SPI Protocol: High-Speed Serial Peripheral Interface Communication

SPI (Serial Peripheral Interface) is one of the most widely used synchronous serial communication protocols in embedded systems, offering high-speed data transfer capabilities with full-duplex communication. Developed by Motorola (now NXP), SPI has become the standard for interfacing with peripherals requiring fast data exchange.

Physical Characteristics

Communication Architecture

- **Type**: **Full-duplex protocol** - simultaneous bidirectional communication
- **Topology**: **Bus protocol** supporting master-slave architecture
- **Speed**: **High-speed** communication (typically MHz range)

Signal Line Requirements

SPI uses a **4-wire protocol** with dedicated signal lines:

Signal	Alternative Names	Function	Direction
MOSI	SDO (Serial Data Output)	Master Out Slave In	Master → Slave
MISO	SDI (Serial Data Input)	Master In Slave Out	Slave → Master
SCLK/SCK	Serial Clock	Clock signal generated by master	Master → Slave
SS/CE	Chip Select/Chip Enable	Slave selection signal	Master → Slave

Internal Architecture

SPI's elegance lies in its simple yet effective internal design:

- **Single Shift Register**: Used in both master and slave for full-duplex communication
- **Circular Data Path**: Data shifts out of master while simultaneously shifting in from slave

- **Clock Synchronization**: All data transfers synchronized to master-generated clock

Voltage Levels

- **Standard**: TTL voltage levels
- **Logic 0**: 0V
- **Logic 1**: 3.3V or 5V (depending on system voltage)

Logical Characteristics and Timing Modes

Clock Configuration Parameters

SPI flexibility comes from two key timing configuration parameters:

CPOL (Clock Polarity)

- **CPOL = 0**: Clock idles at **logic low (0V)**
- **CPOL = 1**: Clock idles at **logic high (3.3V/5V)**

CPHA (Clock Phase)

- **CPHA = 0**: Data captured on **leading edge**, shifted on trailing edge
- **CPHA = 1**: Data captured on **trailing edge**, shifted on leading edge

SPI Modes Combination

Mode	CPOL	CPHA	Clock Idle	Data Capture	Common Usage
Mode 0	0	0	Low	Leading (Rising)	Most common, SD cards
Mode 1	0	1	Low	Trailing (Falling)	Some sensors
Mode 2	1	0	High	Leading (Falling)	Less common

Mode	CPOL	CPHA	Clock Idle	Data Capture	Common Usage
Mode 3	1	1	High	Trailing (Rising)	Some displays

Mode 0 is the most widely supported and should be the default choice unless device datasheet specifies otherwise.

Data Transfer Mechanisms

Single Byte Transfer

The fundamental SPI transaction involves simultaneous bidirectional data exchange:

```
Master initiates transfer:
SS = 0 (Select slave)
→ MOSI: Master sends 8 data bits
← MISO: Slave sends 8 data bits simultaneously
SS = 1 (Deselect slave)
```

Key Points:

- Slave selection is **master's responsibility**
- Data exchange is **always bidirectional** (even if only one direction carries useful data)
- **Byte-oriented**: Standard transfers are 8-bits, though some devices support 16-bit modes

Multi-Byte Write Operation

For writing data to specific device registers or memory locations:

```
Transaction sequence:
SS = 0
```

```
Master → Address1, Address2, Data1, Data2, ..., DataN → Slave
SS = 1
```

Implementation considerations:

- **Internal addressing**: 1-byte or 2-byte addresses depending on device capacity
- **Continuous transfer**: All bytes sent in single SS assertion period
- **Address auto-increment**: Many devices automatically increment address for sequential writes

Multi-Byte Read Operation

Reading data requires address specification followed by data retrieval:

```
Transaction sequence:
SS = 0
Master → Address1, Address2 → Slave
Slave → Data1, Data2, ..., DataN → Master
SS = 1
```

Read operation characteristics:

- **Address phase**: Master specifies starting read address
- **Data phase**: Slave responds with requested data bytes
- **Turnaround time**: Some devices require clock cycles between address and data phases

---

Common SPI Applications and Devices

SPI's high-speed capabilities make it ideal for various embedded applications:

Storage Devices

- **SD Cards**: Mass storage with high-speed data transfer requirements
- **EEPROM**: Non-volatile memory for configuration data
- **Flash Memory**: Program and data storage

Sensor Interfaces

- **Accelerometers**: Motion sensing (e.g., LIS3DSH)
- **Gyroscopes**: Angular velocity measurement
- **Temperature sensors**: High-precision measurements
- **Pressure sensors**: Environmental monitoring

Display Interfaces

- **LCD Displays**: Character and graphic displays
- **7-Segment Displays**: Numeric output with shift register drivers
- **OLED Displays**: High-resolution graphics

