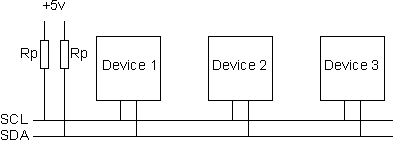
Using the I2C Bus

Using the I2C Bus

Judging from my emails, it is quite clear that the I2C bus can be very confusing for the newcomer. I have lots of examples on using the I2C bus on the website, but many of these are using high level controllers and do not show the detail of what is actually happening on the bus. This short article therefore tries to de-mystify the I2C bus, I hope it doesn't have the opposite effect!

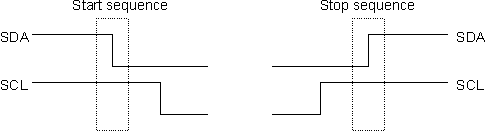
**The physical I2C bus**  
This is just two wires, called SCL and SDA. SCL is the clock line. It is used to synchronize all data transfers over the I2C bus. SDA is the data line. The SCL & SDA lines are connected to all devices on the I2C bus. There needs to be a third wire which is just the ground or 0 volts. There may also be a 5volt wire is power is being distributed to the devices. Both SCL and SDA lines are "open drain" drivers. What this means is that the chip can drive its output low, but it cannot drive it high. For the line to be able to go high you must provide pull-up resistors to the 5v supply. There should be a resistor from the SCL line to the 5v line and another from the SDA line to the 5v line. You only need one set of pull-up resistors for the whole I2C bus, not for each device, as illustrated below:



The value of the resistors is not critical. I have seen anything from 1k8 (1800 ohms) to 47k (47000 ohms) used. 1k8, 4k7 and 10k are common values, but anything in this range should work OK. I recommend 1k8 as this gives you the best performance. If the resistors are missing, the SCL and SDA lines will always be low - nearly 0 volts - and the I2C bus will not work.

**Masters and Slaves**  
The devices on the I2C bus are either masters or slaves. The master is always the device that drives the SCL clock line. The slaves are the devices that respond to the master. A slave cannot initiate a transfer over the I2C bus, only a master can do that. There can be, and usually are, multiple slaves on the I2C bus, however there is normally only one master. It is possible to have multiple masters, but it is unusual and not covered here. On your robot, the master will be your controller and the slaves will be our modules such as the SRF08 or CMPS03. Slaves will never initiate a transfer. Both master and slave can transfer data over the I2C bus, but that transfer is always controlled by the master.

**The I2C Physical Protocol**  
When the master (your controller) wishes to talk to a slave (our CMPS03 for example) it begins by issuing a start sequence on the I2C bus. A start sequence is one of two special sequences defined for the I2C bus, the other being the stop sequence. The start sequence and stop sequence are special in that these are the only places where the SDA (data line) is allowed to change while the SCL (clock line) is high. When data is being transferred, SDA must remain stable and not change whilst SCL is high. The start and stop sequences mark the beginning and end of a transaction with the slave device.



Data is transferred in sequences of 8 bits. The bits are placed on the SDA line starting with the MSB (Most Significant Bit). The SCL line is then pulsed high, then low. Remember that the chip cannot really drive the line high, it simply "lets go" of it and the resistor actually pulls it high. For every 8 bits transferred, the device receiving the data sends back an acknowledge bit, so there are actually 9 SCL clock pulses to transfer each 8 bit byte of data. If the receiving device sends back a low ACK bit, then it has received the data and is ready to accept another byte. If it sends back a high then it is indicating it cannot accept any further data and the master should terminate the transfer by sending a stop sequence.

http://www.robot-electronics.co.uk/images/i2cc.GIF

**How fast?**  
The standard clock (SCL) speed for I2C up to 100KHz. Philips do define faster speeds: Fast mode, which is up to 400KHz and High Speed mode which is up to 3.4MHz. All of our modules are designed to work at up to 100KHz. We have tested our modules up to 1MHz but this needs a small delay of a few uS between each byte transferred. In practical robots, we have never had any need to use high SCL speeds. Keep SCL at or below 100KHz and then forget about it.

