Phần 0. Cách dùng OSC để xem dạng sóng

1. Giới thiệu

Trong hướng dẫn này, chúng ta sẽ tìm hiểu cách chạy mô phỏng giao tiếp SPI trên phần mềm Proteus và cách quan sát dạng sóng một cách hiệu quả nhất. Mục tiêu là giúp bạn khắc phục các vấn đề về méo dạng sóng và hiểu rõ cách thiết lập mô phỏng để đạt kết quả chính xác.

2. Vấn đề gặp phải trong mô phỏng trước đây

2.1. Nhầm lẫn về hàm `Serial.print`

Trong các video trước, có đề cập rằng việc sử dụng hàm `Serial.print` trong mã Arduino có thể gây ra méo dạng sóng trong mô phỏng, dẫn đến xuất hiện các xung hình tam giác thay vì xung vuông như mong đợi.

2.2. Nguyên nhân thực sự gây méo dạng sóng

Tuy nhiên, sau khi tìm hiểu kỹ, nguyên nhân thực sự không phải do hàm `Serial.print`, mà do:

- **Sử dụng sai cách dao động ký trong Proteus**: Việc cấu hình không đúng các tham số trong dao động ký dẫn đến việc hiển thị dạng sóng không chính xác.
- **Vấn đề về tốc độ mô phỏng**: Máy tính không đáp ứng kịp tốc độ xử lý của mô phỏng, khiến dạng sóng bị méo.

3. Hướng dẫn chi tiết cách chạy mô phỏng trên Proteus

3.1. Chuẩn bị và khởi động mô phỏng

- **Mở dự án mô phỏng trong Proteus**: Đảm bảo rằng bạn đã thiết lập đầy đủ các linh kiện cần thiết cho giao tiếp SPI.
- **Chạy mô phỏng**: Nhấn nút "Play" để bắt đầu chạy mô phỏng.

3.2. Cách sử dụng công cụ debug

- **Mở công cụ Debug**: Trong khi mô phỏng đang chạy, mở cửa sổ Debug bằng cách chọn "Debug" > "Open".
- **Chọn bộ phân tích logic (Logic Analyzer)**: Sử dụng công cụ này để bắt và hiển thị dạng sóng của các tín hiệu.

3.3. Thiết lập các kênh và trigger

- **Xóa các thiết lập cũ nếu cần**: Nếu các thiết lập trước đó không chính xác hoặc bị lỗi, hãy xóa chúng và thiết lập lại từ đầu.

- **Thêm các kênh tín hiệu**: Kết nối các chân tín hiệu (SCK, MOSI, MISO, SS) vào các kênh của bộ phân tích logic.
- **Thiết lập trigger (chích gỡ)**:
    - Chọn kênh tín hiệu cần làm trigger, ví dụ kênh SS.
    - Đặt điều kiện trigger là "từ cao xuống thấp" để bắt đầu quá trình truyền dữ liệu khi chân SS được kích hoạt.

3.4. Phóng to và thu nhỏ dạng sóng

- **Sử dụng chuột giữa để phóng to/thu nhỏ**: Lăn con lăn chuột giữa để điều chỉnh mức độ zoom của dạng sóng.
- **Dừng chế độ Auto nếu cần**: Trước khi phóng to, nên dừng chế độ Auto để dạng sóng không bị nhảy và dễ quan sát hơn.
- **Điều chỉnh vị trí dạng sóng**: Sử dụng thanh trượt hoặc kéo dạng sóng để đặt chúng ở vị trí dễ quan sát.

3.5. Điều chỉnh để xem dạng sóng tốt nhất

- **Khắc phục méo dạng sóng**:
    - Nếu dạng sóng bị méo hoặc không rõ ràng, có thể do tốc độ mô phỏng quá nhanh hoặc máy tính không đáp ứng kịp.
    - Để khắc phục, hãy phóng to dạng sóng và đợi một chút để phần mềm vẽ lại chính xác.
- **Sử dụng chức năng Auto và Pause hợp lý**:
    - Chạy mô phỏng ở chế độ Auto để cập nhật dạng sóng liên tục.
    - Dừng mô phỏng khi cần quan sát kỹ một đoạn dạng sóng.

4. Phân tích và đọc dạng sóng

4.1. Quan sát các kênh tín hiệu

- **Kênh SS (Slave Select)**:
    - Khi chân SS chuyển từ mức cao xuống mức thấp, quá trình truyền dữ liệu bắt đầu.
    - Quan sát điểm này để xác định thời điểm bắt đầu giao tiếp.
- **Kênh SCK (Serial Clock)**:
    - Dạng sóng xung clock cần phải đều và chính xác.
    - Mặc dù trong mô phỏng có thể không hoàn toàn đều do giới hạn phần mềm, nhưng cần đảm bảo tần số và chu kỳ đúng.
- **Kênh MOSI và MISO**:
    - Quan sát dữ liệu truyền từ Master đến Slave (MOSI) và ngược lại (MISO).
    - Kiểm tra xem dữ liệu có đúng như mong đợi hay không.

4.2. Kiểm tra tốc độ xung clock

- **Xác định thang đo thời gian**:
    - Kiểm tra đơn vị thời gian trên trục ngang (ví dụ: 50 micro giây mỗi ô).
    - Đảm bảo cài đặt tốc độ xung clock trong mã chương trình phù hợp với thang đo.
- **Tính toán chu kỳ xung clock**:
    - Một chu kỳ xung clock bao gồm một nửa chu kỳ mức cao và một nửa chu kỳ mức thấp.
    - Kiểm tra xem tổng chu kỳ có khớp với cài đặt trong chương trình hay không.

4.3. Xác nhận dữ liệu truyền nhận

- **Ghi lại giá trị dữ liệu**:
    - Sử dụng một trình soạn thảo văn bản để ghi lại các mức logic của từng bit trên các kênh MOSI và MISO theo từng xung clock.
- **So sánh với dữ liệu mong đợi**:
    - Đối chiếu dữ liệu quan sát được với dữ liệu đã lập trình để xác định tính chính xác.
- **Lưu ý về thứ tự bit**:
    - Kiểm tra xem dữ liệu được truyền theo thứ tự MSB trước hay LSB trước và đảm bảo khớp với cấu hình trong mã chương trình.

5. Lưu ý khi sử dụng mô phỏng

5.1. Vấn đề về phóng to và con trỏ chuột

- **Sự lệch vị trí con trỏ chuột**:
    - Khi phóng to màn hình, con trỏ chuột có thể bị lệch so với vị trí thực tế.
    - Ví dụ: khi chỉ vào tọa độ (20, 20), con trỏ có thể hiển thị ở vị trí (50, 50).
- **Cách khắc phục**:
    - Chú ý lắng nghe hướng dẫn và quan sát kỹ trên màn hình.
    - Nếu cần, tạm dừng video để xem chi tiết các bước thực hiện.

5.2. Cách khắc phục méo dạng sóng trong mô phỏng

- **Nguyên nhân**:
    - Do tốc độ máy tính hoặc phần mềm không đáp ứng kịp.
    - Khi phóng to dạng sóng, phần mềm có thể không vẽ chính xác ngay lập tức.
- **Giải pháp**:
    - Sử dụng chế độ Auto và Pause một cách hợp lý.
    - Đợi một khoảng thời gian sau khi phóng to để phần mềm cập nhật dạng sóng chính xác.
    - Nếu dạng sóng vẫn méo, thử thu nhỏ lại và phóng to một lần nữa.

