WIKIPEDIA

I²C

I²C (**Inter-Integrated Circuit**), pronounced *I-squared-C*, is a synchronous, multi-master, multi-slave, packet switched, single-ended, serial computer bus invented in 1982 by Philips Semiconductor (now NXP Semiconductors). It is widely used for attaching lower-speed peripheral ICs to processors and microcontrollers in short-distance, intra-board communication. Alternatively, I²C is spelled **I2C** (pronounced I-two-C) or **IIC** (pronounced I-I-C).

Since October 10, 2006, no licensing fees are required to implement the I²C protocol. However, fees are required to obtain I²C slave addresses allocated by NXP.[1]

Several competitors, such as Siemens (later Infineon Technologies, now Intel Mobile Communications), NEC, Texas Instruments, STMicroelectronics (formerly SGS-Thomson), Motorola (later Freescale, now merged with NXP),[2] Nordic Semiconductor and Intersil, have introduced compatible I²C products to the market since the mid-1990s.

System Management Bus (SMBus), defined by Intel in 1995, is a subset of I²C, defining a stricter usage. One purpose of SMBus is to promote robustness and interoperability. Accordingly, modern I²C systems incorporate some policies and rules from SMBus, sometimes supporting both I²C and SMBus, requiring only minimal reconfiguration either by commanding or output pin use.

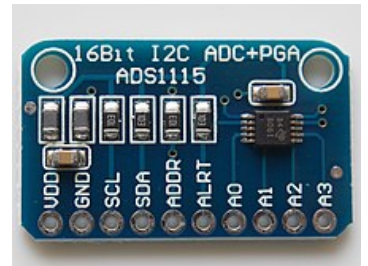| I²C	
Type	Bus
Production history	
Designer	Philips Semiconductor, known today as NXP Semiconductors
Designed	1982
Data	
Data signal	Open-collector or Open-drain
Width	data line (SDA) + clock line (SCL)
Bitrate	0.1 / 0.4 / 1.0 / 3.4 / 5.0 Mbit/s (depending on mode)
Protocol	Serial, half-duplex

Contents

Applications

I²C is appropriate for peripherals where simplicity and low manufacturing cost are more important than speed. Common applications of the I²C bus are:

- Describing connectable devices via small ROM configuration tables to enable "plug and play" operation, such as
  - Serial Presence Detect (SPD) EEPROMs on dual in-line memory modules (DIMMs), and
  - Extended Display Identification Data (EDID) for monitors via VGA, DVI and HDMI connectors.
- System management for PC systems via SMBus;
  - SMBus pins are allocated in both Conventional PCI and PCI Express connectors.
- Accessing real-time clocks and NVRAM chips that keep user settings.
- Accessing low-speed DACs and ADCs.
- Changing contrast, hue, and color balance settings in monitors (via Display Data Channel).
- Changing sound volume in intelligent speakers.
- Controlling small (e.g. feature phone) OLED or LCD displays.
- Reading hardware monitors and diagnostic sensors, e.g. a fan's speed.
- Turning on and turning off the power supply of system components.[4]

STMicroelectronics 24C08: Serial EEPROM with I²C bus[3]

A particular strength of I²C is the capability of a microcontroller to control a network of device chips with just two general-purpose I/O pins and software. Many other bus technologies used in similar applications, such as Serial Peripheral Interface Bus (SPI), require more pins and signals to connect multiple devices.

A 16-bit ADC board with I²C interface

Revisions

The history of I²C specification releases:

- In 1982, the original 100 kbit/s I²C system was created as a simple internal bus system for building control electronics with various Philips chips.
- In 1992, Version 1 added 400 kbit/s *Fast-mode (Fm)* and a 10-bit addressing mode to increase capacity to 1008 nodes. This was the first standardized version.
- In 1998, Version 2 added 3.4 Mbit/s *High-speed mode (Hs)* with power-saving requirements for electric voltage and current.
- In 2000, Version 2.1 clarified version 2, without significant functional changes.
- In 2007, Version 3 added 1 Mbit/s *Fast-mode plus (Fm+)* (using 20 mA drivers), and a device ID mechanism.
- In 2012, Version 4 added 5 Mbit/s *Ultra Fast-mode (UFm)* for new USDA (data) and USCL (clock) lines using push-pull logic without pull-up resistors, and added an assigned manufacturer ID table. It is only a **unidirectional** bus.
- In 2012, Version 5 corrected mistakes.
- In 2014, Version 6 corrected two graphs. This is the most recent standard.[5]

Design

I²C uses only two bidirectional open collector or open drain lines, Serial Data Line (SDA) and Serial Clock Line (SCL), pulled up with resistors.[5] Typical voltages used are +5 V or +3.3 V, although systems with other voltages are permitted.

The I²C reference design has a 7-bit address space, with a rarely used 10-bit extension.[6] Common I²C bus speeds are the 100 kbit/s *standard mode* and the 400 kbit/s *Fast mode*. There is also a 10 kbit/s *low-speed mode*, but arbitrarily low clock frequencies are also allowed. Recent revisions of I²C can host more nodes and run at faster speeds (400 kbit/s *Fast mode*, 1 Mbit/s *Fast mode plus*, 3.4 Mbit/s *High Speed mode,* and 5 Mbit/s *Ultra Fast-mode*). These speeds are more widely used on embedded systems than on PCs.

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

The number of nodes which can exist on a given I²C bus is limited by the address space and also by the total bus capacitance of 400 pF, which restricts practical communication distances to a few meters. The relatively high impedance and low noise immunity requires a common ground potential, which again restricts practical use to communication within the same PC board or small system of boards.

An example schematic with one master (a microcontroller), three slave nodes (an ADC, a DAC, and a microcontroller), and pull-up resistors R_p

Reference design

The aforementioned reference design is a bus with a clock (SCL) and data (SDA) lines with 7-bit addressing. The bus has two roles for nodes: master and slave:

- Master node – node that generates the clock and initiates communication with slaves.
- Slave node – node that receives the clock and responds when addressed by the master.

The bus is a multi-master bus, which means that any number of master nodes can be present. Additionally, master and slave roles may be changed between messages (after a STOP is sent).

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

- master transmit – master node is sending data to a slave,
- master receive – master node is receiving data from a slave,
- slave transmit – slave node is sending data to the master,
- slave receive – slave node is receiving data from the master.

