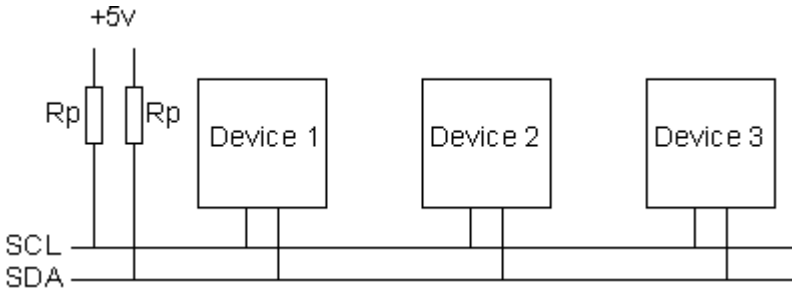
# Using the I2C Bus

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:



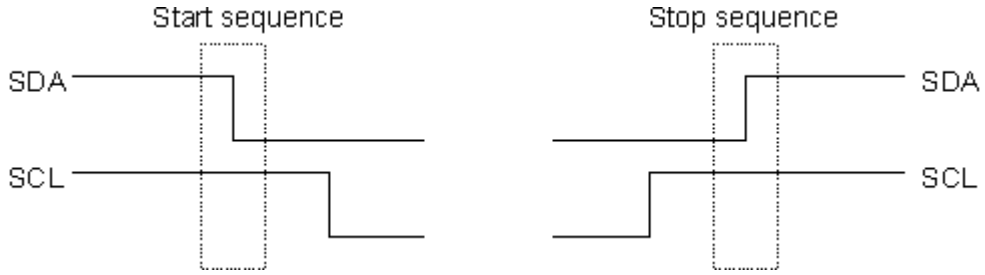

The value of the resistors is not critical. I have seen anything from 1k8 (1800 ohms) to 47k (47000 ohms) used. 1k8, 4k7 and 10k are common values, but anything in this range should work OK. I recommend 1k8 as this gives you the best performance. If the resistors are missing, the SCL and SDA lines will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

**Masters and Slaves**

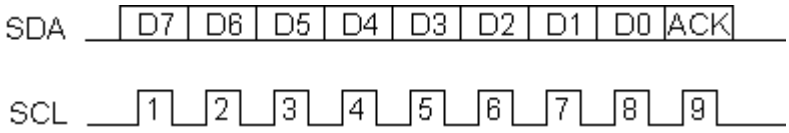
The devices on the I2C bus are either masters or slaves. The master is always the device that drives the SCL clock line. The slaves are the devices that respond to the master. A slave cannot initiate a transfer over the I2C bus, only a master can do that. There can be, and usually are, multiple slaves 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.

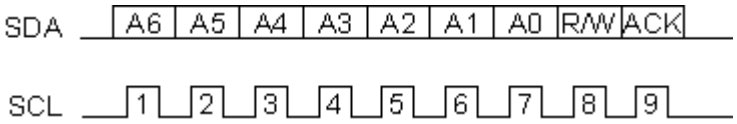



**How fast?**

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 are 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).




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:

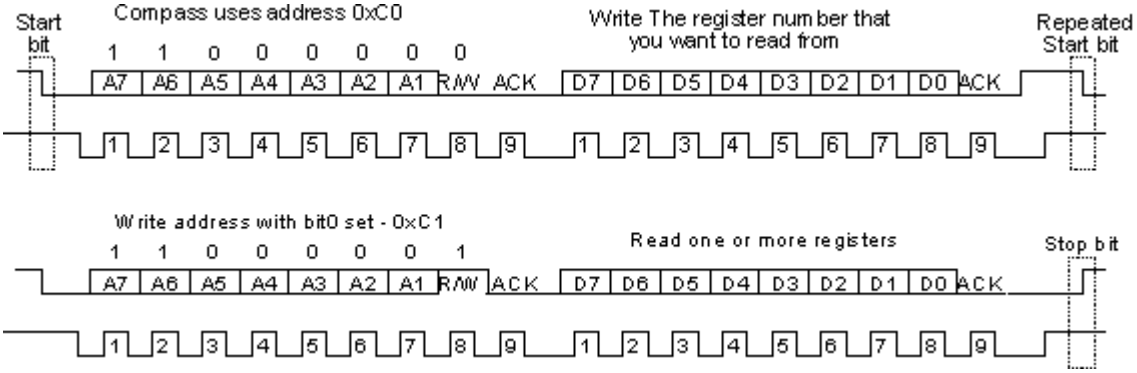
1. Send a start sequence

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?

The definitive specs on the I2C bus can be found on the Philips website. It currently here but if its moved you'll find it easily be googleing on "i2c bus specification".