Chapter 3

Serial Peripheral Interconnect (SPI)

Chapter Introduction

In this chapter we will cover the following topics:

- Asynchronous vs synchronous interface protocols
- Master-slave architecture
- The SPI exchange
- The SPI transaction
- Multi-word transactions
- Multi-slave topologies
- SPI transaction decoding

In cyberphysical systems, an embedded processor must interact with I/O devices, human interface devices, clocks, and networks while also minimizing system cost and complexity, in terms of the number of physical resources and potential points of failure. For this, simplified but relatively low performance interface protocols are used. Potentially one of the most widely used is the Serial Peripheral Interface, or SPI, which uses four wires for bidirectional communication. SPI is **synchronous**, meaning that the channel includes a clock signal. This allows the receiving end of the channel to know exactly when to read the channel's data. SPI also uses a **master-slave architecture**, in which each entity connected to the channel must adopt pre-defined roles, with only one entity able to initiate communication requests and one or more that are obligated to respond to communication requests.

Chapter Objectives

In this chapter, students will learn:

- Serialization and deserialization
- How to interpret SPI exchanges and transactions
- How programmed I/O transactions are performed over an exchange-based protocol

1.1 Interface Channels and Protocols

A **communication channel** is a physical set of wires that convey data between two or more **entities**, where the entities might be modules that share the same semiconductor chip, separate chips that share the same printed circuit board, separate chips connected via flexible wire bundles or cables, or nodes on a network.

A **communication protocol** is a set of rules that govern the operation of the channel. The protocol determines the role of each entity using the channel and controls the timing of data transmitted over the channel and can implement features such as acknowledgements, error control, and arbitration.

Various communication channels and protocols are designed with specific attributes in mind, such as:

- **distance:** e.g. within a single chip (< 10 mm), such as the AXI and Avalon protocolsm, chip-to-chip (< 1 m), such as PCI express, QPI, RapidIO, and local area network (< 100 m), such as Ethernet and Infiniband, and
- **performance and cost:** e.g. PCI express (Gen4 achieves ~4 GB/s per channel), USB 3.1 (~10 Gb/s per channel), SPI (~100 MB/s per channel)

Faster channels are more costly in terms of hardware infrastructure and power. For example, PCI express channels require precise impedance matching for their channel wires on printed circuit boards, while SPI has no such requirement.

1.2 Synchronous vs Asynchronous Protocols

Two important challenges of designing a communication channel is developing a mechanism for the receiver to sample the transmitted data at the same rate at which the transmitter is sending it, and to ensure the transmitter does not change the bit value being transmitted at the same time the receiver is sampling the bit value. In other others, the receiver must set its **sample times**, the moment in time defined by the rising or falling edge of its clock signal where it reads the bit being transmitted, to occur outside of the **aperture time**, which is an interval surrounding the sampling clock edge where the transmitted data should not change to minimize the chance of **metastability**, a type of read error. The aperture time is depicted in Fig. X.

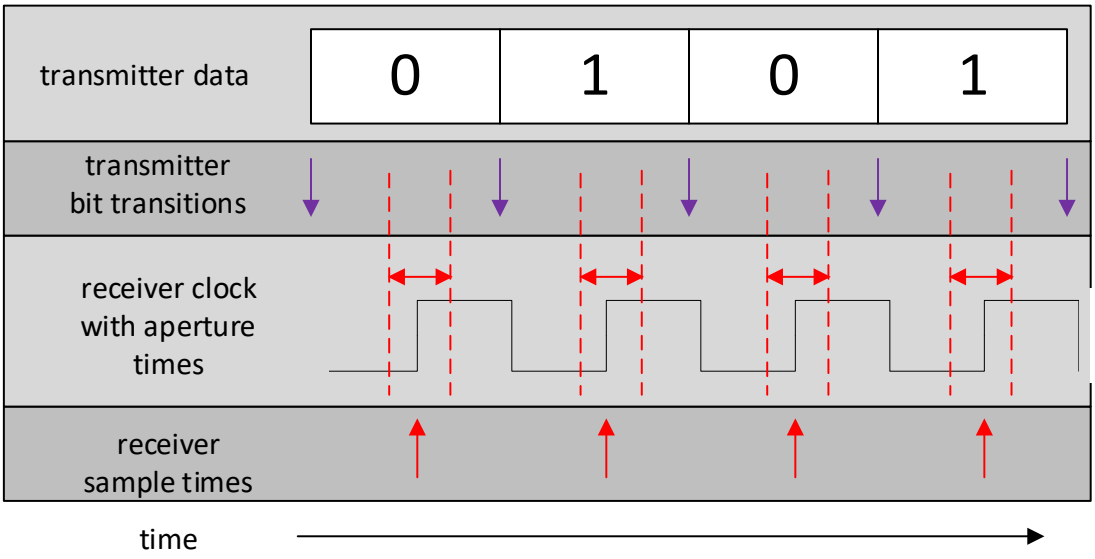


Fig. X: Timeline showing transmitter data and corresponding receiver sample clock. In this case, the receiver is sampling on the rising edge of its clock. Ideally, the transmitter will not change the bit value during the aperture time of the receiver, that is, the purple downward-facing arrows should not fall between the red dotted lines indicating the aperture time.

In a **synchronous protocol**, the transmitter clock is embedded in the channel. This way, this transmitter and the receiver share the same clock for the purpose of communicating on this channel. In an on-chip protocol, such as **Advanced Extensible Interface (AXI)**, the CAD tools that deploy the circuits will ensure

that the channel data is changed and sampled on the same clock edge, but it will also ensure that there is sufficient logic delay between the driving circuit and receiving circuit that there is no violation of the aperture time. For chip-to-chip protocols, opposite edges of the clock are used for changing the bit value and sampling the bit value, to minimize the chance of an aperture time violation.

