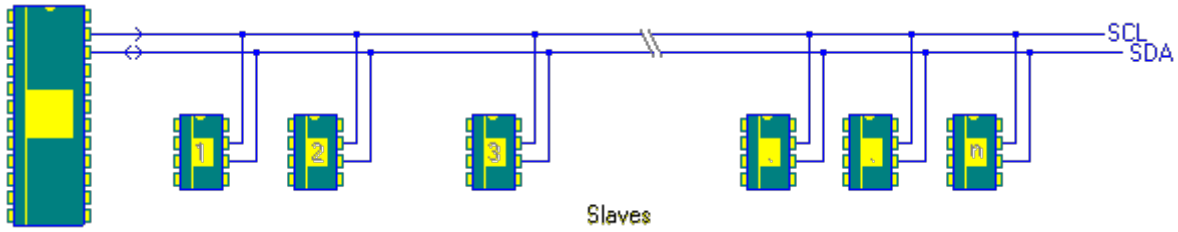
I2C Bus Protocol

The I2C bus physically consists of 2 active wires and a ground connection. The active wires, called SDA and SCL, are both bi-directional. SDA is the Serial DAta line, and SCL is the Serial CLock line.

Every device hooked up to the bus has its own unique address, no matter whether it is an MCU, LCD driver, memory, or ASIC. Each of these chips can act as a receiver and/or transmitter, depending on the functionality. Obviously, an LCD driver is only a receiver, while a memory or I/O chip can be both transmitter and receiver.

The I2C bus is a multi-master bus. This means that more than one IC capable of initiating a data transfer can be connected to it. The I2C protocol specification states that the IC that initiates a data transfer on the bus is considered the Bus Master. Consequently, at that time, all the other ICs are regarded to be Bus Slaves.

As bus masters are generally microcontrollers, let's take a look at a general 'inter-IC chat' on the bus. Lets consider the following setup and assume the MCU wants to send data to one of its slaves (also see here for more information; click here for information on how to receive data from a slave).



First, the MCU will issue a START condition. This acts as an 'Attention' signal to all of the connected devices. All ICs on the bus will listen to the bus for incoming data.

Then the MCU sends the ADDRESS of the device it wants to access, along with an indication whether the access is a Read or Write operation (Write in our example). Having received the address, all IC's will compare it with their own address. If it doesn't match, they simply wait until the bus is released by the stop condition (see below). If the address matches, however, the chip will produce a response called the ACKNOWLEDGE signal.

Once the MCU receives the acknowledge, it can start transmitting or receiving DATA. In our case, the MCU will transmit data. When all is done, the MCU will issue the STOP condition. This is a signal that the bus has been released and that the connected ICs may expect another transmission to start any moment.

We have had several states on the bus in our example: START, ADDRESS, ACKNOWLEDGE, DATA , STOP. These are all unique conditions on the bus. Before we take a closer look at these bus conditions we need to understand a bit about the physical structure and hardware of the bus.

I2C Bus Hardware

Both SDA and SCL are initially bi-directional. This means that in a particular device, these lines can be driven by the IC itself or from an external device. In order to achieve this functionality, these signals use open collector or open drain outputs (depending on the technology).

The bus interface is built around an input buffer and an open drain or open collector transistor. When the bus is IDLE, the bus lines are in the logic HIGH state (note that external pull-up resistors are necessary for this which is easily forgotten). To put a signal on the bus, the chip drives its output transistor, thus pulling the bus to a LOW level. The "pull-up resistor" in the devices as seen in the figure is actually a small current source or even non-existent.

The nice thing about this concept is that it has a "built-in" bus mastering technique. If the bus is "occupied" by a chip that is sending a 0, then all other chips lose their right to access the bus. More will be explained about this in the section about bus arbitration.

However, the open-collector technique has a drawback, too. If you have a long bus, this will have a serious effect on the speed you can obtain. Long lines present a capacitive load for the output drivers. Since the pull-up is passive, you are facing an RC constant which will reflect on the shapes of the signals. The higher this RC constant, the slower you can go. This is due to the effect that it influences the slew rate of the edges on the I2C bus. At a certain point, the ICs will not be able to distinguish clearly between a logic 1 and 0.

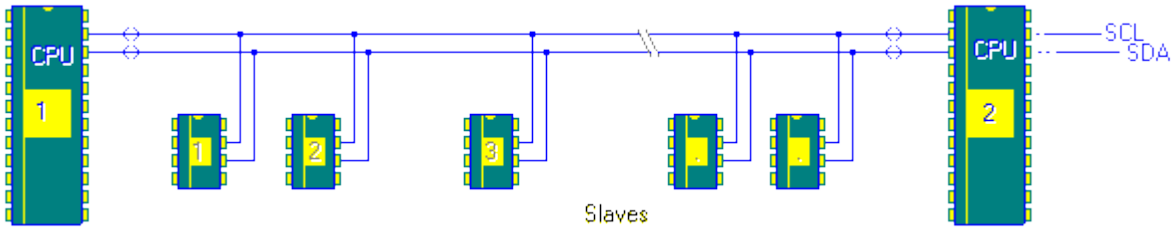
Bus Arbitration

So far we have seen the operation of the bus from the master's point of view and using only one master on the bus.

