Table of Contents

[Explain the physical layer of the I2C protocol 2](#_Toc6591463)

[Explain the operation and frame of I2C protocol 3](#_Toc6591464)

[I2C Data Transfer 3](#_Toc6591465)

[SYNCHRONIZATION 3](#_Toc6591466)

[ARBITRATION 4](#_Toc6591467)

[CLOCK SYNCHRONIZATION AND HANDSHAKING 4](#_Toc6591468)

[COMMUNICATION WITH 7-BIT I2C ADDRESSES 4](#_Toc6591469)

[What is START bit and STOP bit? 5](#_Toc6591470)

[What is the need of repeated start condition? 6](#_Toc6591471)

[What is the standard bus speed in I2C? 6](#_Toc6591472)

[What is the limiting factor as to how many devices can go on the I²C bus? 7](#_Toc6591473)

[Is it possible to have multiple masters in I2C? 7](#_Toc6591474)

[What is a bus arbitration? 7](#_Toc6591475)

[What is I2C clock stretching? 8](#_Toc6591476)

[What is I2C clock synchronization? 9](#_Toc6591477)

[When must data be stable for a correct I²C bus transaction? 10](#_Toc6591478)

[Is Hot swapping possible in I2C protocol? 10](#_Toc6591479)

[If a slave is servicing an internal interrupt, what will it do to avoid losing data? 10](#_Toc6591480)

[Advantages of I2C communication? 10](#_Toc6591481)

[What are the limitations of I2C interface? 10](#_Toc6591482)

[What is the difference between SPI and I2C (I2C vs SPI)? 11](#_Toc6591483)

[I2C is Edge Triggering or Level Triggering? 14](#_Toc6591484)

[Is in I2c two slaves have the same address? 14](#_Toc6591485)

[What is the voltage level for 0 and 1 in I2C? 15](#_Toc6591486)

[What's Wrong with Serial Ports? 17](#_Toc6591487)

[What is A Synchronous Serial communication? 17](#_Toc6591488)

[How is Receiving Data happening on SPI? 18](#_Toc6591489)

[What is the purpose of Slave Select (SS) in SPI? 19](#_Toc6591490)

[Describe Multiple slaves configuration in SPI? 20](#_Toc6591491)

[What are the advantages of SPI? 21](#_Toc6591492)

[What are the disadvantages of SPI? 21](#_Toc6591493)

# Explain the physical layer of the I2C protocol

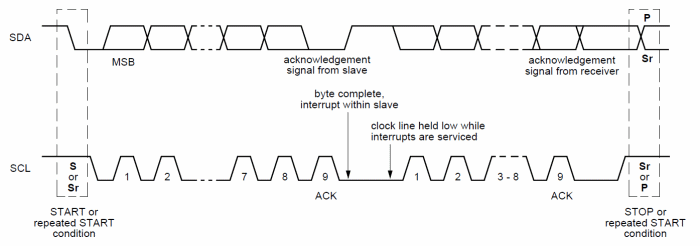
Both signals (SCL and SDA) are bidirectional. They are connected via resistors to a positive power supply voltage. This means that when the bus is free, both lines are high. All devices on the bus must have open-collector or open-drain pins. Activating the line means pulling it down ([wired AND](http://en.wikipedia.org/wiki/Wired_logic_connection)). The number of the devices on a single bus is almost unlimited – the only requirement is that the bus capacitance does not exceed 400 pF. Because logical 1 level depends on the supply voltage, there is no standard bus voltage.

**Serial Data Transfer**

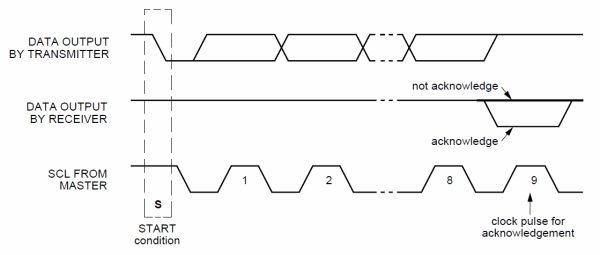
For each clock pulse one bit of data is transferred. The SDA signal can only change when the SCL signal is low – when the clock is high the data should be stable.