In addition to 0 and 1 data bits, the I²C bus allows special START and STOP signals which act as message delimiters and are distinct from the data bits. (This is in contrast to the start bits and stop bits used in asynchronous serial communication, which are distinguished from data bits only by their timing.)

The master is initially in master transmit mode by sending a START followed by the 7-bit address of the slave it wishes to communicate with, which is finally followed by a single bit representing whether it wishes to write (0) to or read (1) from the slave.

If the slave exists on the bus then it will respond with an ACK bit (active low for acknowledged) for that address. The master then continues in either transmit or receive mode (according to the read/write bit it sent), and the slave continues in the complementary mode (receive or transmit, respectively).

The address and the data bytes are sent most significant bit first. The start condition is indicated by a high-to-low transition of SDA with SCL high; the stop condition is indicated by a low-to-high transition of SDA with SCL high. All other transitions of SDA take place with SCL low.

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

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

An I²C transaction may consist of multiple messages. The master terminates a message with a STOP condition if this is the end of the transaction or it may send another START condition to retain control of the bus for another message (a "combined format" transaction).

Message protocols

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

- Single message where a master writes data to a slave.
- Single message where a master reads data from a slave.
- Combined format, where a master issues at least two reads or writes to one or more slaves.
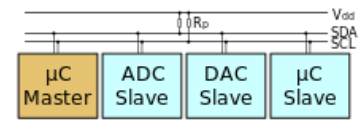
In a combined transaction, each read or write begins with a START and the slave address. The START conditions after the first are also called *repeated START* bits. Repeated STARTs are not preceded by STOP conditions, which is how slaves know that the next message is part of the same transaction.

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

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

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

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

Messaging example: 24C32 EEPROM

One specific example is the 24C32 type EEPROM, which uses two request bytes that are called Address High and Address Low. (Accordingly, these EEPROMs are not usable by pure SMBus hosts, which only support single-byte commands or addresses.) These bytes are used for addressing bytes within the 32 kbit (or 4 kB) EEPROM address space. The same two-byte addressing is also used by larger EEPROMs, like the 24C512 which stores 512 kbits (or 64 kB). Writing and reading data to these EEPROMs uses a simple protocol: the address is written, and then data is transferred until the end of the message. The data transfer part of the protocol can cause trouble on the SMBus, since the data bytes are not preceded by a count, and more than 32 bytes can be transferred at once. I²C EEPROMs smaller than 32 kbit, like the 2 kbit 24C02, are often used on the SMBus with inefficient single-byte data transfers to overcome this problem.

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

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

Physical layer

At the physical layer, both SCL and SDA lines are of open-drain design, thus pull-up resistors are needed. A logic "0" is output by pulling the line to ground, and a logic "1" is output by letting the line float (output high impedance) so that the pull-up resistor pulls it high. A line is never actively driven high. This wiring allows multiple nodes to connect to the bus without short circuits from signal contention. High-speed systems (and some others) may use a current source instead of a resistor to pull-up only SCL or both SCL and SDA, to accommodate higher bus capacitance and enable faster rise times.

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

When used on SCL, this is called *clock stretching* and is a flow-control mechanism for slaves. When used on SDA, this is called arbitration and ensures that there is only one transmitter at a time.

When idle, both lines are high. To start a transaction, SDA is pulled low while SCL remains high. It is illegal[5]:14 to transmit a stop marker by releasing SDA to float high again (although such a "void message" is usually harmless), so the next step is to pull SCL low.

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

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

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

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

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

Only the SDA line changes direction during acknowledge bits; the SCL is always controlled by the master.

After the acknowledge bit, the clock line is low and the master may do one of three things:

- Begin transferring another byte of data: the transmitter sets SDA, and the master pulses SCL high.
- Send a "Stop": Set SDA low, let SCL go high, then let SDA go high. This releases the I²C bus.
- Send a "Repeated start": Set SDA high, let SCL go high, then pull SDA low again. This starts a new I²C bus message without releasing the bus.

Clock stretching using SCL

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

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

Although the master may also hold the SCL line low for as long as it desires (this is not allowed in newest Rev. 6 of the protocol – subsection 3.1.1), the term "clock stretching" is normally used only when slaves do it. Although in theory any clock pulse may be stretched, generally it is the intervals before or after the acknowledgment bit which are used. For example, if the slave is a microcontroller, its I²C interface could stretch the clock after each byte, until the software decides whether to send a positive acknowledgment or a NACK.

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

To ensure a minimal bus throughput, SMBus places limits on how far clocks may be stretched. Hosts and slaves adhering to those limits cannot block access to the bus for more than a short time, which is not a guarantee made by pure I²C systems.

Arbitration using SDA

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

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

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

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

One case which must be handled carefully in multi-master I²C implementations is that of the masters talking to each other. One master may lose arbitration to an incoming message, and must change its role from master to slave in time to acknowledge its own address.

In the extremely rare case that two masters simultaneously send identical messages, both will regard the communication as successful, but the slave will only see one message. For this reason, when a slave can be accessed by multiple masters, every command recognized by the slave either must be idempotent or must be guaranteed never to be issued by two masters at the same time. (For example, a command which is issued by only one master need not be idempotent, nor is it necessary for a specific command to be idempotent when some mutual exclusion mechanism ensures that only one master can be caused to issue that command at any given time.)

Arbitration in SMBus

While I²C only arbitrates between masters, SMBus uses arbitration in three additional contexts, where multiple slaves respond to the master, and one gets its message through.

- Although conceptually a single-master bus, a slave device that supports the "host notify protocol" acts as a master to perform the notification. It seizes the bus and writes a 3-byte message to the reserved "SMBus Host" address (0x08), passing its address and two bytes of data. When two slaves try to notify the host at the same time, one of them will lose arbitration and need to retry.
- An alternative slave notification system uses the separate SMBALERT# signal to request attention. In this case, the host performs a 1-byte read from the reserved "SMBus Alert Response Address" (0x0C), which is a kind of broadcast address. All alerting slaves respond with a data bytes containing their own address. When the slave successfully transmits its own address (winning arbitration against others) it stops raising that interrupt. In both this and the preceding case, arbitration ensures that one slave's message will be received, and the others will know they must retry.
- SMBus also supports an "address resolution protocol", wherein devices return a 16-byte "universal device ID" (UDID). Multiple devices may respond; the one with the lowest UDID will win arbitration and be recognized.

