Microcontroller Interrupt Handling

Interrupt handling is one of the most critical aspects of embedded systems programming, enabling responsive and efficient real-time applications. This comprehensive guide covers interrupt implementation from basic concepts to advanced optimization techniques.

Universal Interrupt Handling

General Implementation Steps

Every microcontroller follows these fundamental steps for interrupt handling:

Step 1: Implement Interrupt Handler and Vector Table Integration

- Create interrupt service routine (ISR) function
- Add function address to **Interrupt Vector Table (IVT)**
- Ensure proper function naming convention for your platform

Step 2: Enable Interrupts at Multiple Levels

- **Peripheral Level**: Enable interrupt generation in the peripheral device
- **Interrupt Controller Level**: Unmask interrupt in the interrupt controller
- **CPU Level**: Enable global interrupts in the CPU

This hierarchical approach provides fine-grained control and prevents unwanted interruptions during system initialization.

STM32 Interrupt Implementation

Initialization Sequence

STM32 microcontrollers use a systematic approach to interrupt configuration:

Step 1: Enable Interrupt in Peripheral

- Configure the peripheral to generate interrupt signals
- Set appropriate trigger conditions (edge/level, rising/falling)

Step 2: Enable Interrupt in NVIC

- Use **Nested Vector Interrupt Controller (NVIC)** to manage interrupt priorities
- Enable specific interrupt lines using `NVIC_EnableIRQ()`

IRQ Handler Implementation

Step 3: Implement IRQHandler Function

- Function name must **exactly match** the vector table entry
- Use the predefined weak function names from startup files

Step 4: Acknowledge Interrupt

- Clear interrupt flags to prevent immediate re-entry
- Essential for edge-triggered interrupts

Step 5: Implement Handling Logic

- Keep ISR code minimal and non-blocking
- Use flags or queues to communicate with main application

STM32 External Interrupt (EXTI) Configuration

External interrupts allow GPIO pins to trigger interrupt events, essential for handling button presses, sensor signals, and external device notifications.

Complete EXTI Setup Example

For a switch connected to **PA.0**, follow this systematic configuration:

GPIO Configuration

```
// Enable GPIO clock for Port A
RCC->AHB1ENR |= BV(0);

// Configure PA.0 as input
GPIOA->MODER &= ~(BV(1) | BV(0));

// Disable pull-up/pull-down resistors
GPIOA->PUPDR &= ~(BV(1) | BV(0));
```

EXTI Configuration

```
// Enable falling edge detection
EXTI->FTSR |= BV(0);

// Unmask EXTI0 interrupt
EXTI->IMR |= BV(0);

// Connect PA.0 to EXTI0 line via SYSCFG
SYSCFG->EXTICR[0] &= ~(BV(3)|BV(2)|BV(1)|BV(0));

// Enable EXTI0 interrupt in NVIC
NVIC_EnableIRQ(6); // EXTI0_IRQn = 6
```

Interrupt Handler Implementation

```
void EXTI0_IRQHandler(void) {
    // Step 1: Acknowledge interrupt by clearing pending flag
    EXTI->PR |= BV(0);

    // Step 2: Implement interrupt handling logic
```

```
    // Keep this section minimal and fast
    interrupt_flag = 1;  // Signal main application
}
```

Key Configuration Points:

- **FTSR**: Falling Trigger Selection Register
- **IMR**: Interrupt Mask Register
- **PR**: Pending Register (for acknowledgment)
- **SYSCFG->EXTICR**: External Interrupt Configuration Register

---

Understanding Weak Functions in GCC

Weak functions provide a elegant mechanism for default implementations that can be overridden without linker errors.

Assembly Implementation

```
.weak   EXTI0_IRQHandler
.thumb_set EXTI0_IRQHandler,Default_Handler
```

C Programming Implementation

```
// Method 1: Using __attribute__
void __attribute__((weak)) EXTI0_IRQHandler(void);

// Method 2: Using #pragma
#pragma weak EXTI0_IRQHandler = Default_Handler
```

Benefits:

- **Graceful Fallback**: Undefined handlers automatically use `Default_Handler`
- **No Linker Errors**: Missing ISR implementations don't cause build failures
- **Flexible Development**: Handlers can be implemented incrementally

---

Interrupt Handling Best Practices

Top-Half vs Bottom-Half Architecture

Modern interrupt design separates time-critical and deferrable operations:

Top-Half (ISR)

- **Purpose**: Handle time-critical, non-blocking operations
- **Operations**:
  - Acknowledge interrupt
  - Read/clear hardware flags
  - Set software flags
  - Minimal data processing
- **Execution Context**: Interrupt context with limited stack