Alternatively, in an **asynchronous protocol**, the receiver must infer the clock used for the transmitter. It can accomplish this using various methods. One method is for the transmitter and receiver to agree ahead of time on the clock period, as is used in the **Universal Asynchronous Receiver/Transmitter (UART) protocol**. Another method, applicable to serial channels only (which send one bit a time), is to encode the channel data using additional bits to maximize the number of serial transitions in the data stream, which allows the receiver to use an analog circuit called a **phase-locked loop** to extract the transmitter's clock from the data.

An example of this encoding technique is **10-bit/8-bit encoding** or **66-bit/64-bit encoding**. In 10-bit/8-bit encoding, even if the transmitter is ending a sequence of 8-bit values consisting of all zeros, the 10-bit encoded data would be 100111011000, which alternates between 0 and 1 five times within the 10 bits.

1.3 Serial Peripheral Interface

Serial Peripheral Interface (SPI) is a bidirectional, single-ended, master-slave, synchronous, serial protocol. This means that:

(1) **bidirectional:** data is sent in both directions,
(2) **single-ended:** one data wire per bit,
(3) **master-slave:** only one entity on the bus, the "master", can initiate communication,
(4) **synchronous:** a clock signal is provided in the channel, and
(5) **serial:** bits are transmitted one at a time.

The SPI channel consists of four physical wires. Three of these wires are driven by the master and only one is driven by the slave. Two of these wires are for bidirectional data, one is for the clock, and one is a signal that allows the master to initiate communication.

The names of the wires vary in different sources, but in this book we will use the following names:

(1) **sclk:** serial clock
(2) **ss:** slave select
(3) **mosi:** master out, slave in data
(4) **miso:** master in, slave out data

The **sclk** signal is only driven during communication and is idle (doesn't change value) when the channel is idle. Its value during idle times is important and is defined by the channel time mode, described below.

The **ss** signal is active-low, meaning that it is logic-1 when the channel is idle and logic-0 when the channel is in use.

The **mosi** and **miso** signals allow for the simultaneous transfer of one bit per cycle between the master and slave. In SPI, all communication occurs in multiples of eight cycles, meaning that every time the channel is invoked, at least eight bits are exchanged between the master and slave.

For this reason, SPI communication is comprised of **exchanges**. Each exchange is comprised of a group of eight cycles when the **ss** signal is low, the **sclk** pulses eight times, and eight bits are sent from both **miso** and **mosi**. However, the SPI protocol does not specify the order that the bits are transmitted, so depending on the behavior of the entities using the channel the bits may be transmitted from most significant to least significant (i.e. left to right), or least significant to most significant (i.e. right to left).

Important: Note that the SPI signal names sclk, ss, mosi, and miso are not standardized and will vary when shown it the data sheets of specific SPI peripherals. For example, sclk may be referred to as "SPC," ss may be referred to as "CS" (chip select), or mosi be referred to "SDI" (serial data out), although the usage of these signals will follow the exchange protocol described in this chapter.

1.3.1 SPI Timing

SPI channels can operate in four different **timing modes**. The timing mode must be agreed upon by both the master and slave entities before the channel is used. If not, transmission errors are likely, and SPI has no inherent mechanism to detect or recover from them.

The timing mode is defined by the values of the clock polarity, or **CPOL**, and the clock phase, or **CPHA**.

The **CPOL** value determines the idle state of the clock, or its value when the channel is not being used.

- A value of 0 means the clock idles at logic-0
- A value of 1 means that the clock idles at logic-1.

CPHA determines when the master and slave must change the data values on **miso** and **mosi**.

- A value of 0 means that the transmitter initially changes the data value one-half cycle prior to the first clock edge, meaning that the receiver must sample the data on the first clock edge.
- A value of 1 means that the transmitter changes the data value on the first clock edge, meaning that the receiver must sample the data on the second clock edge.

Keep in mind that the clock only runs during the exchange, so "first" and "second" clock edge refer to the first and second time the clock changes value during the exchange.

1.3.2 Bit Ordering

Since SPI is serial protocol that transmits one bit at a time, the bits that comprise each byte must be sent one at a time by the sender and reassembled one-at-a-time by the receiver. The SPI protocol does not specify the order in which bits within a byte are transmitted. Depending on the specific implementation used, the bits may be sent from most significant to least significant (i.e. left to right) or least significant to most significant (i.e. right to left). Both master and slave must use the same bit ordering or data will be misinterpreted on the receiving side.

1.3.3 Timing Diagrams

Figure X shows a **timing diagram** of an **SPI exchange**. In a timing diagram, the horizontal axis represents the flow of time. The vertical axis is divided into a set of horizontal strips that each contain a signal that participates in the protocol.

For each single bit signal, each signal strip's vertical space is used to show a high horizontal line to represent logic-1 and a low horizontal line to represent logic-0. Transitions between logic levels is shown with a steep ascending or descending line to represent the **rising edge** or **falling edge** of the signal. The important feature of edges is their **timing**, the relative horizontal position of the edge, which denotes where the edge occurs in relation to the edges of other signals.

During periods where the signal value is irrelevant, or having a **"don't care"** value, a shaded or thatched area is used.

1.3.4 SPI Exchange Examples

Figure X shows an example SPI exchange. Note that the exchange occurs during the period where the ss signal is low. The value of sclk when ss is high is determined by the value of CPOL. In the example, CPOL is set to 0 so the sclk signal is logic-0 during these times.

Since CPHA is 0, the mosi and miso signals transition from "don't care" to the value of the first bit transmitted one-half clock cycle prior to the first edge of the clock. This way, the data is ready to be sampled on the first edge of the clock, which because CPOL is set to 0 will be a rising edge. Thus the eight data bits of the exchange are sampled on the first through eighth rising edges of the clock during the time when ss is set to logic-0.

