

Communication Protocols

Communication protocols in embedded systems are **structured sets of rules** that govern data exchange between devices and components. They ensure accurate, efficient, and reliable information transmission by establishing:

- A common language for device interaction
- Data formatting standards
- Transmission timing and sequencing
- Error detection and correction mechanisms

Key Characteristics of Communication Protocols

Communication Protocol Attributes

1. Communication Types

- **Synchronous:** Devices share a common clock
- **Asynchronous:** No shared clock signal
- **Serial:** Bits transferred one by one
- **Parallel:** Multiple bits transferred simultaneously
- **Duplex Modes:**
 - **Full-Duplex:** Simultaneous two-way communication
 - **Half-Duplex:** One-directional communication at a time
-

Major Communication Protocols

1. I2C (Inter-Integrated Circuit)

Key Features:

- Short-distance communication
- Two-wire interface
- **Components:**
 - Serial Data Line (SDA)
 - Serial Clock Line (SCL)

-
- Uses addressing system (7 or 10-bit)
- Suitable for connecting microcontrollers with peripheral devices

2. SPI (Serial Peripheral Interface)

Key Features:

- High-speed synchronous communication
- Four-wire interface:
 - MISO (Master In Slave Out)
 - MOSI (Master Out Slave In)
 - SCLK (Serial Clock)
 - SS (Slave Select)
-
- Full-duplex communication
- Ideal for rapid data transfer with sensors and memory devices

3. UART (Universal Asynchronous Receiver/Transmitter)

Key Features:

- Simple point-to-point communication
- Asynchronous data transmission
- Used for one-on-one device communication
- Commonly used for debugging and logging

4. CAN (Controller Area Network)

Key Features:

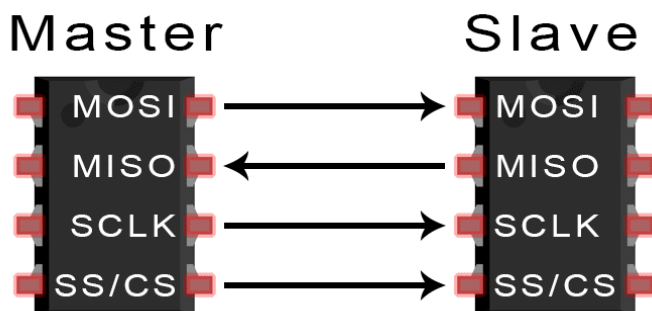
- Robust communication protocol
- Widely used in automotive and industrial systems
- Supports multiple devices on single network
- High reliability in challenging environments
- Message priority handling

5. Ethernet

Key Features:

- Network-based communication
- High-speed data transfer
- Supports multiple devices
- Enables network connectivity for embedded systems

Basics of the SPI Communication Protocol



Communication between electronic devices is like communication between humans. Both sides need to speak the same language. In electronics, these languages are called *communication protocols*. Luckily for us, there are only a few communication protocols we need to know when building most DIY electronics projects.

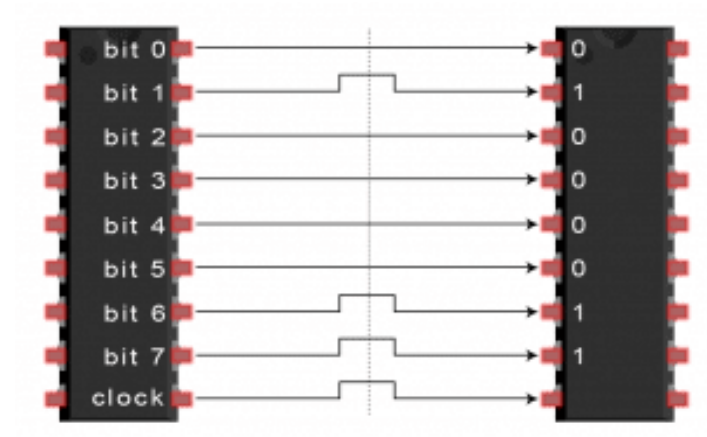
SPI, I2C, and UART are quite a bit slower than protocols like USB, ethernet, Bluetooth, and WiFi, but they're a lot more simple and use less hardware and system resources. SPI, I2C, and UART are ideal for communication between microcontrollers and between

microcontrollers and sensors where large amounts of high speed data don't need to be transferred.

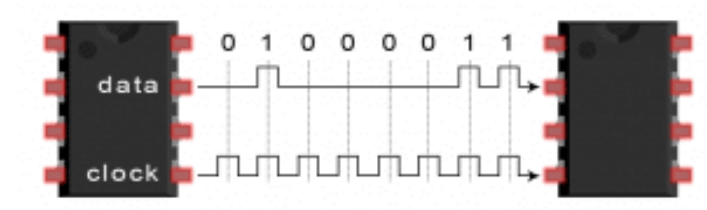
Serial vs. Parallel Communication

Electronic devices talk to each other by sending *bits* of data through wires physically connected between devices. A bit is like a letter in a word, except instead of the 26 letters (in the English alphabet), a bit is binary and can only be a 1 or 0. Bits are transferred from one device to another by quick changes in voltage. In a system operating at 5 V, a 0 bit is communicated as a short pulse of 0 V, and a 1 bit is communicated by a short pulse of 5 V.