Real-Time Clock (RTC)

- Battery-backed timekeeping with non-volatile RAM
- High-precision timing applications

Error Detection and Handling

Common Error Conditions

Read Overrun

- **Cause**: New data arrives before previous data is read from receive buffer
- **Result**: Loss of previous data due to buffer overwrite
- **Prevention**: Implement proper interrupt handling or use DMA transfers

Write Collision

- **Cause**: Attempting to write new data before current transmission completes
- **Result**: Current transmission corrupted or incomplete
- **Prevention**: Check transmission status flags before new writes

Mode Fault

- **Condition**: SPI master mode device forced into slave mode by external signal
- **Detection**: Hardware detection when SS pin driven low while in master mode
- **Recovery**: Reset SPI peripheral and reconfigure as master

Slave Abort

- **Scenario**: Slave device stops responding during active communication
- **Symptoms**: No acknowledge signals or unexpected data patterns
- **Handling**: Implement timeout mechanisms and error recovery procedures

STM32 CRC Error

- **Feature**: Hardware CRC calculation and verification in STM32 SPI peripherals
- **Purpose**: Enhanced data integrity verification
- **Usage**: Enable CRC generation/checking for critical data transfers

---

Advanced SPI Configurations

Multiple Slave Management

Individual Chip Select Method

- **Implementation**: Dedicated GPIO pin for each slave device
- **Advantages**: Simple, independent slave control
- **Disadvantages**: High pin count for multiple slaves

Multiplexer-Based Selection

- **Implementation**: 3-to-8 decoder/multiplexer for slave selection
- **Advantages**: Reduces pin count (3 pins for up to 8 slaves)
- **Disadvantages**: Additional hardware complexity, sequential access only

Daisy Chaining

Daisy chaining enables efficient multi-slave communication with minimal pin usage:

Configuration

- **Single SS**: One chip select controls entire chain
- **Data Flow**: MOSI → Slave1 → Slave2 → ... → SlaveN → MISO
- **Clock Distribution**: Common SCLK to all slaves

Advantages

- **Pin Efficient**: Single SS pin regardless of slave count
- **Synchronized Updates**: All slaves updated simultaneously
- **Cost Effective**: Minimal additional hardware

Applications

- **LED Drivers**: Cascaded shift registers for large displays
- **DAC Chains**: Multi-channel analog output systems
- **Sensor Arrays**: Sequential sensor data collection

References for detailed implementation: Maxim Integrated SPI Daisy Chain Guide and Best Microcontroller Projects SPI Interface

3-Wire SPI Variant

Some applications use a simplified 3-wire SPI configuration:

Implementation

- **Combined Data Line**: MOSI and MISO tied together (bidirectional)
- **Half-Duplex**: Alternating transmit/receive phases
- **Applications**: Simple sensors, basic configuration interfaces

Considerations

- **Timing Critical**: Requires precise turnaround timing
- **Device Support**: Not universally supported by all SPI peripherals
- **Protocol Dependent**: Specific timing requirements vary by device

---

STM32 SPI Implementation

Based on the provided code, here's the complete STM32F407VG SPI implementation:

Hardware Configuration

GPIO Setup for SPI1

```
// Enable GPIO clocks
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN | RCC_AHB1ENR_GPIOEEN;

// Configure PA5 (SCK), PA6 (MISO), PA7 (MOSI) for SPI1
GPIOA->MODER |= BV(5*2+1) | BV(6*2+1) | BV(7*2+1);  // Alt function mode
GPIOA->MODER &= ~(BV(5*2) | BV(6*2) | BV(7*2));

// Set alternate function to AF5 (SPI1)
GPIOA->AFR[0] |= (5 << (5*4)) | (5 << (6*4)) | (5 << (7*4));

// Configure PE3 as GPIO for Chip Select
GPIOE->MODER |= BV(3*2);      // Output mode
GPIOE->MODER &= ~BV(3*2+1);
```

SPI Peripheral Configuration

```
// Enable SPI1 peripheral clock
RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;

// Configure SPI: Master mode, Software SS management, 2MHz bit rate
SPI1->CR1 = SPI_CR1_MSTR |        // Master mode
            SPI_CR1_SSI |         // Internal SS signal
            SPI_CR1_SSM |         // Software SS management
            SPI_CR1_BR_1;         // Baud rate: fpclk/8 = 2MHz

// Standard SPI frame format, no interrupts/DMA
SPI1->CR2 = 0x0000;

// Enable SPI peripheral
SPI1->CR1 |= SPI_CR1_SPE;
```