The I2C bus was originally developed as a multi-master bus. This means that more than one device initiating transfers can be active in the system.

When using only one master on the bus there is no real risk of corrupted data, except if a slave device is malfunctioning or if there is a fault condition involving the SDA / SCL bus lines.

This situation changes with 2 MCU's:



When MCU 1 issues a start condition and sends an address, all slaves will listen (including MCU 2 which at that time is considered a slave as well). If the address does not match the address of CPU 2, this device has to hold back any activity until the bus becomes idle again after a stop condition.

As long as the two MCU's monitor what is going on on the bus (start and stop) and as long as they are aware that a transaction is going on because the last issued command was not a STOP, there is no problem.

Let's assume one of the MCU's missed the START condition and still thinks the bus is idle, or it just came out of reset and wants to start talking on the bus which could very well happen in a real-life

scenario. This could lead to problems.

How can you know if some other device is transmitting on the bus ?

Fortunately, the physical bus setup helps us out. Since the bus structure is a wired AND (if one device pulls a line low it stays low), you can test if the bus is idle or occupied.

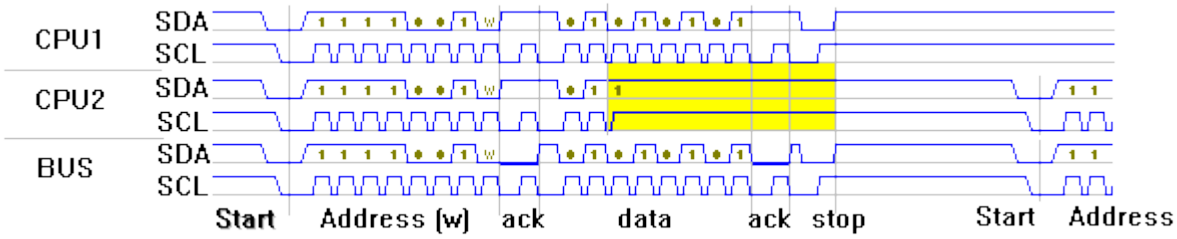
When a master changes the state of a line to HIGH, it MUST always check that the line really has gone to HIGH. If it stays low then this is an indication that the bus is occupied and some other device is pulling the line low.

Therefore the general rule of thumb is: If a master can't get a certain line to go high, it lost arbitration and needs to back off and wait until a stop condition is seen before making another attempt to start transmitting.

What about the risk of data corruption ?

Since the previous rule says that a master loses arbitration when it cannot get either SCL or SDA to go high when needed, this problem does not exist. It is the device that is sending the '0' that rules the bus. One master cannot disturb the other master's transmission because if it can't detect one of the lines to go high, it backs off, and if it is the other master that can't do so, it will behave the same.

This kind of back-off condition will only occur if the two levels transmitted by the two masters are not the same. Therefore, let's have a look at the following figure, where two MCUs start transmitting at the same time:



The two MCU's are accessing a slave in write mode at address 1111001. The slave acknowledges this. So far, both masters are under the impression that they "own" the bus.

Now MCU1 wants to transmit 01010101 to the slave, while MCU 2 wants to transmit 01100110 to the slave. The moment the data bits do not match anymore (because what the MCU sends is different than what is present on the bus) one of them loses arbitration and backs off. Obviously, this is the MCU which did not get its data on the bus. For as long as there has been no STOP present on the bus, it won't touch the bus and leave the SDA and SCL lines alone (yellow zone).

The moment a STOP was detected, MCU2 can attempt to transmit again.

From the example above we can conclude that is the master that is pulling the line LOW in an arbitration situation that always wins the arbitration. The master which wanted the line to be HIGH when it is being pulled low by the other master loses the bus. We call this a loss of arbitration or a

back-off condition.

When a MCU loses arbitration, it has to wait for a STOP condition to appear on the bus. Then it knows that the previous transmission has been completed

Clock Synchronization

All masters generate their own clock on the SCL line to transfer messages on the I2C-bus. Data is only valid during the HIGH period of the clock. A defined clock is therefore needed for the bit-by-bit arbitration procedure to take place.

Clock synchronization is performed using the wired-AND connection of I2C interfaces to the SCL line. This means that a HIGH to LOW transition on the SCL line will cause the devices concerned to start counting off their LOW period and, once a device clock has gone LOW, it will hold the SCL line in that state until the clock HIGH state is reached. However, the LOW to HIGH transition of this clock may not change the state of the SCL line if another clock is still within its LOW period. The SCL line will therefore be held LOW by the device with the longest LOW period. Devices with shorter LOW periods enter a HIGH wait-state during this time.