The bits of data can be transmitted either in parallel or serial form. In parallel communication, the bits of data are sent all at the same time, each through a separate wire. The following diagram shows the parallel transmission of the letter "C" in binary (01000011):



In serial communication, the bits are sent one by one through a single wire. The following diagram shows the serial transmission of the letter "C" in binary (01000011):

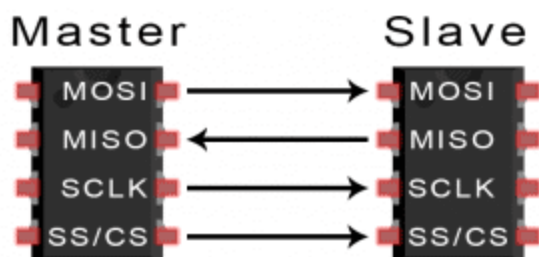


Introduction to SPI Communication

SPI is a common communication protocol used by many different devices. For example, SD card reader modules, RFID card reader modules, and 2.4 GHz wireless transmitter/receivers all use SPI to communicate with microcontrollers.

One unique benefit of SPI is the fact that data can be transferred without interruption. Any number of bits can be sent or received in a continuous stream. With I2C and UART, data is sent in packets, limited to a specific number of bits. Start and stop conditions define the beginning and end of each packet, so the data is interrupted during transmission.

Devices communicating via SPI are in a master-slave relationship. The master is the controlling device (usually a microcontroller), while the slave (usually a sensor, display, or memory chip) takes instruction from the master. The simplest configuration of SPI is a single master, single slave system, but one master can control more than one slave (more on this below).



MOSI (Master Output/Slave Input) – Line for the master to send data to the slave.

MISO (Master Input/Slave Output) – Line for the slave to send data to the master.

SCLK (Clock) – Line for the clock signal.

SS/CS (Slave Select/Chip Select) – Line for the master to select which slave to send data to.

Wires Used	4
Maximum Speed	Up to 10 Mbps
Synchronous or Asynchronous?	Synchronous
Serial or Parallel?	Serial
Max # of Masters	1
Max # of Slaves	Theoretically unlimited*

*In practice, the number of slaves is limited by the load capacitance of the system, which reduces the ability of the master to accurately switch between voltage levels.

How SPI Works

The Clock

The clock signal synchronizes the output of data bits from the master to the sampling of bits by the slave. One bit of data is transferred in each clock cycle, so the speed of data transfer is determined by the frequency of the clock signal. SPI communication is always initiated by the master since the master configures and generates the clock signal.

Any communication protocol where devices share a clock signal is known as *synchronous*. SPI is a synchronous communication protocol. There are also *asynchronous* methods that don't use a clock signal. For example, in UART communication, both sides are set to a pre-configured baud rate that dictates the speed and timing of data transmission.

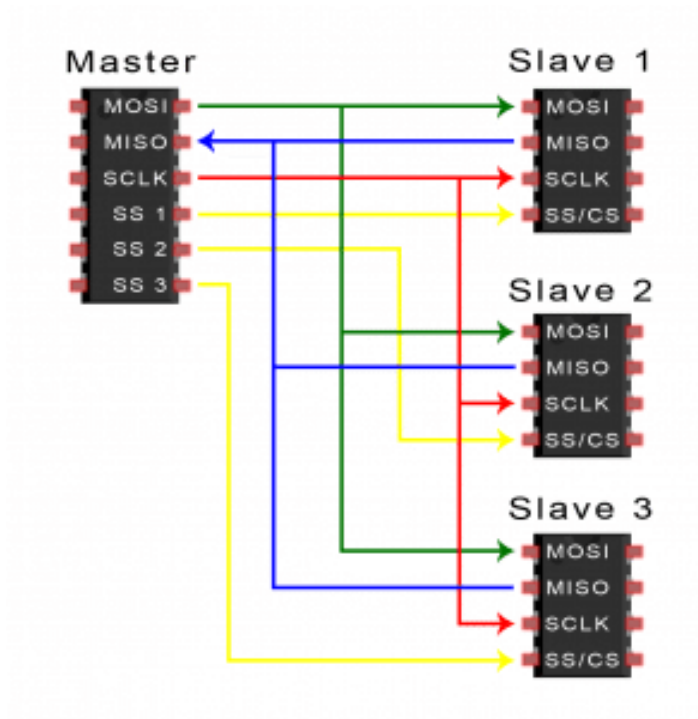
The clock signal in SPI can be modified using the properties of *clock polarity* and *clock phase*. These two properties work together to define when the bits are output and when they are sampled. Clock polarity can be set by the master to allow for bits to be output and sampled on either the rising or falling edge of the clock cycle. Clock phase can be set for output and sampling to occur on either the first edge or second edge of the clock cycle, regardless of whether it is rising or falling.