Arbitration in PMBus

PMBus version 1.3 extends the SMBus alert response protocol in its "zone read" protocol.[7] Slaves may be grouped into "zones", and all slaves in a zone may be addressed to respond, with their responses masked (omitting unwanted information), inverted (so wanted information is sent as 0 bits, which win arbitration), or reordered (so the most significant information is sent first). Arbitration ensures that the highest priority response is the one first returned to the master.

PMBus reserves I²C addresses 0x28 and 0x37 for zone reads and writes, respectively.

Differences between modes

There are several possible operating modes for I²C communication. All are compatible in that the 100 kbit/s *standard mode* may always be used, but combining devices of different capabilities on the same bus can cause issues, as follows:

- *Fast mode* is highly compatible and simply tightens several of the timing parameters to achieve 400 kbit/s speed. *Fast mode* is widely supported by I²C slave devices, so a master may use it as long as it knows that the bus capacitance and pull-up strength allow it.
- *Fast mode plus* achieves up to 1 Mbit/s using more powerful (20 mA) drivers and pull-ups to achieve faster rise and fall times. Compatibility with *standard* and *fast mode* devices (with 3 mA pull-down capability) can be achieved if there is some way to reduce the strength of the pull-ups when talking to them.
- *High speed mode* (3.4 Mbit/s) is compatible with normal I²C devices on the same bus, but requires the master have an active pull-up on the clock line which is enabled during high speed transfers. The first data bit is transferred with a normal open-drain rising clock edge, which may be stretched. For the remaining seven data bits, and the ACK, the master drives the clock high at the appropriate time and the slave may not stretch it. All high-speed transfers are preceded by a single-byte "master code" at fast or standard speed. This code serves three purposes:
  1. it tells high-speed slave devices to change to high-speed timing rules,
  2. it ensures that fast or normal speed devices will not try to participate in the transfer (because it does not match their address), and

3. because it identifies the master (there are eight master codes, and each master must use a different one), it ensures that arbitration is complete before the high-speed portion of the transfer, and so the high-speed portion need not make allowances for that ability.

- *Ultra-Fast mode* is essentially a write-only I²C subset, which is incompatible with other modes except in that it is easy to add support for it to an existing I²C interface hardware design. Only one master is permitted, and it actively drives both clock and data lines at all times to achieve a 5 Mbit/s transfer rate. Clock stretching, arbitration, read transfers, and acknowledgements are all omitted. It is mainly intended for animated LED displays where a transmission error would only cause an inconsequential brief visual glitch. The resemblance to other I²C bus modes is limited to:

  - the start and stop conditions are used to delimit transfers,
  - I²C addressing allows multiple slave devices to share the bus without SPI bus style slave select signals, and
  - a ninth clock pulse is sent per byte transmitted marking the position of the unused acknowledgement bits.

Some of the vendors provide a so called non-standard *Turbo mode* with a speed up to 1.4 Mbit/s.

In all modes, the clock frequency is controlled by the master(s), and a longer-than-normal bus may be operated at a slower-than-nominal speed by underclocking.

Circuit interconnections

I²C is popular for interfacing peripheral circuits to prototyping systems, such as the Arduino and Raspberry Pi. I²C does not employ a standardized connector, however, board designers have created various wiring schemes for I²C interconnections. To minimize the possible damage due to plugging 0.1-inch headers in backwards, some developers have suggested using alternating signal and power connections of the following wiring schemes: (GND, SCL, VCC, SDA) or (VCC, SDA, GND, SCL).[8]

The vast majority of applications use I²C in the way it was originally designed—peripheral ICs directly wired to a processor on the same printed circuit board, and therefore over relatively short distances of less than 1 foot (30 cm), without a connector. However using a differential driver, an alternate version of I²C can communicate up to 20 meters (possibly over 100 meters) over CAT5 or other cable.[9][10]

Several standard connectors carry I²C signals. For example, the UEXT connector carries I²C; the 10-pin iPack connector carries I²C;[11] the 6P6C Lego Mindstorms NXT connector carries I²C;[12][13][14][15] a few people use the 8P8C connectors and CAT5 cable normally used for Ethernet physical layer to instead carry differential-encoded I²C signals[16] or boosted single-ended I²C signals;[17] and every HDMI and most DVI and VGA connectors carry DDC2 data over I²C.

Buffering and multiplexing

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

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

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

Sharing SCL between multiple busses

When having a single master, it is possible to have multiple I²C busses share the same SCL line.[18][19] The packets on each bus are either sent one after the other or at the same time. This is possible, because the communication on each bus can be subdivided in alternating short periods with high SCL followed by short periods with low SCL. And the clock can be stretched, if one bus needs more time in one state.

Advantages are using slaves devices with the same address at the same time and saving connections or a faster throughput by using several data lines at the same time.

Line state table

These tables show the various atomic states and bit operations that may occur during an I²C message.

Line state

Type	Inactive bus (N)	Start (S)	Idle (i)	Stop (P)	Clock stretching (CS)
Note	Free to claim arbitration	Bus claiming (master)	Bus claimed (master)	Bus freeing (master)	Paused by slave
SDA	Passive pullup	**Falling edge (master)**	**Held low (master)**	**Rising edge (master)**	Don't care
SCL	Passive pullup	Passive pullup	Passive pullup	Passive pullup	**Held low (slave)**

Line state

Type	Sending one data bit (1) (0) (SDA is set/sampled after SCL to avoid false state detection)		Receiver reply with ACK bit (Byte received from sender)		Receiver reply with NACK bit (Byte not received from sender)	
	Bit setup (Bs)	**Ready to sample (Bx)**	**Bit setup (Bs)**	**ACK (A)**	**Bit setup (Bs)**	**NACK (A')**
Note	Sender set bit (master/slave)	Receiver sample bit (master/slave)	Sender transmitter hi-Z	Sender sees SDA is low	Sender transmitter hi-Z	Sender sees SDA is high
SDA	Set bit (after SCL falls)	Capture bit (after SCL rises)	Held low by receiver (after SCL falls)		Driven high (or passive high) by receiver (after SCL falls)	
SCL	**Falling edge (master)**	**Rising edge (master)**	**Falling edge (master)**	**Rising edge (master)**	**Falling edge (master)**	**Rising edge (master)**

