I2C – What’s That?

I2C-Bus: What’s that?

The I2C bus was designed by Philips in the early ’80s to allow easy communication between components which reside on the same circuit board. Philips Semiconductors migrated to [NXP](http://www.nxp.com/)in 2006.

The name I2C translates into “Inter IC”. Sometimes the bus is called IIC or I²C bus.

The original communication speed was defined with a maximum of 100 kbit per second and many applications don’t require faster transmissions. For those that do there is a [400 kbit fastmode](https://www.i2c-bus.org/fastmode/) and – since 1998 – a [high speed 3.4 Mbit](https://www.i2c-bus.org/highspeed/) option available. Recently, [*fast mode plus*](https://www.i2c-bus.org/fast-mode-plus/) a transfer rate between this has been specified.  Beyond this, there is the [ultra fast mode UFM](https://www.i2c-bus.org/ultra-fast-mode-ufm/), but it frankly is no real I2C bus.

I2C is not only used on single boards but also to connect components which are linked via cable. Simplicity and flexibility are key characteristics that make this bus attractive to many applications.

Most significant features include:

* Only two bus lines are required
* No strict baud rate requirements like for instance with RS232, the master generates a bus clock
* Simple master/slave relationships exist between all components  
  Each device connected to the bus is software-addressable by a unique address
* I2C is a true [multi-master](https://www.i2c-bus.org/MultiMaster/) bus providing arbitration and collision detection

Want to learn more?

* Check our [I2C Primer](https://www.i2c-bus.org/i2c-primer/).
* The latest [I2C specification](https://www.i2c-bus.org/specification/) is available directly from NXP. Currently, it is the revision 6 dated 4th of April 2014
* Have a look at NXP’s [overview of I2C devices](http://www.nxp.com/products/interface_and_connectivity/i2c/)
* There are a number of I2C-like buses, [see Definitions and Differences Between I2C, ACCESS.bus and SMBus](http://ww1.microchip.com/downloads/en/AppNotes/an617.pdf).

Looking for I2C tools and software?

Whether you need a [high speed I2C](http://www.telos.info/p/hw/traciixl20/) solution or a [USB I2C interface](http://www.telos.info/p/hw/conniimm20/), whether you want to [monitor the bus](http://www.telos.info/p/hw/traciixl20/) or [need a software platform](http://www.telos.info/p/sw/i2cfw/) for your own application, [we can help you!](http://www.telos.info/details/contact/).

# I2C Primer

In order to use the I2C bus properly and efficiently, it is important to understand what happens in the hardware on the electric level. This essay explains

* the hardware setup of an I2C bus
* the requirements for proper operation of the I2C bus
* how to find the cause of problems and how to solve them

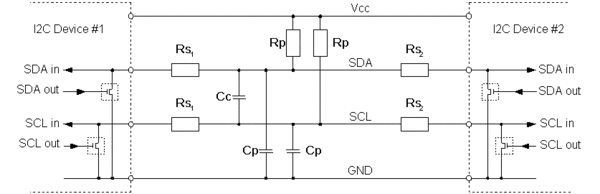
It is assumed that the reader understands the protocol basics of I2C and is thus familiar with terms like I2C master and slave, (re-)start and stop conditions and acknowledges.

For details see the [I2C specification](http://www.nxp.com/documents/user_manual/UM10204.pdf) from NXP, formerly Philips. 2014 NXP released the new version *Rev. 6 — 4 April 2014*.

Note: High-speed I2C has additional aspects which are not addressed here.

The pictures shown in the following sections have been made with [Tracii XL 2.0](http://www.telos.info/p/hw/traciixl20/).

# Typical Setup



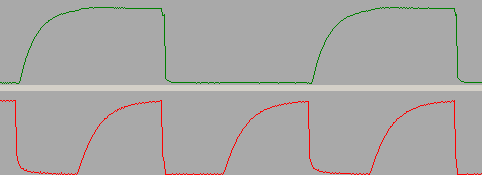
The image shows a simplified equivalent circuit diagram for an I2C connection between two devices (master or slave) containing all relevant factors for I2C.

|  |  |
| --- | --- |
| VCC | I2C supply voltage, typically ranging from 1.2 V to 5.5 V |
| GND | Common ground |
| SDA | Serial data (I2C data line) |
| SCL | Serial clock (I2C clock line) |
| Rp | Pull-up resistance (a.k.a. I2C termination) |
| Rs | Serial resistance |
| Cp | Wire capacitance |
| Cc | Cross channel capacitance |

# How I2C Hardware Works

The I2C bus transmits data and clock with SDA and SCL. The first thing to realize: SDA and SCL are *open-drain* (also known as *open-collector* in the TTL world), that is I2C master and slave devices can only drive these lines low or leave them open. The *termination resistor* Rp pulls the line up to Vcc if no I2C device is pulling it down. This allows for features like the concurrent operation of more than one I2C master (if they are *multi-master* capable) or *stretching* (slaves can slow down communication by holding down SCL).

Together with the wire capacitance Cp, the termination resistor Rp affects the temporal behaviour of the signals on SDA and SCL. While I2C devices pull down the lines with open drain drivers or FETs which can in general drive at least about 10mA or more, the pull-up resistor Rp is responsible for getting the signal back to high level. Rp commonly ranges from 1 kΩ to 10 kΩ, resulting in typical pull-up currents of about 1 mA and less. This is the reason for the characteristic sawtooth-like look of I2C signals. In fact, every ‘tooth’ shows the charge characteristic of the line on the rising edge and the discharge characteristic on the falling edge.

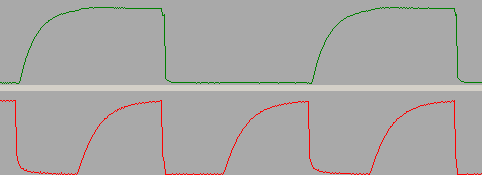
  
SDA (above) and SCL (below) with Rp = 10 kΩ and Cp = 300 pF. The SCL clock runs with 100 kHz (nominal).

# Termination Versus Capacitance

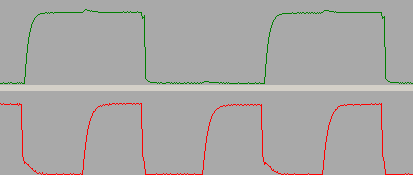
Cp and Rp effectively limit the maximum data rate which can be transferred over SDA and SCL. A high Cp can be compensated with a low Rp and vice versa.

