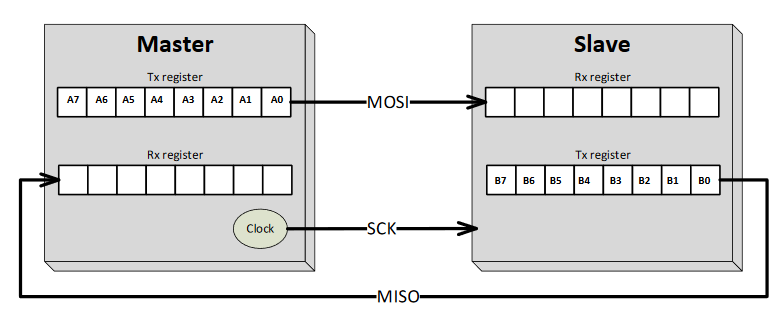
**SPI**

SPI er full duplex kommunikation mellem to devices, den ene er master, den anden er slave. Det er muligt at have flere slaves koblet til den samme master.   
SPI bruger fire signaler:

* SCK: Synkroniserer klokken mellem de to devices.
  + Man bestemmer fra masteren hvilken frekvens der køres på. Det er vigtigt at man tjekker databladet, for hvilken frekvens ens devices/periphirals kan køre.
* MOSI: Master Out Slave In
  + Linjen der sender data fra Master til Slave.
* MISO
  + Linjen der sender data fra Slave til Master.
* SS: Slave select
  + Man vælger fra masteren hvilken slave der skal kommunikeres med. Denne forbindelse er active low, det vil sige at man sætter linjen lav til den slave, der skal kommunikeres med, når der skal kommunikeres med den.

Data sendes mellem de to devices som bit swapping, som vist nedenfor: (taget fra SPI handout)



Typisk kommunikation mellem Master og Slave, vil bestå af:

* Kommando for read/write 11 fra read, 10 for write.
* Addresse på slave der skal read/write
* Data: sendes efterfølgende, ente fra slave til master eller fra master til slave.

Da det er fuld duplex, kan masteren sende en ny kommando, mens den modtager data fra slave.

SPI har 4 modes, hvor en af dem skal vælges og sættes. Tjek konfiguration for den enhed, der skal kommunikeres med, om den har en specifik setting man skal vælge. (billede fra <https://www.youtube.com/watch?v=AuhFr88mjt0>)

SPI Mode O 
SPI Mode 1 
SPI Mode 2 
SPI Mode 3 
Clock Starting 
Position 
High 
High 
Data is received 
on the 
Falling edge of 
the clock 
Rising edge 0 
the clock 
Falling edge Of 
the clock 
Rising edge of 
the clock 

Til kodning kan man tage udgangspunkt i BCM2835 library på RPI, eller et SPI library.

HAL Exercise 7 tager behandler SPI kommunikation fra RPI.

**Introduction –**

**Taget fra:** [**https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all**](https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all)

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 ( SE LINK) –**

**Besrivelser**

**-Serial Communication**

**-Binary**

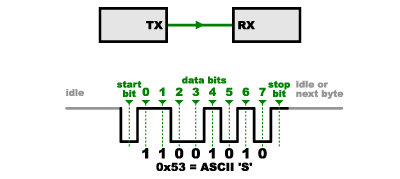
**-Shift Registers**

**-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.

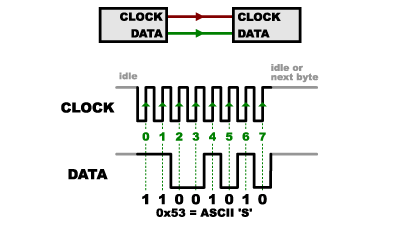
[](https://cdn.sparkfun.com/assets/f/c/6/2/4/52ddb2d5ce395f59658b4567.png)

(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).

[](https://cdn.sparkfun.com/assets/d/6/b/f/9/52ddb2d8ce395fad638b4567.png)

One reason that SPI is so popular is that the receiving hardware can be a simple [shift register](https://www.sparkfun.com/products/733). 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".

