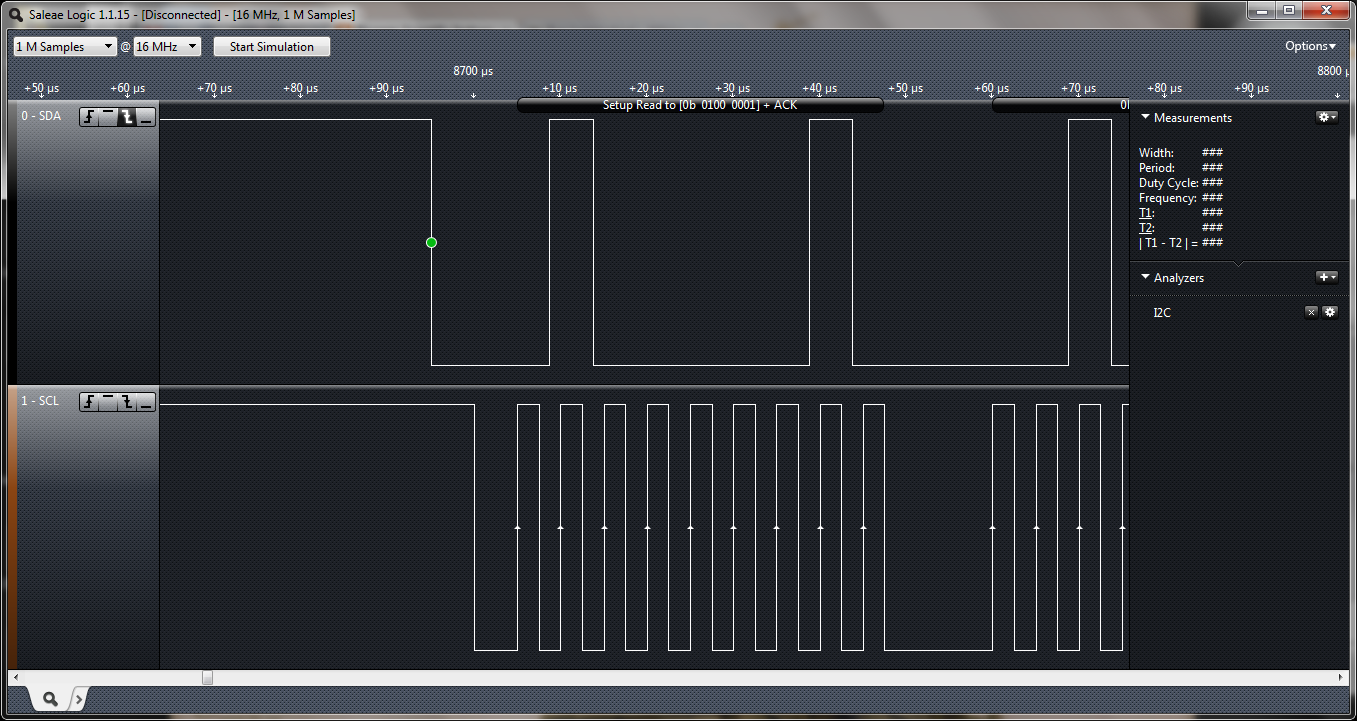
the I2C specification requires devices to only actively pull a line low. The line is tied with a resistor to Vcc, so if no device is active it defaults to a high state. (This reason is similar to the need for the resistor(s) in the SPI example.) In terms of the MSP430 GPIO, think of it as setting the output of a pin to 0, then toggling whether the pin is an output (pulling the line low) or an input (allowing the line to be pulled high). In fact, if you ever find yourself required to bit-bang I2C, this is exactly the technique you'll have to use to prevent any damage to the systems involved.

As a result of the use of resistors to pull the line state high, we are limited on the speed at which we can communicate with I2C. Any capacitance in the line, combined with the resistor, will lengthen the amount of rise time in the line.  Too much resistance, and the line doesn't pull high soon enough to get the right message sent. Too little resistance, and you draw far too much power through the resistor when the line is pulled low. The standard specification for I2C limits transmission speeds to a 100 kHz clock. For the 3 V power used on many MSP430 designs, a resistance between 1 and 10 kΩ is typical; if you're concerned about speed, aim for the 1 kΩ end. Other specifications allow for clock speeds of up to 400 kHz (fast mode), or 1 MHz (fast mode plus). These modes will require smaller resistors, and inherently use more power than standard mode.

 I2C assigns a call sign, of sorts, to each device in the form of a 7-bit address. (An 8th bit is added to specify if the master wants to read from or write to the device.) This address is usually hard-coded into the device, and reading its datasheet will give the information needed to address it correctly. Some devices hard code some of the bits, allowing the user to select the specific address by setting the other bits accordingly. This technique allows the use of more than one of the same device in a system.