The following three pictures show the same part of an I2C transfer. Compared to the first one, the second picture shows the signals with a modified Rp of 2 kΩ, the third one with a lower Cp of 150 pF.

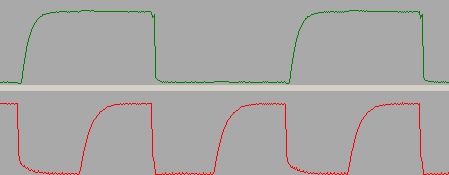
Note: Long wires increase Cp dramatically. I2C connections should always be as short as possible and connected by a suitable wiring pattern (c.f. I2C specification, section 17.3).

****

SDA (above) and SCL (below) with Rp = 10 kΩ and Cp = 300 pF. The SCL clock runs with 100 kHz (nominal).

****

The same transfer as above, but this time with a reduced termination resistance (Rp = 2 kΩ, Cp = 300 pF)

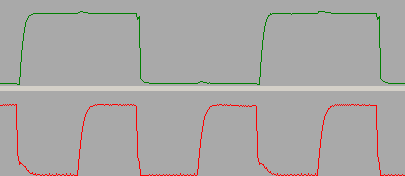
****

The same transfer as above, but this time with a reduced wire capacitance (Rp = 10 kΩ, Cp = 150 pF)

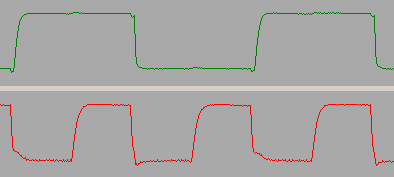
Note that the I2C standard limits Cp to the maximum value of 400 pF. However, with an appropriate termination resistance, it is often possible (although not recommended) to operate I2C buses with higher capacitance.

Usually, if the I2C bus is used solely on one board, serial resistors Rs are very low or not existing at all. In some circuits, e.g. if the I2C bus is externally accessible by a connector, it is desirable to add serial resistors in the SDA and SCL line in order to protect the I2C devices against overcurrent.

Together with the termination resistance Rp, the serial resistance Rs affects the static low level of the I2C lines. The pictures below show parts of a master transmitter transfer on an I2C bus with the serial resistances Rs1 = 250 Ω and Rs2 = 0 Ω, and a pull-up resistance Rp = 1 kΩ. The first one shows the transfer as measured at the clamps of I2C device #1 (here: the master), the second one as measured at the clamps of I2C device #2 (here: the slave). Note the different low-level voltages; the low level measured at the slave is about Rs / (Rs + Rp) \* Vcc = ~1/5 \* Vcc while the low level measured at the master is nearly GND.

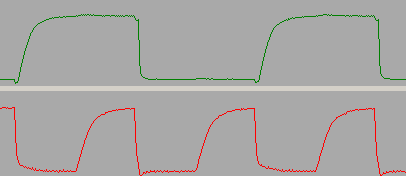
****

SDA (above) and SCL (below) as measured at I2C device #1 (master)

****

Same transfer as above, measured at I2C device #2 (slave)

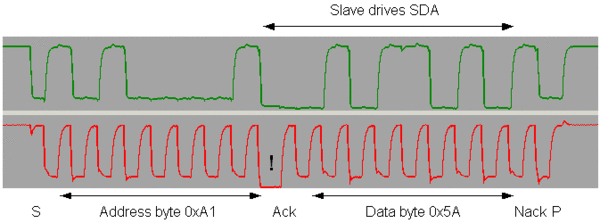
A high low level on SCL and SDA can lead to problems under certain circumstances as some devices may not properly recognize this as a valid low level. If Rs cannot be reduced for some reason, it helps to raise the termination resistance Rp instead. The following picture shows the transfer at the slave clamps with an increased termination resistance of Rp = 10 kΩ.

****

Same transfer as above, measured at I2C device #2 (slave) with increased Rp

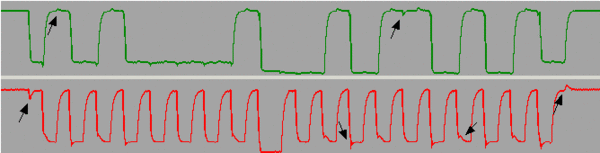
# Serial Resistance And Debugging

Despite their obvious downsides, serial resistances can be useful sometimes when analyzing I2C bus traffic to track down I2C-related problems. As stated above, serial resistances Rs lead to different voltage levels during the low phase, depending on which device currently pulls SDA or SCL low. Thus, it is possible to recognize which I2C device is currently active by analyzing the low level on the I2C lines.The picture below shows a complete master receiver transfer between an I2C master and a slave device. Due to serial resistors, the low level of the slave device is lower than the one of the master device. This applies for both lines, SDA and SCL. Thus, it is possible to see easily that the slave device stretches the SCL line shortly during the acknowledge clock cycle (marked with “!”).

  
A complete master receiver transfer between an I2C master and I2C slave device with Rs = 200 Ω, measured at the slave device.

# Crosstalk Between SDA And SCL

The image below shows a typical I2C transmission. A closer look at the signals discloses small spikes in the signals. These are consequences arising out of cross channel capacitances Cc.In general, crosstalk should not be a problem as long as the resulting spikes are not too large. Cc itself can be minimized by using as short interconnections as possible. Increasing the serial resistors Rs and termination Rp can help to reduce the effects of the cross channel capacitances Cc.

  
Crosstalk between SDA and SCL

# Clock, Stretching, Arbitration

## Clock Generation

The SCL clock is always generated by the I2C master. The specification requires minimum periods for the low and high phases of the clock signal. Hence, the actual clock rate may be lower than the nominal clock rate e.g. in I2C buses with large rise times due to high capacitances.

## Clock Stretching

I2C devices can slow down communication by stretching SCL: During an SCL low phase, any I2C device on the bus may additionally hold down SCL to prevent it from rising again, enabling it to slow down the SCL clock rate or to stop I2C communication for a while. This is also referred to as clock synchronization.

Note: The I2C specification does not specify any timeout conditions for clock stretching, i.e. any device can hold down SCL as long as it likes.

## Arbitration

Several I2C multi-masters can be connected to the same I2C bus and operate concurrently. By constantly monitoring SDA and SCL for start and stop conditions, they can determine whether the bus is currently idle or not. If the bus is busy, masters delay pending I2C transfers until a stop condition indicates that the bus is free again.