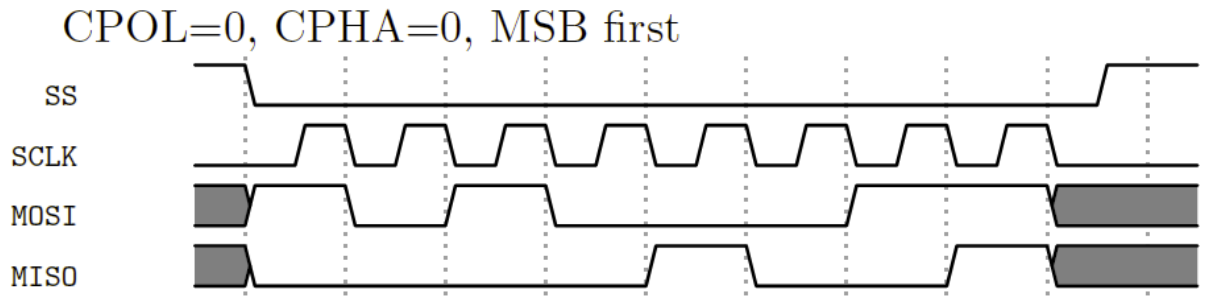


Figure X: SPI exchange example 1.

Figure X shows another example with different data bits transmitted. This time, CPHA is set to 1, so the initial change in mosi and miso occur on the first edge of the clock that again is the rising edge due to CPOL being 0. However, to avoid aperture violations, the data is sampled on the second edge of the clock, which is the falling edge. As a result, the 8 data bits are sampled on the *second* to *ninth falling* edges of sclk.

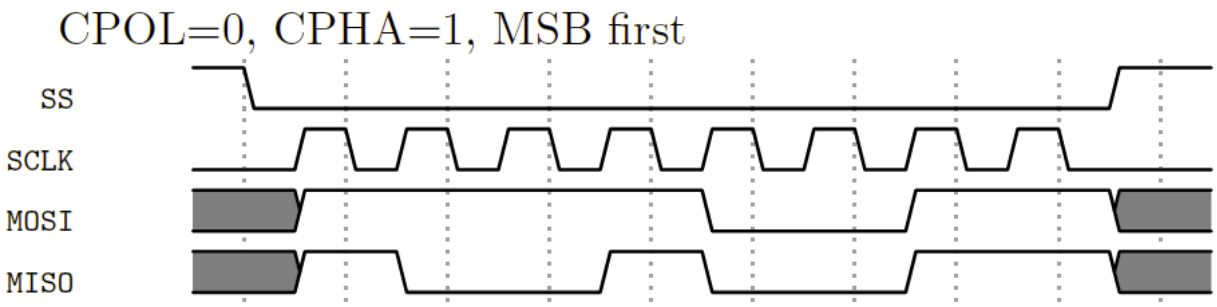


Figure X: SPI exchange example 2.

Figure X and X shows the same two scenarios but this time with CPOL set to 1. In these cases, the first edge of the clock is falling. As a result, when CPHA is set to 0 the data is sampled on the first through eighth falling edges of the clock, bit when CPHA is set to 1 the data is sampled on the second through ninth *rising* edges of the clock.

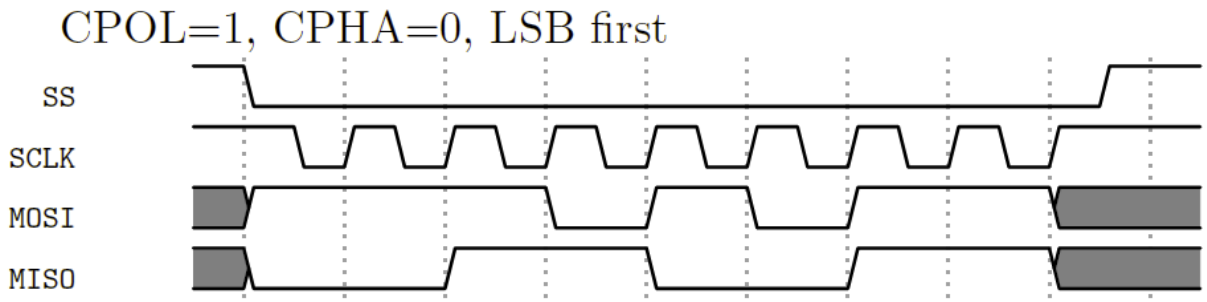


Figure X: SPI exchange example 3.

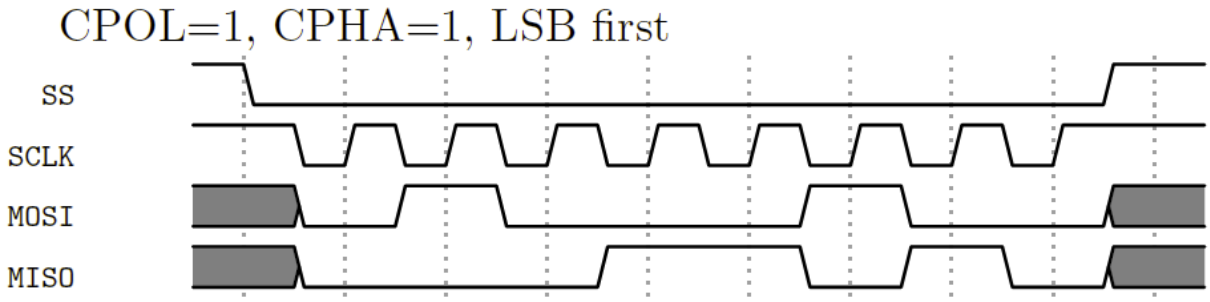
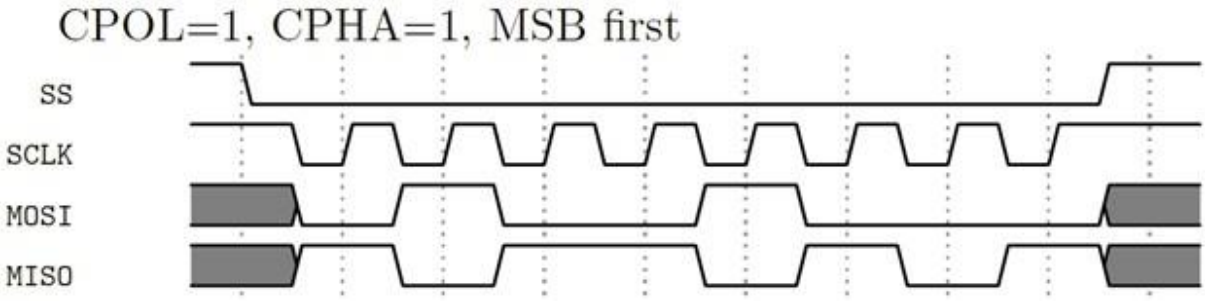
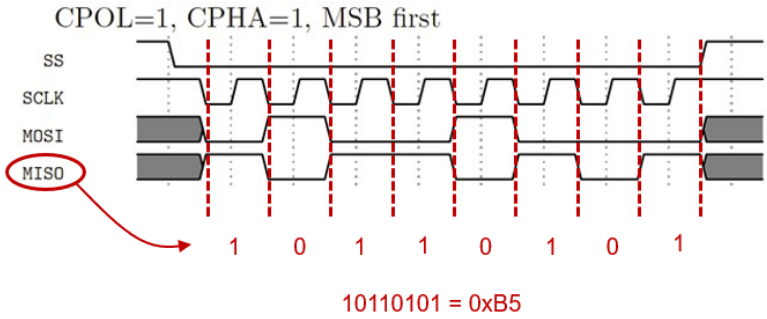


