Serial Peripheral Interface (SPI) a learn.sparkfun.com tutorial

Available online at: http://sfe.io/t16

Contents

Introduction

Serial Peripheral Interface (SPI) is an interface bus commonly used to send data between microcontrollers and small peripherals such as shift registers, sensors, and SD cards. It uses separate clock and data lines, along with a select line to choose the device you wish to talk to.

Suggested Reading

Stuff that would be helpful to know before reading this tutorial:

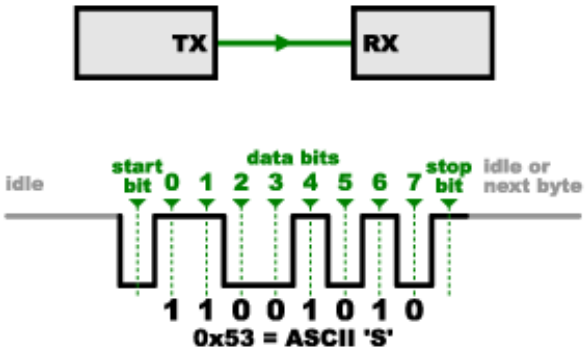
- Binary
- Serial Communication
- Shift registers
- Logic Levels

What's Wrong with Serial Ports?

A common serial port, the kind with TX and RX lines, is called “asynchronous” (not synchronous) because there is no control over when data is sent or any guarantee that both sides are running at precisely the same rate. Since computers normally rely on everything being synchronized to a single “clock” (the main crystal attached to a computer that drives everything), this can be a problem when two systems with slightly different clocks try to communicate with each other.

To work around this problem, asynchronous serial connections add extra start and stop bits to each byte help the receiver sync up to data as it arrives. Both sides must also agree on the transmission speed (such as 9600 bits per second) in advance. Slight differences in the

transmission rate aren't a problem because the receiver re-syncs at the start of each byte.

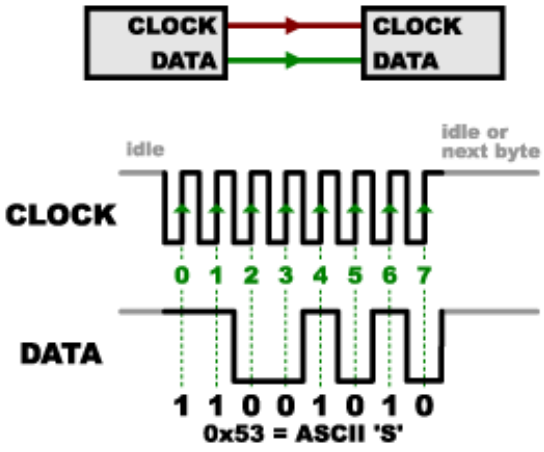


(By the way, if you noticed that "11001010" does not equal 0x53 in the above diagram, kudos to your attention to detail. Serial protocols will often send the least significant bits first, so the smallest bit is on the far left. The lower nybble is actually 0011 = 0x3, and the upper nybble is 0101 = 0x5.)

Asynchronous serial works just fine, but has a lot of overhead in both the extra start and stop bits sent with every byte, and the complex hardware required to send and receive data. And as you've probably noticed in your own projects, if both sides aren't set to the same speed, the received data will be garbage. This is because the receiver is sampling the bits at very specific times (the arrows in the above diagram). If the receiver is looking at the wrong times, it will see the wrong bits.

A Synchronous Solution

SPI works in a slightly different manner. It's a "synchronous" data bus, which means that it uses separate lines for data and a "clock" that keeps both sides in perfect sync. The clock is an oscillating signal that tells the receiver exactly when to sample the bits on the data line. This could be the rising (low to high) or falling (high to low) edge of the clock signal; the datasheet will specify which one to use. When the receiver detects that edge, it will immediately look at the data line to read the next bit (see the arrows in the below diagram). Because the clock is sent along with the data, specifying the speed isn't important, although devices will have a top speed at which they can operate (We'll discuss choosing the proper clock edge and speed in a bit).