Slave Select

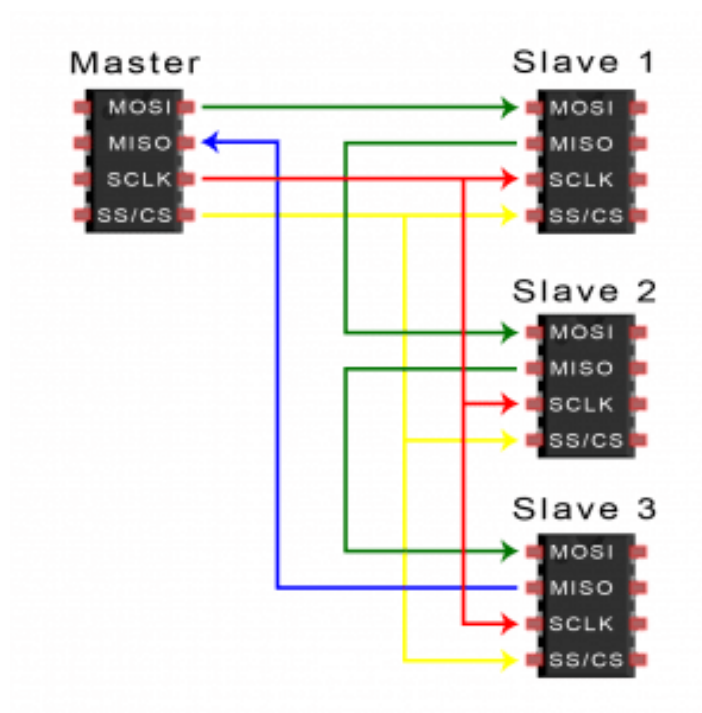
The master can choose which slave it wants to talk to by setting the slave's CS/SS line to a low voltage level. In the idle, non-transmitting state, the slave select line is kept at a high voltage level. Multiple CS/SS pins may be available on the master, which allows for multiple slaves to be wired in parallel. If only one CS/SS pin is present, multiple slaves can be wired to the master by daisy-chaining.

Multiple Slaves

SPI can be set up to operate with a single master and a single slave, and it can be set up with multiple slaves controlled by a single master. There are two ways to connect multiple slaves to the master. If the master has multiple slave select pins, the slaves can be wired in parallel like this:



If only one slave select pin is available, the slaves can be daisy-chained like this:



MOSI and MISO

The master sends data to the slave bit by bit, in serial through the MOSI line. The slave receives the data sent from the master at the MOSI pin. Data sent from the master to the slave is usually sent with the most significant bit first.

The slave can also send data back to the master through the MISO line in serial. The data sent from the slave back to the master is usually sent with the least significant bit first.

Steps of SPI Data Transmission

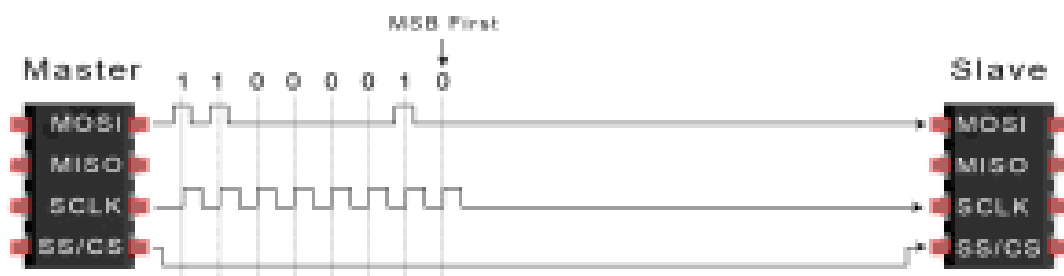
1. The master outputs the clock signal:



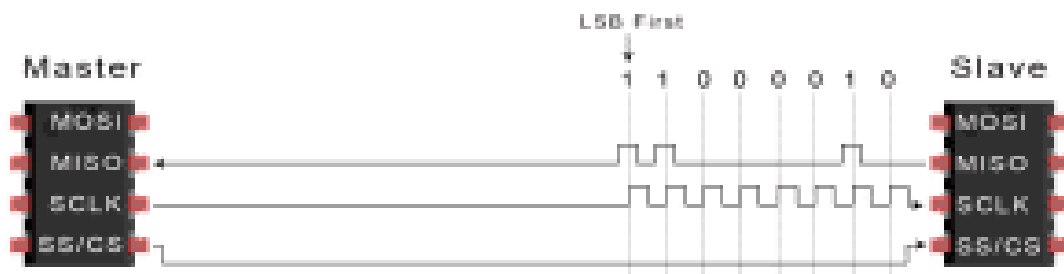
2. The master switches the SS/CS pin to a low voltage state, which activates the slave:



3. The master sends the data one bit at a time to the slave along the MOSI line. The slave reads the bits as they are received:



4. If a response is needed, the slave returns data one bit at a time to the master along the MISO line. The master reads the bits as they are received:



Advantages and Disadvantages of SPI

There are some advantages and disadvantages to using SPI, and if given the choice between different communication protocols, you should know when to use SPI according to the requirements of your project:

Advantages

- No start and stop bits, so the data can be streamed continuously without interruption
- No complicated slave addressing system like I2C

- Higher data transfer rate than I2C (almost twice as fast)
- Separate MISO and MOSI lines, so data can be sent and received at the same time

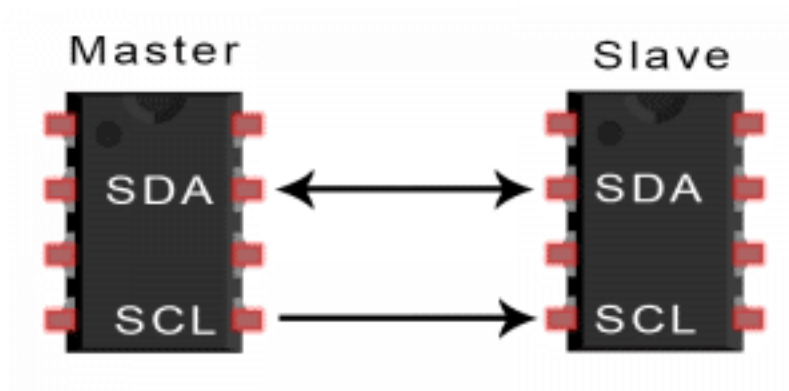
Disadvantages

- Uses four wires (I2C and UARTs use two)
- No acknowledgement that the data has been successfully received (I2C has this)
- No form of error checking like the parity bit in UART
- Only allows for a single master

Introduction to I2C Protocol

I2C combines the best features of SPI and UARTs. With I2C, you can connect multiple slaves to a single master (like SPI) and you can have multiple masters controlling single, or multiple slaves. This is really useful when you want to have more than one microcontroller logging data to a single memory card or displaying text to a single LCD.