6. Kết luận

6.1. Khẳng định lại về hàm `Serial.print`

- **Không gây méo dạng sóng**:
    - Sau khi kiểm tra kỹ lưỡng, xác nhận rằng hàm `Serial.print` không ảnh hưởng đến dạng sóng trong mô phỏng.
    - Vấn đề méo dạng sóng trước đây là do cách sử dụng dao động ký và tốc độ mô phỏng.

6.2. Chia sẻ kinh nghiệm mô phỏng hiệu quả

- **Hiểu rõ công cụ mô phỏng**:
    - Dành thời gian để làm quen với các chức năng và cài đặt của Proteus.
    - Đảm bảo cấu hình đúng các tham số trong dao động ký và bộ phân tích logic.
- **Kiên nhẫn và thử nghiệm**:
    - Mô phỏng có thể không phản ánh hoàn hảo như thực tế do giới hạn phần mềm.
    - Thử nghiệm với các cài đặt khác nhau để đạt kết quả tốt nhất.
- **Chú ý đến chi tiết nhỏ**:
    - Các thao tác như phóng to, thu nhỏ, và thiết lập trigger có thể ảnh hưởng lớn đến việc quan sát dạng sóng.
    - Luôn kiểm tra kỹ từng bước để đảm bảo độ chính xác.

Lời kết:

Qua hướng dẫn này, hy vọng bạn đã nắm được cách chạy mô phỏng giao tiếp SPI trên Proteus một cách hiệu quả và chính xác. Việc hiểu rõ nguyên nhân gây ra các vấn đề và biết cách khắc phục sẽ giúp bạn tiết kiệm thời gian và nâng cao hiệu quả học tập và nghiên cứu.

Phần 1. Lý thuyết cơ bản (Basic Theory)

1. Giới thiệu về giao thức SPI

Tên gọi và khái niệm:

- **SPI** (Serial Peripheral Interface) là một giao thức truyền thông nối tiếp đồng bộ, được sử dụng để truyền dữ liệu giữa vi điều khiển (Master) và các thiết bị ngoại vi (Slave).
- SPI hoạt động theo mô hình **Master-Slave**, trong đó Master điều khiển toàn bộ quá trình truyền thông.

2. Các chân kết nối trong SPI

SPI sử dụng **bốn chân** để thực hiện giao tiếp:

2.1. Chân SCK (Serial Clock)

- **Chức năng:** Tạo tín hiệu xung clock để đồng bộ hóa việc truyền dữ liệu.
- **Đặc điểm:**
    - Được tạo ra bởi Master.
    - Slave chỉ nhận tín hiệu xung clock, không thể tự tạo.

2.2. Chân MOSI (Master Out Slave In)

- **Chức năng:** Truyền dữ liệu từ Master đến Slave.
- **Đặc điểm:**
    - Master xuất dữ liệu qua chân này.
    - Slave nhận dữ liệu vào qua chân này.

2.3. Chân MISO (Master In Slave Out)

- **Chức năng:** Truyền dữ liệu từ Slave đến Master.
- **Đặc điểm:**
    - Slave xuất dữ liệu qua chân này.
    - Master nhận dữ liệu vào qua chân này.

2.4. Chân SS (Slave Select)

- **Chức năng:** Lựa chọn Slave để giao tiếp.
- **Đặc điểm:**
    - Master điều khiển chân này.
    - Mặc định ở mức cao (không hoạt động).
    - Khi kéo xuống mức thấp, Slave tương ứng sẽ được kích hoạt.

3. Cơ chế hoạt động của SPI

3.1. Vai trò của Master và Slave

- **Master:**
    - Tạo xung clock.
    - Điều khiển chân SS để chọn Slave.
    - Truyền dữ liệu qua chân MOSI.
    - Nhận dữ liệu qua chân MISO.
- **Slave:**
    - Chỉ hoạt động khi chân SS được kéo xuống mức thấp.
    - Nhận xung clock từ Master.
    - Nhận dữ liệu qua chân MOSI.
    - Truyền dữ liệu qua chân MISO.

3.2. Nguyên lý truyền dữ liệu

- Khi Master muốn giao tiếp với một Slave, nó sẽ:
    - Kéo chân SS của Slave xuống mức thấp.
    - Bắt đầu tạo xung clock trên chân SCK.
    - Truyền dữ liệu qua chân MOSI.
    - Nhận dữ liệu từ Slave qua chân MISO.
- **Giao tiếp song công (full-duplex):** SPI cho phép truyền và nhận dữ liệu đồng thời.

4. Kết nối SPI với nhiều Slave

SPI hỗ trợ giao tiếp với nhiều Slave theo hai mô hình chính:

4.1. Mô hình sử dụng nhiều chân SS

- **Cách thức kết nối:**
    - Chân SCK, MOSI, MISO được nối chung giữa Master và tất cả các Slave.
    - Mỗi Slave có một chân SS riêng kết nối với Master.
- **Hoạt động:**
    - Master điều khiển từng chân SS để chọn Slave tương ứng.
    - Chỉ có Slave được chọn mới tham gia giao tiếp.
- **Ưu điểm:**
    - Đơn giản, dễ hiểu.
    - Tốc độ truyền dữ liệu nhanh.
- **Nhược điểm:**
    - Cần nhiều chân SS trên Master khi số lượng Slave tăng.

4.2. Mô hình Daisy Chain

- **Cách thức kết nối:**
    - Các Slave được kết nối nối tiếp, tạo thành một chuỗi.
    - Chân MOSI của Master nối với MOSI của Slave đầu tiên.

    - MISO của một Slave nối với MOSI của Slave tiếp theo.
    - Chân SS có thể chung cho tất cả các Slave.
- **Hoạt động:**
    - Dữ liệu truyền từ Master qua từng Slave trong chuỗi.
    - Mỗi Slave xử lý và chuyển tiếp dữ liệu.
- **Ưu điểm:**
    - Tiết kiệm chân SS trên Master.
    - Thích hợp cho hệ thống có nhiều Slave.
- **Nhược điểm:**
    - Độ trễ tăng khi số lượng Slave tăng.
    - Nếu một Slave hỏng, toàn bộ hệ thống bị ảnh hưởng.

4.3. So sánh và đánh giá

- **Mô hình nhiều chân SS:**
    - Phù hợp khi số lượng Slave ít.
    - Đảm bảo tốc độ và độ tin cậy.
- **Mô hình Daisy Chain:**
    - Phù hợp khi cần kết nối nhiều Slave mà hạn chế về chân I/O.
    - Cần cân nhắc về độ trễ và khả năng chịu lỗi.

5. So sánh tốc độ giữa các giao thức truyền thông

5.1. UART

- **Tốc độ truyền tối đa:** Khoảng 1 Mbps.
- **Đặc điểm:**
    - Giao tiếp 1 Master với 1 Slave.
    - Không hỗ trợ nhiều Slave.
    - Sử dụng hai dây (TX và RX).

5.2. I2C

- **Tốc độ truyền:**
    - Cơ bản: 100 Kbps.
    - Nhanh: 400 Kbps.
    - Tốc độ cao: Lên đến 3.4 Mbps.
- **Đặc điểm:**
    - Hỗ trợ nhiều Master và nhiều Slave.
    - Sử dụng hai dây (SDA và SCL).
    - Độ phức tạp cao hơn SPI.