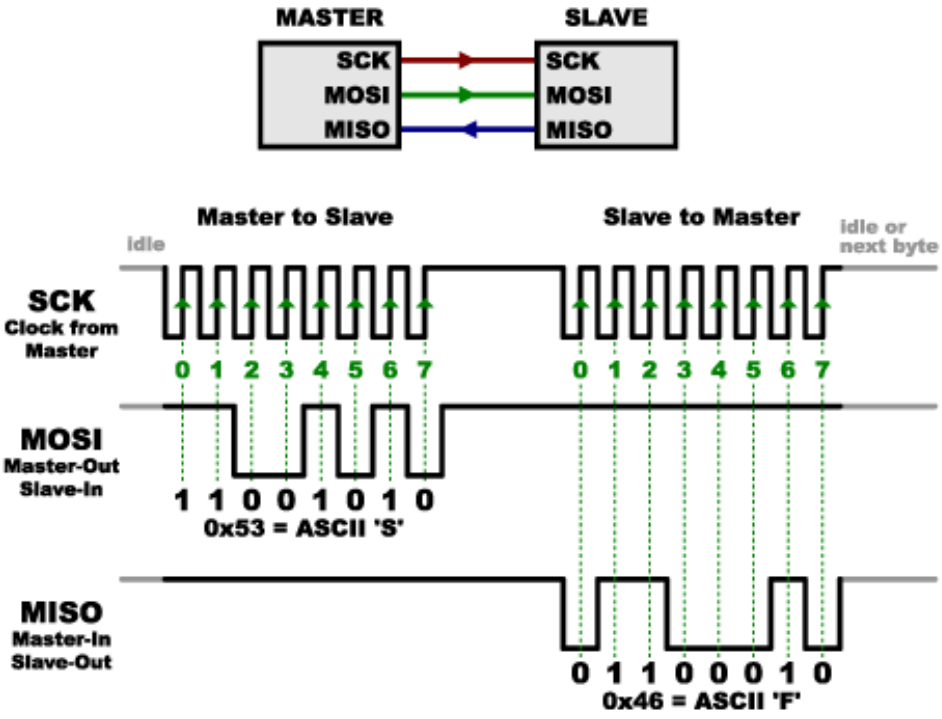
One reason that SPI is so popular is that the receiving hardware can be a simple shift register. This is a much simpler (and cheaper!) piece of hardware than the full-up UART (Universal Asynchronous Receiver / Transmitter) that asynchronous serial requires.

Receiving Data

You might be thinking to yourself, self, that sounds great for one-way communications, but how do you send data back in the opposite direction? Here's where things get slightly more complicated.

In SPI, only one side generates the clock signal (usually called CLK or SCK for Serial ClocK). The side that generates the clock is called the "master", and the other side is called the "slave". There is always only one master (which is almost always your microcontroller), but there can be multiple slaves (more on this in a bit).

When data is sent from the master to a slave, it's sent on a data line called MOSI, for "Master Out / Slave In". If the slave needs to send a response back to the master, the master will continue to generate a prearranged number of clock cycles, and the slave will put the data onto a third data line called MISO, for "Master In / Slave Out".