Like UART communication, I2C only uses two wires to transmit data between devices:



SDA (Serial Data) – The line for the master and slave to send and receive data.

SCL (Serial Clock) – The line that carries the clock signal.

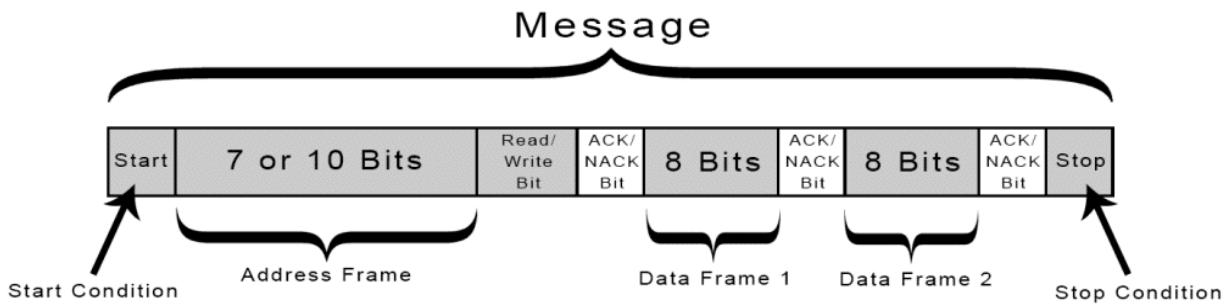
I2C is a serial communication protocol, so data is transferred bit by bit along a single wire (the SDA line).

Like SPI, I2C is synchronous, so the output of bits is synchronized to the sampling of bits by a clock signal shared between the master and the slave. The clock signal is always controlled by the master.

Wires Used	2
Maximum Speed	Standard mode= 100 kbps
	Fast mode= 400 kbps
	High speed mode= 3.4 Mbps
	Ultra fast mode= 5 Mbps
Synchronous or Asynchronous?	Synchronous
Serial or Parallel?	Serial
Max # of Masters	Unlimited
Max # of Slaves	1008

How I2C Works

With I2C, data is transferred in *messages*. Messages are broken up into *frames* of data. Each message has an address frame that contains the binary address of the slave, and one or more data frames that contain the data being transmitted. The message also includes start and stop conditions, read/write bits, and ACK/NACK bits between each data frame:



Start Condition: The SDA line switches from a high voltage level to a low voltage level *before* the SCL line switches from high to low.

Stop Condition: The SDA line switches from a low voltage level to a high voltage level *after* the SCL line switches from low to high.

Address Frame: A 7 or 10 bit sequence unique to each slave that identifies the slave when the master wants to talk to it.

Read/Write Bit: A single bit specifying whether the master is sending data to the slave (low voltage level) or requesting data from it (high voltage level).

ACK/NACK Bit: Each frame in a message is followed by an acknowledge/no-acknowledge bit. If an address frame or data frame was successfully received, an ACK bit is returned to the sender from the receiving device.

Addressing

I2C doesn't have slave select lines like SPI, so it needs another way to let the slave know that data is being sent to it, and not another slave. It does this by *addressing*. The address frame is always the first frame after the start bit in a new message.

The master sends the address of the slave it wants to communicate with to every slave connected to it. Each slave then compares the address sent from the master to its own address. If the address matches, it sends a low voltage ACK bit back to the master. If the address doesn't match, the slave does nothing and the SDA line remains high.

Read/Write Bit

The address frame includes a single bit at the end that informs the slave whether the master wants to write data to it or receive data from it. If the master wants to send data to the slave, the read/write bit is a low voltage level. If the master is requesting data from the slave, the bit is a high voltage level.

The Data Frame

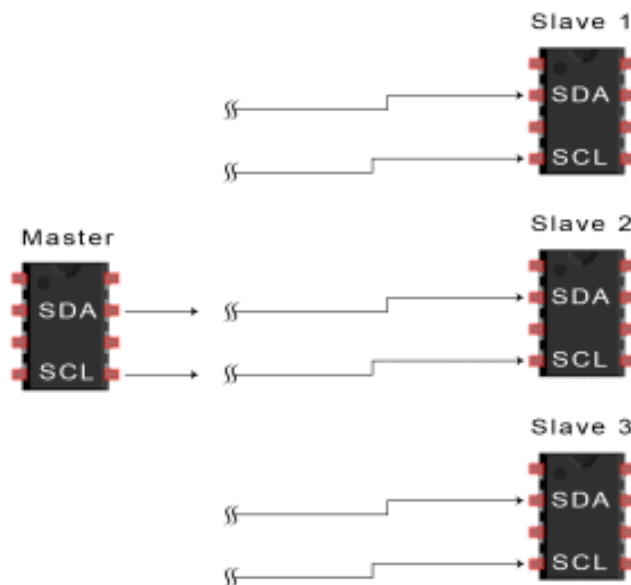
After the master detects the ACK bit from the slave, the first data frame is ready to be sent.