5.3. SPI

- **Tốc độ truyền tối đa:** Lên đến 60 Mbps.

- **Đặc điểm:**
    - Giao tiếp 1 Master với nhiều Slave.
    - Sử dụng bốn dây.
    - Hỗ trợ truyền dữ liệu song công.

6. Các thông số cấu hình trong SPI

6.1. Thứ tự truyền bit (MSB/LSB)

- **MSB (Most Significant Bit) trước:**
    - Bit có trọng số cao nhất được truyền trước.
    - Thường là cấu hình mặc định.
- **LSB (Least Significant Bit) trước:**
    - Bit có trọng số thấp nhất được truyền trước.
    - Cần cấu hình nếu thiết bị yêu cầu.

6.2. Tốc độ truyền dữ liệu

- **Cách điều chỉnh:**
    - Sử dụng bộ chia tần số từ clock hệ thống.
    - Tốc độ có thể từ vài Kbps đến hàng chục Mbps.
- **Lưu ý:**
    - Tốc độ cao yêu cầu thiết kế mạch tốt để giảm nhiễu.
    - Cần đảm bảo cả Master và Slave hỗ trợ tốc độ mong muốn.

6.3. Chế độ hoạt động (Mode)

- **Dựa trên hai thông số CPOL và CPHA:**
    - Xác định cách dữ liệu được lấy mẫu và truyền đi.
    - Có bốn chế độ từ Mode 0 đến Mode 3.

7. Bốn chế độ hoạt động của SPI

7.1. CPOL và CPHA

- **CPOL (Clock Polarity):**
    - **CPOL = 0:** Xung clock ở mức thấp khi nhàn rỗi.
    - **CPOL = 1:** Xung clock ở mức cao khi nhàn rỗi.
- **CPHA (Clock Phase):**
    - **CPHA = 0:** Dữ liệu được lấy mẫu ở cạnh đầu tiên của xung clock.
    - **CPHA = 1:** Dữ liệu được lấy mẫu ở cạnh thứ hai của xung clock.

7.2. Phân tích từng chế độ

- **Mode 0 (CPOL = 0, CPHA = 0):**

    - Xung clock bắt đầu từ mức thấp.
    - Dữ liệu lấy mẫu ở cạnh lên đầu tiên.
- **Mode 1 (CPOL = 0, CPHA = 1):**
    - Xung clock bắt đầu từ mức thấp.
    - Dữ liệu lấy mẫu ở cạnh xuống thứ hai.
- **Mode 2 (CPOL = 1, CPHA = 0):**
    - Xung clock bắt đầu từ mức cao.
    - Dữ liệu lấy mẫu ở cạnh xuống đầu tiên.
- **Mode 3 (CPOL = 1, CPHA = 1):**
    - Xung clock bắt đầu từ mức cao.
    - Dữ liệu lấy mẫu ở cạnh lên thứ hai.
- **Lưu ý:**
    - Master và Slave phải được cấu hình cùng chế độ để giao tiếp đúng.
    - Chế độ mặc định thường là Mode 0.

8. Dạng sóng trong giao tiếp SPI

8.1. Phân tích dạng sóng

- **Chân SS:**
    - Khi được kéo xuống mức thấp, giao tiếp bắt đầu.
    - Sau khi kết thúc truyền dữ liệu, được kéo lên mức cao.
- **Chân SCK:**
    - Xung clock được tạo ra theo chế độ đã cấu hình.
    - Đồng bộ hóa việc truyền và nhận dữ liệu.
- **Chân MOSI và MISO:**
    - Dữ liệu được truyền và nhận đồng thời.
    - Dữ liệu ổn định tại thời điểm lấy mẫu xác định.
- **Khung truyền dữ liệu:**
    - Mỗi lần truyền thường gồm 8 bit (1 byte).
    - Tương ứng với 8 xung clock.

8.2. Ví dụ mô phỏng

- **Mô phỏng dạng sóng:**
    - Giúp hiểu rõ thời điểm dữ liệu được truyền và nhận.
    - Xác định đúng thời điểm lấy mẫu theo chế độ cấu hình.
- **Phân tích:**
    - Quan sát mức logic trên các chân.
    - Đối chiếu với lý thuyết để kiểm tra tính đúng đắn.

9. Chú ý khi lập trình với SPI

9.1. Vai trò của Master và Slave trong lập trình

- **Master:**
    - Chủ động gửi và nhận dữ liệu.
    - Không chờ đợi Slave sẵn sàng.
    - Phải cấu hình đúng chế độ và tốc độ.
- **Slave:**
    - Phải luôn trong trạng thái sẵn sàng.
    - Chỉ hoạt động khi chân SS được kích hoạt.
    - Phản hồi dữ liệu khi nhận được xung clock.

9.2. Lưu ý về chân SS

- **Điều khiển chân SS:**
    - Đảm bảo chân SS được kéo xuống trước khi bắt đầu giao tiếp.
    - Kéo lên mức cao sau khi kết thúc.
- **Quản lý nhiều Slave:**
    - Sử dụng đúng chân SS cho từng Slave.
    - Tránh xung đột khi có nhiều Slave trên bus.
- **Cấu hình đúng chế độ:**
    - Đảm bảo Master và Slave cùng chế độ CPOL và CPHA.
    - Kiểm tra thứ tự truyền bit (MSB/LSB).

10. Tài liệu tham khảo

- **Thư viện và hướng dẫn sử dụng SPI trên Arduino:**
    - Cung cấp hàm và ví dụ để lập trình SPI.
    - Hỗ trợ cấu hình các tham số một cách dễ dàng.
- **Wikipedia về giao thức SPI:**
    - Cung cấp thông tin chi tiết về lịch sử và nguyên lý hoạt động.
- **Bài viết chuyên sâu về SPI:**
    - Hướng dẫn cấu hình thanh ghi trực tiếp trên vi điều khiển.
    - Phân tích ưu nhược điểm và các ứng dụng thực tế.

Kết luận:

Giao thức SPI là một công cụ mạnh mẽ trong việc truyền thông giữa vi điều khiển và các thiết bị ngoại vi. Việc hiểu rõ về các chân kết nối, cơ chế hoạt động, và cách cấu hình chế độ sẽ giúp lập trình viên tối ưu hóa hiệu suất và độ tin cậy của hệ thống. Khi lập trình, cần chú ý đến vai trò của Master và Slave, cũng như quản lý chân SS một cách chính xác để đảm bảo giao tiếp diễn ra suôn sẻ.

Phần 2. Định nghĩa các macro cần cho code Master

1. Giới thiệu

Trong video này, thầy sẽ hướng dẫn các bạn cách viết thư viện giao tiếp SPI cho vi điều khiển, cụ thể là cho Master trên Arduino. Mục tiêu là tạo ra một thư viện SPI tự viết, sử dụng các chân I/O thông thường để thực hiện giao tiếp SPI giữa Master và Slave. Thư viện này có thể áp dụng cho Arduino hoặc bất kỳ vi điều khiển nào khác, miễn là chúng ta cấu hình đúng các chân và thanh ghi tương ứng.