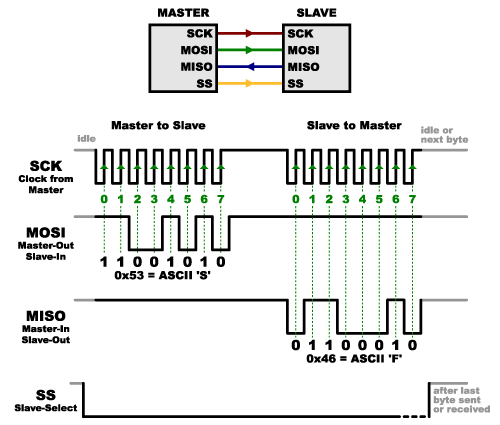
[](https://cdn.sparkfun.com/assets/f/f/e/5/8/52ddb2dbce395fae408b4568.png)

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.

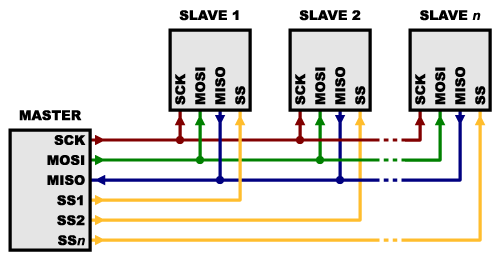
[](https://cdn.sparkfun.com/assets/c/7/8/7/d/52ddb2dcce395fed638b4567.png)

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](https://www.sparkfun.com/products/733), this corresponds to the "latch" input, which transfers the received data to the output lines.

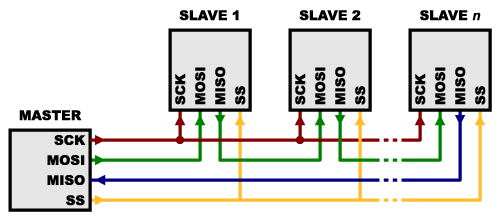
### Multiple slaves

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](https://www.sparkfun.com/products/9577) that can multiply your SS outputs.

[](https://cdn.sparkfun.com/assets/8/f/f/6/5/50e5d529ce395f2f7a000000.png)

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](https://www.sparkfun.com/products/10444).

[](https://cdn.sparkfun.com/assets/e/3/b/d/1/50e5d529ce395fd27b000001.png)

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](http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus).)

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

1. You can use the [shiftIn()](http://arduino.cc/en/Reference/ShiftIn) and [shiftOut()](http://arduino.cc/en/Reference/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](http://arduino.cc/en/Reference/SPI), 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.

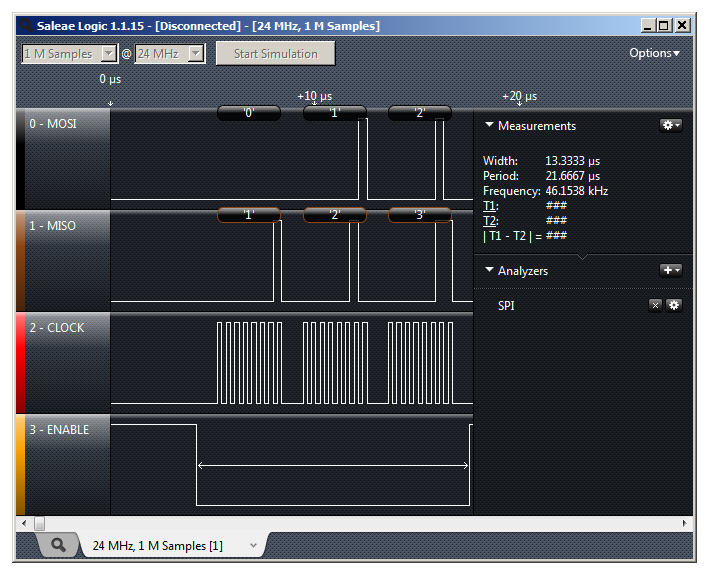
You will need to select some options when setting up your interface. These options must match those of the device you're talking to; check the device's datasheet to see what it requires.

* 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()](http://arduino.cc/en/Reference/SPISetBitOrder) 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()](http://arduino.cc/en/Reference/SPISetDataMode) 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()](http://arduino.cc/en/Reference/SPISetClockDivider) 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](http://arduino.cc/en/Reference/DueExtendedSPI) 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](http://arduino.cc/en/Reference/SPISetClockDivider), and consider using [specialized driver chips](http://www.ti.com/lit/an/slyt441/slyt441.pdf).
* 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](https://www.sparkfun.com/products/8938) can even decode the data bytes for a display or logging.

[](https://cdn.sparkfun.com/assets/a/c/6/3/8/51155d66ce395f443d000002.png)

### 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](http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus), which includes lots of good information on SPI and other synchronous interfaces.