# Explain the operation and frame of I2C protocol

## I2C Data Transfer



Data on the I2C bus is transferred in 8-bit packets (bytes). There is no limitation on the number of bytes, however, each byte must be followed by an Acknowledge bit. This bit signals whether the device is ready to proceed with the next byte. For all data bits including the Acknowledge bit, the master must generate clock pulses. If the slave device does not acknowledges transfer this means that there is no more data or the device is not ready for the transfer yet. The master device must either generate Stop or Repeated Start condition.



## SYNCHRONIZATION

Each master must generate its own clock signal and the data can change only when the clock is low. For successful bus arbitration a synchronized clock is needed. Once a master pulls the clock low it stays low until all masters put the clock into high state. Similarly, the clock is in the high state until the first master pulls it low. This way by observing the SCL signal, master devices can synchronize their clocks.

## ARBITRATION

For normal data transfer on the I2C bus only one master can be active. If for some reason two masters initiate I2C command at the same time, the arbitration procedure determines which master wins and can continue with the command. Arbitration is performed on the SDA signal while the SCL signal is high. Each master checks if the SDA signal on the bus corresponds to the generated SDA signal. If the SDA signal on the bus is low but it should be high, then this master has lost arbitration. Master I2C device that has lost arbitration can generate SCL pulses until the byte ends and must then release the bus and go into slave mode. The arbitration procedure can continue until all the data is transferred. This means that in multi-master system each I2C master must monitor the I2C bus for collisions and act accordingly.

## CLOCK SYNCHRONIZATION AND HANDSHAKING

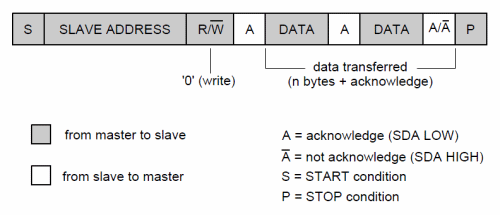
Slave devices that need some time to process received byte or are not ready yet to send the next byte, can pull the clock low to signal to the master that it should wait. Once the clock is released the master can proceed with the next byte.

## COMMUNICATION WITH 7-BIT I2C ADDRESSES

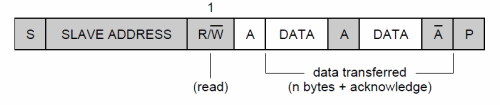


Each slave device on the bus should have a unique 7-bit address. The communication starts with the Start condition, followed by the 7-bit slave address and the data direction bit. If this bit is 0 then the master will write to the slave device. Otherwise, if the data direction bit is 1, the master will read from slave device. After the slave address and the data direction is sent, the master can continue with reading or writing. The communication is ended with the Stop condition which also signals that the I2C bus is free. If the master needs to communicate with other slaves it can generate a repeated start with another slave address without generation Stop condition. All the bytes are transferred with the MSB bit shifted first.

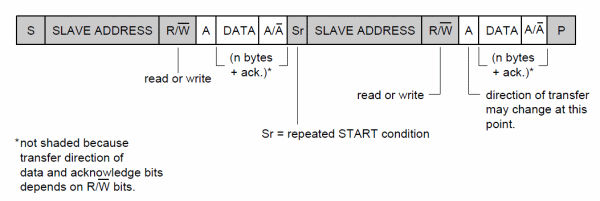
If the master only writes to the slave device then the data transfer direction is not changed.



If the master only needs to read from the slave device then it simply sends the I2C address with the R/W bit set to read. After this the master device starts reading the data.