The data frame is always 8 bits long, and sent with the most significant bit first. Each data frame is immediately followed by an ACK/NACK bit to verify that the frame has been received successfully. The ACK bit must be received by either the master or the slave (depending on who is sending the data) before the next data frame can be sent.

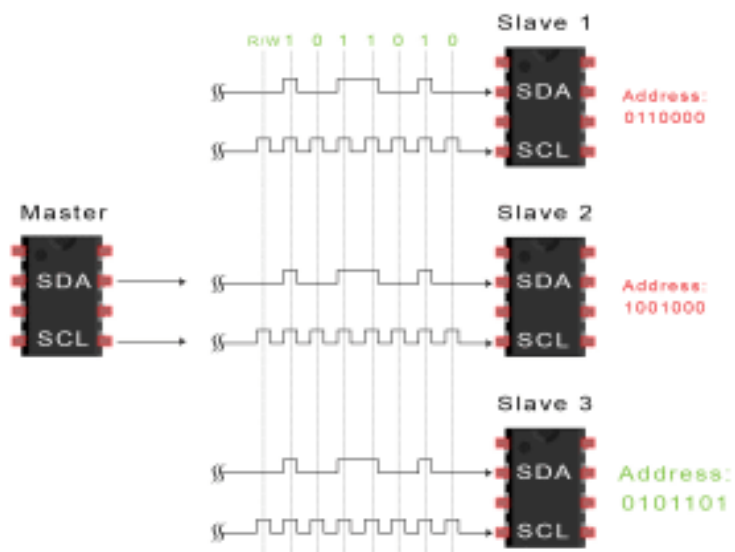
After all of the data frames have been sent, the master can send a stop condition to the slave to halt the transmission. The stop condition is a voltage transition from low to high on the SDA line after a low to high transition on the SCL line, with the SCL line remaining high.

Steps of I2C Data Transmission

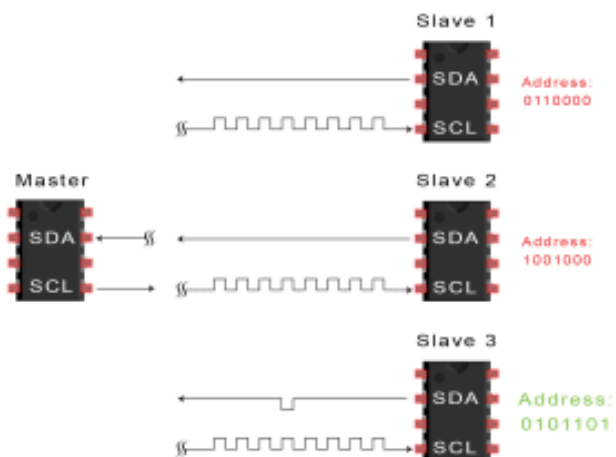
1. The master sends the start condition to every connected slave by switching the SDA line from a high voltage level to a low voltage level *before* switching the SCL line from high to low:



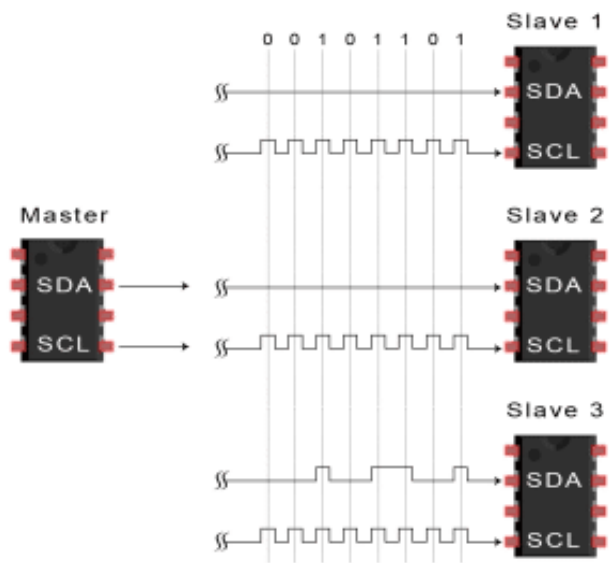
2. The master sends each slave the 7 or 10 bit address of the slave it wants to communicate with, along with the read/write bit:



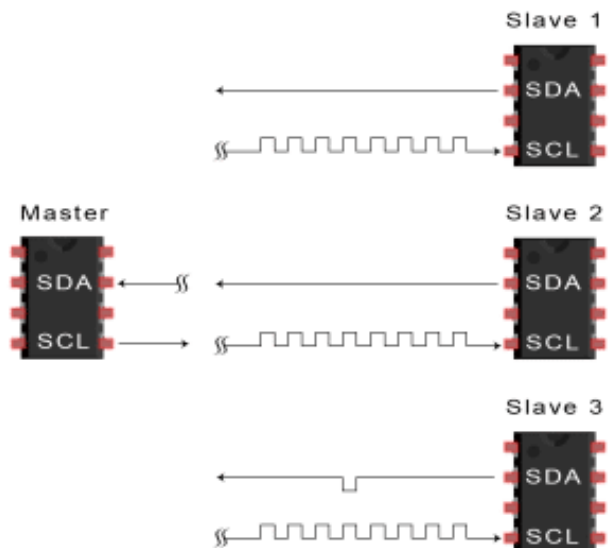
3. Each slave compares the address sent from the master to its own address. If the address matches, the slave returns an ACK bit by pulling the SDA line low for one bit. If the address from the master does not match the slave's own address, the slave leaves the SDA line high.