Line state (repeated start)

Type	Setting up for a (Sr) signal after an ACK/NACK				Repeated start (Sr)
Note	**Start here from ACK**	Avoiding stop (P) state		**Start here from NACK**	Same as start (S) signal
SDA	Was held low for ACK	Rising edge	Passive high	Passive high	**Falling edge (master)**
SCL	**Falling edge (master)**	Held low	**Rising edge (master)**	Passive high	Passive pullup

Addressing structure

7-bit addressing

Field:	S	I²C address field							R/W'	A	I²C message sequences...	P
Type		**Byte 1**									**Byte X etc...**	
Bit position in byte X		7	6	5	4	3	2	1	0	**ACK**	**Rest of the read or write message goes here**	**Stop**
7-bit address pos	**Start**	7	6	5	4	3	2	1				
Note		MSB					LSB	1 = Read				
									0 = Write				

10-bit addressing

Field:	S	10-bit mode indicator					Upper addr		R/W'	A	Lower address field								I2C message sequences	P
Type		Byte 1									Byte 2									
Bit position in byte X		7	6	5	4	3	2	1	0		7	6	5	4	3	2	1	0	Byte X etc.	
Bit value		1	1	1	1	0	X	X	X		X	X	X	X	X	X	X	X	**Rest of the read or write**	
10-bit address pos	Start						10	9		ACK	8	7	6	5	4	3	2	1	**message goes here**	Stop
Note		Indicates 10-bit mode					MSB		1 = Read / 0 = Write									LSB		

Reserved addresses in 7-bit address space

Two groups of addresses are reserved for special functions:

- `0000 XXX`
- `1111 XXX`

Reserved address index	8-bit byte			Description
	7-bit address		R/W value	
	MSB (4-bit)	LSB (3-bit)	1-bit	
1	0000	000	0	General call
2	0000	000	1	Start byte
3	0000	001	X	CBUS address
4	0000	010	X	Reserved for different bus format
5	0000	011	X	Reserved for future purpose
6	0000	1XX	X	HS-mode master code
7	1111	1XX	1	Device ID
8	1111	0XX	X	10-bit slave addressing

SMBus reserves some additional addresses. In particular, `0001 000` is reserved for the SMBus host, which may be used by master-capable devices, `0001 100` is the "SMBus alert response address" which is polled by the host after an out-of-band interrupt, and `1100 001` is the default address which is initially used by devices capable of dynamic address assignment.

Non-reserved addresses in 7-bit address space

MSB (4-bit)	Typical usage[20][21][22][23][24]
0001	Digital receivers, SMBus
0010	TV video line decoders, IPMB
0011	AV codecs
0100	Video encoders, GPIO expanders
0101	ACCESS bus, PMBus
0110	VESA DDC, PMBus
0111	Display controller
1000	TV signal processing, audio processing, SMBus
1001	AV switching, ADCs and DACs, IPMB, SMBus
1010	Storage memory, real-time clock
1011	AV processors
1100	PLLs and tuners, modulators and demodulators, SMBus
1101	AV processors and decoders, audio power amplifiers, SMBus
1110	AV colour space converters

Although MSB 1111 is reserved for Device ID and 10-bit slave addressing, it is also used by VESA DDC display dependent devices such as pointing devices.[23]

Transaction format

An I²C *transaction* consists of one or more *messages*. Each message begins with a start symbol, and the transaction ends with a stop symbol. Start symbols after the first, which begin a message but not a transaction, are referred to as *repeated start* symbols.

Each message is a read or a write. A transaction consisting of a single message is called either a read or a write transaction. A transaction consisting of multiple messages is called a combined transaction. The most common form of the latter is a write message providing intra-device address information, followed by a read message.

Many I²C devices do not distinguish between a combined transaction and the same messages sent as separate transactions, but not all. The device ID protocol requires a single transaction; slaves are forbidden from responding if they observe a stop symbol. Configuration, calibration or self-test modes which cause the slave to respond unusually are also often automatically terminated at the end of a transaction.

Timing diagram
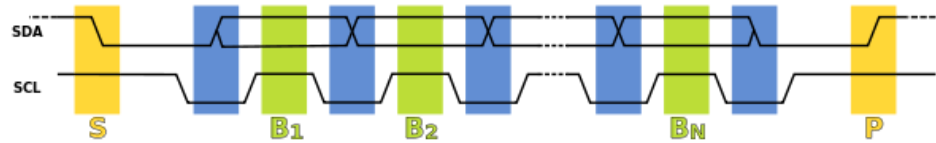
1. Data transfer is initiated with a *start* condition (S) signaled by SDA being pulled low while SCL stays high.
2. SCL is pulled low, and SDA sets the first data bit level while keeping SCL low (during blue bar time).
3. The data are sampled (received) when SCL rises for the first bit (B1). For a bit to be valid, SDA must not change between a rising edge of SCL and the subsequent falling edge (the entire green bar time).
4. This process repeats, SDA transitioning while SCL is low, and the data being read while SCL is high (B2, ...Bn).
5. The final bit is followed by a clock pulse, during which SDA is pulled low in preparation for the *stop* bit.
6. A *stop* condition (P) is signaled when SCL rises, followed by SDA rising.

In order to avoid false marker detection, there is a minimum delay between the SCL falling edge and changing SDA, and between changing SDA and the SCL rising edge. Note that an I²C message containing N data bits (including acknowledges) contains $N+1$ clock pulses.