2. Cấu hình chân cho Master

2.1. Xác định chức năng các chân

Trong giao tiếp SPI, các chân cần được cấu hình như sau:

- **SS (Slave Select):** Chân điều khiển chọn Slave, cần cấu hình là **Output**.
- **MOSI (Master Out Slave In):** Chân Master truyền dữ liệu, cấu hình là **Output**.
- **MISO (Master In Slave Out):** Chân Master nhận dữ liệu, cấu hình là **Input**.
- **SCK (Serial Clock):** Chân tạo xung clock, chỉ Master mới tạo được xung clock, cấu hình là **Output**.

Tóm lại, đối với Master:

- **SS, MOSI, SCK:** Cấu hình **Output**.
- **MISO:** Cấu hình **Input**.

2.2. Gán chân trên Arduino

Chúng ta sẽ gán các chân trên Arduino như sau:

- **SCK:** Chân số 4.
- **MOSI:** Chân số 5.
- **MISO:** Chân số 6.
- **SS:** Chân số 7.

Sử dụng định nghĩa:

```
#define SCK   4
#define MOSI  5
#define MISO  6
#define SS    7
```

3. Viết các macro để cấu hình và điều khiển chân

3.1. Lý do sử dụng macro và thao tác trực tiếp với thanh ghi

- Việc sử dụng các hàm như `pinMode`, `digitalWrite` của Arduino tuy tiện lợi nhưng tốc độ thực thi chậm do phải qua nhiều lớp hàm.
- Để tăng tốc độ và kiểm soát chặt chẽ hơn, chúng ta sẽ thao tác trực tiếp với các thanh ghi của vi điều khiển AVR.
- Sử dụng macro giúp mã nguồn gọn gàng, dễ hiểu và tối ưu hóa tốc độ.

3.2. Cấu hình hướng của chân (Input/Output)

3.2.1. Cấu hình chân Output

Ví dụ cấu hình chân **SS** (chân số 7) là Output:

- Chân số 7 trên Arduino UNO tương ứng với **Port D**, bit số 7.
- Sử dụng thanh ghi **DDRD** để cấu hình hướng của chân.
- Macro để cấu hình chân Output:

```
#define SS_OUTPUT       DDRD |= (1 << PD7)
```

Tương tự, chúng ta viết macro cho các chân khác:

```
#define MOSI_OUTPUT     DDRD |= (1 << PD5)
#define SCK_OUTPUT      DDRD |= (1 << PD4)
```

3.2.2. Cấu hình chân Input

Cấu hình chân **MISO** (chân số 6) là Input:

- Chân số 6 tương ứng với **Port D**, bit số 6.
- Để cấu hình là Input, chúng ta cần xóa bit tương ứng trong thanh ghi **DDRD**.
- Macro:

```
#define MISO_INPUT      DDRD &= ~(1 << PD6)
```

3.3. Ghi mức điện áp ra chân (HIGH/LOW)

3.3.1. Ghi mức HIGH ra chân

Ví dụ ghi mức HIGH ra chân **MOSI**:

- Sử dụng thanh ghi **PORTD** để ghi mức điện áp ra chân.

- Macro:

```
#define MOSI_HIGH      PORTD |= (1 << PD5)
```

3.3.2. Ghi mức LOW ra chân

Ghi mức LOW ra chân **MOSI**:

- Macro:

```
#define MOSI_LOW       PORTD &= ~(1 << PD5)
```

3.3.3. Gộp macro ghi HIGH/LOW bằng toán tử 3 ngôi

Để tiện lợi hơn, chúng ta có thể gộp hai macro trên thành một macro sử dụng toán tử 3 ngôi:

```
#define WRITE_MOSI(value) \
  ((value) ? (PORTD |= (1 << PD5)) : (PORTD &= ~(1 << PD5)))
```

- **Giải thích:**
  - Nếu `value` là **TRUE** (khác 0), thì ghi mức HIGH.
  - Nếu `value` là **FALSE** (0), thì ghi mức LOW.

3.3.4. Macro cho các chân khác

Tương tự, chúng ta viết macro cho các chân **SS** và **SCK**:

- **Ghi mức HIGH/LOW cho chân SS:**

```
#define WRITE_SS(value) \
  ((value) ? (PORTD |= (1 << PD7)) : (PORTD &= ~(1 << PD7)))
```

- **Ghi mức HIGH/LOW cho chân SCK:**

```
#define WRITE_SCK(value) \
  ((value) ? (PORTD |= (1 << PD4)) : (PORTD &= ~(1 << PD4)))
```

3.4. Đọc mức điện áp từ chân MISO

Để đọc dữ liệu từ chân **MISO**, chúng ta sẽ đọc từ thanh ghi **PIND**:

cpp
Sao chép mã
```
#define READ_MISO()  ((PIND & (1 << PD6)) ? 1 : 0)
```

- **Giải thích:**
  - Kiểm tra bit PD6 của **PIND**.
  - Nếu bit PD6 là 1, macro trả về 1.
  - Nếu bit PD6 là 0, macro trả về 0.

4. Hoàn thiện mã nguồn cho Master

4.1. Đặt tất cả các macro vào mã nguồn

Tổng hợp lại, chúng ta có các macro sau:

```
// Định nghĩa chân
#define SCK   4
#define MOSI  5
#define MISO  6
#define SS    7

// Cấu hình hướng chân
#define SS_OUTPUT      DDRD |= (1 << PD7)
#define MOSI_OUTPUT    DDRD |= (1 << PD5)
#define SCK_OUTPUT     DDRD |= (1 << PD4)
#define MISO_INPUT     DDRD &= ~(1 << PD6)

// Ghi mức điện áp ra chân
#define WRITE_MOSI(value) \
  ((value) ? (PORTD |= (1 << PD5)) : (PORTD &= ~(1 << PD5)))

#define WRITE_SS(value) \
  ((value) ? (PORTD |= (1 << PD7)) : (PORTD &= ~(1 << PD7)))

#define WRITE_SCK(value) \
  ((value) ? (PORTD |= (1 << PD4)) : (PORTD &= ~(1 << PD4)))

// Đọc mức điện áp từ chân MISO
#define READ_MISO()  ((PIND & (1 << PD6)) ? 1 : 0)
```

4.2. Sử dụng các macro trong chương trình

Trong hàm `setup()`, chúng ta sẽ cấu hình các chân:

```
void setup() {
  // Cấu hình hướng chân
  SS_OUTPUT;
  MOSI_OUTPUT;
  SCK_OUTPUT;
  MISO_INPUT;

  // Khởi tạo trạng thái ban đầu cho các chân
  WRITE_SS(HIGH);     // Không chọn Slave nào
  WRITE_SCK(LOW);     // Xung clock ở mức LOW
  WRITE_MOSI(LOW);    // Dữ liệu ban đầu là LOW
}
```

Trong hàm `loop()`, chúng ta sẽ sử dụng các macro để thực hiện giao tiếp SPI.