When all devices concerned have counted off their LOW period, the clock line will be released and go HIGH. There will then be no difference between the device clocks and the state of the SCL line, and all the devices will start counting their HIGH periods. The first device to complete its HIGH period will again pull the SCL line LOW. In this way, a synchronized SCL clock is generated with its LOW period determined by the device with the longest clock LOW period, and its HIGH period determined by the one with the shortest clock HIGH period

Using the Clock Synchronizing Mechanism as a Handshake

The I2C protocol also includes a synchronization mechanism. This can be used as a handshake mechanism between slow and fast devices or between masters in a multi-master session.

When a slow slave (slow in terms of internal execution) is attached to the bus then problems may occur. Let's consider a serial EEPROM. The actual writing process inside the EEPROM might take some time. Now if you send multiple bytes to such a device, the risk exists that you send new data to it before it has completed the write cycle. This would corrupt the data or cause data loss.

The slave must have some means to tell the master that it is busy. It could of course simply not respond to the ACK cycle. This would cause the master to send a stop condition and retry. (That's how it is done in hardware in EEPROMs.)

Other cases might not be so simple. Think about an A/D converter. It might take some time for the conversion to complete. If the master would just go on it would be reading the result of the previous conversion instead of the newly acquired data.

Now the synchronization mechanism can come in handy. This mechanism works on the SCL line only. The slave that wants the master to wait simply pulls the SCL low as long as needed. This is like adding "wait states" to the I2C bus cycle.

The master is then not able to produce the ACK clock pulse because it cannot get the SCL line to go high. Of course the master software must check this condition and act appropriately. In this case, the master simply waits until it can get the SCL line to go HIGH and then just goes on with whatever it was doing.

There are a number of minor drawbacks involved when implementing this. If the SCL gets stuck due to an electrical failure of a circuit, the master can go into deadlock. Of course this can be handled by timeout counters. Plus, if the bus gets stuck like this, the communication is not working anyway.

Another drawback is speed. The bus is locked at that moment. If you have rather long delays (long conversion time in our example above), then this penalizes the total bus speed a lot. Other masters cannot use the bus at that time either.

This technique does not interfere with the previously introduced arbitration mechanism because the low SCL line will lead to back-off situations in other devices which possibly would want to "claim" the bus. So there is no real drawback to this technique except the loss of speed / bandwidth and some software overhead in the masters.

You can use this mechanism between masters in a multi-master environment. This can prevent other master from taking over the bus. In a two-master system this is not useful. But as soon as you have three or more masters this is very handy. A third master cannot interrupt a transfer between master 1 and 2 in this way. For some mission-critical situations this can be a very nice feature.

You can make this technique rigid by not pulling only the SCL line low, but also the SDA line. Then any master other than the two masters talking to each other will immediately back off. Before you continue, you first let SCL go back high, and then SDA, representing a stop condition. Any master which attempted to communicate in the meantime would have detected a back-off situation and would be waiting for a STOP to appear.

Special Addresses and Exceptions

In the I2C address map there are so-called "reserved addresses". This section contains some more details on these addresses and what they do. For information about the Extended Addressing Mode, please refer to the corresponding chapter.

Address	R/W	Designation
0000-000	0	General call address (see note 1)
0000-000	1	START byte (note 2)
0000-001	x	Reserved for the obsolete C-Bus format (note 3)

Address	R/W	Designation
0000-010	x	Reserved for a different bus format (note 4)
0000-011	x	Reserved for future purposes (note 5)
0000-1xx	x	Reserved for future purposes
1111-1xx	x	Reserved for future purposes
1111-0xx	x	10-bit slave addressing mode (note 6)

Note 1: The general call address

This address is being used to access all devices on the bus which are capable of handling the general call and need this data. Devices which are capable of handling this general call but do not need it will not answer.

All bytes transferred after this address may or may not be taken in by the slaves that are responding to it. If no slave is acknowledging a transmitted byte, the operation is stopped by issuing a STOP on the bus.

The meaning of the general call address is specified in the 1st byte transmitted after this "general call". This first byte can contain the following information:

If the LSB is set to 0:

0000-0110	Reset and write programmable part of slave address. All devices who respond to this will reset and take in the programmable part of their address. This is done by re-reading the levels on the address select pins of the device (if any). This command can be used to reset an entire I2C system.
0000-0100	The same as above but without the reset . This can be useful if the state of the address select pins of a device are configurable. This way the device address will change.

If the LSB is set to 1:

xxxx-xxx1	This is a Hardware Call. If a device needs urgent attention from a master device without knowing which master it needs to turn to, it can use this call. This is a call "to whom it may concern". The device then embeds its own address into the message. This call means as much as: Please contact me, I need to be serviced. All masters will listen and the master that knows how to handle the device with the address transmitted will contact its slave and act appropriately.

