CMPS03 - Compass Module

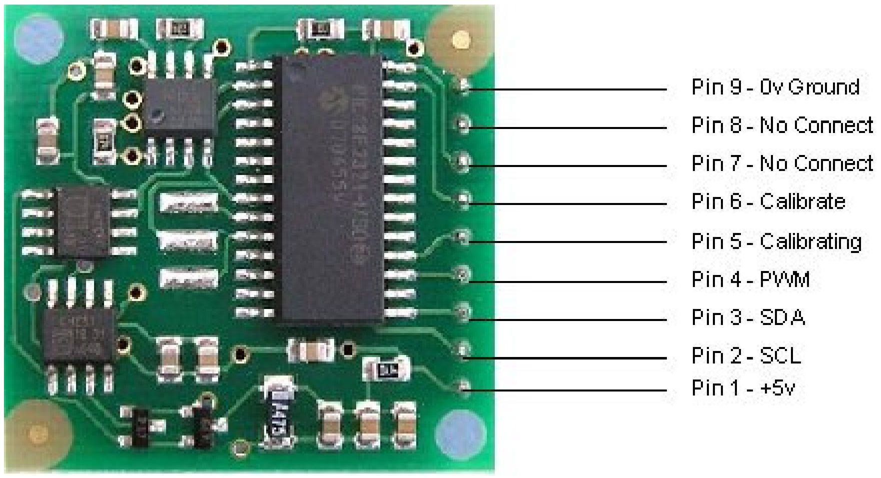
For documentation on CMPS03 revisions prior to Rev14, click here

Earlier versions can be identified by the presence of the silver 8MHz ceramic resonator in the middle of the PCB, this has been removed on new modules.
Rev14 was released March 2007

Overview

This compass module has been specifically designed for use in robots as an aid to navigation. The aim was to produce a unique number to represent the direction the robot is facing. The compass uses the Philips KMZ51 magnetic field sensor, which is sensitive enough to detect the Earths magnetic field. The output from two of them mounted at right angles to each other is used to compute the direction of the horizontal component of the Earths magnetic field. We have examples of using the Compass module with a wide range of popular controllers.

Connections to the compass module



Connections

Pin 1, +5v. The compass module requires a 5v power supply at a nominal 25mA.
There are two ways of getting the bearing from the module. A PWM signal is available on pin 4, or an I2C interface is provided on pins 2,3.

Pins 2,3 are the I2C interface and can be used to get a direct readout of the bearing. If the I2C interface is not used then these pins should be pulled high (to +5v) via a couple of resistors. Around 47k is ok, the values are not at all critical.

Pin 4. The PWM signal is a pulse width modulated signal with the positive width of the pulse representing the angle. The pulse width varies from 1mS (0°) to 36.99mS (359.9°) – in other words 100uS/° with a +1mS offset. The signal goes low for 65mS between pulses, so the cycle time is 65mS + the pulse width - ie. 66ms-102ms. The pulse is generated by a 16 bit timer in the processor giving a 1uS resolution, however I would not recommend measuring this to anything better than 0.1° (10uS). Make sure you connect the I2C pins, SCL and SDA, to the 5v supply if you are using the PWM, as there are no pull-up resistors on these pins.