Data Transfer Functions

```
uint16_t SpiTransfer(uint16_t data) {
    // Wait until transmit buffer empty
    while(!(SPI1->SR & SPI_SR_TXE));

    // Write data to data register
    SPI1->DR = data;

    // Wait until receive buffer full
    while(!(SPI1->SR & SPI_SR_RXNE));

    // Return received data
```

```
    return SPI1->DR;
}

void SpiWrite(uint8_t internalAddr, uint8_t data[], uint8_t size) {
    SpiCSEnable();                    // Assert chip select

    internalAddr &= ~BV(7);           // Clear MSB for write operation
    SpiTransmit(internalAddr);        // Send register address

    for(int i = 0; i < size; i++) {
        SpiTransmit(data[i]);         // Send data bytes
    }

    SpiCSDisable();                   // Deassert chip select
}

void SpiRead(uint8_t internalAddr, uint8_t data[], uint8_t size) {
    SpiCSEnable();                    // Assert chip select

    internalAddr |= BV(7);            // Set MSB for read operation
    SpiTransmit(internalAddr);        // Send register address

    for(int i = 0; i < size; i++) {
        data[i] = SpiReceive();       // Read data bytes
    }

    SpiCSDisable();                   // Deassert chip select
}
```

LIS3DSH Accelerometer: SPI Device Implementation

The LIS3DSH is an excellent example of a modern SPI-based sensor, demonstrating practical SPI protocol usage:

Device Characteristics

- **Interface**: SPI Mode 0 or Mode 3 (configurable)
- **Data Resolution**: 16-bit signed values for each axis
- **Address Space**: 8-bit register addressing
- **Multi-byte Support**: Auto-increment addressing for sequential reads

Key Registers

Control Register 4 (CTRL4 - 0x20)

```
// Configuration for 25 Hz data rate, all axes enabled
#define CR4_ODR_25   BV(6)    // Output data rate: 25 Hz
#define CR4_XYZ_EN   (BV(0) | BV(1) | BV(2))  // Enable X, Y, Z axes
```

Status Register (STATUS - 0x27)

```
// Data availability flags
#define STATUS_XYZ_DA  BV(3)  // XYZ data available
#define STATUS_X_DA    BV(0)  // X-axis data available
#define STATUS_Y_DA    BV(1)  // Y-axis data available
#define STATUS_Z_DA    BV(2)  // Z-axis data available
```

Output Registers

- **ACCEL_X (0x28)**: X-axis low byte, X-axis high byte at 0x29
- **ACCEL_Y (0x2A)**: Y-axis low byte, Y-axis high byte at 0x2B
- **ACCEL_Z (0x2C)**: Z-axis low byte, Z-axis high byte at 0x2D

Accelerometer Implementation

Initialization Sequence

```
void AccelInit(void) {
    // Initialize SPI peripheral
    SpiInit();

    // Configure accelerometer for operation
    uint8_t config = CR4_ODR_25 | CR4_XYZ_EN;  // 25Hz, all axes
    SpiWrite(ACCEL_CR4, &config, 1);
}
```

Data Ready Polling

```
int AccelWaitForChange(void) {
    uint8_t status;

    do {
        SpiRead(ACCEL_STATUS, &status, 1);
    } while(!(status & STATUS_XYZ_DA));    // Wait for data ready

    return 1;  // Data available
}
```

Acceleration Data Reading

```
void AccelRead(AccelData_t *data) {
    uint8_t values[2];

    // Read X-axis (low byte first, then high byte)
    SpiRead(ACCEL_X, values, 2);
    data->x = values[0] | ((int16_t)values[1] << 8);
```

```
    // Read Y-axis
    SpiRead(ACCEL_Y, values, 2);
    data->y = values[0] | ((int16_t)values[1] << 8);

    // Read Z-axis
    SpiRead(ACCEL_Z, values, 2);
    data->z = values[0] | ((int16_t)values[1] << 8);
}
```

SPI Protocol Usage

The LIS3DSH demonstrates several important SPI concepts:

Read/Write Bit Encoding

- **Write Operation**: MSB (bit 7) of address = 0
- **Read Operation**: MSB (bit 7) of address = 1
- **Auto-increment**: Bit 6 = 1 for multi-byte sequential access

Multi-byte Read Operation

```
Transaction for reading X-axis data:
SS Low → Address(0xA8) → Data_Low ← Data_High ← SS High
         (0x28 | 0x80)   (X_L)      (X_H)
```

Multi-byte Write Operation

```
Transaction for configuration:
SS Low → Address(0x20) → Config_Data → SS High
         (Write to CR4)   (0x67)
```

Assignments

1. Set I2C LCD display to 1-Line and write your full name on it. Enable scrolling LCD display.
2. Input a string from UART (using polling). If it starts from "L1", display it on I2C LCD Line 1, otherwise display it on I2C LCD Line 2.