Bottom-Half (Deferred Processing)

- **Purpose**: Handle complex, blocking operations
- **Operations**:
  - Complex algorithms
  - Memory allocation
  - File I/O operations
  - Network communication
- **Execution Context**: Normal application context

ISR Design Guidelines

1. **Keep ISRs Short**: Minimize interrupt latency for other interrupts
2. **Non-Blocking Operations Only**: Avoid delays, polling, or waiting
3. **Priority Management**: Lower priority interrupts are masked during ISR execution
4. **Context Preservation**: Compiler handles register saving/restoring automatically

---

GCC Optimization and the `volatile` Keyword

Optimization Impact on Interrupt Variables

GCC optimization can dramatically affect interrupt-related code behavior:

Without Optimization (-O0)

```
uint32_t exti0_flag = 0;

// Main loop - No optimization
while(exti0_flag == 0)
    ;
/*
 * Assembly: Variable read from memory each iteration
 * LDR r7, =exti0_flag
 * loop:
 *     LDR r0, [r7]     // Read from memory
 *     CMP r0, #0
 *     BEQ loop
 */
```

With Optimization (-O3)

```c
uint32_t exti0_flag = 0;

// Main loop - Optimized (BROKEN!)
while(exti0_flag == 0)
    ;
/*
 * Assembly: Variable read only once, then cached
 * LDR r7, =exti0_flag
 * LDR r0, [r7]        // Read once
 * loop:
 *     CMP r0, #0       // Use cached value
 *     BEQ loop         // Infinite loop!
 */
```

The `volatile` Solution

The `volatile` keyword prevents optimization of variables that can change unexpectedly:

```c
volatile uint32_t exti0_flag = 0;

// Main loop - Correctly optimized
while(exti0_flag == 0)
    ;
/*
 * Assembly: Variable read from memory each iteration
 * LDR r7, =exti0_flag
 * loop:
 *     LDR r0, [r7]     // Always read from memory
 *     CMP r0, #0
 *     BEQ loop
 */
```

When to Use `volatile`

Variables should be declared `volatile` when:

- **Modified in ISRs**: Changed by interrupt handlers
- **Modified in Other Threads**: Multi-threaded environments
- **Memory-Mapped I/O**: Hardware registers that change independently
- **Shared Between Contexts**: Any variable accessed from multiple execution contexts

---

Advanced Memory Qualifiers

`const volatile` Combination

Used for memory-mapped hardware registers that are read-only:

```
// Example: STM32 GPIO Input Data Register
typedef struct {
    // ... other members
    const volatile uint32_t IDR;  // Input Data Register
    // ... other members
} GPIO_TypeDef;

// Correct usage - reading hardware register
while(GPIOA->IDR == 0)  // volatile prevents optimization
    ;

// Compiler error - attempting to write read-only register
// GPIOA->IDR = 0xF;    // const prevents modification
```

Benefits:

- `volatile`: Prevents compiler optimization of register reads

- `const`: Prevents accidental writes to read-only registers
- **Type Safety**: Compiler catches programming errors at build time

`static const volatile` Combination

Adds scope limitation to the memory qualifier:

```
// File scope limitation
static const volatile uint32_t *PERIPHERAL_REG = (uint32_t*)0x40020000;
```

Complete Qualifier Meanings:

- `volatile`: Disable compiler optimization
- `const`: Prevent write operations
- `static`: Limit scope to current file/function

Practical Implementation Examples

UART Receive Interrupt

```
void UART_Init(void) {
    // ... UART configuration code ...
    USART1->CR1 |= USART_CR1_RXNEIE;  // Enable RX interrupt
    NVIC_EnableIRQ(USART1_IRQn);
}

void USART1_IRQHandler(void) {
    if(USART1->SR & USART_SR_RXNE) {
        received_char = USART1->DR;    // Reading DR clears flag
        rx_data_available = 1;         // Signal main application
```

```
    }
}
```

I2C Protocol: A Comprehensive Guide to Inter-Integrated Circuit Communication

I2C (Inter-Integrated Circuit) is one of the most widely used serial communication protocols in embedded systems, designed for short-distance communication between microcontrollers and peripheral devices. Developed by Philips (now NXP) in 1982, I2C has become the backbone of modern embedded system communication.

Physical Characteristics

Communication Architecture

- **Type**: Bus protocol with Master-Slave architecture
- **Communication Mode**: **Half duplex** - bidirectional communication, but not simultaneous
- **Network Topology**: Multi-master, multi-slave bus system

Wiring Requirements

I2C is elegantly simple in its hardware requirements:

Two-Wire Protocol:

- **SDA (Serial Data)**: Bidirectional data line for transmitting and receiving data
- **SCL (Serial Clock)**: Clock line generated by the master device
- **Additional**: Pull-up resistors required on both lines (typically 4.7kΩ for 5V systems)

Note: The minimal wiring is one of I2C's greatest advantages, reducing PCB complexity and cost compared to parallel interfaces.