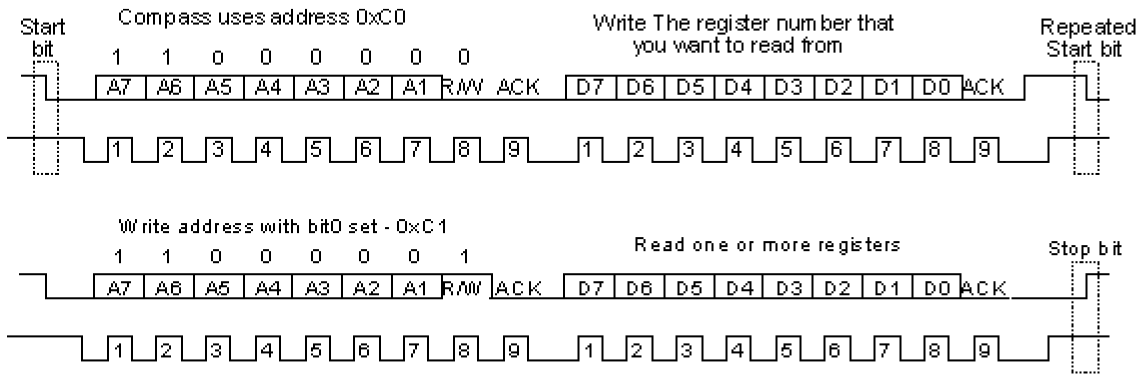
Pin 5 is used to indicate calibration is in progress (active low). You can connect an LED from this pin to +5v via a 390 ohm resistor if you wish.

Pin 6 is one of two ways to calibrate the compass, the other is writing 255 (0xFF) to the command register. Full calibration instructions are further down this page. The calibrate input has an on-board pull-up resistor and can be left unconnected after calibration.

Pins 7 and 8 are currently unused. They have on-board pull-up resistors and should be left unconnected.

Pin 9 is the 0v power supply.

I2C interface.
I2C communication protocol with the compass module is the same as popular eeprom's such as the 24C04.


Start bit
Compass uses address 0xC0
1 1 0 0 0 0 0 0
A7 A6 A5 A4 A3 A2 A1 R/W ACK
Write The register number that you want to read from
D7 D6 D5 D4 D3 D2 D1 D0 ACK
Repeated Start bit
1 2 3 4 5 6 7 8 9
Write address with bit0 set - 0xC1
1 1 0 0 0 0 0 1
Read one or more registers
Stop bit

First send a start bit, the module address (0XC0) with the read/write bit low, then the register number you wish to read. This is followed by a repeated start and the module address again with the read/write bit high (0XC1). You now read one or two bytes for 8bit or 16bit registers respectively. 16bit registers are read high byte first. The compass has a 16 byte array of registers, some of which double up as 16 bit registers as follows;

Register	Function
0	Software Revision Number, Rev14 or higher - for earlier Revisions click here
1	Compass Bearing as a byte, i.e. 0-255 for a full circle
2,3	Compass Bearing as a word, i.e. 0-3599 for a full circle, representing 0-359.9 degrees.
4,5	Internal Test - Sensor1 processed difference signal - 16 bit signed word
6,7	Internal Test - Sensor2 processed difference signal - 16 bit signed word
8,9	Internal Test - Sensor1 raw data - 16 bit signed word
10,11	Internal Test - Sensor2 raw data - 16 bit signed word

12	Unlock code1 - Unlock codes are required for I2C address change or restoring factory calibration
13	Unlock code2
14	Unlock code3
15	Command Register - See text.

Register 0 is the Software revision number (14 at the time of writing). Register 1 is the bearing converted to a 0-255 value. This may be easier for some applications than 0-360 which requires two bytes. For those who require better resolution registers 2 and 3 (high byte first) are a 16 bit unsigned integer in the range 0-3599. This represents 0-359.9°. Registers 4 to 11 are internal test registers. Registers 8,9 and 10,11 contain the raw sensor data. These are the signals coming directly from the sensors, and are the starting point for all the internal calculations which produce the compass bearing. Registers 12,13 and 14 are for writing the unlock codes for I2C address change or restoring factory calibration. Register 15 is the command Register.