**I2C Device Addressing**  
All I2C addresses are either 7 bits or 10 bits. The use of 10 bit addresses is rare and is not covered here. All of our modules and the common chips you will use will have 7 bit addresses. This means that you can have up to 128 devices on the I2C bus, since a 7bit number can be from 0 to 127. When sending out the 7 bit address, we still always send 8 bits. The extra bit is used to inform the slave if the master is  writing to it or reading from it. If the bit is zero the master is writing to the slave. If the bit is 1 the master is reading from the slave. The 7 bit address is placed in the upper 7 bits of the byte and the Read/Write (R/W) bit is in the LSB (Least Significant Bit).

http://www.robot-electronics.co.uk/images/i2cd.GIF

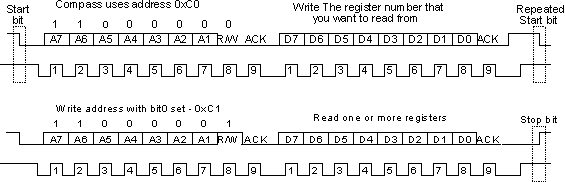
The placement of the 7 bit address in the upper 7 bits of the byte is a source of confusion for the newcomer. It means that to write to address 21, you must actually send out 42 which is 21 moved over by 1 bit. It is probably easier to think of the I2C bus addresses as 8 bit addresses, with even addresses as write only, and the odd addresses as the read address for the same device. To take our CMPS03 for example, this is at address 0xC0 ($C0). You would uses 0xC0 to write to the CMPS03 and 0xC1 to read from it. So the read/write bit just makes it an odd/even address.

**The I2C Software Protocol**The first thing that will happen is that the master will send out a start sequence. This will alert all the slave devices on the bus that a transaction is starting and they should listen in incase it is for them. Next the master will send out the device address. The slave that matches this address will continue with the transaction, any others will ignore the rest of this transaction and wait for the next. Having addressed the slave device the master must now send out the internal location or register number inside the slave that it wishes to write to or read from. This number is obviously dependant on what the slave actually is and how many internal registers it has. Some very simple devices do not have any, but most do, including all of our modules. Our CMPS03 has 16 locations numbered 0-15. The SRF08 has 36. Having sent the I2C address and the internal register address  the master can now send the data byte (or bytes, it doesn't have to be just one). The master can continue to send data bytes to the slave and these will normally be placed in the following registers because the slave will automatically increment the internal register address after each byte. When the master has finished writing all data to the slave, it sends a stop sequence which completes the transaction. So to write to a slave device:   
1. Send a start sequence  
2. Send the I2C address of the slave with the R/W bit low (even address)  
3. Send the internal register number you want to write to  
4. Send the data byte  
5. [Optionally, send any further data bytes]  
6. Send the stop sequence.

As an example, you have an SRF08 at the factory default address of 0xE0. To start the SRF08 ranging you would write 0x51 to the command register at 0x00 like this:  
1. Send a start sequence  
2. Send 0xE0 ( I2C address of the SRF08 with the R/W bit low (even address)  
3. Send 0x00 (Internal address of the command register)  
4. Send 0x51 (The command to start the SRF08 ranging)  
5. Send the stop sequence.

**Reading from the Slave**  
This is a little more complicated - but not too much more. Before reading data from the slave device, you must tell it which of its internal addresses you want to read. So a read of the slave actually starts off by writing to it. This is the same as when you want to write to it: You send the start sequence, the I2C address of the slave with the R/W bit low (even address) and the internal register number you want to write to. Now you send another start sequence (sometimes called a restart) and the I2C address again - this time with the read bit set. You then read as many data bytes as you wish and terminate the transaction with a stop sequence. So to read the compass bearing as a byte from the CMPS03 module:  
1. Send a start sequence  
2. Send 0xC0 ( I2C address of the CMPS03 with the R/W bit low (even address)  
3. Send 0x01 (Internal address of the bearing register)  
4. Send a start sequence again (repeated start)  
5. Send 0xC1 ( I2C address of the CMPS03 with the R/W bit high (odd address)  
6. Read data byte from CMPS03  
7. Send the stop sequence.