Figure X: SPI exchange example 4.

Exercise: In the following SPI exchange, what is the value sent from the slave to the master?



Answer:



Since CPHA is set to 1, the data bits are valid between each pair of edges of sclk starting with the first. This way, the value in the middle of this interval is the value sampled by the receiver and is clearly visible to the reader.

Additionally, since the most significant bit is transmitted first (as denoted by the phrase "MSB first"), the bits should be read from left to right. For miso, the bits would be reassembled to 10110101_2 or $B5_{16}$.

1.4 Building Programmed I/O on Top of SPI

The SPI protocol allows for bidirectional communication, but multiple exchanges are needed to implement a programmed I/O-style interface where the master can transmit an address that can subsequently be read or written.

To avoid confusion, we must be careful with the terminology used. Recall that an **SPI exchange** is the exchange of exactly 8 bits between the master and slave. Using multiple SPI exchanges we can perform an **SPI transaction**, which is generally used to read from an SPI address or write to an SPI address much in the same way as programmed I/O transactions between a CPU and peripherals. In this communication model, the CPU is the SPI master, and the peripheral is the SPI slave.

At a minimum, two SPI exchanges are needed to perform an SPI transaction, although more are needed in cases where the address or data requires more than 8 bits[1] or in cases of burst accesses where multiple values are read or written from one starting address.

The SPI protocol only defines the protocol for the SPI exchange--not the SPI transaction--so peripheral designers and programmers are free to use whatever transaction protocol they wish. There is no standard for this.

1.4.1 Consecutive SPI Exchanges for Address and Data Bytes

It is possible to perform multiple SPI exchanges in consecutive groups of 8 cycles (one group per exchange), if the ss signal is held to logic-0 throughout the sequence of SPI exchanges.

One common protocol to implement an SPI transaction using multiple consecutive SPI exchanges is the following protocol:

1. in the first SPI exchange, use mosi to send a 6-bit address, a read/write flag, and a single or multiple flag, and
2. in subsequent SPI exchanges, use miso to send read data in the case of a read transaction, or use mosi to send write data to the slave in the case of a write transaction.

Notice from this description that the first SPI exchange only requires the use of mosi, while the others either require only miso in the case of a read or mosi in the case of a write.

1.4.2 Example Read Transaction

Figure X shows an example of a read transaction using two consecutive SPI exchanges. In this protocol, the first exchange utilizes the mosi wire for the master to send the slave a read/write flag followed by a 7-bit address ordered from most significant to least significant bit. The read/write flag is set to 0 for a read transaction and 1 for a write transaction. This initial transmitted byte is called the **header**.

The second transaction will behave differently depending on if the transaction is a read or write:

- In a read transaction, the second exchange will utilize the miso wire for the slave to send the data stored at the requested address as specified in the header from the slave to the master.
- In a write transaction, the second exchange will utilize the mosi wire for the master to send the slave the data to be written at the address specified in the header.

Figure X example shows an example read transaction, so the second exchange allows the slave to use the miso wire to send the requested byte back to the master.

[1] Note that one or two bits in the first SPI exchange are usually reserved for flags, so in practice an SPI transaction address is often limited to 6 or 7 bits.

Important: Note that the describe transaction protocol is an example only, and there is variation in transaction protocols among different SPI peripherals! You must refer to the data sheet of each specific SPI peripheral to discover its specific transaction protocol.

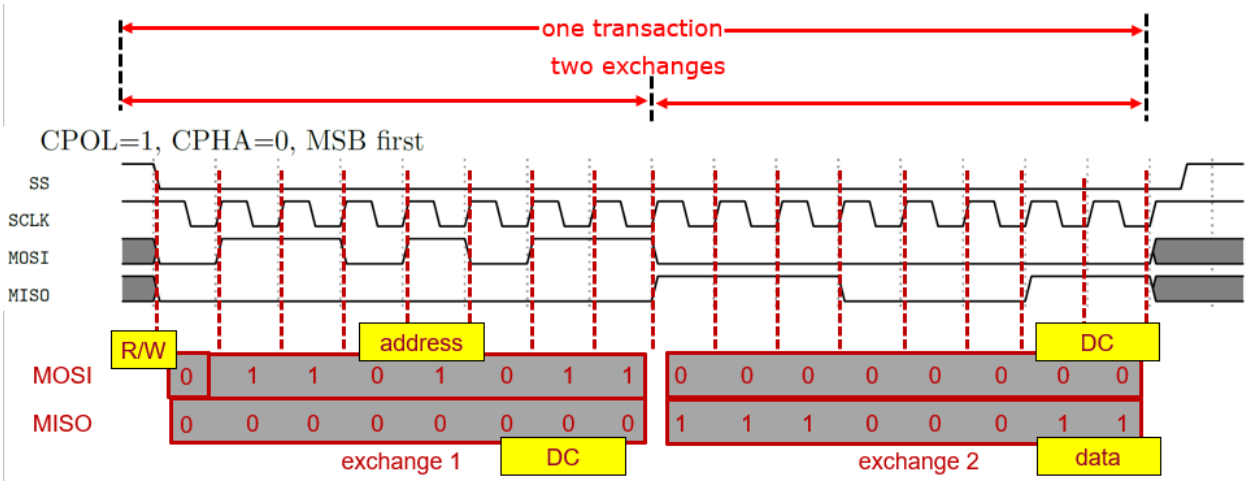


Figure X: Example SPI read transaction.