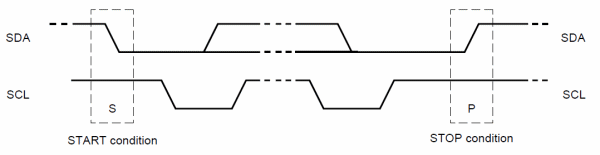
Sometimes the master needs to write some data and then read from the slave device. In such cases it must first write to the slave device, change the data transfer direction and then read the device. This means sending the I2C address with the R/W bit set to write and then sending some additional data like register address. After writing is finished the master device generates repeated start condition and sends the I2C address with the R/W bit set to read. After this the data transfer direction is changed and the master device starts reading the data.



# What is START bit and STOP bit?

Start and Stop Condition

Each I2C command initiated by master device starts with a **START condition** and ends with a **STOP condition**. For both conditions SCL has to be high. A high to low transition of SDA is considered as **START** and a low to high transition as **STOP**.



After the Start condition the bus is considered as busy and can be used by another master only after a Stop condition is detected. After the Start condition the master can generate a repeated Start. This is equivalent to a normal Start and is usually followed by the slave I2C address.

Microcontrollers that have dedicated I2C hardware can easily detect bus changes and behave also as I2C slave devices. However, if the I2C communication is implemented in software, the bus signals must be sampled at least two times per clock cycle in order to detect necessary changes.

**Note: A START and STOP condition always asserted by the master.**

# What is the need of repeated start condition?

A Repeated Start condition is asserted by the master when he does not want to lose their control from the bus. The repeated start is beneficial for the master when it wants to start a new communication without the asserting the stop condition.

**Note: Repeated start is beneficial when more than one master connected with the I2c Bus.**

# What is the standard bus speed in I2C?

There are following speed mode in I2C

|  |  |
| --- | --- |
| MODE | SPEED |
| Standard-mode | 100 kbit/s |
| Fast-mode | 400 kbit/s |
| Fast-mode Plus | 1 Mbit/s |
| High-speed mode | 3.4 Mbit/s |

# What is the limiting factor as to how many devices can go on the I²C bus?

It depends on the total capacitance.

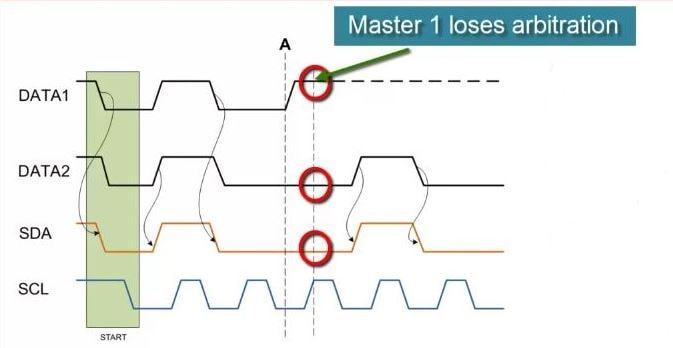
# Is it possible to have multiple masters in I2C?

Yes I2C support multiple master and multiple slaves.

# What is a bus arbitration?

The arbitration is required in case of multi-master, where more than one master is tried to communicate with a slave simultaneously. In I2C arbitration is achieved by the SDA line.

**Example,**  
Suppose two masters in the [**I2C bus**](https://aticleworld.com/i2c-bus-protocol-and-interface/) is tried to communicate with a slave simultaneously then they will assert a start condition on the bus. The SCL clock of the I2c bus would be already synchronized by the wired and logic.

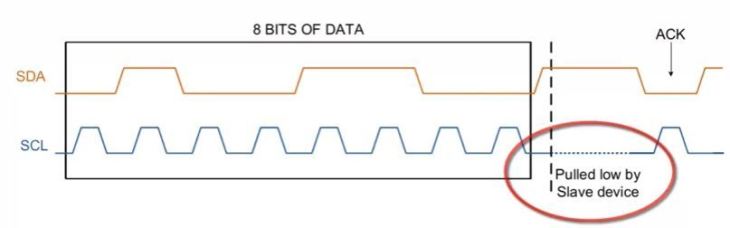


In the above case, everything will be good till the state of SDA line will same what is the masters driving on the bus. If any master sees that the state of SDA line differs, what is it driving then they will exit from the communication and lose their arbitration.