Example of bit-banging the I²C master protocol

Below is an example of bit-banging the I²C protocol as an I²C master. The example is written in pseudo C. It illustrates all of the I²C features described before (clock stretching, arbitration, start/stop bit, ack/nack).[25]

```
 1  // Hardware-specific support functions that MUST be customized:
 2  #define I2CSPEED 100
 3  void I2C_delay(void);
 4  bool read_SCL(void);   // Return current level of SCL line, 0 or 1
 5  bool read_SDA(void);   // Return current level of SDA line, 0 or 1
 6  void set_SCL(void);    // Do not drive SCL (set pin high-impedance)
 7  void clear_SCL(void);  // Actively drive SCL signal low
 8  void set_SDA(void);    // Do not drive SDA (set pin high-impedance)
 9  void clear_SDA(void);  // Actively drive SDA signal low
10  void arbitration_lost(void);
11
12  bool started = false; // global data
13
14  void i2c_start_cond(void) {
15    if (started) {
16      // if started, do a restart condition
17      // set SDA to 1
18      set_SDA();
19      I2C_delay();
20      set_SCL();
21      while (read_SCL() == 0) { // Clock stretching
22        // You should add timeout to this loop
23      }
24
25      // Repeated start setup time, minimum 4.7us
26      I2C_delay();
27    }
28
29    if (read_SDA() == 0) {
30      arbitration_lost();
31    }
32
33    // SCL is high, set SDA from 1 to 0.
34    clear_SDA();
35    I2C_delay();
36    clear_SCL();
37    started = true;
38  }
39
40  void i2c_stop_cond(void) {
41    // set SDA to 0
42    clear_SDA();
43    I2C_delay();
44
45    set_SCL();
46    // Clock stretching
47    while (read_SCL() == 0) {
48      // add timeout to this loop.
49    }
50
51    // Stop bit setup time, minimum 4us
52    I2C_delay();
53
54    // SCL is high, set SDA from 0 to 1
55    set_SDA();
56    I2C_delay();
57
58    if (read_SDA() == 0) {
59      arbitration_lost();
60    }
61
62    started = false;
63  }
64
65  // Write a bit to I2C bus
66  void i2c_write_bit(bool bit) {
67    if (bit) {
68      set_SDA();
69    } else {
70      clear_SDA();
71    }
72
73    // SDA change propagation delay
74    I2C_delay();
75
76    // Set SCL high to indicate a new valid SDA value is available
77    set_SCL();
78
79    // Wait for SDA value to be read by slave, minimum of 4us for standard mode
80    I2C_delay();
81
82    while (read_SCL() == 0) { // Clock stretching
83      // You should add timeout to this loop
84    }
85
86    // SCL is high, now data is valid
87    // If SDA is high, check that nobody else is driving SDA
88    if (bit && (read_SDA() == 0)) {
89      arbitration_lost();
90    }
91
92    // Clear the SCL to low in preparation for next change
93    clear_SCL();
94  }
```

```
 95
 96   // Read a bit from I2C bus
 97   bool i2c_read_bit(void) {
 98     bool bit;
 99
100     // Let the slave drive data
101     set_SDA();
102
103     // Wait for SDA value to be written by slave, minimum of 4us for standard mode
104     I2C_delay();
105
106     // Set SCL high to indicate a new valid SDA value is available
107     set_SCL();
108
109     while (read_SCL() == 0) { // Clock stretching
110       // You should add timeout to this loop
111     }
112
113     // Wait for SDA value to be written by slave, minimum of 4us for standard mode
114     I2C_delay();
115
116     // SCL is high, read out bit
117     bit = read_SDA();
118
119     // Set SCL low in preparation for next operation
120     clear_SCL();
121
122     return bit;
123   }
124
125   // Write a byte to I2C bus. Return 0 if ack by the slave.
126   bool i2c_write_byte(bool send_start,
127                       bool send_stop,
128                       unsigned char byte) {
129     unsigned bit;
130     bool     nack;
131
132     if (send_start) {
133       i2c_start_cond();
134     }
135
136     for (bit = 0; bit < 8; ++bit) {
137       i2c_write_bit((byte & 0x80) != 0);
138       byte <<= 1;
139     }
140
141     nack = i2c_read_bit();
142
143     if (send_stop) {
144       i2c_stop_cond();
145     }
146
147     return nack;
148   }
149
150   // Read a byte from I2C bus
151   unsigned char i2c_read_byte(bool nack, bool send_stop) {
152     unsigned char byte = 0;
153     unsigned char bit;
154
155     for (bit = 0; bit < 8; ++bit) {
156       byte = (byte << 1) | i2c_read_bit();
157     }
158
159     i2c_write_bit(nack);
160
161     if (send_stop) {
162       i2c_stop_cond();
163     }
164
165     return byte;
166   }
167
168   void I2C_delay(void) {
169     volatile int v;
170     int i;
171
172     for (i = 0; i < I2CSPEED / 2; ++i) {
173       v;
174     }
175   }
```

Operating-system support

- In AmigaOS one can use the i2c.resource component[26] for AmigaOS 4.x and MorphOS 3.x or the shared library *i2c.library* by Wilhelm Noeker for older systems.
- Arduino developers can use the "Wire" library.
- Maximite supports I²C communications natively as part of its MMBasic.
- PICAXE uses the i2c and hi2c commands.

- **eCos** supports I²C for several hardware architectures.
- **ChibiOS/RT** supports I²C for several hardware architectures.
- **FreeBSD**, **NetBSD** and **OpenBSD** also provide an I²C framework, with support for a number of common master controllers and sensors.
    - Since **OpenBSD** 3.9 (released 1 May 2006), a central `i2c_scan` subsystem probes all possible sensor chips at once during boot, using an **ad hoc** weighting scheme and a local caching function for reading register values from the I²C slaves;[27] this makes it possible to probe sensors on **general-purpose** **off-the-shelf** i386/amd64 hardware during boot without any configuration by the user nor a noticeable probing delay; the matching procedures of the individual drivers then only has to rely on a string-based "friendly-name" for matching;[28] as a result, most I²C sensor drivers are automatically enabled by default in applicable architectures without ill effects on stability; individual sensors, both I²C and otherwise, are exported to the userland through the sysctl **hw.sensors** framework.
      As of March 2019, OpenBSD has over two dozen device drivers on I²C that export some kind of a sensor through the **hw.sensors** framework, and the majority of these drivers are fully enabled by default in i386/amd64 GENERIC kernels of OpenBSD.
    - In **NetBSD**, over two dozen I²C slave devices exist that feature hardware monitoring sensors, which are accessible through the sysmon **envsys** framework as **property lists**. On general-purpose hardware, each driver has to do its own probing, hence all drivers for the I²C slaves are disabled by default in NetBSD in GENERIC i386/amd64 builds.
- In **Linux**, I²C is handled with a device driver for the specific device, and another for the I²C (or **SMBus**) adapter to which it is connected. Several hundred such drivers are part of current releases.
- In **Mac OS X**, there are about two dozen I²C kernel extensions that communicate with sensors for reading voltage, current, temperature, motion, and other physical status.
- In **Microsoft Windows**, I²C is implemented by the respective device drivers of much of the industry's available hardware.
- **Unison OS**, a POSIX RTOS for IoT, supports I²C for several MCU and MPU hardware architectures.
- In **Windows CE**, I²C is implemented by the respective device drivers of much of the industry's available hardware.
- In **RISC OS**, I²C is provided with a generic I²C interface from the IO controller and supported from the OS module system
- In **Sinclair QDOS** and **Minerva** **QL** **operating systems** I²C is supported by a set of extensions provided by **TF Services**.