1.4.3 Unused Wires

Notice that the miso wire is unused during the first exchange. This is because the slave has nothing to send the master during the first exchange, because the entire first exchange must be performed before the slave can retrieve the requested data for the master.

Likewise, the mosi wire is unused during the second exchange. This is because the master has nothing to send the slave after the first exchange where the read/write bit and address are transmitted.

1.4.4 Performance Implications of the Read Transaction

In the read transaction, the slave is expected to have the requested data ready for the master during the 9^{th} clock cycle, but it doesn't have the address until after the 8^{th} clock cycle.

This means that the slave is not given even a single channel to retrieve the requested data. While this might seem unrealistic, keep in mind that the internal clock on the peripheral is generally operating with a higher frequency than the SPI channel frequency, meaning that it could have multiple internal clock cycles to access the data between the first and second SPI exchanges.

1.4.5 Example Write Transaction

Figure X shows an example write transaction. The transaction is like that of the read transaction except for two differences:

1. In the first exchange, the read/write bit is set to 1, and
2. In the second exchange, the data to be written is sent by the master through the mosi wire, leaving the miso wire idle.

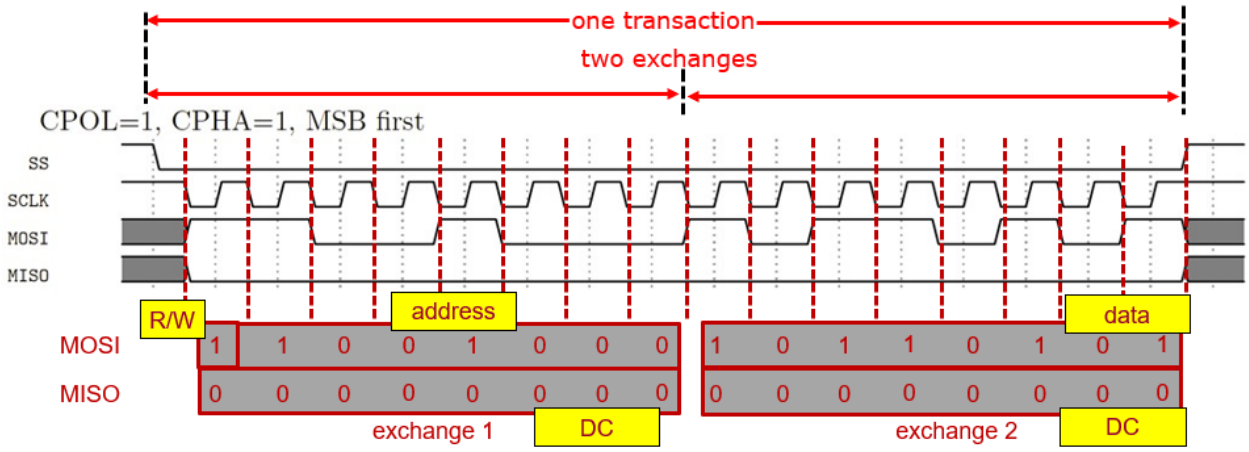


Figure X: Example SPI write transaction.

Notice that in this example the miso wire is idle throughout the entire transaction. Also, since there is no acknowledgement mechanism the master does not receive confirmation from the slave that the write transaction was successful succeeded.

1.5 Multiple Read and Write Transactions

The transaction protocol described above only permits a single byte to be read or written, but in some applications, having the ability in a single transaction to read or write multiple bytes associated with consecutive addresses would offer a significant improvement in channel **throughput**, allowing more bytes to be transmitted per time by avoiding the need having to send a header for each byte read or written.

Peripherals that support this behavior must solve two problems:

1. provide a method for the master to flag a transaction as single- or multi-byte transaction, and
2. provide a method for the master to specify the number of bytes that should be included in the transaction.

For the first challenge, a bit can be reserved in the header to denote a multi-byte operation. For example, the header might be comprised of two bits of flags, read/write and single/multiple, followed by a six-bit address.

Two common methods to address the second challenge are:

1. set the number of bytes in the transaction as $\frac{c-8}{8}$, where c = the number of cycles in which ss is held at logic low; in other words, allow the master to determine the number of bytes accessed by the amount of time it asserts the ss wire, or
2. add a second header byte that contains the number of bytes in the transaction; that is, the first exchange in the transaction will contain the flags and address, the second exchange in the transaction will contain the number of bytes transacted, and subsequent exchanges will contain the data.

Figure X shows an example multi-byte read transaction where the header is comprised of the read/write flag, the single/multiple flag, and a 6-bit address, and with the length of transaction determined by the number of cycles that ss is held low.

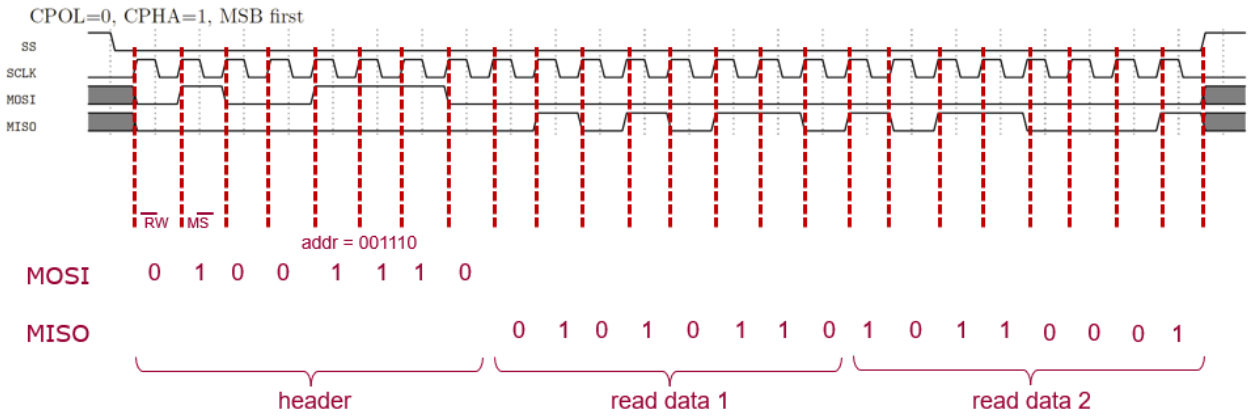


Figure X: Example multiple-byte read transaction.