***Note: Master which is losing their arbitration will wait till the bus become free.***

# What is I2C clock stretching?

In I2c, communication can be paused by the clock stretching to holding the SCL line low and it cannot continue until the SCL line released high again.



In I2C, slave able to receive a byte of data on the fast rate but sometimes slave takes more time in processing the received bytes in that situation slave pull the SCL line to pause the transaction and after the processing of the received bytes, it again released the SCL line high again to resume the communication.

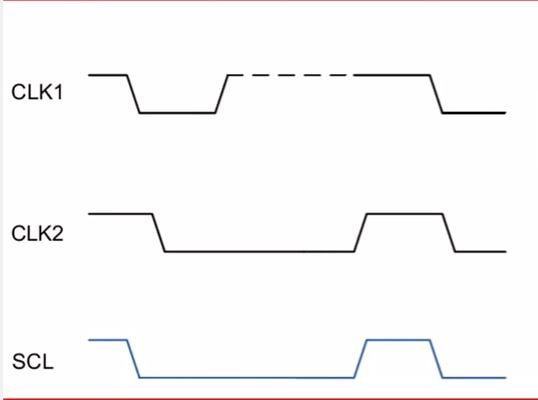
The clock stretching is the way in which slave drive the SCL line but it is the fact, most of the slave does not drive the SCL line

***Note:****In I2c communication protocol, most of the I2C slave devices do not use the clock stretching feature, but every master should support the clock stretching.*

# What is I2C clock synchronization?

Unlike Rs232, I2c is synchronous communication, in which clock is always generated by the master and this clock is shared by both master and slave. In the case of multi-master, all master generate their own SCL clock, hence it is necessary that clock of all master should be synchronized. In the i2C, this clock synchronization is done by wired and logic.

For a better understanding, I am taking an example, where two masters try to communicate with a slave. In that situation, both masters generate their own clock, master M1 generate clk1 and master M2 generate clk2 and clock which observed on the bus is SCL.



The SCL clock would be the Anding (clk1 & clk2) of clk1 and clk2 and most interesting thing is that highest logic 1 of SCL line defines by the CLK which has lowest logic 1.

# When must data be stable for a correct I²C bus transaction?

When the clock is high

# Is Hot swapping possible in I2C protocol?

Yes, hot swapping is possible in I2C.

# If a slave is servicing an internal interrupt, what will it do to avoid losing data?

The slave will stretch the clock until the interrupt servicing is complete.

# Advantages of I2C communication?

There is a lot of advantage of I2C protocol which makes the user helpless to use the I2C protocol in many applications.

* It is the synchronous communication protocol, so there is no need of a precise oscillator for the master and slave.
* It requires only two wire, one wire for the data (SDA) and other wire for the clock (SCL).
* It provides the flexibility to the user to select the transmission rate as per the requirements.
* In I2C Bus, each device on the bus is independently addressable.
* It follows the master and slave relationships.
* It has the capability to handle the multiple masters and multiple slaves on the I2C Bus.
* I2C has some important features like arbitration, clock synchronization, and clock stretching.
* I2C provide ACK/NACK (acknowledgment/ Not-acknowledgement) features which provide help in error handling.

# What are the limitations of I2C interface?

* Half duplex communication, so data is transmitted only in one direction (because of the single data bus) at a time.
* Since the bus is shared by many devices, debugging an I2C bus (detecting which device is misbehaving) for issues is pretty difficult.
* The I2C bus is shared by multiple slave devices if anyone of these slaves misbehaves (pull either SCL or SDA low for an indefinite time) the bus will be stalled. No further communication will take place.
* I2C uses resistive pull-up for its bus. Limiting the bus speed.
* Bus speed is directly dependent on the bus capacitance, meaning longer I2C bus traces will limit the bus speed.