The I2C interface does not have any pull-up resistors on the board, these should be provided elsewhere, most probably with the bus master. They are required on both the SCL and SDA lines, but only once for the whole bus, not on each module. I suggest a value of 1k8 if you are going to be working up to 400KHz and 1k2 or even 1k if you are going up to 1MHz. The compass is designed to work at up to the standard clock speed (SCL) of 100KHz, however the clock speed can be raised to 1MHZ providing the following precaution is taken;
At speeds above around 160KHz the CPU cannot respond fast enough to read the I2C data. Therefore a small delay of 50uS should be inserted either side of writing the register address. No delays are required anywhere else in the sequence. By doing this, I have tested the compass module up to 1.3MHz SCL clock speed. There is an example driver here using the HITECH PICC compiler for the PIC16F877. Note that the above is of no concern if you are using popular embedded language processors such as the OOPic. The compass module always operates as a slave, its never a bus master.

Command Register
Register 15 is the command Register. There are very few commands - 0xC0-CE for I2c address change and 0xF2 for restoring factory calibration - these require unlock codes, see below. Also 255 (0xFF) is the calibrate command. There are no unlock codes required for this.

Changing the I2C address from factory default of 0xC0
With Rev14 onwards, it is now possible to change the I2C address to any of 8 addresses 0xC0, 0xC2, 0xC4, 0xC6, 0xC8, 0xCA, 0xCC or 0xCE. You do this by writing unlock codes to registers 12,13 and 14 and the new address to register 15. Note that the unlock codes are different to the ones which restore factory calibration.

Reg 12	Reg 13	Reg 14	Reg 15
0xA0	0xAA	0xA5	0xC2

The above example will change the address to 0xC2 and the new address will be effective immediately. Don't forget to label you CMPS03 with the new address. You can do this in one

I2C transaction, setting the register address to 12 and writing the four bytes. The internal register pointer is incremented automatically.

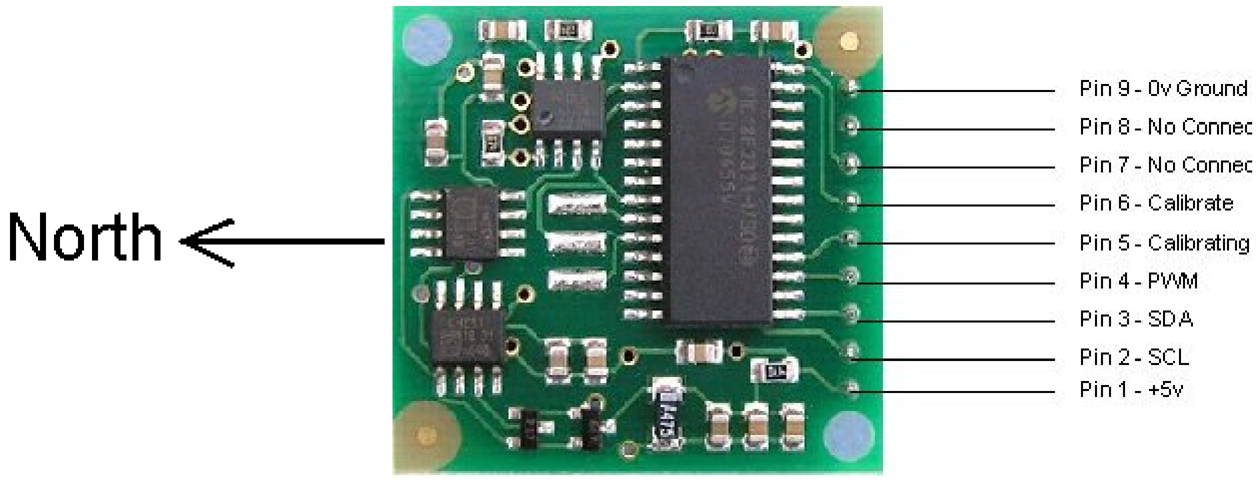
Restoring Factory Calibration
With Rev14 onwards, it is now possible to restore the factory calibration settings. You do this by writing unlock codes to registers 12,13 and 14 and the restore command (0xF2) to register 15. Note that the unlock codes are different to the ones which used for changing the I2C address.