Voltage Levels

- **Standard**: TTL voltage levels (0V = Logic 0, +3.3V/+5V = Logic 1)

- **Pull-up Configuration**: Both SDA and SCL lines are open-drain with external pull-up resistors
- **Logic Levels**: Lines are pulled high by default; devices pull them low to communicate

Operating Frequencies

I2C supports multiple speed modes to accommodate different application requirements:

Mode	Frequency	Typical Use Case
Standard Mode	100 kHz	Basic sensors, RTCs, EEPROMs
Fast Mode	400 kHz	More complex peripherals
Fast Mode Plus	1 MHz	High-speed applications
High-Speed Mode	3.4 MHz	Advanced applications (requires special considerations)
Ultra Fast Mode	5 MHz	Unidirectional, specialized applications

Higher frequencies require shorter bus lengths and careful PCB design considerations.

Logical Characteristics

I2C Device Operating Modes

Every I2C device can operate in one of four modes depending on the current transaction:

1. **Master Transmitter (MT)**: Master sending data to slave
2. **Master Receiver (MR)**: Master requesting data from slave
3. **Slave Transmitter (ST)**: Slave sending data to master
4. **Slave Receiver (SR)**: Slave receiving data from master

Data Transfer Mechanism

Data Frame Structure

Each data transmission consists of:

- **8 Data Bits**: Transmitted MSB first
- **1 ACK/NACK Bit**: Acknowledgment from receiver

Data Transaction Flow:

```
Transmitter → [8 Data Bits] + [ACK=1] → Receiver
Transmitter ← [Acknowledge (ACK=0)] ← Receiver
```

Address Frame Structure

Every communication begins with an address frame:

- **7-bit Slave Address**: Identifies target device
- **1 R/W Bit**: Direction (0=Write, 1=Read)
- **1 ACK Bit**: Acknowledgment from addressed slave

Address Transaction Flow:

```
Master → [7-bit Address] + [R/W'] + [ACK=1] → Slave
Master ← [Acknowledge (ACK=0)] ← Slave
```

I2C Communication Sequences

Single Byte Data Write

Complete sequence for writing one byte to a slave device:

```
START → I2C_Slave_Addr_WR → ACK → Internal_Address → ACK → Data_Byte → ACK → STOP
 (M)         (MT)         (SR)        (MT)         (SR)      (MT)      (SR)   (M)
```

Legend: M=Master, MT=Master Transmitter, SR=Slave Receiver

Multi-Byte Data Write

For writing multiple consecutive bytes:

```
START → I2C_Slave_Addr_WR → ACK → Internal_Address → ACK → Data_Byte1 → ACK → Data_Byte2 → ACK → ... → STOP
 (M)         (MT)         (SR)        (MT)         (SR)      (MT)       (SR)      (MT)       (SR)        (M)
```

Single Byte Data Read

Reading requires a write operation followed by a restart:

```
START → I2C_Slave_Addr_WR → ACK → Internal_Address → ACK → REPEAT_START → I2C_Slave_Addr_RD → ACK → Data_Byte → NACK → STOP
 (M)         (MT)         (SR)        (MT)         (SR)        (M)            (MT)         (SR)      (ST)      (MR)   (M)
```

Legend: ST=Slave Transmitter, MR=Master Receiver

Multi-Byte Data Read

For reading multiple bytes, ACK continues the transfer, NACK terminates:

```
START → I2C_Slave_Addr_WR → ACK → Internal_Address → ACK → REPEAT_START → I2C_Slave_Addr_RD → ACK → Data_Byte1 → ACK →
Data_Byte2 → NACK → STOP
 (M)         (MT)         (SR)        (MT)         (SR)        (M)            (MT)         (SR)      (ST)       (MR)
(ST)       (MR)   (M)
```

Advanced I2C Features

Clock Stretching

A crucial flow control mechanism:

- **Purpose**: Allows slave devices to control communication timing
- **Implementation**: Slave pulls SCL line low when not ready to process data
- **Effect**: Master waits until SCL returns high before continuing communication
- **Benefit**: Prevents data loss when slave needs processing time

Bus Arbitration

Handles multi-master scenarios:

- **Conflict Detection**: When multiple masters attempt simultaneous START conditions
- **Resolution**: Device transmitting logic '0' wins over device transmitting logic '1'
- **Graceful Fallback**: Losing master automatically becomes slave
- **Bus Integrity**: Ensures only one master controls bus at any time

