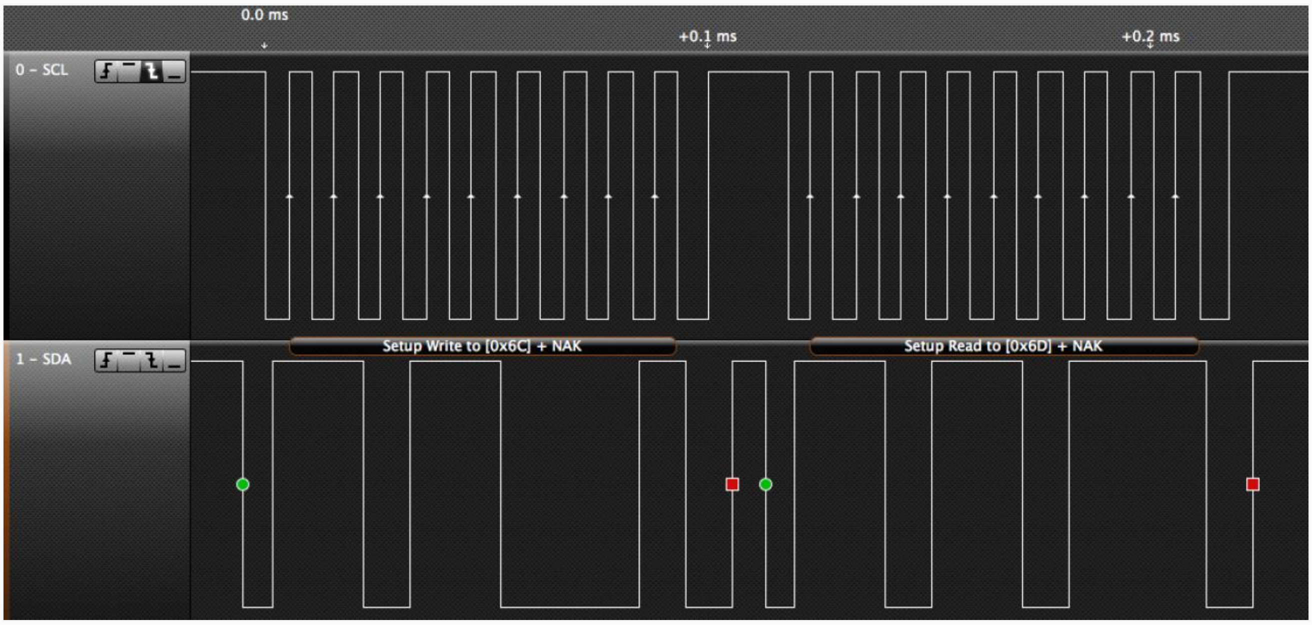
2C - Everything you need to know

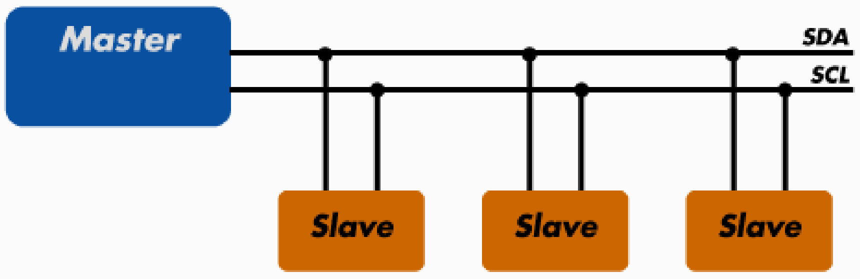
Published: 15/09/2016 | Post categories: Communication Interfaces, Learn | Views: 28364



There is no need for a wordy introduction to **I2C protocol**. We all know it's main parts - 2 wires, multiple slaves, sometimes multiple masters, up to **5MHz** of speed. Often so have we all implemented an I2C connection. Still, every now and then, there's that module that just won't work. This time, we will do an in-depth research about the I2C protocol, and try to cover as much ground as possible.

Overview of the design

I2C works with it's two wires, the **SDA(data line)** and **SCL(clock line)**. Both these lines are open-drain, but are **pulled-up** with resistors. Usually there is one **master** and one or multiple **slaves** on the line, although there can be multiple masters, but we'll talk about that later. Both masters and slaves can transmit or receive data, therefore, a device can be in one of these four states: **master transmit, master receive, slave transmit, slave receive**.