Reg 12	Reg 13	Reg 14	Reg 15
0x55	0x5A	0xA5	0xF2

You can do this in one transaction, setting the register address to 12 and writing the four bytes. The internal register pointer is incremented automatically.

Calibration
Do not attempt this until you have your compass working! Especially if your using the I2C interface - get that fully working first. The module has already been calibrated in our workshop for our inclination, which is 67 degrees. If your location is close to this, you may like to try the compass without re-calibrating at all. Calibration only needs to be done once - the calibration data is stored in EEPROM on the PIC18F2321 chip. You do not need to re-calibrate every time the module is powered up.

Compass module orientation to produce 0 degrees reading.



Before calibrating the compass, you must know **exactly** which direction is North, East, South and West. Remember these are the magnet poles, not the geographic poles. Don't guess at it. Get a magnetic needle compass and check it. When calibrating, make sure the compass is horizontal at all times with components upwards, don't tilt it. Keep all magnetic and ferrous materials away from the compass during calibration - including your wristwatch.

I2C Method
To calibrate using the I2C bus, you only have to write 255 (0xff) to register 15, once for each of

the four major compass points North, East, South and West. The 255 is cleared internally automatically after each point is calibrated. The compass points can be set in any order, but all four points must be calibrated. For example
1. Set the compass module flat, pointing North. Write 255 to register 15, Calibrating pin (pin5) goes low.
2. Set the compass module flat, pointing East. Write 255 to register 15,
3. Set the compass module flat, pointing South. Write 255 to register 15,
4. Set the compass module flat, pointing West. Write 255 to register 15, Calibrating pin (pin5) goes high.
That's it.

Pin Method
Pin 6 is used to calibrate the compass. The calibrate input (pin 6) has an on-board pull-up resistor and can be left unconnected after calibration. To calibrate the compass you only have to take the calibrate pin low and then high again for each of the four major compass points North, East, South and West. A simple push switch wired from pin6 to 0v (Ground) is OK for this. The compass points can be set in any order, but all four points must be calibrated. For example
1. Set the compass module flat, pointing North. Briefly press and release the switch, Calibrating pin (pin5) goes low.
2. Set the compass module flat, pointing East. Briefly press and release the switch,
3. Set the compass module flat, pointing South. Briefly press and release the switch,
4. Set the compass module flat, pointing West. Briefly press and release the switch, Calibrating pin (pin5) goes high.
That's it.

One point which must be emphasized. ***The calibration must be done in exactly four steps***, once for each of the four major compass points North, East, South and West. Previous versions performed part of the calibration at each step and you could actually go back and do a point again, taking 5 or more steps. Only the most recent reading from each point was used. Rev 14 onwards works differently. The 1st step (pulling pin 6 low or writing 255 to register 15) initializes internal construction registers and collects the 1st data set. The remaining steps only collect data. After the final 4th step, multiple processing stages generate and store in EEPROM a number of internal calibration values. When you perform the 1st step, Pin 5 will go low. After the 4th step it will go high again. You can connect an LED from pin 5 to 5v via a 390ohm resistor to indicate calibration is in progress. It should be high (Led off) before you start.

Rev 15 Firmware - April 2007
The default internal scan time of the CPMS03 is 100mS, or 10 updates per second 100mS is a common denominator of 50Hz/20mS (5*) and 60Hz/16.66mS (6*).
In Rev15 we have provided two further options. A 33mS scan time which is 30 updates per second, and 300mS which is 3.3 updates per second. The 33ms scan time can be used for applications that require fast updating. The penalty is that fewer samples can be taken in the available time, so there will be a slightly higher variation in the output angle. A 300mS scan time will increase the stability of the reading slightly and is suitable where the update rate is less important. Most users will not need to change this from the 100mS default.