5. Giải thích chi tiết cách hoạt động

5.1. Cách cấu hình chân bằng thanh ghi

- **Thanh ghi DDRx (Data Direction Register):**
  - Quyết định hướng của chân: Input (0) hoặc Output (1).
  - Ví dụ: `DDRD |= (1 << PD5);` đặt bit PD5 của DDRD lên 1, cấu hình chân PD5 là Output.
- **Thanh ghi PORTx:**
  - Khi chân được cấu hình là Output:
    - Ghi mức HIGH hoặc LOW ra chân.
  - Khi chân được cấu hình là Input:
    - Kích hoạt hoặc tắt **pull-up resistor**.
  - Ví dụ: `PORTD |= (1 << PD5);` đặt bit PD5 của PORTD lên 1, xuất mức HIGH ra chân PD5.
- **Thanh ghi PINx:**
  - Được sử dụng để đọc mức điện áp hiện tại của chân.
  - Ví dụ: `(PIND & (1 << PD6));` đọc giá trị của bit PD6 từ thanh ghi PIND.

5.2. Sử dụng toán tử 3 ngôi trong macro

- Toán tử 3 ngôi có cú pháp:

```
(condition) ? expression_if_true : expression_if_false;
```

- Trong macro `WRITE_MOSI(value)`, nếu `value` là TRUE (khác 0), macro sẽ thực hiện biểu thức trước dấu hai chấm, ngược lại thực hiện biểu thức sau dấu hai chấm.

5.3. Lợi ích của việc thao tác trực tiếp với thanh ghi

- **Tăng tốc độ thực thi:**
  - Tránh được overhead từ các hàm của Arduino.
  - Thao tác trực tiếp với thanh ghi giúp thực hiện lệnh nhanh hơn.
- **Kiểm soát chặt chẽ:**
  - Hiểu rõ hơn về cách vi điều khiển hoạt động.
  - Dễ dàng tối ưu hóa cho các ứng dụng yêu cầu tốc độ cao.

6. Kết luận

- Chúng ta đã hoàn thành việc viết các macro để cấu hình và điều khiển các chân phục vụ cho giao tiếp SPI ở phía Master.
- Bằng cách thao tác trực tiếp với thanh ghi, chúng ta tối ưu hóa được tốc độ và hiệu suất của chương trình.
- Thư viện SPI tự viết này có thể áp dụng cho Arduino và các vi điều khiển khác, chỉ cần điều chỉnh lại các thanh ghi và chân tương ứng.
- Trong các phần tiếp theo, chúng ta sẽ sử dụng các macro này để viết hàm gửi và nhận dữ liệu qua SPI, cũng như viết chương trình cho phía Slave.

Chúc các bạn thành công trong việc tự viết thư viện SPI và hiểu sâu hơn về vi điều khiển!

Phần 3. Viết thư viện truyền nhận 1 byte cho Master

1. Giới thiệu về SPI và mục tiêu của bài viết

SPI (Serial Peripheral Interface) là một giao thức truyền thông nối tiếp đồng bộ tốc độ cao, được sử dụng rộng rãi trong các hệ thống nhúng để giao tiếp giữa vi điều khiển và các thiết bị ngoại vi như cảm biến, bộ nhớ, hoặc các module giao tiếp khác. Mục tiêu của bài viết này là hướng dẫn chi tiết cách viết thư viện SPI cho Arduino, tập trung vào vai trò **Master**, bao gồm:

- Cấu hình ban đầu cho SPI.
- Viết các hàm cần thiết cho việc giao tiếp SPI.
- Giải thích chi tiết quá trình truyền và nhận dữ liệu trong SPI.
- Cách xử lý dữ liệu bit-level để đảm bảo truyền thông chính xác.

2. Cấu hình ban đầu cho SPI

Trong giao tiếp SPI, việc cấu hình ban đầu là rất quan trọng để đảm bảo các chân và tín hiệu hoạt động đúng theo yêu cầu. Các bước cấu hình ban đầu bao gồm:

- Xác định các chân kết nối: **MOSI (Master Out Slave In)**, **MISO (Master In Slave Out)**, **SCK (Serial Clock)**, **SS (Slave Select)**.
- Thiết lập trạng thái mặc định cho các chân: xác định chân nào là **INPUT**, chân nào là **OUTPUT**.
- Cấu hình trạng thái **idle** cho các tín hiệu: xác định mức logic mặc định khi không có giao tiếp.

3. Các macro và định nghĩa cần thiết

Để đơn giản hóa việc lập trình và tăng tính linh hoạt, chúng ta sử dụng các macro và định nghĩa:

- **Macro cho việc thiết lập chân làm INPUT hoặc OUTPUT**:
  - `pinMode(pin, mode)`: thiết lập chế độ cho chân `pin`.
- **Macro cho việc ghi giá trị HIGH hoặc LOW ra chân**:
  - `digitalWrite(pin, value)`: ghi giá trị `value` (HIGH/LOW) ra chân `pin`.
- **Định nghĩa các chân SPI**:
  - `MOSI_PIN`, `MISO_PIN`, `SCK_PIN`, `SS_PIN`: định nghĩa số chân tương ứng.
- **Định nghĩa thời gian trễ cho xung clock**:
  - `T_HALF`: thời gian nửa chu kỳ của xung clock.
  - `T_FULL`: thời gian một chu kỳ đầy đủ của xung clock.

4. Viết hàm cài đặt ban đầu cho SPI (SPI_Setup)

Hàm `SPI_Setup()` chịu trách nhiệm cấu hình ban đầu cho SPI:

- Thiết lập các chân `MOSI`, `SCK`, `SS` làm **OUTPUT**.
- Thiết lập chân `MISO` làm **INPUT**.
- Đặt trạng thái mặc định cho các chân:
    - `digitalWrite(SS_PIN, HIGH)`: chân `SS` ở mức cao.
    - `digitalWrite(SCK_PIN, LOW)`: chân `SCK` ở mức thấp.

Mã nguồn mẫu:

```
void SPI_Setup() {

    // Thiết lập chế độ cho các chân

    pinMode(MOSI_PIN, OUTPUT);

    pinMode(MISO_PIN, INPUT);

    pinMode(SCK_PIN, OUTPUT);

    pinMode(SS_PIN, OUTPUT);


    // Đặt trạng thái mặc định

    digitalWrite(SS_PIN, HIGH);

    digitalWrite(SCK_PIN, LOW);

}
```

5. Viết hàm bắt đầu truyền dữ liệu (SPI_Begin)

Hàm `SPI_Begin()` dùng để bắt đầu quá trình truyền dữ liệu:

- Kéo chân `SS` xuống mức thấp để kích hoạt giao tiếp với Slave.
- Thao tác này báo hiệu cho Slave biết rằng Master sẵn sàng truyền dữ liệu.

Mã nguồn mẫu:

```
void SPI_Begin() {
    digitalWrite(SS_PIN, LOW);
}
```

6. Viết hàm kết thúc truyền dữ liệu (SPI_End)

Hàm `SPI_End()` dùng để kết thúc quá trình truyền dữ liệu:

- Đưa chân `SCK` về mức thấp.
- Kéo chân `SS` lên mức cao để ngừng giao tiếp với Slave.
- Đảm bảo rằng các tín hiệu trở về trạng thái idle.

Mã nguồn mẫu:

```
void SPI_End() {
    digitalWrite(SCK_PIN, LOW);
    digitalWrite(SS_PIN, HIGH);
}
```

7. Viết hàm truyền và nhận dữ liệu (SPI_Transfer)

Hàm `SPI_Transfer(byte dataOut)` thực hiện cả việc truyền và nhận một byte dữ liệu qua SPI.