Figure X shows an example multi-byte write transaction where the header is comprised of the read/write flag, the single/multiple flag, and a 6-bit address, and with the length of transaction determined by the number of cycles that ss is held low.

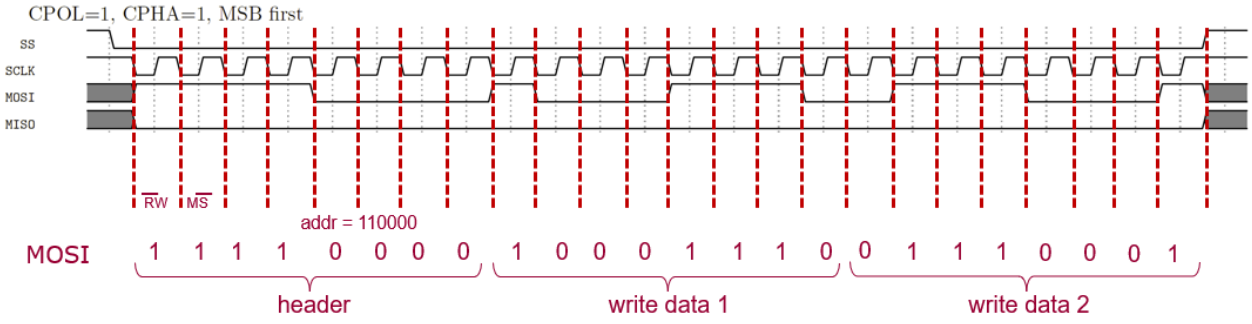


Figure X: Example multiple-byte write transaction.

3.6 SPI with Multiple Slaves

Previously, we described SPI channels that contain only one master and one slave. Although SPI is a single-master bus, it is possible to support multiple slaves per channel.

SPI presents two challenges in how to connect multiple slaves:

1. the master only has a single ss signal to target one slave to initiate an exchange, and
2. there is only one miso line to support a single slave data transmission.

To solve these issues, there are two methods for structuring a multi-slave channel.

The first is to add additional slave select outputs to the master and short all the miso wires. In order to allow all potential slaves to transmit on the same miso wire, they must include a **tristate buffer** on their

miso output pin, which allows each slave to electronically disconnect from the miso wire when they are not participating in an exchange.

Figure X shows this arrangement, which is called "**independent configuration**".

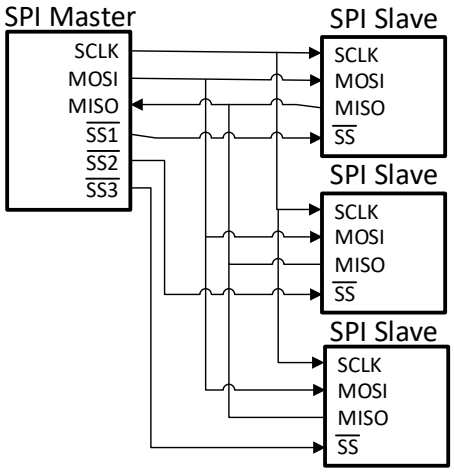


Figure X: Multi-slave topology in "independent" configuration.

TODO: replace this figure with custom version

As shown in Figure X, the second option is to short all the SS wires and configure all the mosi and miso connections in a **daisy-chained configuration**, which is shown in Figure X. In this case, all slaves must participate in any exchanges because there is only one ss wire, and each exchange will be performed not for 8 bits but for $s \times 8$ bits, where s = the number of slaves. In each SPI clock cycle, one bit will be exchanged between each entity, including the master, and its neighbor.

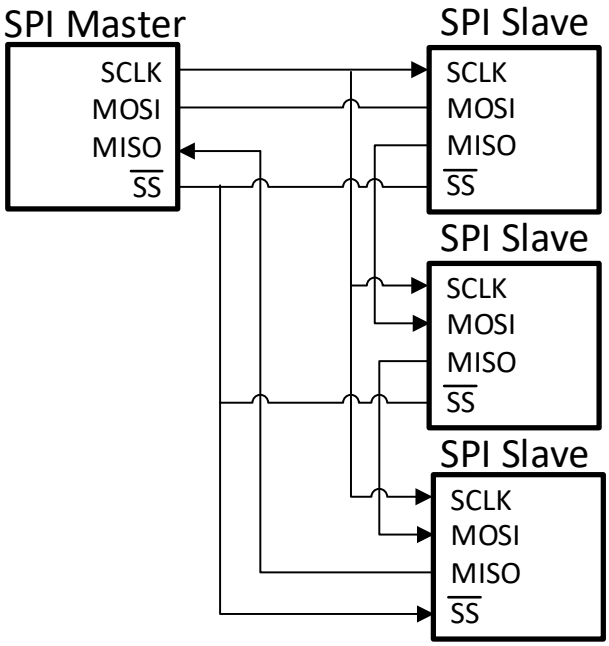


Figure X: Multi-slave topology in "daisy-chained" configuration.

TODO: replace this figure with custom version

For example, in an arrangement with one master and three slaves, each exchange is performed for 24 bits, allowing 8 bits from the master to circulate through all three slaves. This works by allowing bits to flow from the master by way of its mosi to slave 1, then from slave 1 by its miso to slave 2, then from slave 2 by its miso to slave 3, and finally from slave 3 by its miso to the master. In this configuration, slave 1 is the master to slave 2, slave 2 is the master to slave 3.