You change the scan time by writing unlock codes to registers 12,13 and 14 and the new scan command to register 15. The scan commands are:

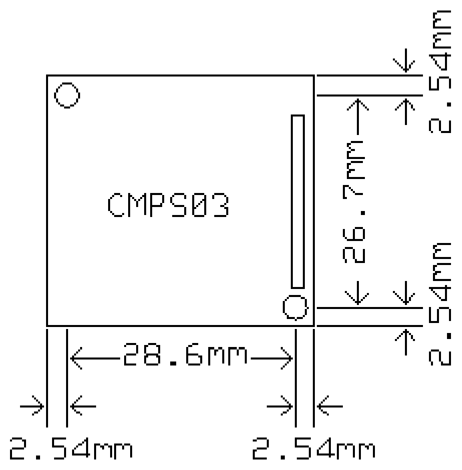
Scan Period	Command
300mS	0x10
100mS	0x11
33mS	0x12

For example, to change to 33mS scan time, write the following to registers 12,13,14 and 15.

Reg 12	Reg 13	Reg 14	Reg 15
0x55	0x5A	0xA5	0x12

The new scan period is stored in EEPROM so you only need to do it once, not every time the module is powered up. When the scan time is changed, the calibration is affected so you will need to recalibrate the compass. If you restore factory calibration then the scan time is reset to 100mS.

PCB Drilling Plan
The following diagram shows the CMPS03 PCB mounting hole positions.



We have examples of using the Compass module with a wide range of popular controllers. There is also an FAQ on the compass module which contains further useful information.

CMPS03 - Compass Module FAQ

Q. Will your compass tell me which direction is north?
A. No.

Q. Ok, will your compass tell me which way the magnetic north pole is?
A. No.

Q. So what does it do?
A. It gives you the direction of the horizontal component of the prevailing magnetic flux. OK, so I'm making a point here and it's this: The magnetic field in a building can vary hugely. Don't expect it to be always pointing north. By walking around my workshop with a standard magnetic needle compass I can make it point in any direction I like by moving past various machines, and I can do the same thing at home by going near the fridge or even the central heating radiators. The compass module will give the same results. It can only provide information about the field at its current location.

Q. Is the Compass affected by Motors, Magnets, Ferrous objects etc?
A. Yes. Anything that affects the local magnetic field will affect the Compass module. Motors in particular contain strong magnets. The only solution is to mount the compass as far away from magnetic/ferrous objects as is practicable. Also see the previous questions and answers above

Q. If the accuracy of the compass is 3-4° , how can you provide a resolution of 0.1° ?
A. Accuracy and resolution are not the same thing. A 3.5 digit multi-meter has a resolution of 1 in 2000 or 0.05%, yet the accuracy on some ranges can be 5% - a hundred time worse. A robot can use the resolution to detect small changes in direction even though there is uncertainty about the absolute direction. Also see previous answer.

Q. How do I select between I2C and PWM outputs?
A. No need to. The PWM output is always present whether you use it or not and does not require a trigger signal. The I2C interface is also always available – just connect it up and start talking.

Q. How many degrees can the module be tilted before the readings become inaccurate?
A. None. Moving the compass off horizontal will result in increasing error. The sensors are sensitive to the vertical component of the Earths magnetic field as well. The angle of the Earths field is not horizontal, it dives into the ground at an angle which varies according to location. It is this which produces an inherent error in the reading, and makes calibration of the compass required at the point where it is to be operated. After calibration you can expect 3-4° accuracy if you keep it horizontal.

Q. When measuring the PWM signal the maximum reading (which should be 359 degrees) is actually 357 degrees (or similar). Why is this?
A. This is because of slight differences in CMPS01/03 and Controller oscillators and the internal software generating and measuring the pulse. If a 359 degree roll-over is important, then you can add a "fiddle factor" to the calculation or alternatively, use the I2C Bus.

Q. Which way up should the Compass be?
A. The Compass module should be horizontal, with the PCB parallel to the earths surface. The IC's should be on top and (for the CMPS01) the Sensors underneath.