The bit sequence will look like this:



**Wait a moment**  
That's almost it for simple I2C communications, but there is one more complication. When the master is reading from the slave, its the slave that places the data on the SDA line, but its the master that controls the clock. What if the slave is not ready to send the data! With devices such as EEPROMs this is not a problem, but when the slave device is actually a microprocessor with other things to do, it can be a problem. The microprocessor on the slave device will need to go to an interrupt routine, save its working registers, find out what address the master wants to read from, get the data and place it in its transmission register. This can take many uS to happen, meanwhile the master is blissfully sending out clock pulses on the SCL line that the slave cannot respond to. The I2C protocol provides a solution to this: the slave is allowed to hold the SCL line low! This is called clock stretching. When the slave gets the read command from the master it holds the clock line low. The microprocessor then gets the requested data, places it in the transmission register and releases the clock line allowing the pull-up resistor to finally pull it high. From the masters point of view, it will issue the first clock pulse of the read by making SCL high and then check to see if it really has gone high. If its still low then its the slave that holding it low and the master should wait until it goes high before continuing. Luckily the hardware I2C ports on most microprocessors will handle this automatically.

Sometimes however, the master I2C is just a collection of subroutines and there are a few implementations out there that completely ignore clock stretching. They work with things like EEPROM's but not with microprocessor slaves that use clock stretching. The result is that erroneous data is read from the slave. Beware!

**Example Master Code**  
This example shows how to implement a software I2C master, including clock stretching. It is written in C for the PIC processor, but should be applicable to most processors with minor changes to the I/O pin definitions. It is suitable for controlling all of our I2C based robot modules. Since the SCL and SDA lines are open drain type, we use the tristate control register to control the output, keeping the output register low. The port pins still need to be read though, so they're defined as SCL\_IN and SDA\_IN. This definition and the initialization is probably all you'll need to change for a different processor.

#define SCL     TRISB4 // I2C bus  
#define SDA     TRISB1 //  
#define SCL\_IN  RB4    //  
#define SDA\_IN  RB1    //

To initialize the ports set the output resisters to 0 and the tristate registers to 1 which disables the outputs and allows them to be pulled high by the resistors.  
SDA = SCL = 1;  
SCL\_IN = SDA\_IN = 0;  
  
We use a small delay routine between SDA and SCL changes to give a clear sequence on the I2C bus. This is nothing more than a subroutine call and return.  
void i2c\_dly(void)  
{  
}

The following 4 functions provide the primitive start, stop, read and write sequences. All I2C transactions can be built up from these.  
void i2c\_start(void)  
{  
  SDA = 1;             // i2c start bit sequence  
  i2c\_dly();  
  SCL = 1;  
  i2c\_dly();  
  SDA = 0;  
  i2c\_dly();  
  SCL = 0;  
  i2c\_dly();  
}  
  
void i2c\_stop(void)  
{  
  SDA = 0;             // i2c stop bit sequence  
  i2c\_dly();  
  SCL = 1;  
  i2c\_dly();  
  SDA = 1;  
  i2c\_dly();  
}

unsigned char i2c\_rx(char ack)  
{  
char x, d=0;  
  SDA = 1;   
  for(x=0; x<8; x++) {  
    d <<= 1;  
    do {  
      SCL = 1;  
    }  
    while(SCL\_IN==0);    // wait for any SCL clock stretching  
    i2c\_dly();  
    if(SDA\_IN) d |= 1;  
    SCL = 0;  
  }   
  if(ack) SDA = 0;  
  else SDA = 1;  
  SCL = 1;  
  i2c\_dly();             // send (N)ACK bit  
  SCL = 0;  
  SDA = 1;  
  return d;  
}

bit i2c\_tx(unsigned char d)  
{  
char x;  
static bit b;  
  for(x=8; x; x--) {  
    if(d&0x80) SDA = 1;  
    else SDA = 0;  
    SCL = 1;  
    d <<= 1;  
    SCL = 0;  
  }  
  SDA = 1;  
  SCL = 1;  
  i2c\_dly();  
  b = SDA\_IN;          // possible ACK bit  
  SCL = 0;  
  return b;  
}  
  
The 4 primitive functions above can easily be put together to form complete I2C transactions. Here's and example to start an SRF08 ranging in cm:

i2c\_start();              // send start sequence  
i2c\_tx(0xE0);             // SRF08 I2C address with R/W bit clear  
i2c\_tx(0x00);             // SRF08 command register address  
i2c\_tx(0x51);             // command to start ranging in cm  
i2c\_stop();               // send stop sequence

Now after waiting 65mS for the ranging to complete (I've left that to you) the following example shows how to read the light sensor value from register 1 and the range result from registers 2 & 3.