Development tools

When developing or troubleshooting systems using I²C, visibility at the level of hardware signals can be important.

I²C host adapters

There are a number of hardware solutions for host computers, running **Linux**, **Mac** or **Windows**, I²C master and/or slave capabilities. Most of them are based on **USB**-to-I²C adapters. Not all of them require proprietary drivers or **APIs**.

I²C protocol analyzers

I²C protocol analyzers are tools that sample an I²C bus and decode the electrical signals to provide a higher-level view of the data being transmitted on the bus.

Logic analyzers

When developing and/or troubleshooting the I²C bus, examination of hardware signals can be very important. **Logic analyzers** are tools that collect, analyze, decode, and store signals, so people can view the high-speed waveforms at their leisure. Logic analyzers display time stamps of each signal level change, which can help find protocol problems. Most **logic analyzers** have the capability to decode bus signals into high-level protocol data and show ASCII data.

Limitations

The assignment of slave addresses is one weakness of I²C. Seven bits is too few to prevent address collisions between the many thousands of available devices. What alleviates the issue of address collisions between different vendors and also allows to connect to several identical devices is that manufacturers dedicate pins that can be used to set the slave address to one of a few address options per device. Two or three pins is typical, and with many devices, there are three or more wiring options per address pin.[29][30][31]

10-bit I²C addresses are not yet widely used, and many host operating systems do not support them.[32] Neither is the complex SMBus "ARP" scheme for dynamically assigning addresses (other than for PCI cards with SMBus presence, for which it is required).

Automatic bus configuration is a related issue. A given address may be used by a number of different protocol-incompatible devices in various systems, and hardly any device types can be detected at runtime. For example, `0x51` may be used by a 24LC02 or 24C32 EEPROM, with incompatible addressing; or by a PCF8563 RTC, which cannot reliably be distinguished from either (without changing device state, which might not be allowed). The only reliable configuration mechanisms available to hosts involve out-of-band mechanisms such as tables provided by system firmware, which list the available devices. Again, this issue can partially be addressed by ARP in SMBus systems, especially when vendor and product identifiers are used; but that has not really caught on. The rev. 03 version of the I²C specification adds a device ID mechanism.

I²C supports a limited range of speeds. Hosts supporting the multi-megabit speeds are rare. Support for the Fm+ 1 Mbit/s speed is more widespread, since its electronics are simple variants of what is used at lower speeds. Many devices do not support the 400 kbit/s speed (in part because SMBus does not yet support it). I²C nodes implemented in software (instead of dedicated hardware) may not even support the 100 kbit/s speed; so the whole range defined in the specification is rarely usable. All devices must at least partially support the highest speed used or they may spuriously detect their device address.

Devices are allowed to stretch clock cycles to suit their particular needs, which can starve bandwidth needed by faster devices and increase latencies when talking to other device addresses. Bus capacitance also places a limit on the transfer speed, especially when current sources are not used to decrease signal rise times.

Because I²C is a shared bus, there is the potential for any device to have a fault and hang the entire bus. For example, if any device holds the SDA or SCL line low, it prevents the master from sending START or STOP commands to reset the bus. Thus it is common for designs to include a reset signal that provides an external method of resetting the bus devices. However many devices do not have a dedicated reset pin, forcing the designer to put in circuitry to allow devices to be power-cycled if they need to be reset.

Because of these limits (address management, bus configuration, potential faults, speed), few I²C bus segments have even a dozen devices. It is common for systems to have several such segments. One might be dedicated to use with high-speed devices, for low-latency power management. Another might be used to control a few devices where latency and throughput are not important issues; yet another segment might be used only to read EEPROM chips describing add-on cards (such as the SPD standard used with DRAM sticks).

Derivative technologies

I²C is the basis for the ACCESS.bus, the VESA Display Data Channel (DDC) interface, the System Management Bus (SMBus), Power Management Bus (PMBus) and the Intelligent Platform Management Bus (IPMB, one of the protocols of IPMI). These variants have differences in voltage and clock frequency ranges, and may have interrupt lines.

High-availability systems (AdvancedTCA, MicroTCA) use 2-way redundant I²C for shelf management. Multi-master I²C capability is a requirement in these systems.

TWI (Two-Wire Interface) or TWSI (Two-Wire Serial Interface) is essentially the same bus implemented on various system-on-chip processors from Atmel and other vendors.[33] Vendors use the name TWI, even though I²C is not a registered trademark as of 2014-11-07.[34] Trademark protection only exists for the respective logo (see upper right corner), and patents on I²C have now lapsed.

In some cases, use of the term "two-wire interface" indicates incomplete implementation of the I²C specification. Not supporting arbitration or clock stretching is one common limitation, which is still useful for a single master communicating with simple slaves that never stretch the clock.

MIPI I3C sensor interface standard (I3C) is a development of I²C, under development in 2017.[35]

See also

- List of network buses
- UEXT Connector
- System Management Bus
- ACCESS.bus
- VESA Display Data Channel
- Power Management Bus
- I3C

References

1. "I²C Licensing Information" (http://www.nxp.com/documents/application_note/AN10216.pdf) (PDF). *nxp.com.* Archived (https://web.archive.org/web/20170110165928/http://www.nxp.com/documents/application_note/AN10216.pdf) (PDF) from the original on 2017-01-10. Retrieved 2018-04-29.

2. "Financial Press Releases-NXP" (http://investors.nxp.com/phoenix.zhtml?c=209114&p=irol-newsArticle&ID=2120581). *investors.nxp.com*. Retrieved 2018-04-29.