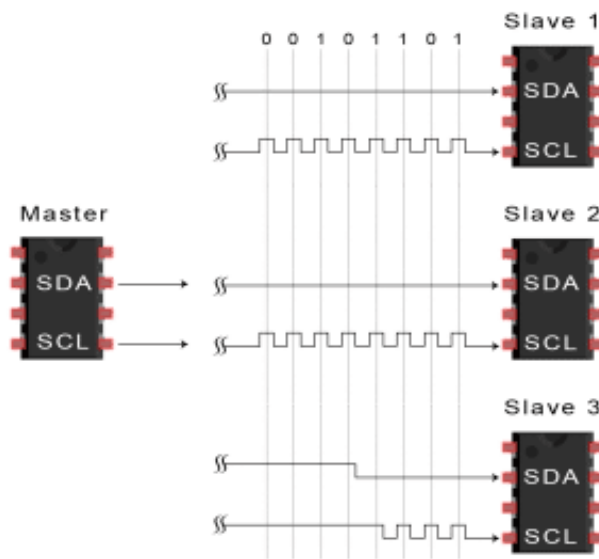
4. The master sends or receives the data frame:



5. After each data frame has been transferred, the receiving device returns another ACK bit to the sender to acknowledge successful receipt of the frame:

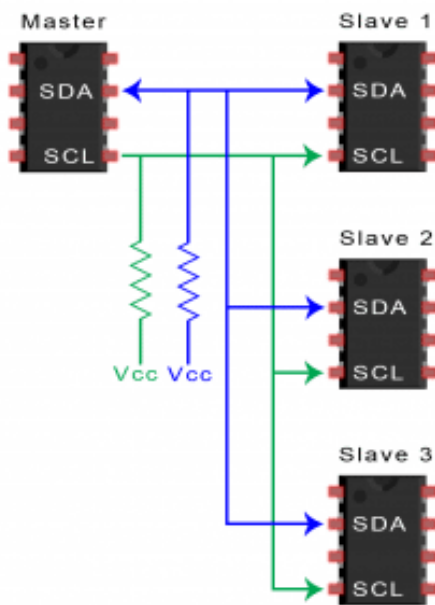


6. To stop the data transmission, the master sends a stop condition to the slave by switching SCL high before switching SDA high:



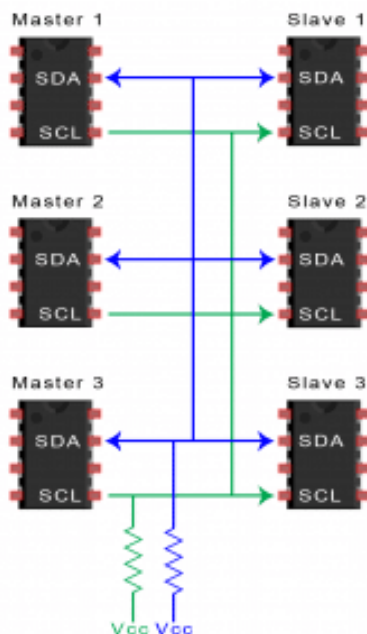
Single Master with Multiple Slaves

Because I2C uses addressing, multiple slaves can be controlled from a single master. With a 7 bit address, 128 (2^7) unique addresses are available. Using 10 bit addresses is uncommon, but provides 1,024 (2^{10}) unique addresses. To connect multiple slaves to a single master, wire them like this, with 4.7K Ohm pull-up resistors connecting the SDA and SCL lines to V_{cc} :



Multiple Masters with Multiple Slaves

Multiple masters can be connected to a single slave or multiple slaves. The problem with multiple masters in the same system comes when two masters try to send or receive data at the same time over the SDA line. To solve this problem, each master needs to detect if the SDA line is low or high before transmitting a message. If the SDA line is low, this means that another master has control of the bus, and the master should wait to send the message. If the SDA line is high, then it's safe to transmit the message. To connect multiple masters to multiple slaves, use the following diagram, with 4.7K Ohm pull-up resistors connecting the SDA and SCL lines to Vcc:



Advantages and Disadvantages of I2C

There is a lot to I2C that might make it sound complicated compared to other protocols, but there are some good reasons why you may or may not want to use I2C to connect to a particular device:

Advantages

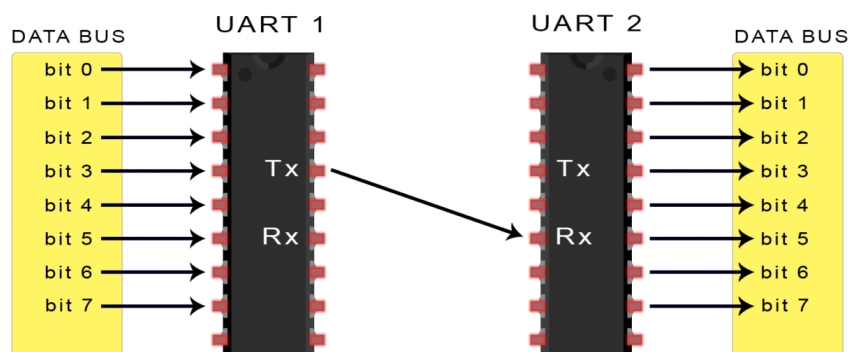
- Only uses two wires
- Supports multiple masters and multiple slaves
- ACK/NACK bit gives confirmation that each frame is transferred successfully
- Hardware is less complicated than with UARTs