i2c\_start();              // send start sequence  
i2c\_tx(0xE0);             // SRF08 I2C address with R/W bit clear  
i2c\_tx(0x01);             // SRF08 light sensor register address  
i2c\_start();              // send a restart sequence  
i2c\_tx(0xE1);             // SRF08 I2C address with R/W bit set  
lightsensor = i2c\_rx(1);  // get light sensor and send acknowledge. Internal register address will increment automatically.  
rangehigh = i2c\_rx(1);    // get the high byte of the range and send acknowledge.  
rangelow = i2c\_rx(0);     // get low byte of the range - note we don't acknowledge the last byte.  
i2c\_stop();               // send stop sequence

Easy isn't it?

I²C uses only two bidirectional [open-drain](https://en.wikipedia.org/wiki/Open_drain) lines, Serial Data Line (SDA) and Serial Clock Line (SCL), [pulled up](https://en.wikipedia.org/wiki/Pull-up_resistor) with [resistors](https://en.wikipedia.org/wiki/Resistor). Typical voltages used are +5 V or +3.3 V although systems with other voltages are permitted.

The I²C [reference design](https://en.wikipedia.org/wiki/I%C2%B2C#Reference_design) has a 7-bit or a 10-bit (depending on the device used) [address space](https://en.wikipedia.org/wiki/Address_space).[[4]](https://en.wikipedia.org/wiki/I%C2%B2C#cite_note-4) Common I²C bus speeds are the 100 [kbit/s](https://en.wikipedia.org/wiki/Kbit/s) *standard mode* and the 10 kbit/s *low-speed mode*, but arbitrarily low clock frequencies are also allowed. Recent revisions of I²C can host more nodes and run at faster speeds (400 kbit/s *Fast mode*, 1 Mbit/s *Fast mode plus* or Fm+, and 3.4 [Mbit/s](https://en.wikipedia.org/wiki/Mbit/s)*High Speed mode*). These speeds are more widely used on embedded systems than on PCs. There are also other features, such as 16-bit addressing.

Note the bit rates are quoted for the transactions between master and slave without clock stretching or other hardware overhead. Protocol overheads include a slave address and perhaps a register address within the slave device as well as per-byte ACK/NACK bits. Thus the actual transfer rate of user data is lower than those peak bit rates alone would imply. For example, if each interaction with a slave inefficiently allows only 1 byte of data to be transferred, the data rate will be less than half the peak bit rate.

The maximum number of nodes is limited by the address space, and also by the total bus [capacitance](https://en.wikipedia.org/wiki/Capacitance) of 400 [pF](https://en.wikipedia.org/wiki/Picofarad), which restricts practical communication distances to a few meters. The relatively high impedance and low noise immunity requires a common ground potential, which again restricts practical use to communication within the same PC board or small system of boards.

**Reference design**[[edit](https://en.wikipedia.org/w/index.php?title=I%C2%B2C&action=edit&section=3" \o "Edit section: Reference design)]

The before mentioned reference design is a bus with a [clock](https://en.wikipedia.org/wiki/Clock_signal) (SCL) and data (SDA) lines with 7-bit addressing. The bus has two roles for nodes: master and slave:

* Master node — node that generates the clock and initiates communication with slaves
* Slave node — node that receives the clock and responds when addressed by the master

The bus is a [multi-master bus](https://en.wikipedia.org/wiki/Multi-master_bus) which means any number of master nodes can be present. Additionally, master and slave roles may be changed between messages (after a STOP is sent).

There may be four potential modes of operation for a given bus device, although most devices only use a single role and its two modes:

* master [transmit](https://en.wikipedia.org/wiki/Transmission_(telecommunications)) — master node is sending data to a slave
* master receive — master node is receiving data from a slave
* slave transmit — slave node is sending data to the master
* slave receive — slave node is receiving data from the master

The master is initially in master transmit mode by sending a [start bit](https://en.wikipedia.org/wiki/Start_bit) followed by the 7-bit address of the slave it wishes to communicate with, which is finally followed by a single bit representing whether it wishes to write(0) to or read(1) from the slave.

If the slave exists on the bus then it will respond with an [ACK](https://en.wikipedia.org/wiki/Acknowledgement_(data_networks)) bit (active low for acknowledged) for that address. The master then continues in either transmit or receive mode (according to the read/write bit it sent), and the slave continues in its complementary mode (receive or transmit, respectively).