Q. What is the difference between the CMPS01 and the CMPS03?
A. For most purposes, not a lot really. The coils around the KMZ10 sensors on the underside were proving way to expensive to assemble and still sell the module at such a low cost. When Philips produced the KMZ51 sensor, an 8 pin surface mount chip with the coils built in, it was time to change. The PCB is the same physical size as the CMPS01 with the same connections and uses the same software. Calibrating the CMPS03 is the same as CMPS01 Rev7.

Q. Can the Compass be mounted near to the speaker on the SP03 speech synthesizer?
A. The speaker magnet will effect the Compass. The effects reduce with distance and are negligible at 10-12 inches

Q. My software master I2C code does not read correct data from the compass, but its works fine with an I2C EEPROM chip. Why is this?
A. The most likely cause is the master code not waiting for the I2C bus hold. This is where the slave can hold the SCL line low until it is ready. When the master releases SCL (remember it's a passive pull-up, not driven high) the slave may still be holding it low. The master code should then wait until it actually does go high before proceeding. If you are writing your own code, have a look at our I2C Tutorial.

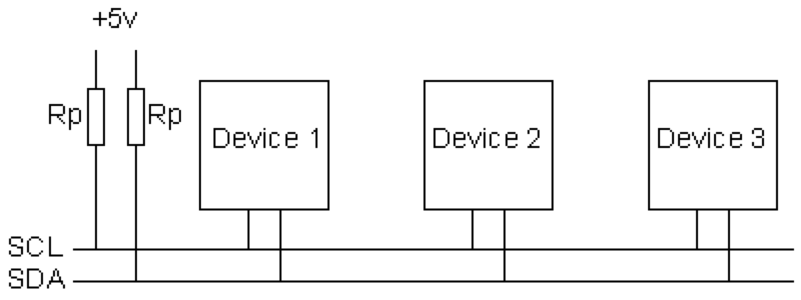
If you have any questions of your own, Please email me.
Gerald Coe.

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:



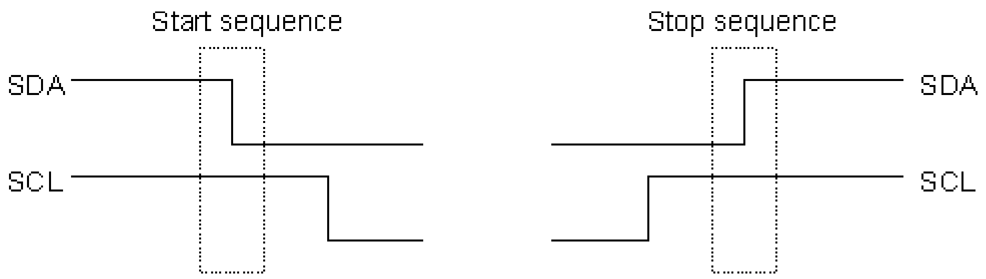
The value of the resistors is not critical. I have seen anything from 1k8 (1800 ohms) to 47k (47000 ohms) used. 1k8, 4k7 and 10k are common values, but anything in this range should work OK. I recommend 1k8 as this gives you the best performance. If the resistors are missing, the SCL and SDA lines will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

Masters and Slaves

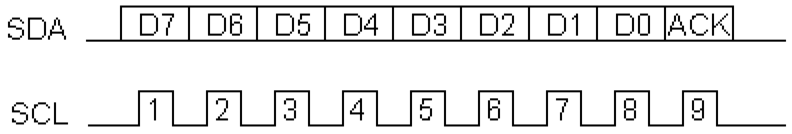
The devices on the I2C bus are either masters or slaves. The master is always the device that drives the SCL clock line. The slaves are the devices that respond to the master. A slave cannot initiate a transfer over the I2C bus, only a master can do that. There can be, and usually are, multiple slaves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