Note 2: The START address

This can be used between masters. A master which does not have an I2C interface in hardware but in software needs to monitor the bus all the time. Since this can require a lot of processing time, the START address was introduced. The masters can sample the bus at a low rate. As soon as they detect that the SDA line is low (it is held low for over 7 clock periods) it can switch to a higher sampling rate to detect the Start condition.

This address is not followed by a stop condition but rather by a repeated start condition.

Notes 3 and 4: These addresses are used when data other than I2C data has to be transmitted over the bus.

Note 5: These addresses are for further expansion and are currently not allowed.

Note 6: This will be discussed in detail in the section about 10-bit addressing.

Enhanced I2C (FAST Mode)

Since the first I2C spec release (which dates back from 1982), a couple of improvements have been made. In 1992, a newer version of the I2C spec was released. This new spec contained some additional sections covering FAST mode and 10-bit addressing.

In the FAST mode, the physical bus parameters are not altered. The protocol, bus levels, capacitive load etc. remain unchanged. However, the data rate has been increased to 400 Kbit/s and a constraint has been set on the level of noise that can be present in the system. To accomplish this task, a number of changes have been made to the I2C bus timing.

Since all CBUS activities have been canceled, there is no compatibility anymore with CBUS timing. The development of ICs with CBUS interface has long been stopped. The existing CBUS IC's are discontinued. Furthermore, the CBUS devices cannot handle these higher clock rates.

The input of the FAST mode devices all include Schmitt triggers to suppress noise. The output buffers include slope control for the falling edges of the SDA and SCL signals. If the power supply of a FAST mode device is switched off, the bus pins must be floating so that they do not obstruct the bus.

The pull-up resistor must be adapted. For loads up to 200 pF, a resistor is sufficient. For loads between 200pF and 400pF, a current source (active pull-up) is preferred.

High-Speed I2C (HS-Mode)

High-speed mode (Hs-mode) devices offer a quantum leap in I2C-bus transfer speeds. Hs-mode devices can transfer information at bit rates of up to 3.4 Mbit/s, yet they remain fully downward compatible with Fast- or Standard-mode (F/S-mode) devices for bi-directional communication in a mixed-speed bus system. With the exception that arbitration and clock synchronization is not performed during the Hs-mode transfer, the same serial bus protocol and data format is maintained as with the F/S-mode system. Depending on the application, new devices may have a Fast or Hs-mode I2C-bus interface, although Hs-mode devices are preferred as they can be designed-in to a greater number of applications.

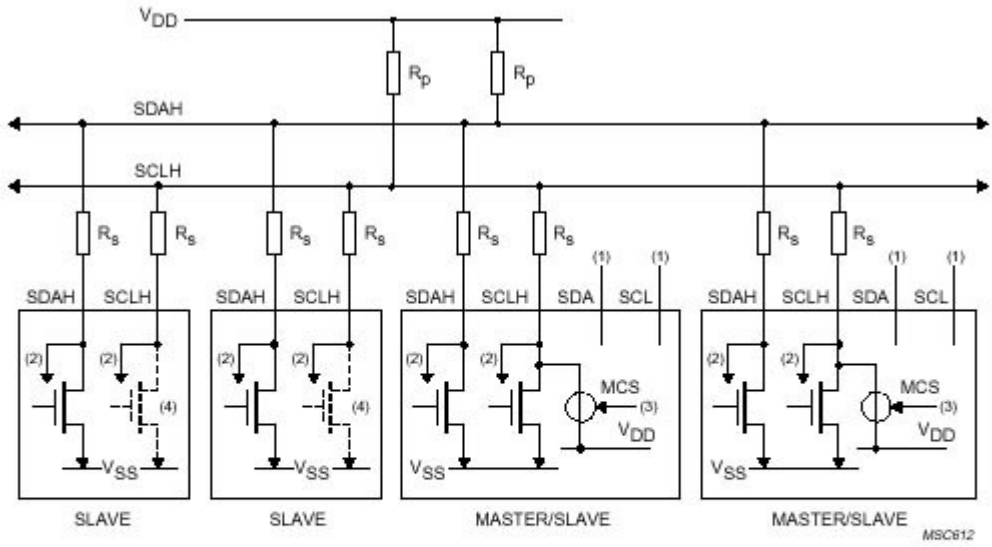
High speed transfer

To achieve a bit transfer of up to 3.4 Mbit/s the following improvements have been made to the regular I 2 C-bus specification:

- Hs-mode master devices have an open-drain output buffer for the SDAH signal and a combination of an open-drain pull-down and current-source pull-up circuit on the SCLH output (1) . This current-source circuit shortens the rise time of the SCLH signal. Only the current-source of one master is enabled at any one time, and only during Hs-mode.
- No arbitration or clock synchronization is performed during Hs-mode transfer in multi-master systems, which speeds-up bit handling capabilities. The arbitration procedure always finishes after a preceding master code transmission in F/S-mode.