- Well known and widely used protocol

Disadvantages

- Slower data transfer rate than SPI
- The size of the data frame is limited to 8 bits
- More complicated hardware needed to implement than SPI

Basics of UART Communication



Remember when printers, mice, and modems had thick cables with those huge clunky connectors? The ones that literally had to be screwed into your computer? Those devices were probably using UARTs to communicate with your computer. While USB has almost completely replaced those old cables and connectors, UARTs are definitely not a thing of the past. You'll find UARTs being used in many DIY electronics projects to connect GPS

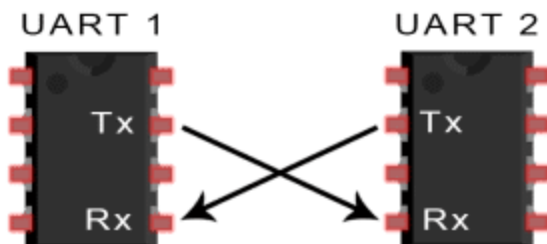
modules, Bluetooth modules, and RFID card reader modules to your Raspberry Pi, Arduino, or other microcontrollers.

UART stands for Universal Asynchronous Receiver/Transmitter. It's not a communication protocol like SPI and I2C, but a physical circuit in a microcontroller, or a stand-alone IC. A UART's main purpose is to transmit and receive serial data.

One of the best things about UART is that it only uses two wires to transmit data between devices.

Introduction to UART Communication

In UART communication, two UARTs communicate directly with each other. The transmitting UART converts parallel data from a controlling device like a CPU into serial form, transmits it in serial to the receiving UART, which then converts the serial data back into parallel data for the receiving device. Only two wires are needed to transmit data between two UARTs. Data flows from the Tx pin of the transmitting UART to the Rx pin of the receiving UART:



UARTs transmit data *asynchronously*, which means there is no clock signal to synchronize the output of bits from the transmitting UART to the sampling of bits by the receiving UART. Instead of a clock signal, the transmitting UART adds start and stop bits to the data packet being transferred. These bits define the beginning and end of the data packet so the receiving UART knows when to start reading the bits.

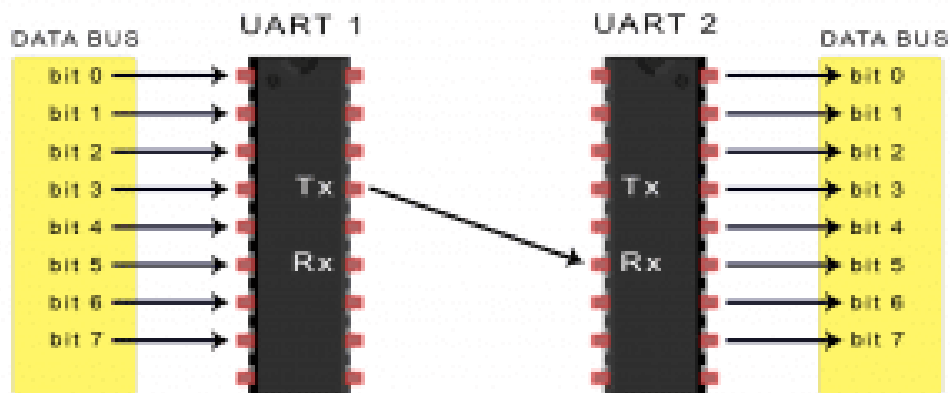
When the receiving UART detects a start bit, it starts to read the incoming bits at a specific frequency known as the *baud rate*. Baud rate is a measure of the speed of data transfer, expressed in bits per second (bps). Both UARTs must operate at about the same baud rate. The baud rate between the transmitting and receiving UARTs can only differ by about 10% before the timing of bits gets too far off.

Both UARTs must also must be configured to transmit and receive the same data packet structure.

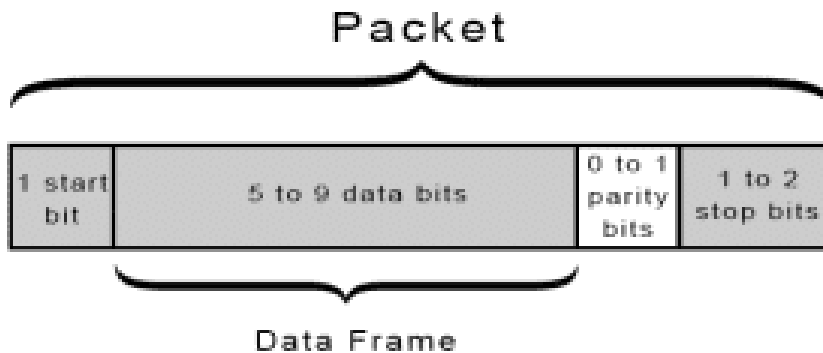
Wires Used	2
Maximum Speed	Any speed up to 115200 baud, usually 9600 baud
Synchronous or Asynchronous?	Asynchronous
Serial or Parallel?	Serial
Max # of Masters	1
Max # of Slaves	1