3.7 Lab Project

In this lab, you will implement a protocol decoder which can decode an SPI signal. This lab has several components. In the first component, you will decode simple two-exchange transactions, next you will extend your code to decode both two-exchange and streaming transactions. Finally, you will add support for handling arbitrary values of CPOL and CPHA.

This lab requires that you download support code from github.com/HeRCLab/XXX.git.

3.7.1 Trace File

This lab will require that you write code that reads a **signal trace file**, which is a text file that contains the value of signals at specific sample times.

For example, the example signals shown in the waveform in Figure X are sampled at times 1 ms, 3 ms, 4 ms, 5 ms, 8 ms, 9 ms, 10ms, 11 ms, 12 ms, 15 ms, 18 ms, and 20 ms, as shown by the vertical dotted lines.

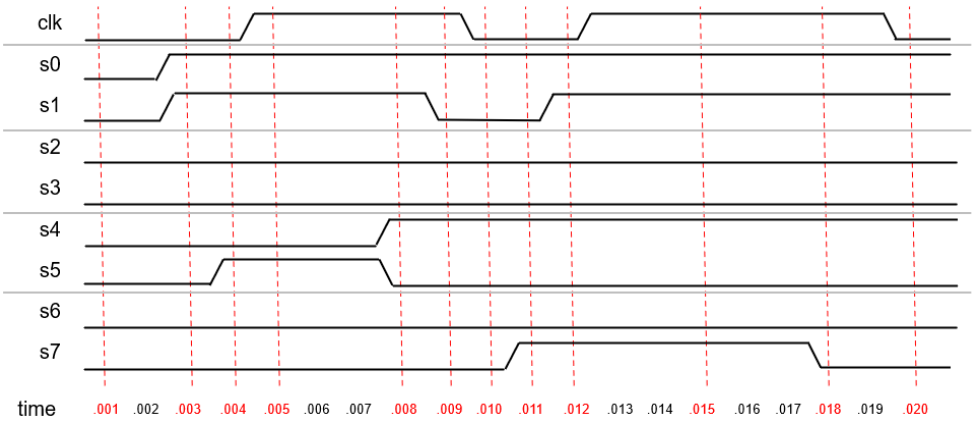


Figure X: Example signal trace.

Figure X shows the contents of the corresponding trace file:

- the first line contains the number of samples,
- the second line contains a list of signal names separated by whitespace,
- the third line contains the number of bits in each signal, and
- the fourth to 15 lines show the sample time followed by the state of each signal at that sample time.

12 # samples

clk	s0	s1	s2	s3	s4	s5	s6	s7	
1	1	1	1	1	1	1	1	1	
0.001	0	0	0	0	0	0	0	0	0
0.003	0	1	1	0	0	0	0	0	0
0.004	0	1	1	0	0	0	1	0	0
0.005	1	1	1	0	0	0	1	0	0
0.008	1	1	1	0	0	1	0	0	0
0.009	1	1	0	0	0	1	0	0	0
0.010	0	1	0	0	0	1	0	0	0
0.011	0	1	0	0	0	1	0	0	1
0.012	0	1	1	0	0	1	0	0	1
0.015	1	1	1	0	0	1	0	0	1
0.018	1	1	1	0	0	1	0	0	0
0.020	0	1	1	0	0	1	0	0	0

Figure X: Example signal trace file.

The support software includes a C and Python library for parsing the signal trace file.

3.7.2 Lab Requirements

You will write a program which reads in pre-recorded signal data, which will allow it to determine the value of various digital signals at specific points in time. The following signals are present in the input waveform:

- cpha - CPHA value for the SPI bus (guaranteed to be 0 except in part 3).
- cpol - CPOL value for the SPI bus (guaranteed to be 0 except in part 3).
- miso - MISO (master in slave out) signal for the SPI bus.
- mosi - MOSI (master out slave in) signal for the SPI bus.
- sclk - SCLK (serial clock) signal for the SPI bus.
- ss - SS (slave select) signal for the SPI bus - **active low**.

Note: note that a correct solution to part 2 is also a correct solution to part 1, and a correct solution to part 3 is also a correct solution to parts 1 and 2. Thus, you do not need to write multiple separate programs.

Note: as in the previous assignment, all hexadecimal numbers should be formatted without a leading 0x and in all-lowercase, and le_-zer-padded to be a fully byte long.

Note: for all parts of this assignment, you should assume that SPI data is transmitted in MSB-first bit order.

Note: for all parts of this assignment, you may assume that the bus does not have any signal transmission errors, and that all transactions are fully complete (e.g. we don't expect you to handle invalid, corrupted, or partial transactions).

3.7.3 Part 1: Two-Exchange Transactions

In Part 1, the SPI bus will always have the settings CPHA=0, CPOL=0, and will only contain transactions that contain exactly two exchanges. Each transaction can represent either a read or a write. Both types of transactions begin with a transaction from the master to the slave, where the slave transmits only 0 values to the master, and the master transmits a byte with the following format:

bits 7 (MSB)…2	bit 1	bit 0 (LSB)
address	read/write	stream

Figure X: Header format.
TODO: convert to regular table

Thus, the master transmits a 6 bit address, followed by a 1 if a write is requested, and a 0 if a read is requested. Finally, the stream field will always be 0 for this part of the lab.

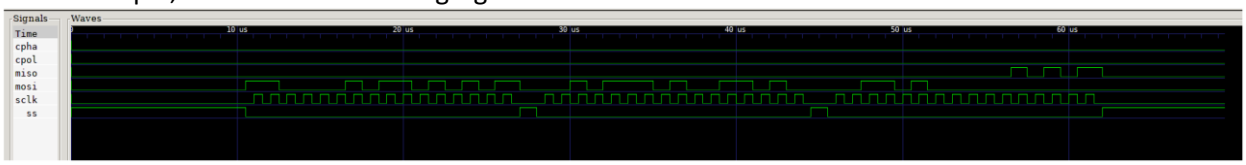
If the first exchange was a read request, the second exchange will send 0 from the master to the slave, and an arbitrary data byte from the slave to the master.