3. "8-Kbit serial I²C bus EEPROM (PDF)" (https://www.st.com/resource/en/datasheet/m24c08-f.pdf) (PDF). *STMicroelectronics*. October 2017. Archived (https://web.archive.org/web/20191018114246/https://www.st.com/resource/en/datasheet/m24c08-f.pdf) (PDF) from the original on 2019-10-18. Retrieved 19 November 2019.

4. "Benefits of Power Supplies Equipped with I2C Ethernet Communications" (http://aegispower.com/index.php/2015-01-15-19-35-10/178-benefits-of-power-supplies-equipped-with-i2c-ethernet-communications). *Aegis Power Systems, Inc*. Aegis Power Systems, Inc. Archived (http://archive.wikiwix.com/cache/20151221174119/http://aegispower.com/index.php/2015-01-15-19-35-10/178-benefits-of-power-supplies-equipped-with-i2c-ethernet-communications) from the original on 2015-12-21. Retrieved 2015-12-21.

5. "I²C-bus specification and user manual" (http://www.nxp.com/documents/user_manual/UM10204.pdf) (PDF). Rev. 6. NXP. 2014-04-04. UM10204. Archived (https://web.archive.org/web/20130511150526/http://www.nxp.com/documents/user_manual/UM10204.pdf) (PDF) from the original on 2013-05-11.

6. "7-bit, 8-bit, and 10-bit I2C Slave Addressing" (http://www.totalphase.com/support/kb/10039/). *Total Phase*. Archived (https://web.archive.org/web/20130601201106/http://www.totalphase.com/support/kb/10039/) from the original on 2013-06-01. Retrieved 2018-04-29.

7. *Using The ZONE_READ And ZONE_WRITE Protocols* (http://pmbus.org/Assets/PDFS/Public/PMBus_AN001_Rev_1_0_1_20160107.pdf) (PDF) (Application Note). Revision 1.0.1. System Management Interface Forum. 2016-01-07. AN001. Archived (https://web.archive.org/web/20170922194245/http://pmbus.org/Assets/PDFS/Public/PMBus_AN001_Rev_1_0_1_20160107.pdf) (PDF) from the original on 2017-09-22.

8. "Is there any definitive I2C pin-out guidance out there? Not looking for a "STANDARD" " (http://electronics.stackexchange.com/a/48343). StackExchange.

9. *NXP Application note AN11075: Driving I2C-bus signals over twisted pair cables with PCA9605* (https://web.archive.org/web/20170816015822/http://www.nxp.com/docs/en/application-note/AN11075.pdf) (PDF), 2017-08-16, archived from the original (http://www.nxp.com/docs/en/application-note/AN11075.pdf) (PDF) on 2017-08-16

10. Vasquez, Joshua (2017-08-16), *Taking the leap off board: An introduction to I2C over long wires* (https://web.archive.org/web/20170816012615/http://hackaday.com/2017/02/08/taking-the-leap-off-board-an-introduction-to-i2c-over-long-wires/), archived from the original (http://hackaday.com/2017/02/08/taking-the-leap-off-board-an-introduction-to-i2c-over-long-wires/) on 2017-08-16

11. *iPack Stackable Board Format* (https://web.archive.org/web/20170819074710/http://www.mcc-us.com/ipack1.htm), 2017-08-19, archived from the original (http://www.mcc-us.com/ipack1.htm) on 2017-08-19

12. Ferrari, Mario; Ferrari, Giulio (2018-04-29). *Building Robots with LEGO Mindstorms NXT* (https://web.archive.org/web/20180429135500/https://books.google.com/books?id=1LizU1nKZO0C). pp. 63–64. ISBN 9780080554334. Archived from the original (https://books.google.com/books?id=1LizU1nKZO0C) on 2018-04-29.

13. Gasperi, Michael; Hurbain, Philippe (2010), "Chapter 13: I²C Bus Communication" (https://books.google.com/books?id=vtUPNDYSTssC), *Extreme NXT: Extending the LEGO MINDSTORMS NXT to the Next Level*, ISBN 9781430224549

14. Philo. "NXT connector plug" (http://www.philohome.com/nxtplug/nxtplug.htm) Archived (https://web.archive.org/web/20170820003822/http://www.philohome.com/nxtplug/nxtplug.htm) 2017-08-20 at the Wayback Machine

15. Sivan Toledo. "I2C Interfacing Part 1: Adding Digital I/O Ports" (http://www.tau.ac.il/~stoledo/lego/i2c-8574/) Archived (https://web.archive.org/web/20170812150214/http://www.tau.ac.il/~stoledo/lego/i2c-8574/) 2017-08-12 at the Wayback Machine. 2006

16. "Sending I2C reliably over Cat5 cables" (https://electronics.stackexchange.com/questions/107663/sending-i2c-reliably-over-cat5-cables) Archived (https://web.archive.org/web/20170818215121/https://electronics.stackexchange.com/questions/107663/sending-i2c-reliably-over-cat5-cables) 2017-08-18 at the Wayback Machine

17. "I2C Bus Connectors & Cables" (https://www.i2cchip.com/i2c_connector.html) Archived (https://web.archive.org/web/20170818022842/http://www.i2cchip.com/i2c_connector.html) 2017-08-18 at the Wayback Machine

18. https://github.com/Testato/SoftwareWire/wiki/Multiple-I2C-busses

19. https://www.microchip.com/forums/m474935.aspx

20. "I²C Address Allocation Table" (https://web.archive.org/web/20171016173844/http://simplemachines.it/doc/IC12_97_I2C_ALLOCATION.pdf) (PDF) (Selection Guide). Philips Semiconductors. 1999-08-24. Archived from the original (http://simplemachines.it/doc/IC12_97_I2C_ALLOCATION.pdf) (PDF) on 2017-10-16. Retrieved 2017-10-01.

21. Data Handbook IC12: I2C Peripherals, Philips ordering code 9397 750 00306

22. "System Management Bus (SMBus) Specification" (http://www.smbus.org/specs/SMBus_3_0_20141220.pdf#page=81) (PDF). Version 3.0. System Management Interface Forum. 2014-12-20. pp. 81–82. Archived (https://web.archive.org/web/20160129154849/http://smbus.org/specs/SMBus_3_0_20141220.pdf#page=81) (PDF) from the original on 2016-01-29. Retrieved 2017-12-01.