- Hs-mode master devices generate a serial clock signal with a HIGH to LOW ratio of 1 to 2. This relieves the timing requirements for set-up and hold times.
- As an option, Hs-mode master devices can have a built-in bridge (1) . During Hs-mode transfer, the high speed data (SDAH) and high-speed serial clock (SCLH) lines of Hs-mode devices are separated by this bridge from the SDA and SCL lines of F/S-mode devices. This reduces the capacitive load of the SDAH and SCLH lines resulting in faster rise and fall times.
- The only difference between Hs-mode slave devices and F/S-mode slave devices is the speed at which they operate. Hs-mode slaves have open-drain output buffers on the SCLH and SDAH outputs. Optional pull-down transistors on the SCLH pin can be used to stretch the LOW level of the SCLH signal, although this is only allowed after the acknowledge bit in Hs-mode transfers.
- The inputs of Hs-mode devices incorporate spike suppression and a Schmitt trigger at the SDAH and SCLH inputs.
- The output buffers of Hs-mode devices incorporate slope control of the falling edges of the SDAH and SCLH signals.

The figure below shows the physical I 2 C-bus configuration in a system with only Hs-mode devices. Pins SDA and SCL on the master devices are only used in mixed-speed bus systems and are not connected in an Hs-mode only system. In such cases, these pins can be used for other functions.



(1) SDA and SCL are not used here but may be used for other functions.
(2) To input filter.
(3) Only the active master can enable its current-source pull-up circuit
(4) Dotted transistors are optional open-drain outputs which can stretch the serial clock signal SCLH.

Optional series resistors Rs protect the I/O stages of the I2C-bus devices from high-voltage spikes on the bus lines and minimize ringing and interference. Pull-up resistors Rp maintain the SDAH and SCLH lines at a HIGH level when the bus is free and ensure the signals are pulled up from a LOW to a HIGH level within the required rise time. For higher capacitive bus-line loads (>100 pF), the resistor Rp can be replaced by external current source pull-ups to meet the rise time requirements.

Unless proceeded by an acknowledge bit, the rise time of the SCLH clock pulses in Hs-mode transfers is shortened by the internal current-source pull-up circuit MCS of the active master.

Extended Addressing (10-bit)

Due to the increasing popularity of the I2C bus the 7-bit address space got exhausted. This started posing problems for people currently in the phase of designing a new I2C compatible IC. Therefore the I2C standard has been updated to implement a 10-bit addressing mode.

A chip that conforms to the new standard receives two address bytes. The first consists of the extended addressing reserved address including the 2 MSB's of the device address and the Read/Write bit. The second byte contains the 8 LSB's of the address.

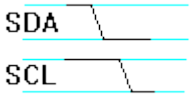
This scheme insures that the 10 bit addressing mode stays completely transparent for the other devices on the bus. Any new design should implement this new addressing scheme.

I2C Bus Events

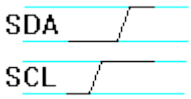
Start and Stop Conditions

Prior to any transaction on the bus, a START condition needs to be issued on the bus. The start condition acts as a signal to all connected IC's that something is about to be transmitted on the bus. As a result, all connected chips will listen to the bus.

After a message has been completed, a STOP condition is sent. This is the signal for all devices on the bus that the bus is available again (idle). If a chip was accessed and has received data during the last transaction, it will now process this information (if not already processed during the reception of the message).

Start 

The chip issuing the Start condition first pulls the SDA (data) line low, and next pulls the SCL (clock) line low.

Stop 

The Bus Master first releases the SCL and then the SDA line.

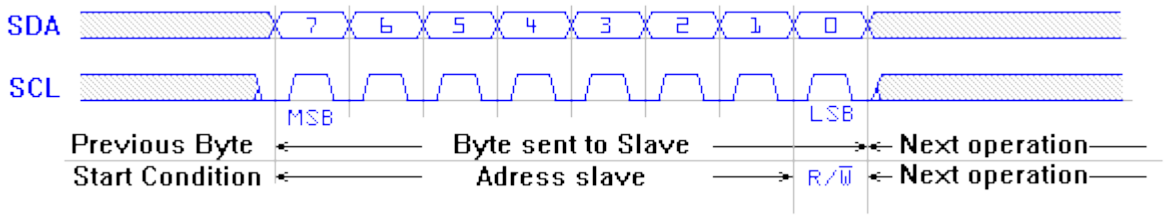
A few notes about start and stop conditions:

- A single message can contain multiple Start conditions. The use of this so-called "repeated start" is common in I2C.
- A Stop condition ALWAYS denotes the END of a transmission. Even if it is issued in the middle of a transaction or in the middle of a byte. It is "good behavior" for a chip that, in this case, it disregards the information sent and resumes the "listening state", waiting for a new start condition.

Transmitting a Byte to a Slave Device

Once the start condition has been sent, a byte can be transmitted by the MASTER to the SLAVE.

This first byte after a start condition will identify the slave on the bus (address) and will select the mode of operation. The meaning of all following bytes depends on the slave.



