Embedded_session_4

Enhanced Embedded System Session 4 - Interrupts

Created: 2025-07-16
Author: Fares Hesham Mahmoud
Tags: AVR, Embedded system, External Interrupt
Status: #Formatted

Interrupt Fundamentals

An interrupt is a signal that temporarily halts the normal execution of a program to handle a time-sensitive event. Think of it like receiving an urgent phone call during a meeting - you must decide whether to answer immediately or wait until the meeting ends.

Real-World Analogy

- Main Program: Your daily schedule
- Interrupt: Emergency phone call
- ISR (Interrupt Service Routine): Handling the emergency
- Return: Continuing your original schedule

Interrupt Classification

1. External Interrupts

Generated by external devices or peripherals:

- Hardware Interrupts: ADC, TIMER, SPI, UART, External Interrupt pins, I2C
- Source: Outside the processor core but within the microcontroller
- Examples: Button press, sensor trigger, communication completion

2. Internal Interrupts

Generated within the processor core itself:

Traps (Exception Handling)

Automatic responses to error conditions:

- Division by Zero: Mathematical error handling
- Invalid Opcode: Illegal instruction detection
- Stack Overflow: Memory protection
- Page Fault: Memory management errors

Software Interrupts (SWI)

Programmatically triggered interrupts:

- Purpose: System calls, OS services
- Usage: ARM architectures, advanced embedded systems
- Example: Requesting OS services from user applications

Core Peripheral Interrupts

Interrupts from processor-integrated peripherals:

- SysTick Timer: System timing in ARM Cortex-M
- NVIC: Nested Vectored Interrupt Controller

Interrupt Management

For any interrupt to function, a hierarchical enabling system must be configured:

Three-Level Enable System

```
GIE (Global) → PIE (Peripheral) → Event Occurs → PIF (Flag Set)
```

1. PIF (Peripheral Interrupt Flag)

- Control: Hardware-controlled, software-readable
- Function: Indicates interrupt occurrence
- Clearing: Usually automatic or manual depending on peripheral

2. PIE (Peripheral Interrupt Enable)

- Control: Software-controlled
- Function: Enables/disables specific peripheral interrupts
- Values: 0 = Disabled, 1 = Enabled

3. GIE (Global Interrupt Enable)

- Control: Software-controlled
- Function: Master interrupt enable/disable

- Values: 0 = All interrupts disabled, 1 = Interrupts enabled

Programming Approaches

Super Loop System (Polling)

Structure:

```c
int main(void) {
    // Initialize peripherals
    while(1) {
        // Check peripheral 1
        // Check peripheral 2
        // Check peripheral N
        // Process events
    }
}
```

Advantages:

- Simplicity: Easy to understand and debug
- Predictability: Sequential execution flow
- Low Overhead: No context switching
- Resource Efficiency: Minimal memory usage

Disadvantages:

- Poor Responsiveness: Delayed event handling
- CPU Waste: Continuous polling even when idle
- Blocking: Long tasks delay other operations
- Scalability Issues: Difficult to manage multiple events
- Timing Jitter: Variable response times

Interrupt-Driven System (ISR)

Structure:

```c
// Background (Main Program)
int main(void) {
    // Initialize system
    // Enable interrupts
    while(1) {
        // Non-critical tasks
        // Sleep/idle when possible
    }
}

// Foreground (Interrupt Handlers)
ISR(TIMER0_OVF_vect) {
    // Handle timer overflow
    // Set flags, update variables
    // Keep it SHORT!
}
```

Advantages:

- **High Responsiveness**: Immediate event handling
- **Efficiency**: CPU idle when no events occur
- **Power Savings**: Sleep modes between interrupts
- **Concurrency**: Multiple event handling
- **Scalability**: Easier to manage complex systems
- **Precision**: Accurate timing for critical operations

Disadvantages:

- **Complexity**: Harder to design and debug
- **Context Switching**: CPU overhead for interrupt handling
- **Shared Resource Issues**: Race conditions possible
- **ISR Constraints**: Must be short and efficient
- **Debugging Challenges**: Asynchronous execution

PIC (Programmable Interrupt Controller)

The PIC manages interrupt priorities and vectoring:

Priority Queue Example

```
Interrupt Queue:    Execution Order:
1. Timer           1. Timer (highest priority)
2. ADC             2. ADC
3. SPI             3. SPI pending...
4. ADC             4. ADC (preempts SPI)
5. Timer           5. Timer (preempts ADC)
                   6. Return to ADC
                   7. Return to SPI
```