In addition to knowing who needs to respond, devices (including the master) need to know when they can send signals; if the line is tied up by another device, other devices must hold on their own messages until the line is signaled as being free. This coordination is effected in I2C by the timing of changes in the data line compared to the clock line. Communication in I2C is initiated by a falling edge in the data line, and marked complete by a rising edge in the data line. Since this is the same line where data is transmitted, I2C specifies that a "start condition" occurs when the data line is pulled low while the clock is high. A "stop condition" occurs when the line is pulled high while the clock is high. Data transmission occurs between these two conditions.

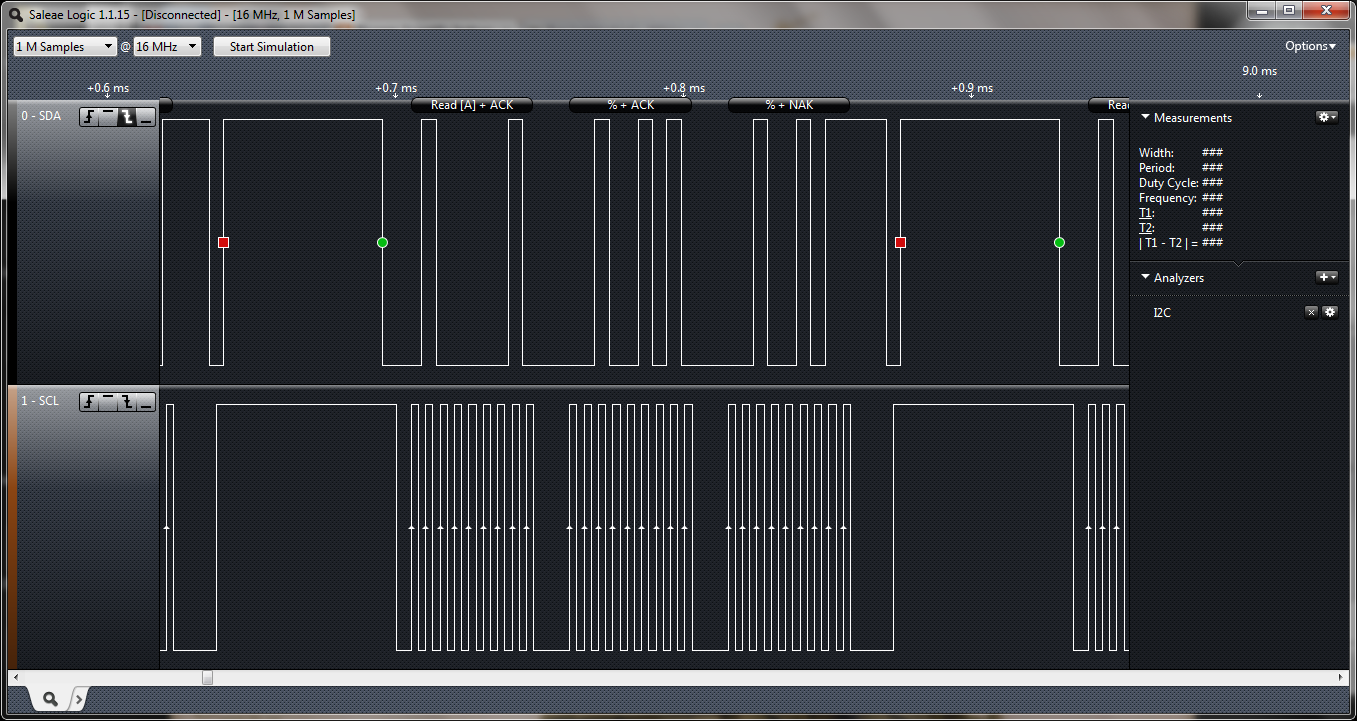
Data is sent after signaling a start condition by changing the line state while the clock is low, and the listening device reading the state while the clock is high. (Note that this is essentially equivalent to Mode 3 for SPI.) It is essential that devices sending messages *do not* change the state of the data line while the clock is high unless marking a start or stop condition. Other devices need to be aware of start and stop conditions to avoid sending messages of their own at the same time. If a device sees a start condition, it will listen for the address. If not its own, it must wait until the line is freed up by a stop condition before sending its message.