However, it may happen that two masters start a transfer at the same time. During the transfer, the masters constantly monitor SDA and SCL. If one of them detects that SDA is low when it should actually be high, it assumes that another master is active and immediately stops its transfer. This process is called arbitration.

Requirements For Devices

The I2C specification poses several requirements to I2C devices in order to ensure proper operation in different environments. The most important ones are:

* Both, SDA and SCL, must be open drain and must not be driven high by any device attached to the I2C bus.
* In most I2C buses, the low and high input voltage level thresholds of SDA and SCL must depend on Vcc. E.g. an SDA voltage level of 1.1 V will be interpreted as low in an I2C bus with Vcc = 5 V and as high in an I2C bus with Vcc = 1.2 V.
* The SCL and SDA signals must be sampled by Schmitt Trigger inputs, i.e. with a certain hysteresis.
* Spikes in SCL and SDA signals must be filtered up to a certain amount (only for full speed I2C).
* Setup and hold times; this includes a specified maximum SCL clock rate (100 kHz for normal speed, 400 kHz for full speed).

Most off-the-shelf standard I2C ICs fulfill these requirements while e.g. I2C software implementations in microcontrollers often do not. This does not necessarily need to be a problem as long as the environment does not require a feature that one of the devices on the I2C bus does not support, but has to be kept in mind when tracking down I2C bus-related problems.

Common Problems In Systems

Often, problems in I2C bus systems have a simple cause and can easily be verified by checking with simple equipment.

* *Is there a stable supply voltage Vcc?* Often, external I2C devices (like I2C masters or monitors) must be provided with Vcc.
* *Are appropriate termination resistances attached between SDA, SCL and Vcc?* The voltage level on SDA and SCL must be Vcc as long as the bus is idle and drop near GND if shorted to GND. [Note: Very few I2C masters exist which drive SCL high and low, i.e. the SCL line is not open-drain. In this case, a termination resistor is not needed and SCL cannot be pulled low. These masters will not work together with other masters (as they have no multi-master support) and may not be used with devices which stretch SCL during transfers.]
* *Are SDA and SCL mixed up?* This may accidentally happen e.g. when connecting I2C buses with cables or connectors.
* *Do all I2C devices support the I2C supply voltage used on the bus?*
* *Do all I2C devices support the maximum SCL clock rate used on the bus?*
* If more than one I2C master is connected to the bus: *do all masters provide multi-master support?*

Further diagnostics is possible with Vcc, SCL and SDA connected to an oscilloscope and/or an appropriate I2C monitor device:

* *Are the high and low-level voltages on SDA and SCL correct during I2C transfers?* The I2C standard defines the low-level threshold with 0.3 Vcc, the high-level threshold with 0.7 Vcc. Modifying the termination resistance Rp, the serial resistors Rs or lowering the SCL clock rate could help here.
* *Are there spikes or noise on SDA, SCL or even Vcc?* They may result from interferences from other components or because the capacitances Cp and/or Cc are too high. The effects can often be reduced by using shorter interconnections.

Obscure Problems In Systems

Additional problems may arise if I2C devices which do not fully conform to the I2C specification are used on the bus. They are often hard to find without appropriate equipment like sophisticated I2C monitors or logic analysers, especially if they are not easily reproducible. Potential sources for non-obvious problems are:

* Some I2C masters (especially if implemented in software) have no multi-master support and/or cannot be used with certain devices which stretch SCL during transfers. Note that fully functioning multi-master support in software-implemented I2C masters is not trivial.
* Very simple I2C master implementations may not even recognize error conditions like negative acknowledges from slave devices.
* Some I2C devices do not use the correct voltage thresholds to recognize high and low levels on SDA and SCL, leaving them especially vulnerable to wrong combinations of serial and termination resistors.
* Some I2C devices do not have Schmitt trigger inputs and/or spike filters, which makes them especially susceptible to crosstalk and noise.
* Some I2C devices cancel an I2C transfer after a certain period of inactivity on the I2C bus and consider the bus idle afterwards. The I2C specification does not specify any timeout conditions – an I2C device can occupy the bus for an arbitrary time period.

# Analysing Obscure Problems

As already mentioned, the causes for some problems are hard to track down. In general, these problems only occur occasionally and often are difficult to reproduce and analyse without adequate tools like I2C monitors.

[Tracii XL 2.0](http://www.telos.info/p/hw/traciixl20/) is an I2C monitor which can detect I2C error conditions that are very hard or even nearly impossible to find with usual oscilloscopes, it helps to analyse them by triggering oscilloscopes or recording digital and even analog traces of SDA and SCL.

# Blocked I2C Bus

## Did you ever see a blocked I2C bus?

No communication after reset?

There are many reasons for a micro controller to go through a reset condition. Watchdog triggering, user events and intentional reconfiguration are only some examples.

For systems which employ an I2C bus such reset of a micro controller may have rather unexpected side effects if other components continue to operate during the reset phase.

I2C slaves work as a state machine. After they have been addressed they keep receiving bytes until they see another start condition or a stop condition send by the I2C bus master.

But what if the I2C master goes through a reset right in the middle of transmitting or receiving a byte from the I2C bus slave?  
The slave is not aware of the reset and since I2C doesn’t really define timeouts it may well wait for the next clock event to send or receive a bit, not listening to any start condition which is likely to occur after a reset as startup sequence.

**In other words, the bus is stuck, your I2C bus is blocked.**

One rather clumsy but easy to implement solution is to toggle the clock line multiple (16) times before doing any I2C operation after power-up of the micro controller i.e. after it has possibly gone through reset. This sequence can be followed by a stop condition.  
In many cases this will advance the state machine of any blocked slave to a point where it accepts the next start condition again.  
A safer but more complex solution is to power cycle all I2C devices after a reset condition.

The key point to keep in mind is that an I2C slave is not stateless and therefore may not function properly out of the box after reset or – equally common – a spike caused by EMC influence interpreted as clock signal.

No Acknowledge From I2C Slave

During a master transmitter transfer, the slave suddenly does not acknowledge the bytes from the master anymore. This may happen directly after the address byte or later during the data transfer stage. Possible reasons are:

* The I2C slave could not correctly interpret the data on SDA because the SDA high or low-level voltages do not reach its appropriate input thresholds.
* The I2C slave missed an SCL cycle because the SCL high or low-level voltages do not reach its appropriate input thresholds.
* The I2C slave accidently interpreted a spike etc. as an SCL cycle.