7.1. Giải thích về quá trình truyền và nhận dữ liệu trong SPI

- **Truyền dữ liệu**: Master gửi dữ liệu tới Slave thông qua chân `MOSI`.
- **Nhận dữ liệu**: Master nhận dữ liệu từ Slave thông qua chân `MISO`.
- **Xung clock (`SCK`)**: Master tạo xung clock để đồng bộ hóa quá trình truyền nhận.
- **Quá trình full-duplex**: SPI cho phép truyền và nhận dữ liệu đồng thời trong cùng một chu kỳ xung clock.

7.2. Cách lấy từng bit dữ liệu để truyền

- Dữ liệu cần truyền (`dataOut`) là một byte (8 bit).

- Sử dụng phép toán bitwise để lấy từng bit từ bit cao đến bit thấp:
    - Sử dụng biến `mask` bắt đầu từ `0x80` (10000000b).
    - Trong mỗi vòng lặp, kiểm tra bit hiện tại:
        - Nếu `dataOut & mask` khác 0, bit hiện tại là 1.
        - Nếu không, bit hiện tại là 0.
    - Ghi giá trị bit ra chân `MOSI`.
    - Dịch `mask` sang phải một bit cho vòng lặp tiếp theo.

Mã nguồn mẫu cho phần truyền bit:

```
for (byte mask = 0x80; mask; mask >>= 1) {
    if (dataOut & mask)
        digitalWrite(MOSI_PIN, HIGH);
    else
        digitalWrite(MOSI_PIN, LOW);
    // Tạo xung clock và đọc dữ liệu ở đây
}
```

7.3. Cách đọc từng bit dữ liệu nhận được

- Trong cùng vòng lặp truyền bit, sau khi thiết lập giá trị trên `MOSI`, thực hiện:
    - Tạo xung clock bằng cách kéo `SCK` lên cao.
    - Đọc giá trị từ chân `MISO`.
    - Dịch giá trị đọc được vào biến `dataIn`.

Mã nguồn mẫu cho phần nhận bit:

```
   // Sau khi ghi MOSI
    digitalWrite(SCK_PIN, HIGH);
    if (digitalRead(MISO_PIN))
        dataIn |= mask;
    // Đợi nửa chu kỳ, sau đó kéo SCK xuống thấp
```

```
digitalWrite(SCK_PIN, LOW);
```

7.4. Cách tạo xung clock và đồng bộ hóa

- **Xung clock (`SCK`)**:
  - Mỗi bit dữ liệu được truyền trong một chu kỳ xung clock.
  - Xung clock bắt đầu ở mức thấp, lên mức cao, sau đó trở về mức thấp.
- **Đồng bộ hóa**:
  - Dữ liệu được ghi ra `MOSI` trước khi xung clock lên cao.
  - Dữ liệu được đọc từ `MISO` khi xung clock ở mức cao.
- **Thời gian trễ**:
  - Sử dụng hàm `delayMicroseconds(T_HALF)` để tạo thời gian trễ nửa chu kỳ giữa các thay đổi mức logic.

Mã nguồn mẫu cho xung clock và đồng bộ hóa:

```
// Ghi MOSI
// ...
delayMicroseconds(T_HALF);
digitalWrite(SCK_PIN, HIGH);
// Đọc MISO
// ...
delayMicroseconds(T_HALF);
digitalWrite(SCK_PIN, LOW);
```

8. Kết luận

Trong bài viết này, chúng ta đã đi qua từng bước chi tiết để xây dựng một thư viện SPI cho Arduino ở vai trò Master:

- Hiểu rõ cấu trúc và hoạt động của giao thức SPI.
- Biết cách cấu hình các chân và tín hiệu ban đầu.
- Sử dụng các macro và định nghĩa để tăng tính linh hoạt trong lập trình.

- Viết các hàm cơ bản cho việc bắt đầu và kết thúc giao tiếp SPI.
- Hiểu và triển khai quá trình truyền và nhận dữ liệu ở mức bit-level, đảm bảo đồng bộ hóa chính xác với xung clock.

Với kiến thức này, bạn có thể mở rộng và tùy chỉnh thư viện SPI cho các ứng dụng cụ thể, cũng như viết thư viện tương tự cho vai trò Slave, đảm bảo sự giao tiếp hiệu quả giữa các thiết bị trong hệ thống nhúng.

Phần 4. Viết thư viện truyền nhận 1 byte cho Slave

1. Giới thiệu và mục tiêu

Trong video trước, chúng ta đã viết thư viện SPI cho vai trò **Master**. Trong video này, chúng ta sẽ tiếp tục với việc viết thư viện SPI cho vai trò **Slave**. Mục tiêu chính bao gồm:

- Thiết lập các chân giao tiếp cho SPI Slave.
- Viết các hàm cần thiết để SPI Slave có thể truyền và nhận dữ liệu.
- Hiểu rõ cách thức hoạt động của SPI ở phía Slave, đặc biệt là cách đồng bộ với Master.
- Đảm bảo quá trình truyền nhận dữ liệu diễn ra chính xác và hiệu quả.

2. Thiết lập các chân cho SPI Slave

2.1. Chọn chân kết nối

Chúng ta sẽ sử dụng các chân sau cho SPI Slave (giống như bên Master):

- **SCK_PIN** (Serial Clock): Chân số 4
- **MOSI_PIN** (Master Out Slave In): Chân số 5
- **MISO_PIN** (Master In Slave Out): Chân số 6
- **SS_PIN** (Slave Select): Chân số 7

Lưu ý: Các chân này có thể thay đổi tùy theo thiết kế của bạn, miễn là đồng bộ với Master.

2.2. Định nghĩa các chân trong mã nguồn

Chúng ta định nghĩa các chân bằng cách sử dụng `#define`:

```
#define SCK_PIN   4
#define MOSI_PIN  5
#define MISO_PIN  6
#define SS_PIN    7
```

3. Cấu hình hướng và trạng thái cho các chân

3.1. Cấu hình hướng vào/ra cho các chân

- **SCK_PIN**: INPUT (Slave nhận xung clock từ Master)
- **MOSI_PIN**: INPUT (Slave nhận dữ liệu từ Master)
- **MISO_PIN**: OUTPUT (Slave gửi dữ liệu tới Master)

- **SS_PIN**: INPUT (Slave nhận tín hiệu chọn từ Master)

3.2. Viết hàm cài đặt ban đầu cho SPI Slave

Hàm `SPI_Slave_Init()` dùng để cấu hình hướng cho các chân:

```
void SPI_Slave_Init() {
    // Thiết lập hướng cho các chân
    pinMode(SCK_PIN, INPUT);
    pinMode(MOSI_PIN, INPUT);
    pinMode(MISO_PIN, OUTPUT);
    pinMode(SS_PIN, INPUT);
}
```

4. Viết các macro đọc/ghi cho các chân

4.1. Macro đọc chân SCK

```
#define READ_SCK()  digitalRead(SCK_PIN)
```

4.2. Macro đọc chân MOSI

```
#define READ_MOSI() digitalRead(MOSI_PIN)
```

4.3. Macro đọc chân SS

```
#define READ_SS()   digitalRead(SS_PIN)
```

4.4. Macro ghi chân MISO

```
#define WRITE_MISO(value) digitalWrite(MISO_PIN, value)
```

5. Viết hàm truyền và nhận dữ liệu (SPI_Slave_Transfer)

Hàm `SPI_Slave_Transfer(byte dataOut)` thực hiện việc truyền và nhận một byte dữ liệu giữa Slave và Master.