The address and the data bytes are sent [most significant bit](https://en.wikipedia.org/wiki/Most_significant_bit) first. The start bit is indicated by a high-to-low transition of SDA with SCL high; the stop bit is indicated by a low-to-high transition of SDA with SCL high. All other transitions of SDA take place with SCL low.

If the master wishes to write to the slave then it repeatedly sends a byte with the slave sending an ACK bit. (In this situation, the master is in master transmit mode and the slave is in slave receive mode.)

If the master wishes to read from the slave then it repeatedly receives a byte from the slave, the master sending an ACK bit after every byte but the last one. (In this situation, the master is in master receive mode and the slave is in slave transmit mode.)

The master then either ends transmission with a [stop bit](https://en.wikipedia.org/wiki/Stop_bit), or it may send another START bit if it wishes to retain control of the bus for another transfer (a "combined message").

**Message protocols**[[edit](https://en.wikipedia.org/w/index.php?title=I%C2%B2C&action=edit&section=4" \o "Edit section: Message protocols)]

I²C defines basic types of messages, each of which begins with a START and ends with a STOP:

* Single message where a master writes data to a slave;
* Single message where a master reads data from a slave;
* Combined messages, where a master issues at least two reads and/or writes to one or more slaves.

In a combined message, each read or write begins with a START and the slave address. After the first START in a combined message these are also called *repeated START* bits. Repeated START bits are not preceded by STOP bits, which is how slaves know the next transfer is part of the same message.

Any given slave will only respond to certain messages, as specified in its product documentation.

Pure I²C systems support arbitrary message structures. [SMBus](https://en.wikipedia.org/wiki/SMBus) is restricted to nine of those structures, such as *read word N* and *write word N*, involving a single slave. [PMBus](https://en.wikipedia.org/wiki/PMBus) extends SMBus with a *Group* protocol, allowing multiple such SMBus transactions to be sent in one combined message. The terminating STOP indicates when those grouped actions should take effect. For example, one PMBus operation might reconfigure three power supplies (using three different I²C slave addresses), and their new configurations would take effect at the same time: when they receive that STOP.

With only a few exceptions, neither I²C nor SMBus define message semantics, such as the meaning of data bytes in messages. Message semantics are otherwise product-specific. Those exceptions include messages addressed to the I²C *general call*address (0x00) or to the SMBus *Alert Response Address*; and messages involved in the SMBus *Address Resolution Protocol* (ARP) for dynamic address allocation and management.

In practice, most slaves adopt request/response control models, where one or more bytes following a write command are treated as a command or address. Those bytes determine how subsequent written bytes are treated and/or how the slave responds on subsequent reads. Most SMBus operations involve single byte commands.

**Messaging example: 24c32 EEPROM**[[edit](https://en.wikipedia.org/w/index.php?title=I%C2%B2C&action=edit&section=5" \o "Edit section: Messaging example: 24c32 EEPROM)]

One specific example is the 24c32 type [EEPROM](https://en.wikipedia.org/wiki/EEPROM), which uses two request bytes that are called Address High and Address Low. (Accordingly, these EEPROMs are not usable by pure SMBus hosts, which only support single byte commands or addresses.) These bytes are used to address bytes within the 32 [kbit](https://en.wikipedia.org/wiki/Kilobit) (4 [kB](https://en.wikipedia.org/wiki/Kilobyte)) supported by that EEPROM; the same two byte addressing is also used by larger EEPROMs, such as 24c512 ones storing 512 kbits (64 kB). Writing and reading data to these EEPROMs uses a simple protocol: the address is written, and then data is transferred until the end of the message. (That data transfer part of the protocol also makes trouble for SMBus, since the data bytes are not preceded by a count and more than 32 bytes can be transferred at once. I²C EEPROMs smaller than 32 kbits, such as 2 kbit 24c02 ones, are often used on SMBus with inefficient single byte data transfers.)

A single message writes to the EEPROM. After the START, the master sends the chip's bus address with the direction bit clear (*write*), then sends the two byte address of data within the EEPROM and then sends data bytes to be written starting at that address, followed by a STOP. When writing multiple bytes, all the bytes must be in the same 32 byte page. While it is busy saving those bytes to memory, the EEPROM will not respond to further I²C requests. (That is another incompatibility with SMBus: SMBus devices must always respond to their bus addresses.)