With adequate serial resistors between master and slave, an analog shot of the signals at the slave’s SDA and SCL pins provides a clue whether the slave acknowledges and to which SCL clock pulse. The different SDA low levels due to the serial resistor make it possible to distinguish acknowledges from the slave from data bits from the master.

# Master Reports Arbitration Lost

Sometimes during an I2C transfer, the master reports “arbitration lost” or something similar and cancels the transfer, although there is no other active master on the bus.

Possible reasons are the same as the ones described in “No Acknowledge From I2C Slave”, but here they provoke the slave to pull down SDA when it should not. The master detects this as an arbitration lost event and stops the transfer.

Data Bytes From Slave Are 0xff

A slave addressed by a master receiver transfer acknowledges the address byte and starts to send its data correctly, but suddenly all bytes sent by the slave are 0xff. Possible reasons are:

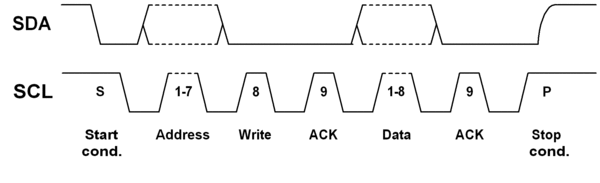
* The I2C slave missed an SCL cycle because the SCL high or low-level voltages do not reach its appropriate input thresholds.
* The I2C slave accidently interpreted a spike etc. as an SCL cycle.

After having received the last byte during a master receiver transfer, the master leaves SDA high during the acknowledge clock cycle – this causes the slave to not send data anymore, regardless whether a stop condition follows or not. Here, the slave is not synchronized anymore with the master and interprets a high SDA as a “not acknowledge” from the master.

# Addressing

The first byte of an I2C transfer contains the slave address and the data direction.

The address is 7 bits long, followed by the direction bit. Like all data bytes, the address is transferred with the most significant bit first.

  
A seven bit wide address space theoretically allows 128 I2C addresses – however, some addresses are reserved for special purposes. Thus, only 112 addresses are available with the 7 bit address scheme. To get rid of this a special method for using [10 bit addresses is defined](https://www.i2c-bus.org/addressing/10-bit-addressing/).

The following table shows I2C addresses reserved for special purposes:

|  |  |
| --- | --- |
| 10 bit adresses, binary noted, MSB is left | Purpose |
| 0000000 0 | [General Call](https://www.i2c-bus.org/addressing/general-call-address/) |
| 0000000 1 | [Start Byte](https://www.i2c-bus.org/addressing/start-byte/) |
| 0000001 X | [CBUS Addresses](https://www.i2c-bus.org/addressing/cbus-addresses/) |
| 0000010 X | Reserved for [Different Bus Formats](https://www.i2c-bus.org/addressing/different-bus-formats/) |
| 0000011 X | Reserved for future purposes |
| 00001XX X | [High-Speed Master Code](https://www.i2c-bus.org/addressing/high-speed/) |
| 11110XX X | [10-bit Slave Addressing](https://www.i2c-bus.org/addressing/10-bit-addressing/) |
| 11111XX X | Reserved for future purposes |

# General Call Address

The general call addresses all devices on the bus using the [I2C address 0](https://www.i2c-bus.org/addressing/).

If a device does not need the information provided, it simply does nothing. Devices processing the message acknowledge this address and behave as slave receiver.

The master cannot detect how many devices are using the message.

The second byte contains a command. The possible commands are described in the [I2C specification](https://www.i2c-bus.org/references/#c3153) chapter 3.13. The value 0 e.g. is a Software Reset.

Not every microcontroller connected to the I2C bus has an integrated I2C controller. These microcontrollers have to observe the I2C lines permanently to detect an I2C transmission. This consumes, mostly done by polling, much CPU time. To reduce this waste of CPU power, an I2C transfer can be established with a slower arbitration method.

For this, the master transmits the start-condition, followed by the start byte (‘00000001’), a dummy acknowledge pulse and a repeated start condition. The observing microcontroller has to detect only one of the seven zeros on SDA to detect an I2C transmission. This can be done with a relatively slow polling rate. As soon as the controller detects that SDA is low, it can switch to a higher polling rate in order to await the repeated start condition and the following I2C transfer.

After the transfer has ended, it can switch back to the CPU-power saving slow polling rate in order to detect the next transmission.

# CBUS Addresses

The CBUS is a three wire bus, using a different transmission format than I2C. To be able to connect CBUS receivers to an I2C bus, this special address has been reserved (‘0000001X’). I2C devices have to ignore messages to this address.

The CBUS is no longer in use.

# Different Bus Formats

The address ‘0000010X’ is intended to interconnect I2C devices with devices using different protocols on the same bus. Only I2C devices which are able to operate using these protocols are allowed to reply to the messages.

High Speed

The addressing scheme for [high speed](https://www.i2c-bus.org/highspeed/) transfers differs from the normal addressing procedure.

After the start condition, a so-called master code is transmitted ‘00001XXX’, followed by a mandatory not-acknowledge bit. The master code is sent in [Fast-](https://www.i2c-bus.org/fastmode/) or [Standard-mode](https://www.i2c-bus.org/standard-mode/) (this is with at most 400bkit/s).

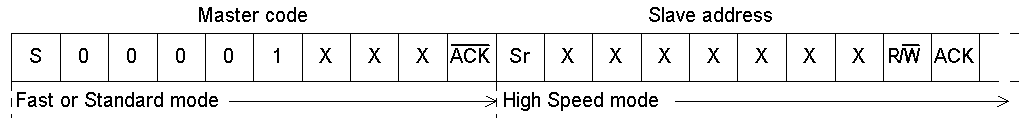
The three lowest bits are used to identify different I2C masters on the same bus – each one has its unique identifier. During transmission of the master code, [arbitration](https://www.i2c-bus.org/MultiMaster/) takes place, so that only the winning master can perform the following high speed transfer.

The master codes are selectable by the designer and allow up to eight high speed masters to be connected in one system (master code ‘00001000’ should be reserved for test and diagnostic purposes).