Look at this image of the first part of an I2C transmission-- this portion highlights the start condition and the addressing portions. Note that the state of the clock (bottom curve) is idling high. The clock does not start until after the master pulls SDA low. The start condition is signalle by the falling edge on SDA while SCL is high. The start signal is followed by 9 pulses of the clock on SCL. Remember: at this point, neither the master nor the slave changes the state of SDA while the clock is high. Notice that the changes in SDA sending the data occur between pulses-- you can read out the transmitted value at the points where the clock is high. In this example, the data reads 0b01000001. The first 7 bits of this value are the address of the slave (in this case, 0b0100000). These bits are followed by a final 1, which designates a read command. The following data will be sent by the slave, reporting the data it has ready to send.

Did you note that there were 9 pulses, but only 8 bits? The 9th clock is done for some basic handshaking, and is called the "acknowledge" bit. In this case, the master addresses the slave at the specified address and releases the line after sending the 8 bits. If the slave recognizes the call, it pulls the line low; the master reads the 9th bit to confirm that the slave has heard the instruction. What happens if the slave doesn't hear, or fails to acknowledge the signal? Since the line is tied high and the master has relinquished control, the line automatically pulls high. If the master reads the 9th bit and sees it high, it aborts the transmission.

At this point (assuming the slave acknowledged the instruction), it is important that the master not signal a stop condition (by sending a rising edge on SDA while the clock is high) to give the slave the time to report back the data it was asked to read.



 For simplicity, let's assume that we're dealing with a device that has a single, 2-byte register. No further addressing is necessary, so when the slave receives the instruction to read, it reports back the 16 bit value it has. Data is transmitted one byte at a time, with an acknowledge bit between bytes. In this example, we see the slave reporting back the two byte value corresponding to the ASCII characters "%%". After the first byte, the slave relinquishes SDA and watches for the master to acknowledge receiving the data (again, by holding SDA low). After the final byte, notice that the acknowledge bit is now high. On the last cycle, the master does not send an acknowledge bit, indicating that no more data is expected.

Finally, the master pulls SDA low again, then releases the clock so that SCL is high. The master then releases SDA to give a stop condition.  At this point, any other devices needing to start a transmission are free to do so; if between the start and stop any other device has a message to send, it must wait until it sees the stop condition before doing so.

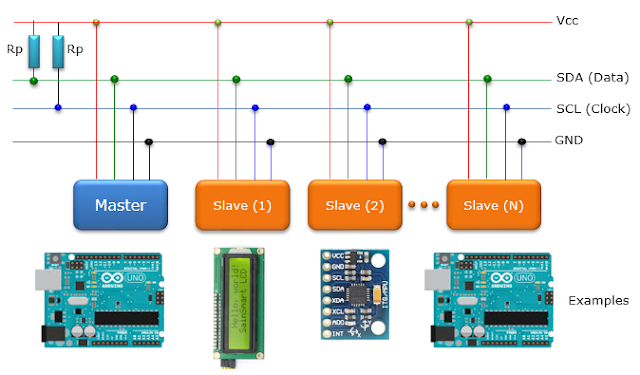
I2C is 2 wire communication i.e. SDA and SCL. When voltage is high it considered as logic 1 and logic 0 when voltages is 0.

**I2C Bus Communication Protocol Tutorial with Example**

**What is I2C:**

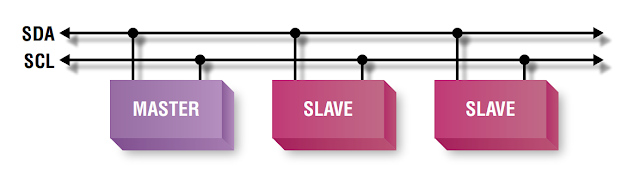
- Inter Integrated Circuit

- Bidirectional Data Transfer  
- Half duplex (have only one data line)  
- Synchronous bus so data is clocked with clock signal  
- Clock is controlled when data line is changed

[](https://2.bp.blogspot.com/-DsqX-sVffdo/WSF6aymTH3I/AAAAAAAAARY/b2ETGc3t2CIwR4Ly37a2tzS-2qnL95GCwCLcB/s1600/EmbeddedSystem_I2C_03.png)

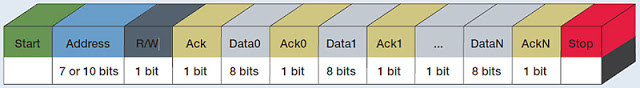
**Speed of I2C:**

low (under 100 kbps)  
Fast (400 kbps)  
high speed (3.4 mbps) I2C V2.0  
2 wire communication :  
SDA and SCL  
Vtg high = 1, low = 0