A number of addresses have been reserved for special purposes. One of these addresses is reserved for the "Extended Addressing Mode". As the I2C bus gained popularity, it was soon discovered that the number of available addresses was too small. Therefore, one of the reserved addresses has been allocated to a new task to switch to 10-bit addressing mode. If a standard slave (not able to resolve extended addressing) receives this address, it won't do anything (since it's not its address).

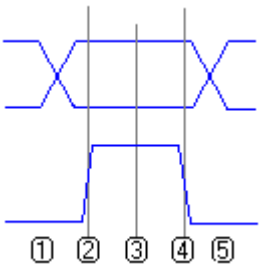
If there are slaves on the bus that can operate in the extended 10-bit addressing mode, they will ALL respond to the ACK cycle issued by the master. The second byte that gets transmitted by the master will then be taken in and evaluated against their address.

Note: Even in 10-bit extended addressing mode, Bit 0 of the first byte after the Start condition determines the slave access mode ('1' = read / '0' = write)

Receiving a Byte From a Slave Device

Once the slave has been addressed and the slave has acknowledged this, a byte can be received from the slave if the R/W bit in the address was set to READ (set to '1').

The protocol syntax is the same as in transmitting a byte to a slave, except that now the master is not allowed to touch the SDA line. Prior to sending the 8 clock pulses needed to clock in a byte on the SCL line, the master releases the SDA line. The slave will now take control of this line. The line will then go high if it wants to transmit a '1' or, if the slave wants to send a '0', remain low.

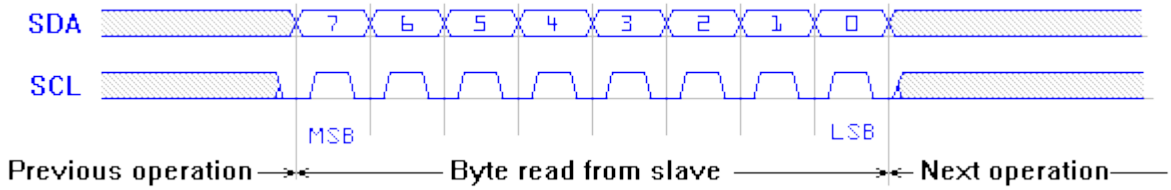


All the master has to do is generate a rising edge on the SCL line (2), read the level on SDA (3) and generate a falling edge on the SCL line (4). The slave will not change the data during the time that

SCL is high. (Otherwise a Start or Stop condition might inadvertently be generated.)

During (1) and (5), the slave may change the state of the SDA line.

In total, this sequence has to be performed 8 times to complete the data byte. Bytes are always transmitted MSB first.

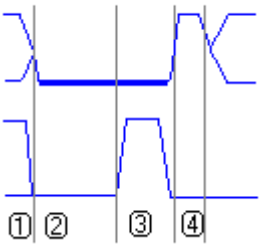


The meaning of all bytes being read depends on the slave. There is no such thing as a "universal status register". You need to consult the data sheet of the slave being addressed to know the meaning of each bit in any byte transmitted.

Getting Acknowledge from a Slave Device

When an address or data byte has been transmitted onto the bus then this must be ACKNOWLEDGED by the slave(s). In case of an address: If the address matches its own then that slave and only that slave will respond to the address with an ACK. In case of a byte transmitted to an already addressed slave then that slave will respond with an ACK as well.

The slave that is going to give an ACK pulls the SDA line low immediately after reception of the 8th bit transmitted, or, in case of an address byte, immediately after evaluation of its address. In practical applications this will not be noticeable.



This means that as soon as the master pulls SCL low to complete the transmission of the bit (1), SDA will be pulled low by the slave (2).

The master now issues a clock pulse on the SCL line (3). the slave will release the SDA line upon completion of this clock pulse (4).

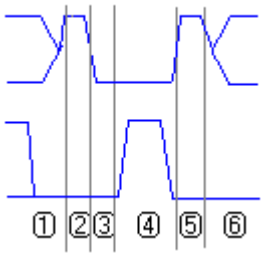
The bus is now available again for the master to continue sending data or to generate a stop condition.

In case of data being written to a slave, this cycle must be completed before a stop condition can be generated. The slave will be blocking the bus (SDA kept low by slave) until the master has generated a clock pulse on the SCL line.

Giving Acknowledge to a Slave Device

Upon reception of a byte from a slave, the master must acknowledge this to the slave device.

The master is in full control of the SDA and the SCL line.



After transmission of the last bit to the master (1) the slave will release the SDA line.

The SDA line should then go high (2). The Master will now pull the SDA line low (3) .

Next, the master will put a clock pulse on the SCL line (4). After completion of this clock pulse, the master will again release the SDA line (5).

The slave will now regain control of the SDA line (6).

Note: The above waveform is slightly exaggerated. You will not notice SDA going high in (2) and (5). A small spike might barely be visible.

Note: An Acknowledge of a byte received from a slave is always necessary, EXCEPT on the last byte received.