After the acknowledge phase following the master code, the high speed transfer begins with a repeated start condition, followed by the slave address and the succeeding data, just like in Fast or Standard mode, but with a higher bit rate.

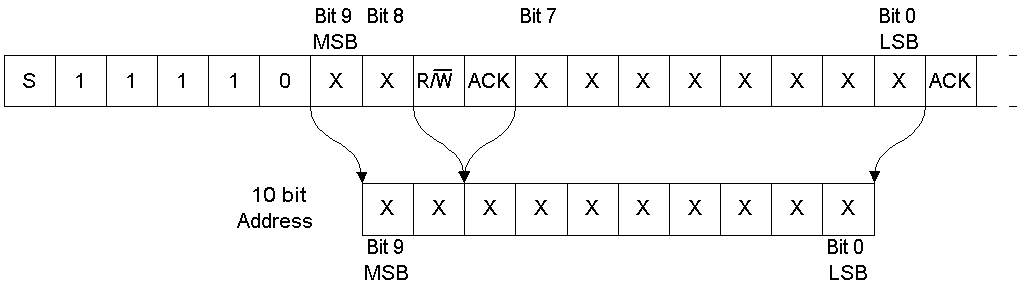
The high speed mode remains active until a stop condition is transmitted, on which the connected high speed devices switch back to slow transmission rates like Fast or Standard mode.

The picture below shows the beginning of a high speed transfer.



# 10 Bit Addressing

In order to prevent address clashes, due to the limited range of the 7 bit addresses, a new 10 bit address scheme has been introduced. This enhancement can be mixed with 7 bit addressing and increases the available address range about ten times.After the start condition, a leading ‘11110’ introduces the 10 bit addressing scheme.The last two address bits of the first byte concatenated with the eight bits of the second byte of the whole 10 bit address. Devices which only use 7 bit addressing simply ignore messages with the leading ‘11110’.The following picture shows the first two bytes of a transfer with a 10 bit address.



A master transmitter addresses the slave with two address bytes as described above with the RW-Bit=’0′ followed by data bytes from the master. The master receiver transfer is only possible with a [Combined Transfer](https://www.i2c-bus.org/repeated-start-condition/) due to the fact that the second address byte can only be transmitted if the RW-Bit of the first address byte is ‘0’. Hence, the start of a master receiver transfer will be the same as a master transmitter transfer followed by a repeated start condition and the first byte of address byte with RW-Bit=’1′ (switching to slave transmitter mode). Please refer to the following master receiver sequence:

\* Start condition

\* First address byte, RW-Bit=’0′, ACK from the slave

\* Second address byte, ACK from the slave

\* Repeated start condition (no stop condition!)

\* First address byte again, RW-Bit=’1′, ACK from the slave, slave switches to transmit mode

\* Slave transmits data bytes, ACK from master

\* After the last data byte, the master sends a NACK

\* Stop condition

Please consult the I2C specification for details.

# Alternatives

# I2C Alternatives

There are numerous ways to establish a connection between devices and whilst I2C is the ideal solution for many applications the following briefly describes and compares some alternatives.

## SPI

SPI is a synchronous serial connection implemented by one clock line, one data line in each direction and an optional chip- select signal. In its most simple form, only one data line (either input or output) is present. Unlike the I2C bus, SPI is based on push/pull technology which allows running at much higher speeds than I2C. An SPI interface is not exactly a bus since there is no way to address different devices by an address. Data is sent synchronized by the clock line which is driven by the master. Using a different chip-select line for each device it is, however, possible to share wires between components. Speed and simplicity are the advantages of SPI and therefore, it is popular for applications like EEPROM/Flash programming, display control, sensor data acquisition etc.

## Serial

By far the most often used interconnection is the UART (Universal Asynchronous Receiver/Transmitter) in various forms. It is based on a fixed baud rate which must be guaranteed and which is best achieved using dedicated hardware. Except for some special protocols, UARTS do not allow different devices on one bus either. But one huge advantage of them is the ability to serve as master and slave at the same time, i.e. allow a free bi-directional communication without the need to define master/slave roles.

## CAN

The CAN bus was originally developed for automotive applications but it has found its way into almost all industrial areas. The CAN protocol is quite complex allowing for data integrity checks, device addressing, error recovery and several advanced features. There is, however, a large number of hardware CAN controllers available which hide this complexity from the developer. CAN can use the same physical layer as a UART allowing to bridge larger distances than an I2C or SPI connection.

## 1-Wire®

Dallas Semiconductors is the inventor of [1-Wire](http://www.maxim-ic.com/products/1-wire/)

# Auto Increment

# A faster way for data access

Even though it is frequently related to I2C, automatic incrementation is not part of any I2C standard but rather a common feature found in many I2C devices. What it means is that the device maintains an internal pointer which is automatically incremented upon data read or write activities and which can be manually set to a fixed value. This comes in handy when storing larger amounts of data for instance in an ordinary I2C RAM or EEPROM.

# A typical communication scheme looks like this

* Send a start condition
* Send an address byte with read/write bit, for instance, 0xa1 for a device with address 0x50 if we want to set the autoincrement pointer
* Write value of the auto increment pointer e.g. 0x00
* Send a [Repeated Start Condition](https://www.i2c-bus.org/repeated-start-condition/) because we want to read data starting at location 0 now
* Again send an address byte (0xa0 in our example)
* Read an arbitrary number of bytes from the device – the autoincrement feature will internally add 1 to the address pointer allowing us to obtain consecutive bytes starting with the data byte at location 0 we originally wrote to the bus
* Send a stop condition

# Clock Stretching

In an I2C communication, the master device determines the clock speed. Unlike RS232 the I2C bus provides an explicit clock signal which relieves master and slave from synchronizing exactly to a predefined baud rate.

However, there are situations where an I2C slave is not able to co-operate with the clock speed given by the master and needs to slow down a little. This is done by a mechanism referred to as *clock stretching*.

An I2C slave is allowed to hold down the clock if it needs to reduce the bus speed. The master, on the other hand, is required to read back the clock signal after releasing it to the high state and wait until the line has actually gone high.

# Bandwith

Clock stretching sounds a bit odd but is common practice. However, the total bandwidth of the shared bus might be significantly decreased. So, especially for I2C buses shared by multiple devices, it is important to estimate the impacts of clock stretching. So do not make the slowest I2C device dominate your bus performance.