Vector Table

- Purpose: Maps interrupts to ISR addresses
- Structure: Fixed order based on priority
- Dynamic: ISR addresses may change with code organization
- Access: Hardware automatically vectors to correct ISR

ATmega32 External Interrupts

Available External Interrupts

- INT0: Pin PD2, highest priority
- INT1: Pin PD3, medium priority
- INT2: Pin PB2, lowest priority

Trigger Modes

INT0 & INT1 (Advanced Triggering)

MCUCR Register Control:

- ISC01:ISC00 for INT0
- ISC11:ISC10 for INT1

ISC01	ISC00	Trigger Mode
0	0	Low Level
0	1	Any Change
1	0	Falling Edge
1	1	Rising Edge

INT2 (Simple Triggering)

MCUCSR Register Control:

- ISC2 bit controls trigger mode

ISC2	Trigger Mode
0	Falling Edge
1	Rising Edge

Enable Configuration

GICR Register (General Interrupt Control Register):

- INT2: Bit 5
- INT0: Bit 6
- INT1: Bit 7

Implementation Example

```c
void EXT_INT0_Init(void) {
    // Configure trigger mode (falling edge)
    MCUCR |= (1 << ISC01);
    MCUCR &= ~(1 << ISC00);

    // Enable INT0
    GICR |= (1 << INT0);

    // Enable global interrupts
    sei();
}

ISR(INT0_vect) {
    // Handle external interrupt
    // Keep processing minimal
}
```

Best Practices

ISR Design Rules

1. Keep ISRs Short: Minimize processing time
2. Use Flags: Signal main program for complex processing
3. Avoid Blocking Operations: No delays, loops, or I/O in ISRs
4. Protect Shared Data: Use atomic operations or disable interrupts
5. Volatile Variables: Mark ISR-modified variables as volatile

Debugging Tips

1. LED Indicators: Visual interrupt confirmation
2. Serial Output: Debug information (use buffering)
3. Logic Analyzer: Hardware signal verification

4. Simulator: Step-through debugging in controlled environment

Performance Optimization

1. Interrupt Nesting: Allow higher priority interrupts
2. Priority Assignment: Critical tasks get higher priority
3. Load Balancing: Distribute processing between ISR and main
4. Power Management: Use sleep modes effectively

Common Pitfalls

1. ISR Too Long: Causes system instability
2. Shared Variable Issues: Race conditions and data corruption
3. Missing Volatile: Compiler optimization issues
4. Interrupt Storms: Too frequent interrupts overwhelming system
5. Stack Overflow: Deep interrupt nesting

Advanced Topics

Callback Functions

Function pointers that allow flexible ISR behavior:

```c
typedef void (*InterruptCallback)(void);
InterruptCallback ext_int0_callback = NULL;

void EXT_INT0_SetCallback(InterruptCallback callback) {
    ext_int0_callback = callback;
}

ISR(INT0_vect) {
    if(ext_int0_callback != NULL) {
        ext_int0_callback();
    }
}
```

Interrupt Latency

Time from interrupt signal to ISR execution:

- Hardware Latency: Signal detection and vectoring
- Software Latency: Context saving and ISR start
- Minimization: Optimize ISR entry and keep ISRs short

Maskable vs Non-Maskable Interrupts

Feature	Maskable Interrupts	Non-Maskable Interrupts
Control	Software enable/disable	Cannot be disabled
Priority	Programmable via vector table	Highest priority
Use Case	Normal event handling	Critical system events
Examples	Timer, ADC, UART	Reset, Watchdog, NMI
Response	Can be delayed	Immediate system action

Extra tips:

1. Non-printable ASCII Values

The range `0-31` and `127-255` includes control characters or extended characters not properly rendered by standard HD44780 LCDs.

- Values like `0x00` (null), `0x01`, ..., `0x1F` have no visible character (except custom characters `0x00-0x07` if defined).
- Values from `0x80` onward are not standard and may display garbage unless the LCD has a character ROM that supports extended ASCII or specific symbols.

```
for(uint16_t i = 32; i <= 126; i++){
  LCD_voidWriteData((u8)i);
   _delay_ms(200);
}
```

```
f ! " # $ % & ' ( ) * + , - . / 0 1 2 ... ~
```

Summary

Interrupts are fundamental to responsive embedded systems. While polling systems are simpler, interrupt-driven systems provide better performance and power efficiency for complex applications. Proper interrupt management requires understanding the hardware, careful ISR design, and attention to system-wide effects like shared resource protection and timing requirements.

The key to successful interrupt programming is balancing responsiveness with system stability through proper design, testing, and optimization practices.