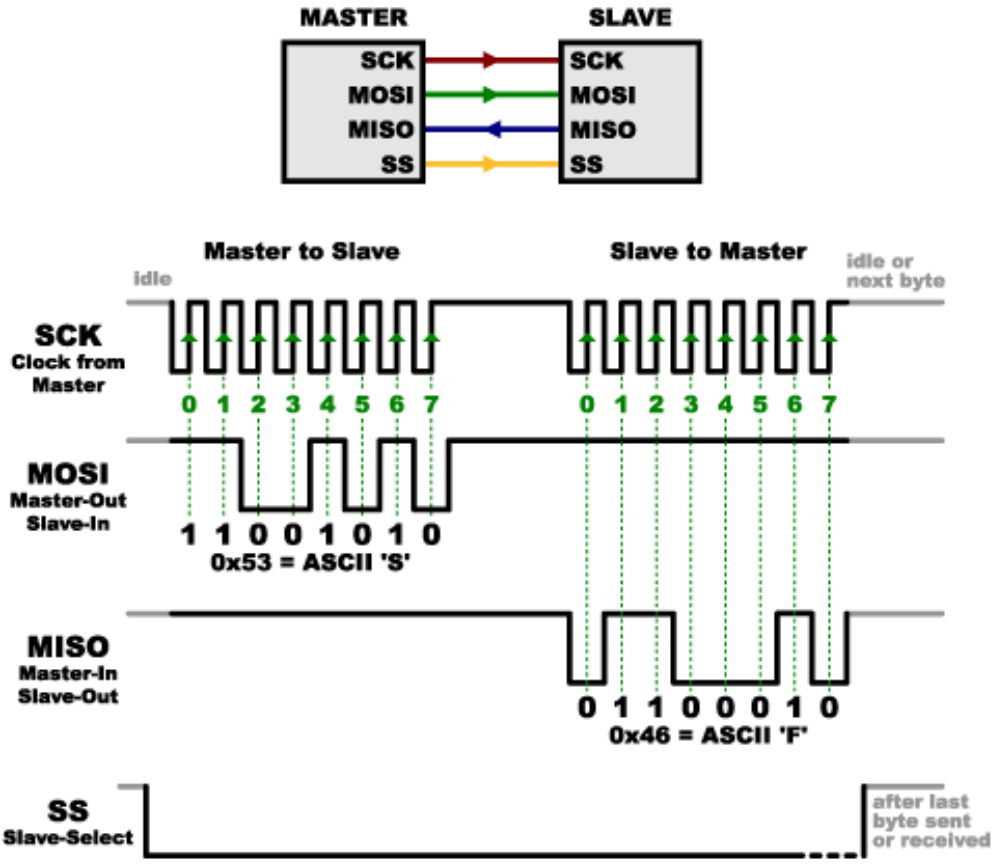


Notice we said "prearranged" in the above description. Because the master always generates the clock signal, it must know in advance when a slave needs to return data and how much data will be returned. This is very different than asynchronous serial, where random amounts of data can be sent in either direction at any time. In practice this isn't a problem, as SPI is generally used to talk to sensors that have a very specific command structure. For example, if you send the command for "read data" to a device, you know that the device will always send you, for example, two bytes in return. (In cases where you might want to return a variable amount of data, you could always return one or two bytes specifying the length of the data and then have the master retrieve the full amount.)

Note that SPI is “full duplex” (has separate send and receive lines), and, thus, in certain situations, you can transmit and receive data *at the same time* (for example, requesting a new sensor reading while retrieving the data from the previous one). Your device’s datasheet will tell you if this is possible.

Slave Select (SS)

There’s one last line you should be aware of, called SS for Slave Select. This tells the slave that it should wake up and receive / send data and is also used when multiple slaves are present to select the one you’d like to talk to.



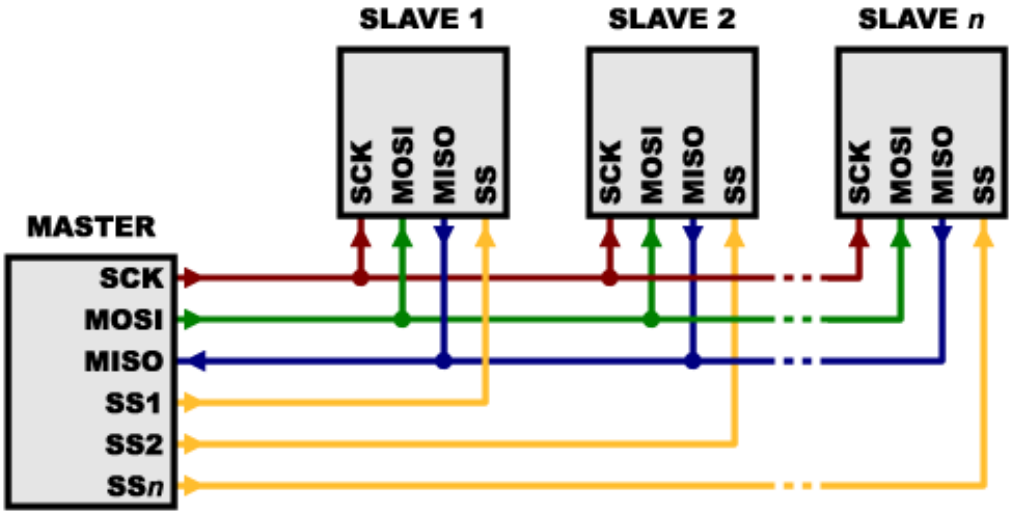
The SS line is normally held high, which disconnects the slave from the SPI bus. (This type of logic is known as “active low,” and you’ll often see used it for enable and reset lines.) Just before data is sent to the slave, the line is brought low, which activates the slave. When you’re done using the slave, the line is made high again. In a shift register, this corresponds to the “latch” input, which transfers the received data to the output lines.

Multiple slaves

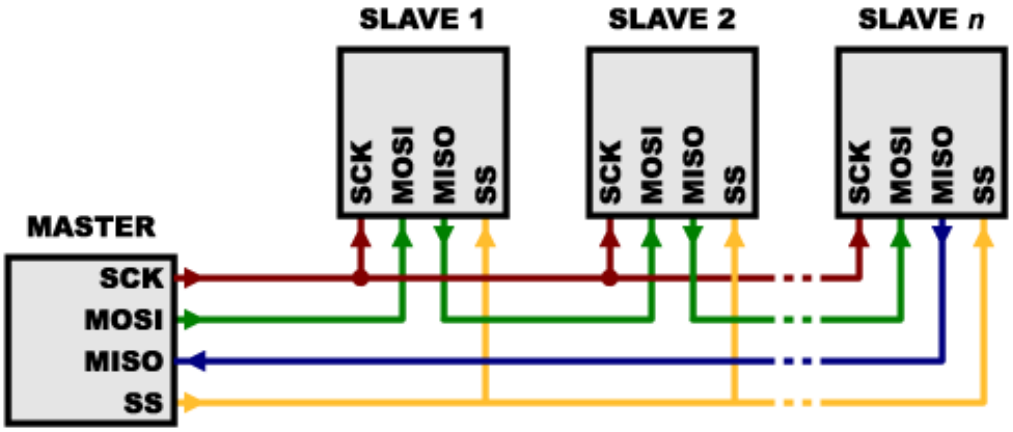
There are two ways of connecting multiple slaves to an SPI bus:

1. In general, each slave will need a separate SS line. To talk to a particular slave, you’ll make that slave’s SS line low and keep the rest of them high (you don’t want two slaves activated at the same time, or they may both try to talk on the same MISO line resulting in garbled data). Lots of slaves will require lots of SS lines; if you’re running

low on outputs, there are [binary decoder chips] that can multiply your SS outputs.



1. On the other hand, some parts prefer to be daisy-chained together, with the MISO (output) of one going to the MOSI (input) of the next. In this case, a single SS line goes to *all* the slaves. Once all the data is sent, the SS line is raised, which causes all the chips to be activated simultaneously. This is often used for daisy-chained shift registers and [addressable LED drivers].



Note that, for this layout, data overflows from one slave to the next, so to send data to any *one* slave, you'll need to transmit enough data to reach *all* of them. Also, keep in mind that the *first* piece of data you transmit will end up in the *last* slave.

This type of layout is typically used in output-only situations, such as driving LEDs where you don't need to receive any data back. In these cases you can leave the master's MISO line disconnected. However, if data does need to be returned to the master, you can do this by closing the daisy-chain loop (blue wire in the above diagram). Note that if you do this, the return data from slave 1 will need to pass through *all* the slaves before getting back to the master, so be sure to send enough receive commands to get the data you need.

Programming for SPI

Many microcontrollers have built-in SPI peripherals that handle all the details of sending

and receiving data, and can do so at very high speeds. The SPI protocol is also simple enough that you (yes, you!) can write your own routines to manipulate the I/O lines in the proper sequence to transfer data. (A good example is on the Wikipedia SPI page.)

If you're using an Arduino, there are two ways you can communicate with SPI devices:

1. You can use the shiftIn() and shiftOut() commands. These are software-based commands that will work on any group of pins, but will be somewhat slow.

2. Or you can use the SPI Library, which takes advantage of the SPI hardware built into the microcontroller. This is vastly faster than the above commands, but it will only work on certain pins.

You will need to select some options when setting up your interface. These options must match those of the device you're talking to; check the device's datasheet to see what it requires.

- The interface can send data with the most-significant bit (MSB) first, or least-significant bit (LSB) first. In the Arduino SPI library, this is controlled by the setBitOrder() function.

- The slave will read the data on either the rising edge or the falling edge of the clock pulse. Additionally, the clock can be considered "idle" when it is high or low. In the Arduino SPI library, both of these options are controlled by the setDataMode() function.

- SPI can operate at extremely high speeds (millions of bytes per second), which may be too fast for some devices. To accommodate such devices, you can adjust the data rate. In the Arduino SPI library, the speed is set by the setClockDivider() function, which divides the master clock (16MHz on most Arduinos) down to a frequency between 8MHz (/2) and 125kHz (/128).

- If you're using the SPI Library, you must use the provided SCK, MOSI and MISO pins, as the hardware is hardwired to those pins. There is also a dedicated SS pin that you can use (which must, at least, be set to an output in order for the SPI hardware to function), but note that you can use any other available output pin(s) for SS to your slave device(s) as well.