To read starting at a particular address in the EEPROM, a combined message is used. After a START, the master first writes that chip's bus address with the direction bit clear (*write*) and then the two bytes of EEPROM data address. It then sends a (repeated) START and the EEPROM's bus address with the direction bit set (*read*). The EEPROM will then respond with the data bytes beginning at the specified EEPROM data address -— a combined message, first a write then a read. The master issues an ACK after each read byte except the last byte, and then issues a STOP. The EEPROM increments the address after each data byte transferred; multi-byte reads can retrieve the entire contents of the EEPROM using one combined message.

**Physical layer**[[edit](https://en.wikipedia.org/w/index.php?title=I%C2%B2C&action=edit&section=6" \o "Edit section: Physical layer)]

At the [physical layer](https://en.wikipedia.org/wiki/Physical_layer), both SCL and SDA lines are of [open-drain](https://en.wikipedia.org/wiki/Open-drain) design, thus, [pull-up resistors](https://en.wikipedia.org/wiki/Pull-up_resistor) are needed. Pulling a line to ground signals a logic '0' and driving it [high impedance](https://en.wikipedia.org/wiki/High_impedance) to be pulled high signals a logic '1'. This wire-ANDing allows multiple nodes to connect to the bus without short circuits from signal contention. High speed systems (and some others) may use a [current source](https://en.wikipedia.org/wiki/Current_source) pull-up on SCL or both SCL and SDA, to accommodate higher bus capacitance and enables faster rise times.

An important consequence of this is that multiple nodes may be driving the lines simultaneously. If *any* node is driving the line low, it will be low. Nodes that are trying to transmit a logical one (i.e. letting the line float high) can detect this and conclude that another node is active at the same time.

When used on SCL, this is called *clock stretching* and used as a flow control mechanism for slaves. When used on SDA, this is called [arbitration](https://en.wikipedia.org/wiki/Arbitration) and ensures there is only one transmitter at a time.

When idle, both lines are high. To start a transaction, SDA is pulled low while SCL remains high. Releasing SDA to float high again would be a stop marker, signaling the end of a bus transaction. Although legal, this is typically pointless immediately after a start, so the next step is to pull SCL low.

Except for the start and stop signals, the SDA line only changes while the clock is low; transmitting a data bit consists of pulsing the clock line high while holding the data line steady at the desired level.

While SCL is low, the transmitter (initially the master) sets SDA to the desired value and (after a small delay to let the value propagate) lets SCL float high. The master then waits for SCL to actually go high; this will be delayed by the finite rise-time of the SCL signal (the [RC time constant](https://en.wikipedia.org/wiki/RC_time_constant) of the [pull-up resistor](https://en.wikipedia.org/wiki/Pull-up_resistor) and the [parasitic capacitance](https://en.wikipedia.org/wiki/Parasitic_capacitance) of the bus), and may be additionally delayed by a slave's clock stretching.

Once SCL is high, the master waits a minimum time (4 μs for standard speed I²C) to ensure the receiver has seen the bit, then pulls it low again. This completes transmission of one bit.

After every 8 data bits in one direction, an "acknowledge" bit is transmitted in the other direction. The transmitter and receiver switch roles for one bit, and the original receiver transmits a single 0 bit (ACK) back. If the transmitter sees a 1 bit (NACK) instead, it learns that:

* (If master transmitting to slave) The slave is unable to accept the data. No such slave, command not understood, or unable to accept any more data.
* (If slave transmitting to master) The master wishes the transfer to stop after this data byte.

During the acknowledgment, SCL is always controlled by the master.

After the acknowledge bit, the master may do one of three things:

* Prepare to transfer another byte of data: the transmitter set SDA, and the master pulses SCL high.
* Send a "Stop": Set SDA low, let SCL go high, then let SDA go high. This releases the I²C bus.
* Send a "Repeated start": Set SDA high, let SCL go high, and pull SDA low again. This starts a new I²C bus transaction without releasing the bus.