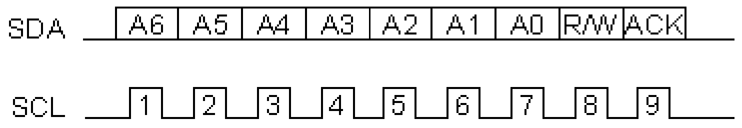


Data is transferred in sequences of 8 bits. The bits are placed on the SDA line starting with the MSB (Most Significant Bit). The SCL line is then pulsed high, then low. Remember that the chip cannot really drive the line high, it simply "lets go" of it and the resistor actually pulls it high. For every 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:

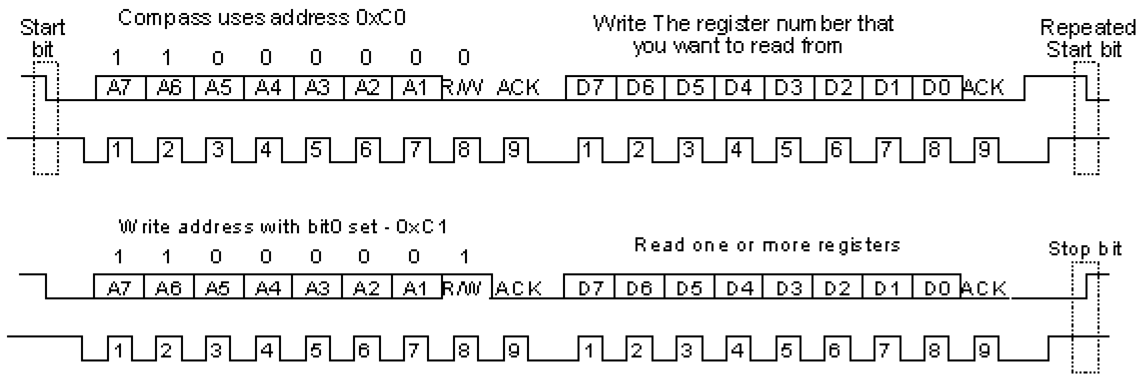
1. Send a start sequence
2. Send 0xE0 (I2C address of the SRF08 with the R/W bit low (even address)
3. Send 0x00 (Internal address of the command register)
4. Send 0x51 (The command to start the SRF08 ranging)
5. Send the stop sequence.

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:


Start bit
Compass uses address 0xC0
1 1 0 0 0 0 0 0
A7 A6 A5 A4 A3 A2 A1 R/W ACK
Write The register number that you want to read from
D7 D6 D5 D4 D3 D2 D1 D0 ACK
Repeated Start bit
1 2 3 4 5 6 7 8 9
Write address with bit0 set - 0xC1
1 1 0 0 0 0 0 1
Read one or more registers
Stop bit