---

Error Detection and Handling

I2C protocol includes several error detection mechanisms:

Common Error Conditions

1. **Bus Error**
    - **Cause**: Invalid START/STOP condition during active communication
    - **Master Behavior**: Retains bus ownership until proper termination
    - **Recovery**: May require bus reset

2. **Acknowledgment Error**
    - **Detection**: No slave pulls SDA low after address/data transmission
    - **Indication**: Addressed device not present or not responding
    - **Handling**: Master should terminate transaction with STOP condition
3. **Arbitration Lost Error**
    - **Scenario**: Multi-master environment collision
    - **Action**: Losing master switches to slave mode
    - **Recovery**: Automatic, transparent to application
4. **Read Overrun**
    - **Problem**: Microcontroller fails to read received data before next byte arrives
    - **Result**: Previous data overwritten
    - **Prevention**: Implement proper interrupt handling or use DMA
5. **CRC Error (STM32 specific)**
    - **Feature**: Advanced microcontrollers may implement CRC checking
    - **Purpose**: Enhanced data integrity verification
    - **Benefit**: Detects communication errors beyond simple ACK/NACK

Common I2C Device Examples

Understanding real-world implementations helps solidify I2C concepts:

DS1307 (Real-Time Clock)

- **Slave Addresses**: 0xD0 (Write) | 0xD1 (Read)
- **Internal Addressing**: 1 byte
    - RTC registers: Seconds, Minutes, Hours, Day, Date, Month, Year
    - NVRAM: 56 bytes of battery-backed memory

- **Typical Use**: Timekeeping in embedded systems

AT24C256 (EEPROM)

- **Slave Addresses**: 0xA0 (Write) | 0xA1 (Read)
- **Internal Addressing**: 2 bytes (16-bit memory address)
- **Capacity**: 256 Kbit (32K x 8) non-volatile storage
- **Page Write**: Supports up to 64-byte page writes for efficiency

PCF8574 (I/O Expander)

- **Slave Addresses**: 0x4E (Write) | 0x4F (Read)
- **Internal Addressing**: None (direct I/O port access)
- **Functionality**: 8-bit bidirectional I/O expansion
- **Applications**: LED control, switch input, GPIO expansion

---

Advanced Addressing: 10-bit Extension

While 7-bit addressing supports 128 devices (with some reserved), 10-bit addressing extends this capability:

10-bit Address Structure

- **First Byte**: `11110XX0/1` (where XX are upper 2 bits of address)
- **Second Byte**: Lower 8 bits of the 10-bit address
- **Total Devices**: Up to 1024 unique addresses

Example Implementation

For 10-bit address `0b1010101010` (0x2AA):

First Byte: `11110` + `10` + `0` = `0xF4` (for write) **Second Byte**: `10101010` = `0xAA`

Advantages

- **Scalability**: Supports 1024 devices vs 128 for 7-bit
- **Collision Avoidance**: Reduces address conflicts in complex systems
- **Future-Proofing**: Accommodates growing system complexity

Implementation Considerations

- **Backward Compatibility**: 7-bit devices can coexist on same bus
- **Software Support**: Requires driver support for 10-bit addressing
- **Overhead**: Slightly longer address phase but negligible performance impact

---

I2C State Machine Implementation

I2C devices operate as state machines with standardized status codes:

Status Code System

- **Format**: 5-bit status codes
- **Purpose**: Indicates current transaction state and success/failure conditions
- **Implementation**: Allows robust error handling and transaction management
- **Benefits**: Enables predictable communication protocols and debugging

Typical State Progression

1. **Idle State**: Bus available, both lines high
2. **Start Condition**: Master initiates communication
3. **Address Phase**: Master transmits slave address
4. **Data Phase**: Actual data transmission
5. **Stop Condition**: Master terminates communication
6. **Error States**: Various error conditions with specific codes

---

Best Practices and Design Considerations

Hardware Design

- **Pull-up Resistors**: Choose appropriate values (1.8kΩ-10kΩ depending on bus capacitance)
- **Bus Length**: Keep traces short for higher frequencies
- **Decoupling**: Use proper power supply decoupling on all devices

Software Implementation

- **Timeout Handling**: Implement timeouts for all I2C operations
- **Error Recovery**: Plan for bus error recovery mechanisms
- **Address Management**: Maintain clear documentation of device addresses
- **Clock Stretching Support**: Ensure master supports slave clock stretching

Performance Optimization

- **Burst Transfers**: Use multi-byte operations when possible
- **Frequency Selection**: Choose appropriate clock frequency for your application
- **Bus Loading**: Consider total capacitive load when adding devices