If the first exchange was a write request, the second exchange will send an arbitrary data byte from the master to the slave, and 0 from the slave to the master.

Your program must process the data from the SPI bus, and output a textual log of what events occurred on the bus. Your program will output one line of output for each transaction, in the following format:

- For a read request, the output should be RD <address> <value>, where <address> is replaced by the address in hexadecimal, and <value> is replaced with the value the slave responded with, also in hexadecimal.
- For a write request, the output should be WR <address> <value> where <address> is replaced by the address in hexadecimal, and <value> is replaced with the value the master wrote to the slave.

As an example, consider the following signal trace:



This trace contains three two-exchange transactions:

- The first writes to address 0x30 the value 0xd5.
- The second writes to address 0x0b the value 0x9a.
- The third reads from the address 0x0d and gets the value 0x15 as a result.

This should correspond to the following output:

```
WR 30 d5
WR 0b 9a
RD 0d 15
```

You can test your code against just this simple example using the command
sh grade.sh –only case000.

You can test your code against just the test cases for part 1 using this command:
sh grade.sh --only $(cd test_cases ; echo part1_*).

3.7.4 Part 2: Streaming Transactions

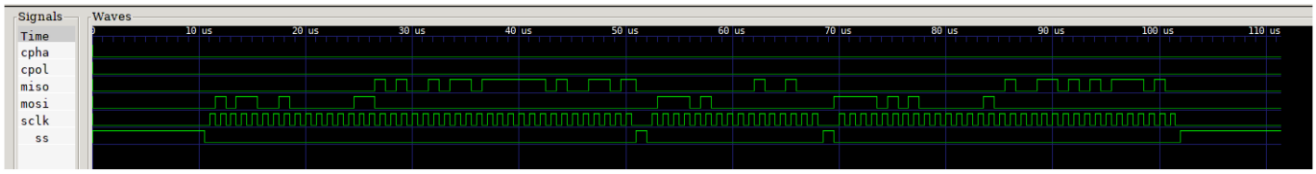
In part 2, the SPI bus will work exactly as in part 1, except that for some transactions, the stream bit may be set to 1. In this case, the second exchange will always be from the master to the slave, and will encode a value N (in this exchange, the slave always transmits 0 to the master).

The subsequent N many exchanges will be sent either from the master to the slave or the slave to the master (depending on whether the read/write flag is 1 or 0 respectively). Your program must still output data in the same format as in part 1, but if a transaction is encountered where the stream field is 1, it should instead output data in the following format:

- For a read request, the output should be RD STREAM <address> <value 1> <value 2> ... <value n>, where the <address> is replaced with the address read, and the <value >s replaced with the values streamed in order.
- For a read request, the output should be WR STREAM <address> <value 1> <value 2> … <value n>, where the <address> is replaced with the address read, and the <value >s replaced with the values streamed in order.

NOTE: N will never be greater than 32.

NOTE: the output from your program is whitespace-sensitive, take care not to accidentally leave a trailing whitespace while concatenating the list of values. As an example, consider the following example:



This signal trace contains three transactions and 11 exchanges, which are shown in the following table:

timestamp	slave to master	master to slave
1100.000000	00	59
1900.000000	00	03
2700.000000	a5	00
3500.000000	bf	00
4300.000000	4d	00
5250.000000	00	74
6050.000000	24	00
7000.000000	00	f5
7800.000000	00	02
8600.000000	9a	00
9400.000000	ba	00

Figure X: List of exchanges in sample waveform.
TODO: convert to regular table.

Transaction 1:

- The first exchange sends the byte 0x59 to the slave, which corresponds to an address of 0x16, with a read/write flag of 0 (signifying a read), and a stream flag of 1, indicating this is a streaming transaction.
- The next exchange sends the byte 0x03 to the slave, indicating a streaming read of 3 bytes.
- The subsequent 3 transactions send the values 0x15, 0xbf, and 0x4d from the slave to the master.

Transaction 2:

- The first exchange sends the byte 0x74 to the slave, indicating an address of 0x1d, a read/write flag of 0, and a stream flag of 0, indicating this is a single read operation.
- The second exchange sends the byte 0x24 from the slave to the master.

Transaction 3:

- The first exchange sends the byte 0xf5 from the master to the slave, indicating an address of 0x3d, a read/write flag of 0, and a stream flag of 1, indicating a streaming read.
- The second exchange sends the byte 0x02 from the master to the slave, indicating that the streaming read is of length 2.
- The final two exchanges send the bytes 0x9a and 0xba from the slave to the master.

This should correspond to the following output:

```
RD STREAM 16 a5 bf 4d
RD 1d 24
RD STREAM 3d 9a ba
```

You can test your code against just this simple example using the command
`sh grade.sh –only case001`.

You can test your code against just the test cases for part 2 using this command:
`sh grade.sh --only $(cd test_cases ; echo part2_*)`.

3.7.5 Part 3: Flexible Timing Modes

In part 3, the SPI bus will work exactly as in part 2, and the output format of your program will not change. However, your program must read the value of the cpol and cpha signals when it first starts up and configure itself accordingly. The observed behavior of the other bus signals is guaranteed to be consistent with the settings for CPOL and CPHA.

Note: the value of CPOL and CPHA is guaranteed not to change during the course of a given test case, but may vary between test cases.

Hint: you can reasonably expect that few if any test cases in part 3 will have CPOL=0 and CPHA=0, so as to differentiate them from the test cases of part 2.

You can test your code against just the test cases for part 3 using this command:
`sh grade.sh --only $(cd test_cases ; echo part3_*)`.

3.8 Chapter Summary

This chapter covered the fundamental concepts of SPI, a single-master serial synchronous protocol, including:

1. The capabilities of an SPI channel
2. The components of an SPI channel
3. The timing of SPI exchanges
4. Example transaction protocols for SPI
5. Example multi-byte SPI transactions
6. Multi-slave SPI channel