**Clock stretching using SCL**[[edit](https://en.wikipedia.org/w/index.php?title=I%C2%B2C&action=edit&section=7" \o "Edit section: Clock stretching using SCL)]

One of the more significant features of the I²C protocol is clock stretching. An addressed slave device may hold the clock line (SCL) low after receiving (or sending) a byte, indicating that it is not yet ready to process more data. The master that is communicating with the slave may not finish the transmission of the current bit, but must wait until the clock line actually goes high. If the slave is clock stretching, the clock line will still be low (because the connections are [open-drain](https://en.wikipedia.org/wiki/Open-drain)). The same is true if a second, slower, master tries to drive the clock at the same time. (If there is more than one master, all but one of them will normally lose arbitration.)

The master must wait until it observes the clock line going high, and an additional minimum time (4 μs for standard 100 kbit/s I²C) before pulling the clock low again.

Although the master may also hold the SCL line low for as long as it desires (this is not allowed in newest Rev. 6 of the protocol - subsection 3.1,1), the term "clock stretching" is normally used only when slaves do it. Although in theory any clock pulse may be stretched, generally it is the intervals before or after the acknowledgment bit which are used. For example, if the slave is a [microcontroller](https://en.wikipedia.org/wiki/Microcontroller), its I²C interface could stretch the clock after each byte, until the software decides whether to send a positive acknowledgment or a NACK.

Clock stretching is the only time in I²C where the slave drives SCL. Many slaves do not need to clock stretch and thus treat SCL as strictly an input with no circuitry to drive it. Some masters, such as those found inside custom [ASICs](https://en.wikipedia.org/wiki/ASIC) may not support clock stretching; often these devices will be labeled as a "two-wire interface" and not I²C.

To ensure a minimum bus [throughput](https://en.wikipedia.org/wiki/Throughput), [SMBus](https://en.wikipedia.org/wiki/SMBus) places limits on how far clocks may be stretched. Hosts and slaves adhering to those limits cannot block access to the bus for more than a short time, which is not a guarantee made by pure I²C systems.

**Arbitration using SDA**[[edit](https://en.wikipedia.org/w/index.php?title=I%C2%B2C&action=edit&section=8" \o "Edit section: Arbitration using SDA)]

Every master monitors the bus for start and stop bits, and does not start a message while another master is keeping the bus busy. However, two masters may start transmission at about the same time; in this case, arbitration occurs. Slave transmit mode can also be arbitrated, when a master addresses multiple slaves, but this is less common. In contrast to protocols (such as [Ethernet](https://en.wikipedia.org/wiki/Ethernet)) that use random back-off delays before issuing a retry, I²C has a deterministic arbitration policy. Each transmitter checks the level of the data line (SDA) and compares it with the levels it expects; if they do not match, that transmitter has lost arbitration, and drops out of this protocol interaction.

If one transmitter sets SDA to 1 (not driving a signal) and a second transmitter sets it to 0 (pull to ground), the result is that the line is low. The first transmitter then observes that the level of the line is different from that expected, and concludes that another node is transmitting. The first node to notice such a difference is the one that loses arbitration: it stops driving SDA. If it's a master, it also stops driving SCL and waits for a STOP; then it may try to reissue its entire message. In the meantime, the other node has not noticed any difference between the expected and actual levels on SDA, and therefore continues transmission. It can do so without problems because so far the signal has been exactly as it expected; no other transmitter has disturbed its message.

If the two masters are sending a message to two different slaves, the one sending the lower slave address always "wins" arbitration in the address stage. Since the two masters may send messages to the same slave address—and addresses sometimes refer to multiple slaves—arbitration must continue into the data stages.

Arbitration occurs very rarely, but is necessary for proper multi-master support. As with clock-stretching, not all devices support arbitration. Those that do generally label themselves as supporting "multi-master" communication.

In the extremely rare case that two masters simultaneously send identical messages, both will regard the communication as successful, but the slave will only see one message. Slaves that can be accessed by multiple masters must have commands that are [idempotent](https://en.wikipedia.org/wiki/Idempotent) for this reason.

**Arbitration in SMBus**[[edit](https://en.wikipedia.org/w/index.php?title=I%C2%B2C&action=edit&section=9" \o "Edit section: Arbitration in SMBus)]

While I²C only arbitrates between masters, [SMBus](https://en.wikipedia.org/wiki/SMBus) uses arbitration in three additional contexts, where multiple slaves respond to the master, and one gets its message through.