If the master wants to stop receiving data from the slave, it must be able to send a stop condition.

Since the slave regains control of the SDA line after the ACK cycle issued by the master, this could lead to problems.

Let's assume the next bit ready to be sent to the master is a 0. The SDA line would be pulled low by the slave immediately after the master takes the SCL line low. The master now attempts to generate a Stop condition on the bus. It releases the SCL line first and then tries to release the SDA line - which is held low by the slave. Conclusion: No Stop condition has been generated on the bus.

This condition is called a NACK : Not ACKnowledge . Do not confuse this with No ACKnowledge:

Condition	Can Only Occur...
Not acknowledge (NACK)	After a master has read a byte from a slave
No acknowledge	After a master has written a byte to a slave

No Acknowledge Condition

This is not exactly a condition. It is merely a state in the data flow between master and slave.

If, after transmission of the 8th bit from the master to the slave the slave does not pull the SDA line low, then this is considered a No ACK condition.

This means that either :

- The slave is not there (in case of an address)
- The slave missed a pulse and got out of sync with the SCL line of the master.
- The bus is "stuck". One of the lines could be held low permanently.

In any case the master should abort by attempting to send a stop condition on the bus.

A test for a "stuck bus" can be performed in the stop condition cycle.

Overview of the Different Versions of the I2C Specification

The very first specification dates back to the year 1982. It only covered Standard mode (up to 100 kbit/s) and 7-bit addressing. Extensions like Fast Mode, Hs-Mode or 10-bit addressing were added in later versions.

Version 1.0 - 1992

This version of the 1992 I2C-bus specification included the following modifications:

- Programming of a slave address by software has been omitted. The realization of this feature was rather complicated and had not been used.
- The "low-speed mode" has been omitted. This mode was, in fact, a subset of the total I2C-bus specification and did not need to be specified explicitly.
- The Fast-mode was added. This allows a fourfold increase of the bit rate up to 400 kbit/s. Fast-mode devices are downwards compatible i.e. they can be used in a 0 to 100 kbit/s I2C-bus system.
- 10-bit addressing was added. This allows 1024 additional slave addresses.
- Slope control and input filtering FOR FAST-MODE DEVICES was specified to improve the EMC behavior. NOTE: Neither the 100 kbit/s I2C-bus system nor the 100 kbit/s devices have been changed.

Version 2.0 - 1998

As the I2C-bus became a de facto world standard implemented in over 1000 different ICs and licensed to more than 50 companies, an update of the specification became necessary as many of the newer applications required higher bus speeds and lower supply voltages. This version 2.0 of the I2C-bus specification met those requirements and included the following modifications:

- The High-speed mode (Hs-mode) was added. This allows an increase in the bit rate up to 3.4 Mbit/s. Hs-mode devices can be mixed with Fast- and Standard-mode devices on the one I2C-bus system with bit rates from 0 to 3.4 Mbit/s.
- The low output level and hysteresis of devices with a supply voltage of 2 V and below has been adapted to meet the required noise margins and to remain compatible with higher supply voltage devices.
- The 0.6 V at 6 mA requirement for the output stages of Fast-mode devices has been omitted.
- The fixed input levels for new devices were replaced by bus voltage-related levels.
- Application information for bi-directional level shifter was added.

Version 2.1 - 2000

Version 2.1 of the I2C-bus specification is the most current version. It includes the following minor modifications:

- After a repeated START condition in Hs-mode, it is possible to stretch the clock signal SCLH.
- Some timing parameters in Hs-mode have been relaxed.

I2C Driver in Pseudocode

It is written in PseudoCode which is an imaginary programming language that any programmer should be capable of porting to his/her favorite language.

First we will define a set of basic interface routines. All text between / / is considered as remark.

Following variables are used :