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
```

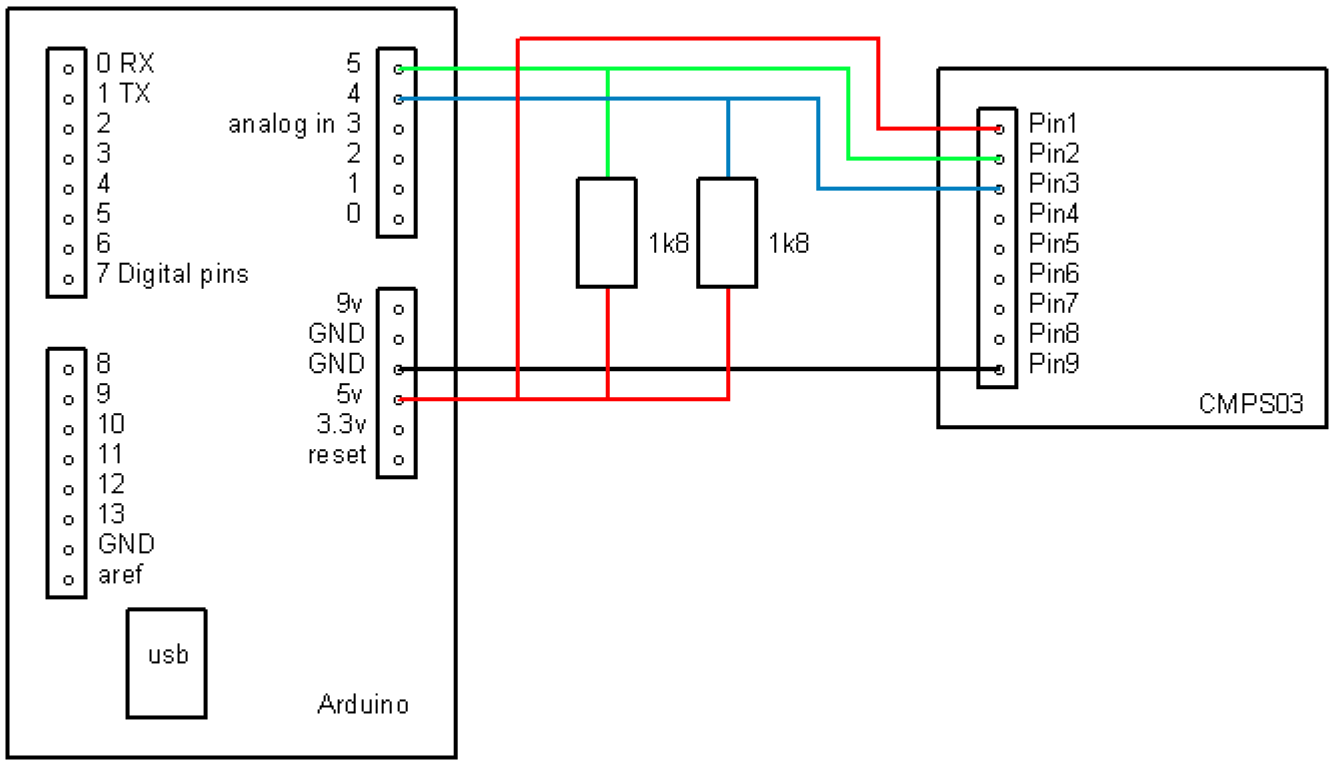
Now after waiting 65mS for the ranging to complete (I've left that to you) the following example shows how to read the light sensor value from register 1 and the range result from registers 2 & 3.

```
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".

CMPS03 Magnetic Compass
This uses the I2C bus to connect the Arduino to the CMPS03. It reads the bearing as a two byte integer and displays the bearing as a number 0-359 on the PC.



```
/*
CMPS03 with arduino I2C example

This will display a value of 0 - 359 for a full rotation of the compass.

The SDA line is on analog pin 4 of the arduino and is connected to pin 3 of
the CMPS03.
The SCL line is on analog pin 5 of the arduino and is conected to pin 2 of
the CMPS03.
Both SDA and SCL are also connected to the +5v via a couple of 1k8 resistors.
A switch to callibrate the CMPS03 can be connected between pin 6 of the
CMPS03 and the ground.


*/
#include <Wire.h>

#define address 0x60 //defines address of compass

void setup(){
  Wire.begin(); //conects I2C
  Serial.begin(9600);
}
```

```
void loop(){
  byte highByte;
  byte lowByte;

   Wire.beginTransmission(address);      //starts communication with cmps03
   Wire.send(2);                         //Sends the register we wish to read
   Wire.endTransmission();

   Wire.requestFrom(address, 2);        //requests high byte
   while(Wire.available() < 2);         //while there is a byte to receive
   highByte = Wire.receive();           //reads the byte as an integer
   lowByte = Wire.receive();
   int bearing = ((highByte<<8)+lowByte)/10;

   Serial.println(bearing);
   delay(100);
}
```