23. "VESA Display Data Channel Command Interface (DDC/CI) Standard" (http://www.chrisbot.com/uploads/1/3/8/4/13842915/ddcciv1r1.pdf#page=15) (PDF). Version 1.1. VESA. 2004-10-29. pp. 15–16. Archived (https://web.archive.org/web/20160909200724/http://www.chrisbot.com/uploads/1/3/8/4/13842915/ddcciv1r1.pdf#page=15) (PDF) from the original on 2016-09-09. Retrieved 2017-12-01.

24. "Intelligent Platform Management Interface Specification Second Generation V2.0" (https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ipmi-second-gen-interface-spec-v2-rev1-1.pdf#page=589) (PDF). Document Revision 1.1. Intel, NEC, Hewlett-Packard & Dell. 2013-10-01. p. 563. Archived (https://web.archive.org/web/20160327190023/http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ipmi-second-gen-interface-spec-v2-rev1-1.pdf#page=589) (PDF) from the original on 2016-03-27. Retrieved 2017-12-01. "The 7-bit portion of the slave address for the BMC is 0010_000b"

25. TWI Master Bit Band Driver; Atmel; July 2012 (http://www.atmel.com/Images/doc42010.pdf) Archived (https://web.archive.org/web/20170329171649/http://www.atmel.com/Images/doc42010.pdf) 2017-03-29 at the Wayback Machine.

26. i2c.resource component (http://www.os4depot.net/?function=showfile&file=driver/misc/i2c.resource.lha) Archived (https://web.archive.org/web/20110724123557/http://www.os4depot.net/?function=showfile&file=driver%2Fmisc%2Fi2c.resource.lha) 2011-07-24 at the Wayback Machine for AmigaOS 4.x.

27. Theo de Raadt (2015-05-29). "/sys/dev/i2c/i2c_scan.c#probe_val" (http://bxr.su/o/sys/dev/i2c/i2c_scan.c#probe_val). *Super User's BSD Cross Reference*. OpenBSD. Retrieved 2019-03-04. "static u_int8_t probe_val[256];"

28. Constantine A. Murenin (2010-05-21). "5.2. I²C bus scan through i2c_scan.c". *OpenBSD Hardware Sensors — Environmental Monitoring and Fan Control* (http://cnst.su/MMathCS) (MMath thesis). University of Waterloo: UWSpace. hdl:10012/5234 (https://hdl.handle.net/10012%2F5234). Document ID: ab71498b6b1a60ff817b29d56997a418.

29. Linear Technology's LTC4151 (http://cds.linear.com/docs/en/datasheet/4151ff.pdf) Archived (https://web.archive.org/web/20170809062230/http://cds.linear.com/docs/en/datasheet/4151ff.pdf) 2017-08-09 at the Wayback Machine has two pins for address selection, each of which can be tied high or low or left unconnected, offering 9 different addresses.

30. Maxim's MAX7314 (https://datasheets.maximintegrated.com/en/ds/MAX7314.pdf) Archived (https://web.archive.org/web/20170713011748/http://datasheets.maximintegrated.com/en/ds/MAX7314.pdf) 2017-07-13 at the Wayback Machine has a single pin for address selection to be tied high or low or connected to SDA or SCL, offering 4 different addresses.

31. TI's UCD9224 (http://www.ti.com/lit/ds/symlink/ucd9224.pdf) Archived (https://web.archive.org/web/20171107012229/http://www.ti.com/lit/ds/symlink/ucd9224.pdf) 2017-11-07 at the Wayback Machine uses two ADC channels discriminating twelve levels each to select any valid 7-bit address.

32. Delvare, Jean (2005-08-16). "Re: [PATCH 4/5] add i2c_probe_device and i2c_remove_device" (https://lkml.org/lkml/2005/8/16/156). *linux-kernel* (Mailing list). <20050816183839.0eda95e3.khali@linux-fr.org>. Archived (https://web.archive.org/web/20160817223546/https://lkml.org/lkml/2005/8/16/156) from the original on 2016-08-17.

33. avr-libc: Example using the two-wire interface (TWI) (http://www.nongnu.org/avr-libc/user-manual/group__twi__demo.html) Archived (https://web.archive.org/web/20070527183044/http://www.nongnu.org/avr-libc/user-manual/group__twi__demo.html) 2007-05-27 at the Wayback Machine.

34. "TESS -- Error" (http://tmsearch.uspto.gov/bin/showfield?f=toc&state=4803:nk9lwv.1.1&p_search=searchss&p_L=50&BackReference=&p_plural=yes&p_s_PARA1=&p_tagrepl~:=PARA1$LD&expr=PARA1+AND+PARA2&p_s_PARA2=i2c&p_tagrepl~:=PARA2$COMB&p_op_ALL=AND&a_default=search&a_search=Submit+Query&a_search=Submit+Query). *tmsearch.uspto.gov*. Retrieved 2018-04-29.

35. Thornton, Scott (2017-11-29). "The improved inter-integrated circuit (I3C)" (https://www.microcontrollertips.com/improved-inter-integrated-circuit-i3c/). *Microcontroller Tips*. Archived (https://web.archive.org/web/20180203235738/https://www.microcontrollertips.com/improved-inter-integrated-circuit-i3c/) from the original on 2018-02-03.

Further reading

- Himpe, Vincent (2011). *Mastering the I²C Bus*. ISBN 978-0-905705-98-9. (248 pages)
- Paret, Dominique (1997). *The I2C Bus: From Theory to Practice*. ISBN 978-0-471-96268-7. (314 pages)

External links

- Official I2C specification (free) (http://www.nxp.com/documents/user_manual/UM10204.pdf), NXP
- Detailed Introduction, Primer (https://www.i2c-bus.org)
- I²C Bus Technical Overview and Frequently Asked Questions (http://www.esacademy.com/faq/i2c/)
- Introduction to SPI and I2C protocols (http://www.byteparadigm.com/kb/article/AA-00255/22/Introduction-to-SPI-and-IC-protocols.html)
- Using the I²C Bus with Linux (https://www.linuxjournal.com/article/1342)
- OpenBSD iic(4) manual page (http://man.openbsd.org/iic.4)
- Serial buses information page (http://www.epanorama.net/links/serialbus.html)