[This page](https://www.pjrc.com/better-spi-bus-design-in-3-steps/) presents a more correct way to set up an SPI network amongst your embedded devices, particularly for use with an [Arduino](https://learn.sparkfun.com/tutorials/what-is-an-arduino) microcontroller.

A number of SparkFun products have SPI interfaces. For example, the [Bar Graph Breakout kit](https://www.sparkfun.com/products/10936) has an easy-to-use SPI interface that you can use to turn any of 30 LEDs on or off.

**Udover Disse er der links til:**

[**http://maxembedded.com/2013/11/serial-peripheral-interface-spi-basics/**](http://maxembedded.com/2013/11/serial-peripheral-interface-spi-basics/)

**Taget fra GFV**

[**https://blackboard.au.dk/bbcswebdav/pid-2294027-dt-content-rid-7002800\_1/courses/BB-Cou-UUVA-86301/SPI%281%29.pdf**](https://blackboard.au.dk/bbcswebdav/pid-2294027-dt-content-rid-7002800_1/courses/BB-Cou-UUVA-86301/SPI%281%29.pdf)

**Taget fra GFV (Dette er PDF slides.)**

**https://blackboard.au.dk/bbcswebdav/pid-2294027-dt-content-rid-7002800\_1/courses/BB-Cou-UUVA-86301/SPI%281%29.pdf**

**Samme som den overnævnte:**

[**https://blackboard.au.dk/bbcswebdav/pid-2294028-dt-content-rid-7002802\_1/courses/BB-Cou-UUVA-86301/BB-Cou-UUVA-69270\_ImportedContent\_20170609013941/SPI%20Block%20Guide%20v04.01%281%29.pdf**](https://blackboard.au.dk/bbcswebdav/pid-2294028-dt-content-rid-7002802_1/courses/BB-Cou-UUVA-86301/BB-Cou-UUVA-69270_ImportedContent_20170609013941/SPI%20Block%20Guide%20v04.01%281%29.pdf)

**Youtube film om SPI taget fra BB**

**Part 1:**

<https://www.youtube.com/watch?v=o7VLQzWfGF0>

Part 2:

<https://www.youtube.com/watch?v=aXV_EPm1TTc>

Film der introducerer BCM2835 library fra RPI, som vi muligvis kan bruge til kodning af SPI.

<https://www.youtube.com/watch?v=0QcteUtzB4o>

Klip taget fra ovenstående film:

ech 
Microcontroller 
Microcontroller 
Microcontroller 
SPI Physical Layer 
—MOSI 
CE (aka SS. aka 
aka SYNC) 
OSI 
PI Dev 
(Can have as many 
as there are 
available chip 
SPI Device 
(Single device does 
not need chip sel 
line) 
(GPIO_GCLK) 
(GPIO_GENO) 
(GPIO_GEN2) 
(GPIO_GEN3) 
(SPI_scLK) 
3V3 
toon 
1005 
GP102 
GPIOt7 
GPIOIO 
GP109 
ID SD 
2 
40W O. 1" PIN HDR 
1 oon 
1 oos 
GP1014 
GP1023 
GP1025 
GP1012 
GP1016 
t TXDO) 
(RXDO) 
(GPIO_GENI) 
(GPIO_GEN4) 
(GPIO_GENS) 
OSI 
SPI Device 
(If device does not 
send data, MISO is 
not needed) Common Steps (with BCM2835 Library) 
1. 
2. 
3. 
4. 
5. 
6. 
7. 
8. 
Import the library 
Initialize the library 
Begin the SPI 
Set the bit order 
Set the data mode 
Set the SPI clock divider 
Set the chip select 
Set the chip select polarity 
Lines of code to use with AD5592 
#include <bcm2835.h> 
if( !bcm2835_init())return 1; 
bcm2835_spi_begin(); 
spi 
spi 
_spi_ 
bcm2835 
bcm2835 
bcm2835 
bcm2835 
bcm2835 
_setBitOrder(...MSBFlRST); 
_setDataMode(...MODE1); 
setClockDivider(...16); //15 MHz 
_spi_chipSelect(...CSO); 
_spi_chipSelectPolarity(...CSO, LOW); 

Noter:

* + Det er vigtigt at tjekke datasheet for:
    - Hvilken frekvens SPI kan sættes til.
    - Hvilken størrelse ord ens device ønsker at modtage (8 bit eller 16 bit fx).
      * Hvis det er 16 skal ordet deles op i 2x8bit ord, for at blive sendt.
    - Check om device ønsker MSB eller LSB sendt først.