## Clock Stretching in High Speed Mode

Clock stretching in [High-Speed-Mode](https://www.i2c-bus.org/highspeed/) is only allowed after the ACK bit (and before the 1st bit of the next byte). Stretching between bits 2-9 is illegal because the edges of these bits are boosted with an additional current source. See I2C specification Rev. 03 chapter 5.3.1 for further details.

# Fast Mode

The I2C fast mode defines transfer rates up to 400 kbit/s whilst the first I2C specification dated 1982 had a limit of [100 kbit/s](https://www.i2c-bus.org/standard-mode/). To accomplish 400 kbit/s the timing requirement were defined more strictly.

In May 2006 NXP, formerly Philips introduced [Fast-mode Plus](https://www.i2c-bus.org/fast-mode-plus/) allowing up to 1 Mbit/s.

Additionaly, there is the [high speed mode](https://www.i2c-bus.org/highspeed/) running the bus with with up to 3.4 Mbit/s

# Fast Mode Plus

# Making the fast mode faster

The [I2C fast mode](https://www.i2c-bus.org/fastmode/) was originally specified to run at 400 khz. So what keeps people for ignoring this 400 kilobit limit and run the same bus with the same logic at higher speeds? There are a couple of factors which limit the maximum speed on the bus. First, there’s the capacitance which introduces a rise time for the signals. As a “counter measure” the current on the bus can be increased by lowering the resistor values for the termination pull-ups.

Secondly, the bus has some timing tolerances and if the speed is increased these tolerances need to be more restrictive. The Fast-mode plus specification *FM+* introduced by Philips Semiconductors (now: NXP) in April 2006 defines such a bus with a maximum speed of 1 Mhz. Unlike the [highspeed mode](https://www.i2c-bus.org/highspeed/) there is no additional logic to implement. Therefore, fast-mode plus devices are downward compatible with standard and fast-mode devices. In fact, many I2C slaves on the market like RAMs and EEPROMs will already tolerate higher bus frequencies.

The specification for FM+ is now part of the current I²C-bus specifiation and user manual (document UM10204 from NXP), available from [www.nxp.com/acrobat\_download/usermanuals/UM10204\_3.pdf](http://www.nxp.com/documents/usermanuals/UM10204_3.pdf).

NXP provides an overview of [I2C Fast-mode Plus devices](http://www.nxp.com/products/interface_and_connectivity/i2c/i2c_fast_mode_plus/).

# High Speed Mode

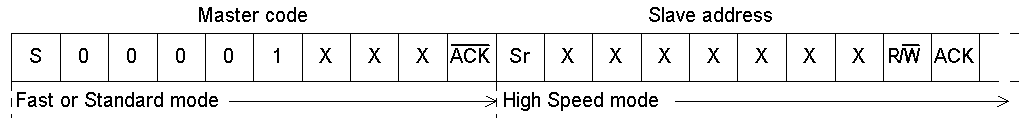
There are applications where the I2C transfer speed is a limiting factor. To allow for higher transmission rates while retaining a certain amount of compatibility Philips has introduced the HS I2C standard. Here are some details and particularities:

# Electrical Characteristics

* The high-speed variant of the I2C bus allows communication up to 3.4 Mbit per second.
* Both, master and slave device must be highspeed-enabled in order to benefit from this increase.
* High-speed IC devices are downward compatible allowing for mixed bus systems.
* In order to shorten signal rise time HS mode master devices have a combination of an open-drain pull-down and current-source pull-up circuit on the SCL output.
* HS IC masters can actually source current to the bus which is referred to as boosting.
* This current source is enabled only (!) during HS operation and just for one master.
* HS mode master devices generate a serial clock signal with a HIGH to LOW ratio of 1 to 2.
* HS mode master devices can have a built-in bridge to separate lower speed devices from the bus during HS transfer. The main purpose of such bridge is to reduce the capacitive load on the bus and to avoid conflicts caused by low speed devices.

# Transmission Format

A high-speed transmission starts up in full-speed mode, i.e. at max. 400 kbit.



* Start condition is sent
* 8 bit ‘master code’ is sent at 400 kbit max
* Master code is ‘not acknowledged’
* Active master switches to high-speed communication
* The ‘master code’ is a reserved value. 8 of them are available
* Arbitration and clock synchronisation only take place during master code transmission, not during HS transfer
* The current-source circuit is enabled after the transmission of the master code
* The active master sends out a repeated start condition followed by the address of the desired slave
* This address is acked or nacked
* Transmission continues in the known manner
* The current source circuit is disabled after each repeated start condition and after each ack or nack to give slaves a chance to stretch the clock
* It is re-enabled once SCL has been released by all devices
* All devices return to fast mode operation after a stop condition is sent
* More details on the addressing scheme for high-speed transfers is available [on this page](https://www.i2c-bus.org/addressing/high-speed/).

## Clock Stretching

Clock stretching during this mode has a special rule: It is only allowed after the ACK bit (and before the 1st bit of the next byte). Stretching between bits 2-9 is illegal because the edges of these bits are boosted with an additional current source. See I2C specification rev. 03 chapter 5.3.1 for further details.

## Interfaces

The I2C HS-mode is rarely supported by I2C interfaces. [Tracii XL 2.0](http://www.telos.de/p/hw/traciixl20/" \o "Opens external link in new window" \t "_blank) features this mode.

# I2C Interface

The term “I2C Interface” usually refers to means for connecting an I2C bus to a PC. However, there are certainly other I2C interfaces which provide connectivity for this universally used bus to non-PC devices.

There are a number of ways to attach the I2C lines to a PC. There is even a zero cost solution available which takes advantage of the PC parallel port pins. A bit-banging driver is then used to toggle the signals to implement a simple I2C single master device. Provided that the parallel port of the used PC is tolerant enough with respect to various electrical characteristics such approach may well work for hobbyists. Since it is quite unreliable a direct (hardwareless) parallel port interface should never be applied in commercial applications.

Philips provided schematics for an adapter called *Single-master,* which essentially works like a hardwareless solution but compensates for some parallel port-related issues by using a driver chip between the PC port pins and the I2C SDA and SCL lines. Such interface can only be operated as single master and the maximum speed highly depends on the drivers and operating system of the PC. Since each bit is toggled individually no timing can be guarranteed under Windows or Linux with such connection. Another disadvantage is the potential danger of damaging the PC with electronic discharge or faulty connections of the external lines. This is particularly annoying if the parallel port functionality is integrated into a multifunctional non-replaceable chip as is the case with all modern PCs and notebooks.