* Although conceptually a single-master bus, a slave device that supports the "host notify protocol" acts as a master to perform the notification. It seizes the bus and writes a 3-byte message to the reserved "SMBus Host" address (0x08), passing its address and two bytes of data. When two slaves try to notify the host at the same time, one of them will lose arbitration and need to retry.
* An alternative slave notification system uses the separate SMBALERT# signal to request attention. In this case, the host performs a 1-byte read from the reserved "SMBus Alert Response Address" (0x0c), which is a kind of broadcast address. All alerting slaves respond with a data bytes containing their own address. When the slave successfully transmits its own address (winning arbitration against others) it stops raising that interrupt. In both this and the preceding case, arbitration ensures that one slave's message will be received, and the others will know they must retry.
* SMBus also supports an "address resolution protocol", wherein devices return a 16-byte "universal device ID" ([UDID](https://en.wikipedia.org/wiki/UDID)). Multiple devices may respond; the one with the lowest UDID will win arbitration and be recognized.

**Circuit interconnections**[[edit](https://en.wikipedia.org/w/index.php?title=I%C2%B2C&action=edit&section=10" \o "Edit section: Circuit interconnections)]

I²C is popular for interfacing peripheral circuits to prototyping systems, such as the [Arduino](https://en.wikipedia.org/wiki/Arduino) and [Raspberry Pi](https://en.wikipedia.org/wiki/Raspberry_Pi). I²C does not employ a standardized connector, however, and board designers have created various wiring schemes for I²C interconnections. To minimize the possible damage due to plugging 0.1-inch headers in backwards, some developers have suggested using alternating signal and power connections of the following wiring schemes: (GND, SCL, VCC, SDA) or (VCC, SDA, GND, SCL).[[5]](https://en.wikipedia.org/wiki/I%C2%B2C#cite_note-5)

**Buffering and multiplexing**[[edit](https://en.wikipedia.org/w/index.php?title=I%C2%B2C&action=edit&section=11" \o "Edit section: Buffering and multiplexing)]

When there are many I²C devices in a system, there can be a need to include bus [buffers](https://en.wikipedia.org/wiki/Data_buffer) or [multiplexers](https://en.wikipedia.org/wiki/Multiplexer) to split large bus segments into smaller ones. This can be necessary to keep the capacitance of a bus segment below the allowable value or to allow multiple devices with the same address to be separated by a multiplexer. Many types of multiplexers and buffers exist and all must take into account the fact that I²C lines are specified to be bidirectional. Multiplexers can be implemented with analog switches which can tie one segment to another. Analog switches maintain the bidirectional nature of the lines but do not isolate the capacitance of one segment from another or provide buffering capability.

Buffers can be used to isolate capacitance on one segment from another and/or allow I²C to be sent over longer cables or traces. Buffers for bi-directional lines such as I²C must use one of several schemes for preventing latch-up. I²C is open-drain so buffers must drive a low on one side when they see a low on the other. One method for preventing latch-up is for a buffer to have carefully selected input and output levels such that the output level of its driver is higher than its input threshold, preventing it from triggering itself. For example, a buffer may have an input threshold of 0.4 V for detecting a low, but an output low level of 0.5 V. This method requires that all other devices on the bus have thresholds which are compatible and often means that multiple buffers implementing this scheme cannot be put in series with one another.

Alternatively, other types of buffers exist that implement current amplifiers, or keep track of the state (i.e. which side drove the bus low) to prevent latch-up. The state method typically means that an unintended pulse is created during a hand-off when one side is driving the bus low, then the other drives it low, then the first side releases (this is common during an I²C acknowledgement).

**Timing diagram**[[edit](https://en.wikipedia.org/w/index.php?title=I%C2%B2C&action=edit&section=12" \o "Edit section: Timing diagram)]

[](https://en.wikipedia.org/wiki/File:I2C_data_transfer.svg)

1. Data Transfer is initiated with a START bit (S) signaled by SDA being pulled low while SCL stays high.
2. SDA sets the 1st data bit level while keeping SCL low (during blue bar time) .
3. The data is sampled (received) when SCL rises (green) for the first bit (B1).
4. This process repeats, SDA transitioning while SCL is low, and the data being read while SCL is high (B2, Bn).
5. A STOP bit (P) is signaled when SDA is pulled high while SCL is high.

In order to avoid false marker detection, SDA is changed on the SCL falling edge and is sampled and captured on the rising edge of SCL.