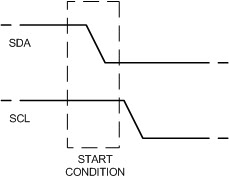
[](https://4.bp.blogspot.com/-NZdsla0-xSg/WSF6puYzzsI/AAAAAAAAARc/wF-jo6EZCR0bIJ3eRp2Tvvp9RuisQ8W3wCLcB/s1600/I2C.PNG)

when SCL = 1 data is a valid data, when SCL = 0 data changes  
Basic protocol is master slave protocol  
- Master controls the clock  
- Slave device may hold the clock low to prevent data transfer  
- no data transfer is present when clock is low  
- It is a kind of wired and connection  
- need to put pullup resistor  
- default it is a open-drain or open-collector, so that adding pull up resistor is necessary so that it will have only two states that is 1.floating high and 2.drive low  
- Default state is high when no device is pulling it low

**Packet format:**

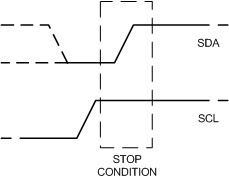
[](https://4.bp.blogspot.com/-zn5eF5Bdw7M/WSF63PDNWDI/AAAAAAAAARg/orhhNW2DLjkIkrC-rWfSjomFr9daq9m9wCLcB/s1600/tektronixi2cdebugfig1.jpg)

**Start condition:**

[](https://3.bp.blogspot.com/-ND39xU0nbac/WSF7FAGLdbI/AAAAAAAAARk/JCu_knsGs84uBzHP6veOYPFcqR6a7ZCKgCLcB/s1600/start_cond.jpg)

SDA changes High to Low when SCL is High

**Stop Condition:**

[](https://2.bp.blogspot.com/-FAlzyVIPMf8/WSF7MhQETFI/AAAAAAAAARo/8wcbPCcIMv42YYV8FSV4NKqU5EPDxUO4QCLcB/s1600/stop_cond.jpg)

SDA changes Low to High when SCL is High

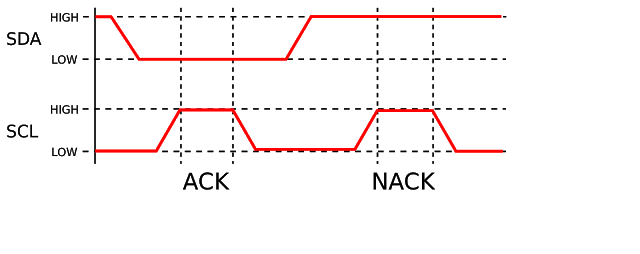
**Repeated Start:**

Incase of repeated start START condition is generated instead of STOP after a frame.

**Data Transfer:**

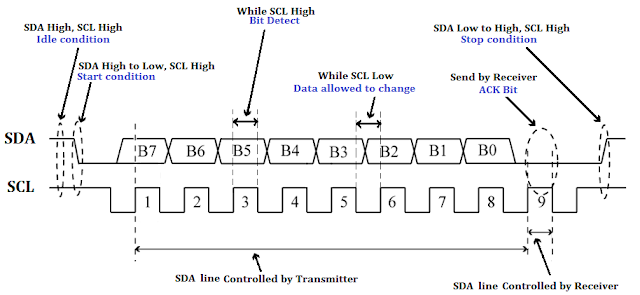
It is byte oriented (8bit)  
Ack transmited by recepient of the data  
MSB first  
First byte is address  
First byte is transmitted by master and addressed slave is the recepient  
Next byte is based on the last bit (R/W)  
7bit address  
1 bit R/W  
0 - master write  
1 - master receive

**9th Pulse ACK**

[](https://4.bp.blogspot.com/-H38C306_YOw/WSF7cx4TcNI/AAAAAAAAARs/uQX3E2JbWb85Qmk0nZD_1MDMizNxy0ujgCLcB/s1600/2000px-I2C_ACK.svg.png)

 ACK => SDA - low  
master only generates clock and slave pulls low the SDA line

**Full I2C Data transfer**

[](https://2.bp.blogspot.com/-oOqG6rQkPmk/WSF7oUcG4NI/AAAAAAAAARw/u2ki6rEiDr4Wxc15SENGmUK6Z1cqYbjTgCLcB/s1600/I2C-Data-Frame.png)

**I2C Multi Master:**

It is a multi master bus  
So bus arbitration is required  
When two device tries to drive SDA to different value  
It is necessary to be sure that is not interfiering with another message  
If a device is trying to send logic one but hears logic 0, it immediately stops transmission and gives the other sender priority  
Synch needed in SCL

**Advantages:**

Good for comm in On-board devices  
Easy to link multiple devices because of addressing scheme  
Cost and complexity do not scale up with the num of devices

**Disadvantages:**

The complexity of supporting software components can be higher than that of scheme(EX. SPI - No need of address in SPI)