# What is the difference between SPI and I2C (I2C vs SPI)?

|  |  |
| --- | --- |
| **I2C** | **SPI** |
| I2C can be multi-master and multi-slave, which means there can be more than one master and slave attached to the I2C bus | SPI can be multi-save but does not a multi-master serial protocol, that means there can be only one master attached to SPI bus. |
| I2C is half-duplex communication protocol. | SPI is a full duplex commination protocol. |
| I2C has the feature of clock stretching, that means if the slave cannot able to send fast data as fast enough then it suppresses the clock to stop the communication. | Clock stretching is not the feature of SPI. |
| I2C is used only two wire for the communication, one wire is used for the data and the second wire is used for the clock. | SPI needs three or four wire for communication ((depends on requirement), MOSI, MISO, SCL and Chip-select pin. |
| I2C is slower than SPI. | In comparison to I2C, SPI is faster. |
| I2C draws more power than SPI. | Draws less power as compared to I2C. |
| I2C is less susceptible to noise than SPI | SPI is more susceptible to noise than I2C. |
| I2C is cheaper to implement than the SPI communication protocol. | Costly as compare to I2C. |
| I2C work on wire and logic and it has a pull-up resistor. | There is no requirement of pull-up resistor in case of the SPI. |
| In I2C communication we get the acknowledgment bit after each byte. | Acknowledgment bit is not supported by the SPI communication protocol. |
| I2C ensures that data sent is received by the slave device. | SPI does not verify that data is received correctly or not. |
| I2C support the multi-master communication. | SPI does not support multi -master communication. |
| I2C is a multi-master communication protocol that’s why it has the feature of arbitration. | SPI is not a multi-master communication protocol, so it does not consist the properties of arbitration. |
| I2C is the address base bus protocol, you have to send the address of the slave for the communication. | In case of the SPI, you have to select the slave using the slave select pin for the communication. |
| I2C has some extra overhead due to start and stop bits. | SPI does not have a start and stop bits. |
| I2C supports multiple devices on the same bus without any additional select lines (work on the basis of device address). | SPI requires additional signal (slave select lines) lines to manage multiple devices on the same bus. |
| I2C is better for long distance. | SPI is better for the short distance. |
| I2C is developed by NXP. | SPI is developed by Motorola. |

# I2C is Edge Triggering or Level Triggering?

Edge triggered. circuit becomes active at negative or positive edge of the clock signal which is known as positive and negative edge triggering respectively

# Is in I2c two slaves have the same address?

Theoretically, there will be conflict.

Master will put data on bus but which slave (with same addresses) going to receive it can not be determined.

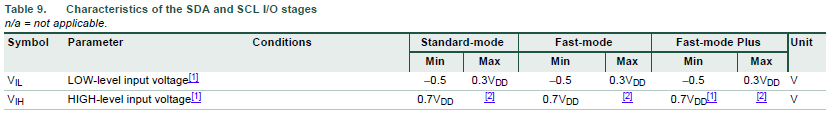
Once i faced same problem. I was interfacing two temperature sensors with same I2C address. So what i did is , i introduced a another GPIO pin to control Vcc of temp sensor such that when one sensor is ON other should be OFF (NOTed them). When i want to read first temp sensor value then the GPIO pin should be high. So first sensor will power ON and other will be OFF.

You can go for your way if you have multiple devices. You can use multiplexers .

# What is the voltage level for 0 and 1 in I2C?

"Due to the variety of different technology devices (CMOS, NMOS, bipolar) that can be connected to the I2C-bus, the levels of the logical ‘0’ (LOW) and ‘1’ (HIGH) are not fixed and depend on the associated level of VDD. Input reference levels are set as 30 % and 70 % of VDD; VIL is 0.3VDD and VIH is 0.7VDD. See Figure 38, timing diagram. Some legacy device input levels were fixed at VIL = 1.5 V and VIH = 3.0 V, but all new devices require this 30 %/70 % specification. See Section 6 for electrical specifications." *(page 9)*

Deeper in the spec, you'll see that this 0.7×VDD0.7×VDD is the *minimum* logic high voltage:



For your 5V system:

0.7×5V=3.5V0.7×5V=3.5V

0.3×5V=1.5V0.3×5V=1.5V

To me, the 3.3 V pull-up looks marginal, especially if any of your 5V devices use the 'new' standard of 0.7×VDD0.7×VDD for logic HIGH.

