<<ISC01) ; Set ISC01 bit = 1 | trigger on falling edge
    sts EICRA, r16 ; write to EICRA register to set the interrupt trigger condition
    ldi r16, (1<<INT0) ; set INT0 bit = 1 | enable external interrupt 0
    sts EIMSK, r16 ; write to EIMSK register to enable the interrupt

    ; enable global interrupts
    sei

INT0_vect:
    ; main interrupt logic here
    in r16, PORTB
    ldi r17, (1 << 5)
    eor r16, r17 ; Toggle PB5
    out PORTB, r16
    reti
```

Let's break it down register by register:

EICRA (External Interrupt Control Register A)

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	

- This register is used to configure the trigger condition for external interrupts. For INT0, we need to set the ISC01 bit to 1 and ISC00 bit to 0 for falling edge trigger. This is done by loading the value `(1<<ISC01)` into register r16 and then writing it to EICRA.

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)

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	

- This register is used to enable or disable external interrupts. To enable INT0, we need to set the INT0 bit to 1. This is done by loading the value `(1<<INT0)` into register r16 and then writing it to EIMSK.

sei (Set Global Interrupt Enable)

This instruction enables global interrupts. It **MUST be** called after setting up the individual interrupt configurations to allow the microcontroller to respond to interrupts.

You should get the rough idea for INT1 as well. You just need to set the ISC11 and ISC10 bits in EICRA for the trigger condition and set the INT1 bit in EIMSK to enable it. Read the datasheet for more details on the trigger conditions for INT1.

4. Internal/Timer Interrupts

Internal Interrupts

Now Internal interrupts or Timer interrupts is somewhat more complex then external interrupts. Before going for implementation let's understand on what use case we can use timer interrupts:

- **Replacing Delay Loops**: Instead of using blocking delay loops, timer interrupts can be used to perform tasks at regular intervals without halting the main program execution.
- **Real-Time Clock**: Timer interrupts can be used to create a real-time clock by counting the number of timer overflows or compare matches to keep track of time.
- **Event Scheduling**: Timer interrupts can be used to schedule events or tasks to occur at specific intervals, allowing for multitasking in embedded applications.

For now we will try replacing delay loops with TIMER1 compare match interrupt. We will set up the timer to generate an interrupt every 5 seconds and toggle an LED in the interrupt handler.

```
#define __SFR_OFFSET 0
#include <avr/io.h>

.global main
.global TIMER1_COMPA_vect ; The linker looks for this specific name

main:
    ; Set PB5 as output
    sbi DDRB, 5

    ; Clear r16 to use as a zero register
    clr r16
    sts TCCR1A, r16

    ; Set prescaler to 1024 and enable CTC mode (Clear Timer on Compare)
    ; CTC mode is better for toggling so you don't have to manually reset TCNT1
    ldi r17, (1 << WGM12) | (1 << CS12) | (1 << CS10)
    sts TCCR1B, r17

    ; Reset timer count
```

```
    sts TCNT1H, r16
    sts TCNT1L, r16

    ; Set compare match value (62499 = 0xF423)
    ldi r17, 0xF4
    sts OCR1AH, r17
    ldi r17, 0x23
    sts OCR1AL, r17

    ; Enable timer compare match interrupt
    ldi r17, (1 << OCIE1A)
    sts TIMSK1, r17

    sei ; Enable global interrupts

loop:
    rjmp loop

TIMER1_COMPA_vect:
    ; toggle LED on PB5
    in r16, PORTB
    ldi r17, (1 << 5)
    eor r16, r17 ; Toggle PB5
    out PORTB, r16
    reti
```

TCCR1A: Timer/Counter Control Register A for Timer1

- This register is used to configure the behavior of Timer1. In this code, it is set to 0, which means normal operation mode (no PWM or special modes).

TCCR1B: Timer/Counter Control Register B for Timer1

- This register is used to set the prescaler for Timer1. In this code, it is set to (1<<CS12 | 1<<CS10), which means a prescaler of 1024 is selected. This means the timer will count at a rate of the CPU clock divided by 1024.

TCNT1L and TCNT1H: Timer/Counter Register for Timer1

- These registers hold the current count value of Timer1. They are reset to 0 at the beginning of the main function to start counting from 0. TCNTL will increment with each timer tick, and when it reaches the value set in OCR1A, it will trigger the compare match interrupt.

OCR1AL and OCR1AH: Output Compare Register for Timer1

- These registers hold the value that Timer1 will compare against. When the timer count (TCNT1) matches the value in OCR1A, the compare match interrupt will be triggered. In this code, OCR1A is set to 62499, which corresponds to a 5-second interval with a 16 MHz clock and a prescaler of 1024.

TIMSK: Timer/Counter Interrupt Mask Register

- This register is used to enable or disable specific timer interrupts. In this code, it is set to (1<<OCIE1A), which enables the Timer1 Compare Match A interrupt.

sei: Set Global Interrupt Enable

- This instruction enables global interrupts, allowing the microcontroller to respond to interrupt requests. It must be called after configuring the interrupts to ensure that the microcontroller can handle them when they occur.

TO DO: Leaern more about timer interrupts (TIMER0, TIMER2) from the official datasheet