- n,x = a general purpose BYTE
- SIZE = a byte holding the maximum number of transferred data at a time
- DATA(SIZE) = an array holding up to SIZE number of bytes. This will contain the data we want to transmit and will store the received data.
- BUFFER = a byte value holding immediate received or transmit data.

```
/ $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ /
/ **** I2C Driver V1.1 Written by V.Himpe. Released as Public Domain **** /
/ $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ /
DECLARE N,SIZE,BUFFER,X Byte
DECLARE DATA() Array of SIZE elements
SUBroutine I2C_INIT / call this immediately after power-on /
```

```
7.      SDA=1
8.      SCK=0
9.      FOR n = 0 to 3
10.         CALL STOP
11.     NEXT n
12.ENDsub
13.SUBroutine START
14.     SCK=1 / BUGFIX !/
15.     SDA=1 / Improvement /
16.     SDA=0
17.     SCK=0
18.     SDA=1
19.ENDsub
20.SUBroutine STOP
21.     SDA=0
22.     SCK=1
23.     SDA=1
24.ENDsub
25.SUBroutine PUTBYTE(BUFFER)
26.     FOR n = 7 TO 0
27.         SDA= BIT(n) of BUFFER
28.         SCK=1
29.         SCK=0
30.     NEXT n
31.     SDA=1
32.ENDsub
33.SUBroutine GETBYTE
34.     FOR n = 7 to 0
35.         SCK=1
36.         BIT(n) OF BUFFER = SDA
37.         SCK=0
38.     NEXT n
39.     SDA=1
40.ENDsub
41.SUBroutine GIVEACK
42.     SDA=0
43.     SCK=1
44.     SCK=0
45.     SDA=1
46.ENDsub
47.SUBroutine GETACK
48.     SDA=1
49.     SCK=1
50.     WAITFOR SDA=0
```

```
51.     SCK=0
52.ENDSUB
53./ this concludes the low-level set of instructions for the I2C driver
54.The next functions will handle the telegram formatting on a higher level /
55.SUBroutine READ(Device_address,Number_of_bytes)
56.     Device_adress=Device_adress OR (0000.0001)b /This sets the READ FLAG/
57.     CALL START
58.     CALL PUTBYTE(Device_adress)
59.     CALL GETACK
60.     FOR x = 0 to Number_of_bytes
61.          CALL GETBYTE DATA(x)=BUFFER /Copy received BYTE to DATA array /
62.          IF X< Number_of_bytes THEN /Not ack the last byte/
63.               CALL GIVEACK
64.          END IF
65.     NEXT x
66.     CALL STOP
67.ENDsub
68.SUBroutine WRITE(Device_address,Number_of_bytes)
69.     Device_adress=Device_adress AND (1111.1110)b / This clears READ flag /
70.     CALL START
71.     CALL PUTBYTE(Device_adress)
72.     CALL GETACK
73.     FOR x = 0 to Number_of_bytes
74.          CALL PUTBYTE (DATA(x))
75.          CALL GETACK
76.     NEXT x
77.     CALL STOP
78.ENDsub
79.SUBroutine RANDOMREAD(Device_adress,Start_adress,Number_of_bytes)
80.     Device_adress=Device_adress AND (1111.1110)b / This clears READ flag /
81.     CALL START
82.     CALL PUTBYTE(Device_adress)
83.     CALL GETACK
84.     CALL PUTBYTE(Start_adress)
85.     CALL GETACK
86.     CALL START /create a repeated start condition/
87.     Device_adress=Device_adress OR (0000.0001)b /This sets the READ FLAG/
88.     CALL PUTBYTE(Device_adress)
89.     CALL GETACK
90.     FOR x = 0 to Number_of_bytes
91.          CALL GETBYTE
92.          DATA(x)=BUFFER
93.          CALL GIVEACK
94.     NEXT x
```

```
    CALL STOP
ENDsub
SUBroutine RANDOMWRITE(Device_adress,Start_adress,Number_of_bytes)
    Device_adress=Device_adress AND (1111.1110)b / This clears READ flag /
    CALL START
    CALL PUTBYTE(Device_adress)
    CALL GETACK
    CALL PUTBYTE(Start_adress)
    CALL GETACK
    FOR x = 0 to Number_of_bytes
        CALL PUTBYTE (DATA(x))
        CALL GETACK
    NEXT x
    CALL STOP
ENDsub
/ $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ /
/ **** I2C Driver . (c)95-97 V.Himpe . Public Domain release *** /
/ $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ /
```

Some notes about the high level routines.

The READ and WRITE routine read / write one or more byte(s) from / to a slave device. Generally this will be used only with Number_of_bytes set to 1. An example:

```
PCD8574=(0100.0000)b
CALL READ(PCD8574,1)
result = DATA(0)
/ will read the status of the 8 bit input port of a PCD8574. /
DATA(0)=(0110.01010)b
CALL WRITE(PCD8574,1)
/ will write 0110.0101 to the 8 bit port of the PCD8574 /
```

When do you need a multi-read ? Consider a PCF8582 EEPROM. You want to read its contents in one time.

```
PCF8582=(1010.0000)b
CALL READ(PCF8582,255)
```

You can do the same with WRITE for the EEPROM with the restriction that Number_of_bytes is not larger than 4 AND that you write on page boundaries (multiple of 4 for offset).

You will have to check the components datasheets.

The most useful instructions are RANDOMREAD and RANDOMWRITE.

Write 4 bytes of data to location 20h of the EEPROM:

1. DATA(0)=(1010.0011)b
2. DATA(1)=(1110.0000)b
3. DATA(2)=(0000.1100)b
4. DATA(3)=(1111.0000)b
5. CALL RANDOMWRITE (PCF8582,(20)h,3)

The same goes for reading 16 bytes from the EEPROM starting at address 42h:

1. CALL RANDOMREAD(PCF8582,(42)h,15)

The results are stored in DATA. All you have to do is read them out of the array for processing. When you give the devices address to these routines you don't have to care about the R/W flag. It will be automatically set to the right state inside the routines.