How UART Works

The UART that is going to transmit data receives the data from a data bus. The data bus is used to send data to the UART by another device like a CPU, memory, or microcontroller. Data is transferred from the data bus to the transmitting UART in parallel form. After the transmitting UART gets the parallel data from the data bus, it adds a start bit, a parity bit, and a stop bit, creating the data packet. Next, the data packet is output serially, bit by bit at the Tx pin. The receiving UART reads the data packet bit by bit at its Rx pin. The receiving UART then converts the data back into parallel form and removes the start bit, parity bit, and stop bits. Finally, the receiving UART transfers the data packet in parallel to the data bus on the receiving end:



UART transmitted data is organized into *packets*. Each packet contains 1 start bit, 5 to 9 data bits (depending on the UART), an optional *parity* bit, and 1 or 2 stop bits:



Start Bit

The UART data transmission line is normally held at a high voltage level when it's not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.

Data Frame

The data frame contains the actual data being transferred. It can be 5 bits up to 8 bits long if a parity bit is used. If no parity bit is used, the data frame can be 9 bits long. In most cases, the data is sent with the least significant bit first.

Parity

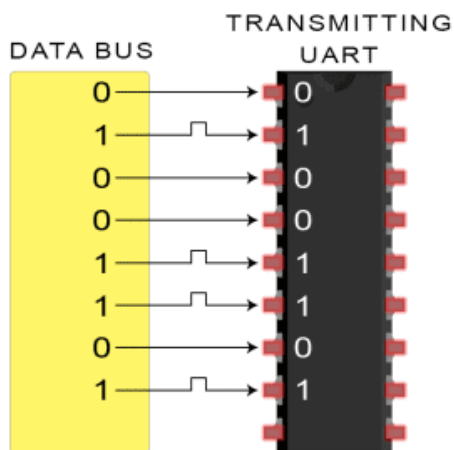
Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long distance data transfers. After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the 1 bits in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bits in the data frame should total to an odd number. When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd; or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.

Stop Bits

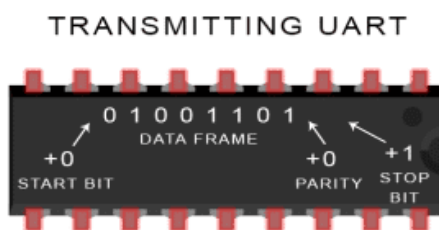
To signal the end of the data packet, the sending UART drives the data transmission line from a low voltage to a high voltage for at least two bit durations.

Steps of UART Transmission

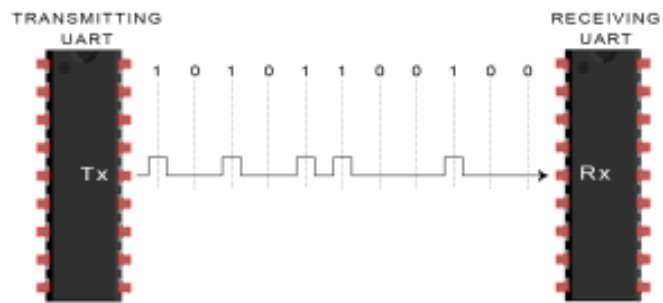
1. The transmitting UART receives data in parallel from the data bus:



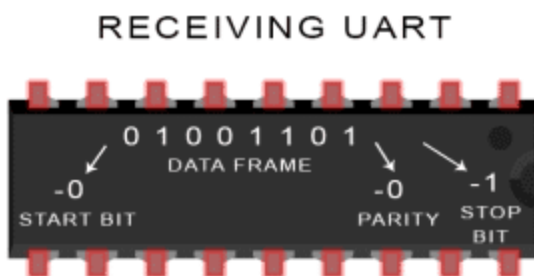
2. The transmitting UART adds the start bit, parity bit, and the stop bit(s) to the data frame:



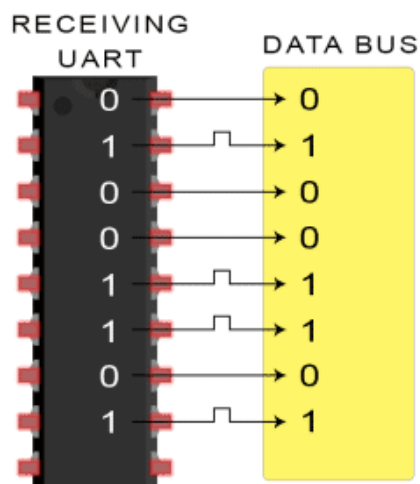
3. The entire packet is sent serially from the transmitting UART to the receiving UART. The receiving UART samples the data line at the pre-configured baud rate:




4. The receiving UART discards the start bit, parity bit, and stop bit from the data frame:



5. The receiving UART converts the serial data back into parallel and transfers it to the data bus on the receiving end:



Advantages and Disadvantages of UARTs



No communication protocol is perfect, but UARTs are pretty good at what they do. Here are some pros and cons to help you decide whether or not they fit the needs of your project:

Advantages

- Only uses two wires
- No clock signal is necessary
- Has a parity bit to allow for error checking
- The structure of the data packet can be changed as long as both sides are set up for it
- Well documented and widely used method

Disadvantages

- The size of the data frame is limited to a maximum of 9 bits
- Doesn't support multiple slave or multiple master systems
- The baud rates of each UART must be within 10% of each other