A much more reliable, powerful and safe alternative are interfaces which provide their own I2C logic such as [Connii MM 2.0](http://www.telos.info/p/hw/conniimm20/) or [Tracii XL 2.0](http://www.telos.info/p/hw/traciixl20/). These devices are connected via USB, in addition to being able to operate in multi-master mode they provide a number of important and helpful additional features. Since they communicate with the PC software on a more efficient level than bit banging, the transfer speed is considerably higher and the load on the PC is much lower.

So what are the considerations when choosing a professional I2C interface?

* An I2C interface should not only consist of wiring to the parallel port of a PC. This approach can cause damage to your PC or notebook and is not very reliable.
* If you intend to use the interface with another I2C master simultaneously, you should opt for a multimaster interface in order to avoid arbitration conflicts on the bus.
* The interface you chose should be capable of running at the same speed as your other I2C devices.
* If you have slaves which are implemented using a microcontroller or if you know that a slave device requires clock stretching, you need to make sure that your interface supports it.
* Choose an interface which has a user interface and programming API which you can easily use and which is supported.
* If possible, an USB interface is preferable over a plug-in card for your PC. This reduces the cable length you need extra on the I2C bus.
* Choose a vendor who offers reliable support and long-term market commitment. As operating systems tend to change from time to time you will need upgrades in order to be able to use such new versions.

# Level Shifting

I2C buses are not limited to a defined voltage. In some applications different I2C reference voltages are used for different ICs. Sometimes it is necessary to have all of them sharing the same bus. To be able to recognize what a logical zero and logical one is, a level-shifter is necessary.

Different to most interfaces the I2C bus does not have a dedicated direction, e.g. multiple devices may pull the SCL line to ground.

This causes serious problems building level shifters since they have to take care of moving a request from one side to the other and vice versa without introducing back coupling and oscillation. The basic idea to handle this is to limit the bandwidth on the bus. This gives poor transfer-rates of course.

[A basic idea on how to build a level shifter](http://www.semiconductors.philips.com/markets/mms/protocols/i2c/facts/#levelshifting)

Commercial level shifters as [Levii](http://www.telos.info/p/hw/levii/" \t "_blank) and [Optii](http://www.telos.info/p/hw/optii/) are using different approaches providing higher bandwidths. Optii has some additional, outstanding feature: it separates the I2C buses galvanically, so two I2C buses can be logically connected but electrically insulated.

The semiconductors company NXP offers some [I2C Voltage level translators](https://www.nxp.com/products/analog/interfaces/ic-bus/ic-voltage-level-translators).

MultiMaster

There are I2C environments where multiple masters are driving the bus.

In such case each device needs to be able to cooperate with the fact that another device is currently talking and the bus is therefore busy.

This translates into:

a) Being able to follow arbitration logic. If two devices start to communicate at the same time the one writing more zeros to the bus (or the slower device) wins the arbitration and the other device immediately discontinues any operation on the bus.

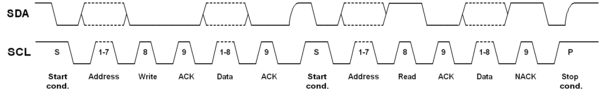
b) Bus busy detection. Each device must detect an ongoing bus communication and must not interrupt it. This is achieved by recognizing traffic and waiting for a stop condition to appear before starting to talk on the bus.

If you plan to use a multimaster device on a bus it is essential that **all masters** are multimasters. A single-master is simply a device, which does not understand the above mechanisms. If a [singlemaster](https://www.i2c-bus.org/SingleMaster/) and a multimaster are connected, the singlemaster may well interrupt the multimaster causing unpredictable results.

Repeated Start Condition

A way to claim the bus

During an I2C transfer there is often the need to first send a command and then read back an answer right away. This has to be done without the risk of another (multimaster) device interrupting this atomic operation. The I2C protocol defines a so-called repeated start condition. After having sent the address byte (address and read/write bit) the master may send any number of bytes followed by a stop condition. Instead of sending the stop condition it is also allowed to send another start condition again followed by an address (and of course including a read/write bit) and more data. This is defined recursively allowing any number of start conditions to be sent. The purpose of this is to allow combined write/read operations to one or more devices without releasing the bus and thus with the guarantee that the operation is not interrupted.

  
Regardless of the number of start conditions sent during one transfer the transfer must be ended by exactly one stop condition.

# SingleMaster

A single master environment is defined by the fact that there is only one device present on the bus which acts as I2C master.

This is by far the easiest implementation of an I2 C bus as it drops the need for arbitration detection and handling. Unless there is a slave present which makes use of [clock stretching](https://www.i2c-bus.org/clock-stretching/) the master can simply write out commands without reading back the clock line. This behaviour is certainly not recommended, but it works in many cases, especially when writing to EEPROMS or other slave devices which are implemented as a hardware state machine. One important thing to consider when working with PC [I2C interfaces](https://www.i2c-bus.org/i2c-Interface/) is that the system will become a [multimaster](https://www.i2c-bus.org/MultiMaster/) environment if the PC is used to send commands to a running I2C system, i.e. if the PC is used to talk to slave devices while there is also a microcontroller on the target possibly talking to the I2C bus.

# Slave

In an I2C communication each device takes a designated role. It is either a master or a slave. These terms refer to the one who dictates the speed of the clock, i.e. who actually generates it. Unlike an RS232 scenario the actual Baud rate is not fixed to a constant value. Instead, the master device pulses the clock for each transmitted bit. The slave has to follow this speed. Since even slaves need to rest sometimes, it is possible for a slave device to hold down the clock line until it is ready to continue communication. This mechanism is referred to as clock stretching and assures that a slave device has the ability to protect itself from beeing addressed too quickly.

Whilst it is next to trivial to implement an I2C single master on a microcontroller the same is merely possible in many cases for a slave. A master generates a clock and drives the communication whenever it is ready to do so. In contrast, a slave device has to always be ready to detect and process a start condition and to recognize its address. This is relatively easy to achieve with a hardware state machine and extremely difficult with a microcontroller, especially if it is busy with other tasks, too. The challenge is to assure that at any point in time and regardless of the application state the microcontroller is ready to process a master request at the maximum clock speed allowed for the communication.