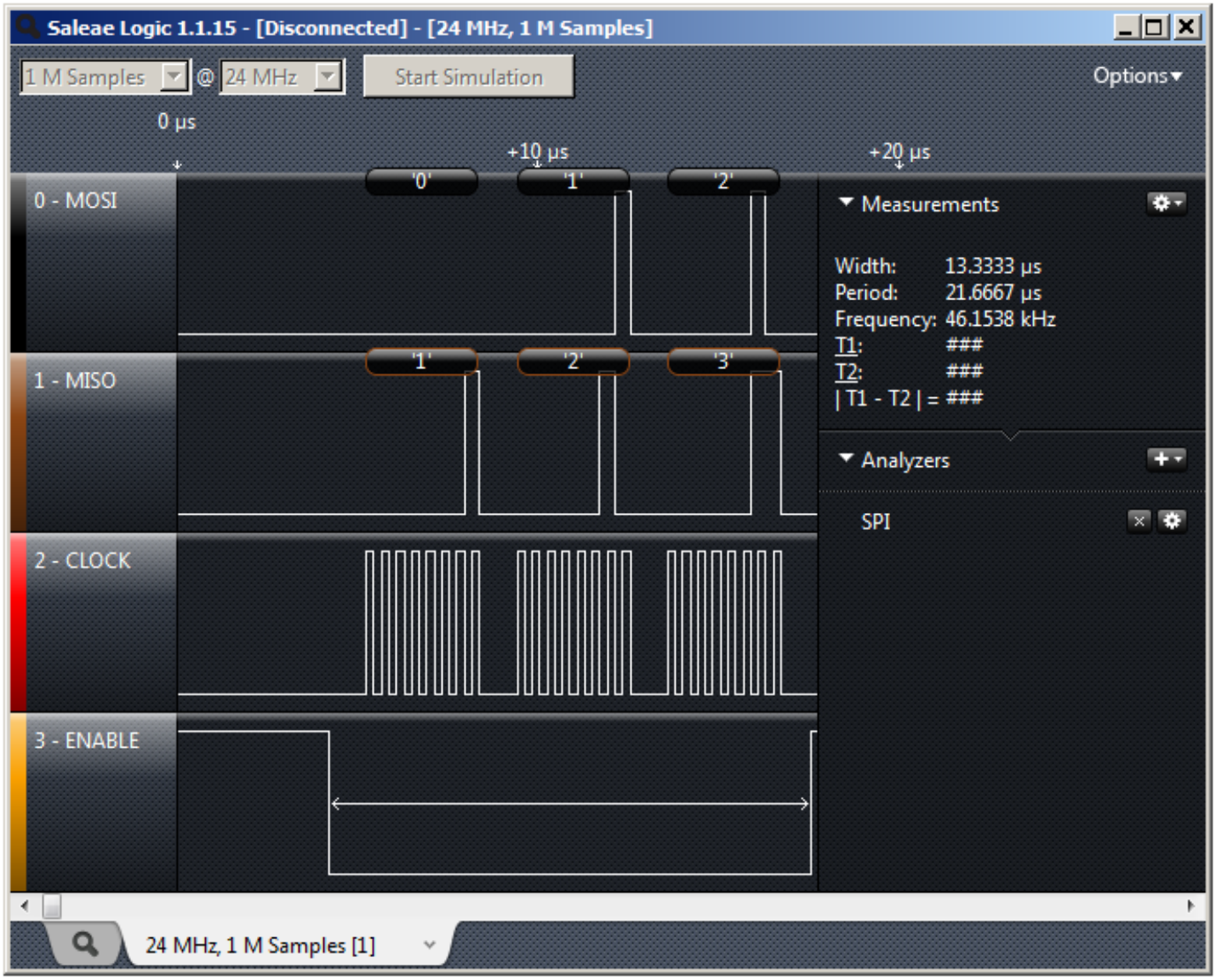
- On older Arduinos, you'll need to control the SS pin(s) yourself, making one of them low before your data transfer and high afterward. Newer Arduinos such as the Due can control each SS pin automatically as part of the data transfer; see the Due SPI documentation page for more information.

Resources and Going Further

Tips and Tricks

- Because of the high speed signals, SPI should only be used to send data over short distances (up to a few feet). If you need to send data further than that, lower the clock speed, and consider using specialized driver chips.

- If things aren't working the way you think they should, a logic analyzer is a very helpful tool. Smart analyzers like the Saleae USB Logic Analyzer can even decode the data bytes for a display or logging.



Advantages of SPI:

- It's faster than asynchronous serial
- The receive hardware can be a simple shift register
- It supports multiple slaves

Disadvantages of SPI:

- It requires more signal lines (wires) than other communications methods
- The communications must be well-defined in advance (you can't send random amounts of data whenever you want)

- The master must control all communications (slaves can't talk directly to each other)
- It usually requires separate SS lines to each slave, which can be problematic if numerous slaves are needed.

Further Reading

Check out the Wikipedia page on SPI, which includes lots of good information on SPI and other synchronous interfaces.

A number of SparkFun products have SPI interfaces. For example, the Bar Graph Breakout kit has an easy-to-use SPI interface that you can use to turn any of 30 LEDs on or off.

Other communication options:

- Serial Communication
- Analog to Digital Conversion
- I^2C

Now that you're a pro on SPI, here are some other tutorials to practice your new skills:

- Serial 7-Segment Display Quickstart
- Elevator Tardis
- Graphic LCD Quickstart