5.1. Khởi tạo biến và chuẩn bị dữ liệu

- Khai báo biến lưu dữ liệu nhận được:

```
byte dataIn = 0;
```

- Biến `mask` để xử lý từng bit:

```
byte mask = 0x80; // Bắt đầu từ bit cao nhất
```

5.2. Chờ tín hiệu SS từ Master

Slave cần chờ đến khi chân SS được kéo xuống mức thấp:

```
while (READ_SS() == HIGH) {
    // Chờ Master chọn Slave
}
```

5.3. Vòng lặp xử lý 8 bit dữ liệu

5.3.1. Bắt đầu vòng lặp

Chúng ta sử dụng vòng lặp `for` để xử lý 8 bit:

```
for (int i = 0; i < 8; i++) {
    // Nội dung xử lý từng bit
}
```

5.3.2. Ghi dữ liệu ra MISO

- Kiểm tra bit hiện tại của `dataOut` và ghi ra MISO:

```
if (dataOut & mask) {
    WRITE_MISO(HIGH);
} else {
    WRITE_MISO(LOW);
}
```

5.3.3. Chờ SCK lên mức cao

- Slave chờ SCK từ Master để đồng bộ:

```
while (READ_SCK() == LOW) {
    // Chờ SCK lên cao
}
```

5.3.4. Đọc dữ liệu từ MOSI

- Khi SCK ở mức cao, đọc dữ liệu từ MOSI:

```
if (READ_MOSI() == HIGH) {
    dataIn |= mask;
}
```

5.3.5. Chờ SCK xuống mức thấp

- Đảm bảo SCK đã xuống thấp trước khi chuyển sang bit tiếp theo:

```
while (READ_SCK() == HIGH) {
    // Chờ SCK xuống thấp
}
```

5.3.6. Dịch `mask` sang phải

- Chuẩn bị cho bit tiếp theo:

```
mask >>= 1;
```

5.4. Kết thúc truyền dữ liệu

- Sau khi hoàn thành 8 bit, trả về dữ liệu nhận được:

```
return dataIn;
```

6. Giải thích chi tiết quá trình truyền và nhận dữ liệu

6.1. Chờ tín hiệu SS

- Slave không chủ động bắt đầu giao tiếp, phải chờ Master kéo SS xuống thấp.
- Điều này đảm bảo rằng Slave chỉ hoạt động khi được Master yêu cầu.

6.2. Truyền dữ liệu từ Slave tới Master

- Slave ghi dữ liệu ra MISO trước khi SCK lên cao.
- Dữ liệu phải ổn định trước khi Master đọc.

6.3. Nhận dữ liệu từ Master

- Slave đọc dữ liệu từ MOSI khi SCK ở mức cao.
- Điều này đảm bảo dữ liệu đã ổn định do Master đã ghi trước đó.

6.4. Đồng bộ hóa với SCK

- Slave không tạo xung clock, nên phải đồng bộ hoàn toàn với SCK từ Master.
- Việc chờ SCK lên cao và xuống thấp đảm bảo đồng bộ hóa.

7. So sánh hoạt động giữa Master và Slave

Hoạt động	Master	Slave
Khởi tạo giao tiếp	Kéo SS xuống thấp	Chờ SS xuống thấp
Tạo xung clock	Chủ động tạo SCK	Chờ SCK từ Master
Truyền dữ liệu	Ghi dữ liệu ra MOSI	Ghi dữ liệu ra MISO
Nhận dữ liệu	Đọc dữ liệu từ MISO	Đọc dữ liệu từ MOSI
Kết thúc giao tiếp	Kéo SS lên cao	Giao tiếp kết thúc khi SS lên cao

8. Lưu ý quan trọng trong việc lập trình SPI Slave

8.1. Đảm bảo đồng bộ thời gian

- Do Slave không kiểm soát SCK, cần chờ chính xác cạnh lên và cạnh xuống của SCK.

8.2. Xử lý dữ liệu bit-level

- Sử dụng biến `mask` để xử lý từng bit từ cao đến thấp.
- Đảm bảo dữ liệu được truyền và nhận chính xác theo thứ tự bit.

8.3. Trạng thái của các chân khi không hoạt động

- Khi không giao tiếp, MISO nên ở trạng thái LOW hoặc HIGH tùy theo yêu cầu.
- Tránh trạng thái lơ lửng (floating) gây nhiễu.

9. Kết luận

Trong video này, chúng ta đã hoàn thiện việc viết thư viện SPI cho vai trò Slave với các bước:

- Thiết lập các chân và hướng vào/ra phù hợp.
- Viết các macro để đơn giản hóa việc đọc/ghi các chân.
- Viết hàm `SPI_Slave_Transfer` để xử lý việc truyền và nhận dữ liệu.
- Hiểu rõ cách Slave đồng bộ với Master thông qua SCK và SS.
- Đảm bảo quá trình truyền nhận dữ liệu diễn ra chính xác và hiệu quả.

Với thư viện này, chúng ta có thể thực hiện giao tiếp SPI giữa Master và Slave một cách hiệu quả, đáp ứng được yêu cầu của các ứng dụng thực tế trong hệ thống nhúng.

10. Hướng phát triển tiếp theo

- **Mô phỏng và kiểm tra**: Trong video tiếp theo, chúng ta sẽ tiến hành mô phỏng để kiểm tra quá trình truyền nhận dữ liệu giữa Master và Slave.
- **Xử lý lỗi**: Bổ sung cơ chế xử lý khi gặp lỗi trong quá trình truyền nhận.
- **Mở rộng chức năng**: Thêm các chức năng nâng cao như hỗ trợ nhiều Slave, điều chỉnh tốc độ giao tiếp,...

Phần 5. Chạy mô phỏng kết quả Giao tiếp Master-Slave

1. Giới thiệu

Trong video này, chúng ta sẽ tiến hành mô phỏng và kiểm tra quá trình truyền nhận dữ liệu giữa hai vi điều khiển Arduino đóng vai trò **Master** và **Slave** trong giao thức SPI. Trước đó, chúng ta đã viết thư viện SPI cho cả Master và Slave. Mục tiêu chính của bài này là:

- Chạy mô phỏng để kiểm tra tính đúng đắn của thư viện SPI đã viết.
- Quan sát và phân tích dạng sóng trên các đường truyền SPI.
- Xử lý các lỗi phát sinh trong quá trình mô phỏng.
- Hiểu rõ hơn về quá trình truyền nhận dữ liệu giữa Master và Slave.

2. Thiết lập mô phỏng

2.1. Cấu hình chân kết nối

Cả hai vi điều khiển (Master và Slave) sẽ sử dụng chung các chân sau cho giao tiếp SPI:

- **SCK_PIN** (Serial Clock): Chân số 4
- **MOSI_PIN** (Master Out Slave In): Chân số 5
- **MISO_PIN** (Master In Slave Out): Chân số 6
- **SS_PIN** (Slave Select): Chân số 7

2.2. Khai báo và khởi tạo

Trong mã nguồn của cả Master và Slave, chúng ta cần:

- Khai báo các chân SPI tương ứng.
- Gọi hàm khởi tạo SPI trong hàm `setup()`.
- Tạo các biến lưu trữ dữ liệu truyền và nhận.