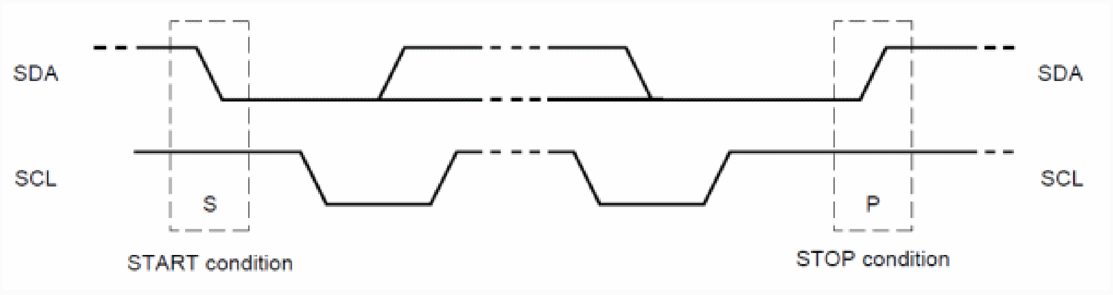


Simple overview of the I2C wiring

Start from the start

The master initiates the communication by sending a **START bit**, this bit alerts all the slaves that some data is coming and they need to listen. After the start bit, 7 bits of a **unique slave address**is sent. Each slave has it's own slave address, this way, only one slave will respond to the data. The last sent bit is the **read/write bit**. If this bit is **0**, it means that the master wishes to **write** data to a register of the slave, if this bit is **1**, it means that the master wishes to **read** data from a register of a slave. Note that all the data is sent **MSB first**.

The start bit is generated by pulling the **SDA line low**, while the **SCL is high**. And the stop bit is indicated by releasing the **SDA** to high, along with the **SCL being high**. So, whenever a module wants to initialize communication, it has to assert **SDA** line. In our I2C library, which you can find in our compilers, the start bit is sent by calling the "**I2C_Start();**" function.



Start and stop conditions

REPEATED START

A module can also initiate what is called a **repeated start**. The repeated start differs from the start condition in the way that there is **no stop condition before it**. This way, the slave knows that the incoming bytes are parts of the same message/conversation.

Sending a message

Data is transferred on the bus in packages of **8 bits**. Whenever a device receives a byte, it has to send an **acknowledgment bit** (**ACK** bit) back to the master (this also goes for the first, address, byte). This bit signalizes if the device has received data successfully. Once the sending device is done sending the data through the bus, it will generate a **STOP** bit, to signalize the end of that conversation, or generate a **repeated start**, to hold the bus again for some other data transmission.

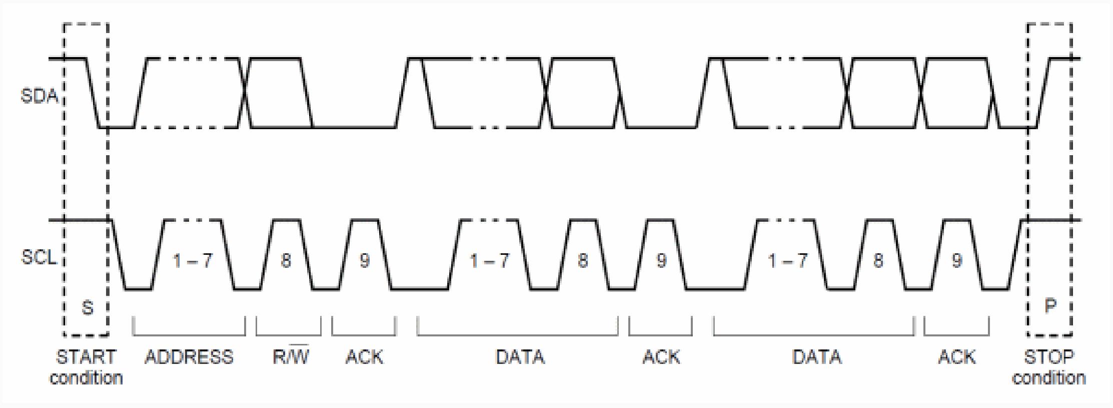


Diagram showing data transfer over I2C

Multiple masters, multiple slaves

I2C supports having multiple devices on the same bus. With multiple masters and multiple slaves, signal collision is bound to happen. Fortunately, I2C was designed to have multiple ways of eliminating errors in data transmission: clock stretching, and arbitration.