# I2C Slave on a PC

Another rather tricky issue is to implement a slave interface for a PC.

The purpose of such interface is usually to simulate an I2C slave by PC software. This in turn means that the interface needs to talk to a PC application in order to know how to react to a master request. This introduces a timing issue and since none of the popular PC operating systems has real-time capabilities there are no easy solutions to this.

# Communication Example

Let’s assume we want to build a slave device which returns the square of a given number. It listens to the I2C address 0x56, receives a byte as input and returns the square value of this byte (without any carry) in response to a subsequent read operation.

Now the master puts the 0x46 on the bus in write mode and sends a 7 afterwards.

The slave interface recognizes that it has been addressed and reads in the 7. Now it needs to do the processing and has to contact the PC for this to calculate the 49.

Depending on the used connection this takes anything between 80 µs and several ms. The interface could stretch the clock, but unfortunately there are many master mode implementations out that behave like dispots and do not allow clock stretching. In any case, many masters have a timeout for maximum clock stretching somewhere around 1 ms, which comes quite close to the delay caused in PC communication.

# Dedicated Slaves

There are I2C interfaces for PCs which include a slave. E.g. Tracii 400 implements a slave mode and since it uses a parallel port connection the delays introduced are quite acceptable for many applications.

# Standard Mode

Standard-mode refers to the initial transfer speed mode of the I2C specification which allows up to 100 kbit/s.

The [fast-mode](https://www.i2c-bus.org/fastmode/) features 400 kbit/s, [fast-mode plus](https://www.i2c-bus.org/fast-mode-plus/) up to 1000 kbit/s,  whilst the [high speed](https://www.i2c-bus.org/highspeed/) *HS*-mode runs with up to 3.4 Mbit/s.

# Termination

# Every I2C bus must be terminated

The I2C bus relies on open drain technology. Except for one special case in conjunction with the high speed mode no I2C device is allowed to drive the bus, i.e. to send a 1 by putting voltage on the bus. Instead, the bus is terminated to a high level, e.g. 5 V and both lines, SCL and SDA, remain there during idle mode. A device sends a 0 by pulling the bus to ground level. A 1 is sent by doing nothing, i.e. leaving the bus at high level. The termination of I2C lines is achieved by connecting the bus to the reference voltage of the bus over an appropriate resistor. Missing or wrong termination is the source of many field problems. As a rule of thumb, the higher the termination the better the signals. On the other hand a high termination bares a potential risk of damaging components. By the way, in I2C engineering terms a high termination means a strong pull-up to VCC which translates into a low value for the resistor.

Voltage Level

In the beginning the I2C bus focussed on 5 volt logic. With the I2C specification 2.0 released 1998 the possible I2C reference voltage was decreased to 2 volt.

Since the I2C bus lines are used bidirectionally, interfacing I2C devices with different voltages is not straightforward, special [level-shifting](https://www.i2c-bus.org/level-shifting/) devices are necessary for this purpose.

**Moving down the voltage scale**

Things to consider when working with low voltage I2C systems.

When I2C was first introduced the typical electrical VCC level of electronics was 5V and the initially specified speed was a maximum of 100 kHz. With these parameters it was possible to operate over a wide range of electrical parameters in terms of bus termination and capacitance.

With the definition of fast mode, fast mode plus and high speed the timing requirements became more strict and this implied the need for higher terminations, lower capacitance and minimized serial resistance.

Apart from the bus speed more and more circuitry is moving to 3.3V or even 1.8V and below logic levels. An example is NXP’s IO expander [PCA9554C](http://www.nxp.com/products/interface_and_connectivity/i2c/i2c_general_purpose_i_o/series/PCA9554B_PCA9554C.html).

Since I2C is an open drain concept the VCC level as such is not critical for the operation as long as all components on the bus can accept the voltage on the IO pins and are able to detect the logic levels.

*There is however one aspect which is often forgotten*:

Cables and connectors add capacitance to the system. This causes the signals to rise and fall more slowly but it also creates additional cross talk due to capacitive cupeling. Low voltage systems tend to me more sensitive to these effects then 5V or 3.3V setups.

Even though there might be low capacitance cables on the market there is not much one can do about this except to lower the speed and to possible use a lower bus termination.

As a general rule, on low voltage I2C systems the cables should be kept as short as possible and ribbon cables are preferable.

In addition it is a good idea to work with the slowest possible speed to have the option of a lower termination.

Another option is to use a [dedicated device for bus signal enhancement](http://www.linear.com/product/LTC4311).

# PMBus

# Features On Top

Power management on top of SMBus

Just like the I2C-Bus, the [SMBus](https://www.i2c-bus.org/smbus/) defines a protocol for inter-device communication. It adds timeouts and standards for data transfer formats, however, it does not define the content of transmitted data.

The new PMBus standard is based on the [SMBus](https://www.i2c-bus.org/smbus/). It defines a set of commands and data structures required by power control and management components.

This allows interoperability between ICs not only on the electrical level but also regarding the command semantics.

As an example, the PMBus defines a command for setting and reading over voltage level – something that is essential in the power management domain.

In this sense, the PMBus is not a new bus but a protocol layer on top of the SM-Bus.

More information about the PMBus can be found at [www.pmbus.org.](http://pmbus.org/Home)

The PMBus name and logo are trademarks of SMIF, Inc.

# SMBus

The System Management Bus (SMBus) is more or less a derivative of the I2C bus. The standard has been developed by Intel and is now maintained by the SBS Forum.

The main application of the SMBus is to monitor critical parameters on PC motherboards and in embedded systems. For example there a lot of supply voltage monitor, temperature monitor, and fan monitor/control ICs with a SMBus interface available.

There are some interesting differences between the I2C bus and the SMBus:

* Packet Error Checking (PEC)
* Timeout for transfers
* Standardized transfer types
* ALERT line
* SUSPEND line
* Power down/up
* max. bitrate of 100 kHz/s

# TWI Bus

TWI stands for Two Wire Interface and, for the most part, this bus is identical to I²C. The name TWI was introduced by Atmel and other companies to avoid conflicts with trademark issues related to I²C. A description of the capabilities of TWI interfaces can be found in the data sheets of corresponding devices. Expect TWI devices to be compatible with I²C devices except for some particularities like general broadcast or 10 bit addressing.

For the time being there is also no TWI high speed mode.

If you have an I²C analyser it will work with a TWI set-up as well without modifications.