Ví dụ cho Master:

```
void setup() {
    SPI_Master_Init();
}

void loop() {
    byte dataReceived;
    SPI_Master_Begin();
    dataReceived = SPI_Master_Transfer('a');
    SPI_Master_End();
```

```
    delay(1000);
}
```

Ví dụ cho Slave:

```
void setup() {
    SPI_Slave_Init();
}

void loop() {
    byte dataReceived;
    dataReceived = SPI_Slave_Transfer('A');
}
```

3. Quá trình truyền nhận dữ liệu

3.1. Truyền dữ liệu từ Master đến Slave

- Master gửi ký tự `'a'` (mã ASCII 97) đến Slave.
- Slave nhận dữ liệu từ đường MOSI.

3.2. Truyền dữ liệu từ Slave đến Master

- Slave gửi ký tự `'A'` (mã ASCII 65) đến Master.
- Master nhận dữ liệu từ đường MISO.

3.3. Đồng bộ hóa quá trình truyền nhận

- Master tạo xung clock và điều khiển chân SS.
- Slave chờ tín hiệu từ Master để bắt đầu truyền nhận.

4. Hiển thị dữ liệu truyền nhận

4.1. Sử dụng Serial Monitor

Để kiểm tra dữ liệu nhận được, chúng ta sử dụng Serial Monitor:

- Khởi tạo Serial trong hàm `setup()` với tốc độ baud phù hợp.
- Sử dụng `Serial.print()` hoặc `Serial.println()` để hiển thị dữ liệu.

Ví dụ cho Master:

```
void setup() {
    Serial.begin(9600);
    SPI_Master_Init();
}

void loop() {
    byte dataReceived;
    SPI_Master_Begin();
    dataReceived = SPI_Master_Transfer('a');
    SPI_Master_End();
    Serial.println((char)dataReceived);
    delay(1000);
}
```

Ví dụ cho Slave:

```
void setup() {
    Serial.begin(9600);
    SPI_Slave_Init();
}

void loop() {
    byte dataReceived;
    dataReceived = SPI_Slave_Transfer('A');
    Serial.println((char)dataReceived);
}
```

4.2. Kết quả mong đợi

- **Master** sẽ nhận được ký tự `'A'` từ Slave.
- **Slave** sẽ nhận được ký tự `'a'` từ Master.

5. Phân tích dạng sóng SPI

5.1. Sử dụng công cụ mô phỏng

- Sử dụng phần mềm mô phỏng Proteus để quan sát dạng sóng.
- Kết nối các chân SPI với các kênh của oscilloscope trong Proteus.

5.2. Quan sát các tín hiệu

- **SS (Slave Select)**: Thấp khi bắt đầu truyền, cao khi kết thúc.
- **SCK (Serial Clock)**: Xung clock do Master tạo ra.
- **MOSI (Master Out Slave In)**: Dữ liệu từ Master đến Slave.
- **MISO (Master In Slave Out)**: Dữ liệu từ Slave đến Master.

5.3. Phân tích dữ liệu trên dạng sóng

- Kiểm tra xem các bit dữ liệu có truyền đúng theo thứ tự bit không.
- Đảm bảo rằng dữ liệu trên MOSI và MISO khớp với dữ liệu mong đợi.

6. Xử lý lỗi và vấn đề thường gặp

6.1. Dữ liệu nhận được không đúng

- Kiểm tra kết nối giữa các chân SPI.
- Đảm bảo rằng Master và Slave sử dụng cùng một cấu hình SPI (chế độ, tốc độ).

6.2. Dạng sóng không ổn định

- Có thể do tốc độ truyền quá cao, giảm tốc độ truyền để ổn định tín hiệu.
- Đảm bảo rằng thời gian trễ trong mã nguồn phù hợp.

6.3. Xung clock bị méo hoặc không đều

- Xem xét lại hàm tạo xung clock trong mã nguồn của Master.
- Đảm bảo rằng không có hàm hoặc tác vụ nào gây trễ không mong muốn trong quá trình tạo xung.

7. Điều chỉnh và tối ưu hóa mã nguồn

7.1. Tránh sử dụng hàm Serial trong quá trình truyền nhận

- Hàm `Serial.print()` có thể gây trễ và làm sai lệch quá trình truyền nhận.
- Chỉ nên sử dụng `Serial` để hiển thị kết quả sau khi hoàn tất truyền nhận.

7.2. Tối ưu hóa hàm truyền nhận

- Giảm thiểu các tác vụ không cần thiết trong vòng lặp truyền nhận.

- Sử dụng các biến và hàm hiệu quả để tăng tốc độ truyền.

7.3. Điều chỉnh tốc độ truyền

- Thử nghiệm với các tốc độ truyền khác nhau để tìm ra tốc độ tối ưu.
- Lưu ý rằng tốc độ truyền quá cao có thể gây lỗi trong mô phỏng.

8. Mở rộng ứng dụng

8.1. Truyền nhiều byte dữ liệu

- Có thể truyền một mảng dữ liệu bằng cách lặp lại hàm truyền nhận.
- Đảm bảo rằng Master và Slave đồng bộ trong quá trình truyền nhiều byte.

Ví dụ:

```
// Master
byte dataToSend[3] = {'a', 'b', 'c'};
for (int i = 0; i < 3; i++) {
    dataReceived = SPI_Master_Transfer(dataToSend[i]);
}

// Slave
byte dataToSend[3] = {'1', '2', '3'};
for (int i = 0; i < 3; i++) {
    dataReceived = SPI_Slave_Transfer(dataToSend[i]);
}
```

8.2. Giao tiếp với các thiết bị SPI thực tế

- Sử dụng thư viện đã viết để giao tiếp với các module SPI như cảm biến, màn hình, thẻ nhớ,...
- Cần xem xét các yêu cầu cụ thể của thiết bị (chế độ SPI, tốc độ, định dạng dữ liệu).

9. Kết luận

Trong bài viết này, chúng ta đã:

- Thiết lập mô phỏng giao tiếp SPI giữa Master và Slave.
- Viết mã nguồn cho cả hai bên để truyền nhận dữ liệu.
- Sử dụng công cụ mô phỏng để quan sát và phân tích dạng sóng.

- Xử lý các lỗi và tối ưu hóa mã nguồn để đảm bảo truyền nhận chính xác.
- Mở rộng ứng dụng của giao tiếp SPI trong các dự án thực tế.

10. Hướng phát triển tiếp theo

- **Giao tiếp với nhiều Slave**: Triển khai giao tiếp SPI với nhiều thiết bị Slave.
- **Chuyển đổi chế độ SPI**: Thử nghiệm với các chế độ SPI khác nhau (CPOL, CPHA).
- **Tối ưu hóa hiệu năng**: Sử dụng các kỹ thuật lập trình nâng cao để tăng tốc độ truyền nhận.
- **Ứng dụng thực tế**: Áp dụng giao tiếp SPI để kết nối với các cảm biến, module RF, màn hình,...

11. Lời cảm ơn

Cảm ơn các bạn đã theo dõi bài viết. Hy vọng thông qua bài này, các bạn đã hiểu rõ hơn về giao tiếp SPI và cách triển khai nó trong các dự án của mình. Nếu có bất kỳ thắc mắc hoặc góp ý nào, xin vui lòng để lại bình luận. Chúc các bạn thành công!