The I2C standard limits the maximum allowed capacitance on the bus to 400 pF for I2C fast mode (FM) and 550 pF for I2C fast mode plus (FM+). With ever-growing system complexity more and more integrated circuits (IC’s) are added to the I2C bus and complying with the I2C spec capacitance limit has become a concern. Each IC added results in a capacitance increase up to approximately 15 pF on the I2C bus. I2C repeaters are circuits which provide a solution to the previously described problem by isolating the capacitance between two I2C bus, hence, allowing greater capacitance on an I2C bus for a given timing budget.

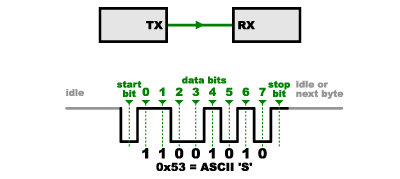
The maximum clock frequency (fSCL (max)) is specified to be up to 400 kHz for I2C FM and up to 1000 kHz for FM+ spec. With the increasing number of devices, application requirements also tend to dictate faster operating frequencies to improve overall system response time. Since I2C repeaters typically buffer both the clock (SCL) and the data (SDA) lines, an I2C system utilizing I2C repeaters must properly account for the propagation delays through the repeater when determining the optimal operating frequency.



# 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.

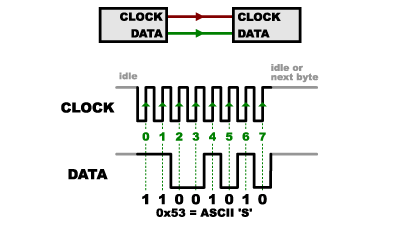
[](https://cdn.sparkfun.com/assets/f/c/6/2/4/52ddb2d5ce395f59658b4567.png)

(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 nibble is 0011 = 0x3, and the upper nibble 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.

# What is A Synchronous Serial communication?

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.

# How is Receiving Data happening on SPI?

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".

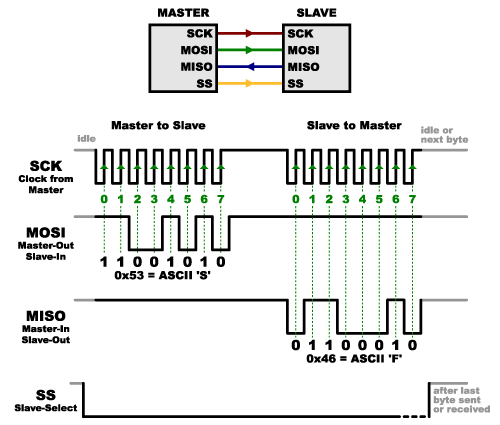
[](https://cdn.sparkfun.com/assets/f/f/e/5/8/52ddb2dbce395fae408b4568.png)

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.

# What is the purpose of Slave Select (SS) in SPI?

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.

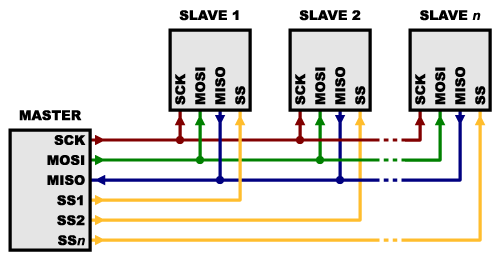
[](https://cdn.sparkfun.com/assets/c/7/8/7/d/52ddb2dcce395fed638b4567.png)

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.

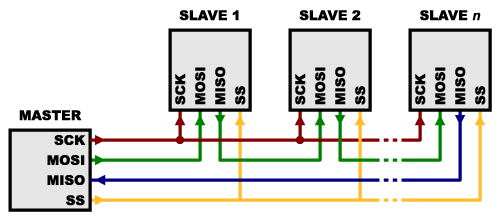
# Describe Multiple slaves configuration in SPI?

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.



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).



# What are the advantages of SPI?

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

# What are the 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.