CLOCK STRETCHING

The term clock stretching refers to a situation when the addressed slave, which is receiving or sending data, **holds** the clock line low for a desired time period, thus preventing the master to send any more data. The slave usually does this in cases when it needs more time to process incoming data, or data needed to be sent. When the clock is stretched, the master must wait for the clock to go **high** again. Clock stretching can also be done by the master, but the term usually applies to the scenario in which the slave drives the SCL, since usually only the master is driving the SCL.

ARBITRATION

Masters monitor the bus and wait for the line to be accessible in order to start sending data over it. When two masters send bits over the bus, **arbitration** occurs in order to evade data collision. How arbitration works: each master checks the **SDA** line, and compares it to the level that it expects it to be. If the level of the SDA doesn't match the expected level, that device loses arbitration and waits for the line to be free again.

For example: one master sends a logical 1 on the SDA, another master sends a logical 0. After comparing, the first master will see that the SDA is 0, but is expected to be 1. Hence, the first master loses arbitration and stops sending data over the SDA. When two masters are sending a slave address, the one with the lower address wins arbitration.

A simple example, with possible bugs

Now that we have gone over all that theory, it's time for some practical work. We will make a small code example of I2C communication, and go line by line to see where possible bugs can occur.

```
#include

#define SLAVE_ADDRESS 0xC0

void main()
{
  uint8_t write_register = 0x02;
  uint8_t read_value = 0;

  I2C1_Init(100000);         // initialize I2C communication
  I2C1_Start();              // issue I2C start signal
  I2C1_Wr(SLAVE_ADDRESS);    // send byte via I2C  (device address + W)
  I2C1_Wr(write_register);   // send byte (register address)
  I2C1_Wr(0xAA);             // send data (data to be written)
  I2C1_Stop();               // issue I2C stop signal

  Delay_100ms();

  I2C1_Start();                // issue I2C start signal
  I2C1_Wr(SLAVE_ADDRESS);      // send byte via I2C  (device address + W)
  I2C1_Wr(write_register);     // send byte (register address)
  I2C1_Repeated_Start();       // issue I2C signal repeated start
  I2C1_Wr(SLAVE_ADDRESS + 1);  // send byte (device address + R)
  read_value = I2C1_Rd(0u);    // Read the data (NO acknowledge)
  I2C1_Stop();                 // issue I2C stop signal
}
```

The first chunk of code which you see is the process of writing data to an I2C device. First we send the start bit by calling "I2C1_Start();", after that, we address our desired device by sending the device address + the write bit (0). After that, we are sending the adress of the register in the memory of the slave device in which we want to write that data. Last but not least, we send our desired value, and issue a STOP signal.

The second chunk of data is the process of reading data, this can be a little bit confusing, but bear with me. Again, we are issuing the start bit, and sending the device adress with the write bit (yes, write bit). This is because we first want the slave to know from which register we want to read data. After sending the slave address, we send the register from which we want to read. Now, we issue a repeated start, and we are ready to read. We send the slave device address, but this time with the read bit (1). Now the slave device will send us the value from the register which we pointed to in our last I2C_Write, that is why we first needed to write to the device.

Little summation of the read/write process:

1. **Send the device address**
2. **Send the register address**
3. **Read from that register or write to that register**

"MY CODE HANGS HERE..."

These two routines often encounter problems, so let's see where those problems may be.

Code hangs on "I2C_Start();" - the device is waiting for the SDA and SCL lines to be high, so that it can assert them. Check if one or both of these lines are being pulled low somewhere.
Code hangs on "I2C_Write(SLAVE_ADDRESS);" - after sending the addres byte, the master will wait for the slave to respon with the ACK bit, if the address byte is wrong, or the slave is not physically present, the master will be stuck waiting for a response.

Code hangs on "I2C_Read();" - this happens rarely, usually if the slave device is all of a sudden disconnected or interrupted by something else, the master will be stuck waiting for the response from the slave.

Conclusion

There it is! A basic overlook of I2C communication, how it works and how to implement it. Going over the basic problems of implementing an I2C communication, you'll be on the right path to never have problems with I2C ever again, whether you're a firmware developer programming a multi-slave device, or a hardware engineer coming up with a brand new board!