WIKIPEDIA

I^{2}C

I^{2}C (**Inter-Integrated Circuit**, *eye-squared-C*), alternatively known as **I2C** or **IIC**, is a synchronous, multi-controller/multi-target (controller/target), packet switched, single-ended, serial communication bus invented in 1982 by Philips Semiconductors. It is widely used for attaching lower-speed peripheral ICs to processors and microcontrollers in short-distance, intra-board communication.

Several competitors, such as Siemens, NEC, Texas Instruments, STMicroelectronics, Motorola,[1] Nordic Semiconductor and Intersil, have introduced compatible I^{2}C products to the market since the mid-1990s.

System Management Bus (SMBus), defined by Intel in 1995, is a subset of I^{2}C, defining a stricter usage. One purpose of SMBus is to promote robustness and interoperability. Accordingly, modern I^{2}C systems incorporate some policies and rules from SMBus, sometimes supporting both I^{2}C and SMBus, requiring only minimal reconfiguration either by commanding or output pin use.

I^{2}C bus

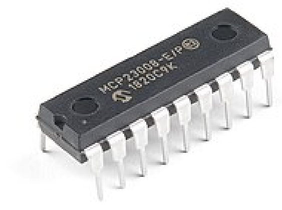
Type	Serial communication bus
Production history	
Designer	Philips Semiconductor, known today as NXP Semiconductors
Designed	1982
Data	
Data signal	Open-collector or open-drain
Width	1-bit (SDA) with separate clock (SCL)
Bitrate	0.1, 0.4, 1.0, 3.4 or 5.0 Mbit/s depending on mode
Protocol	Serial, half-duplex

Contents

Applications

I^2C is appropriate for peripherals where simplicity and low manufacturing cost are more important than speed. Common applications of the I^2C bus are:

- Describing connectable devices via small ROM configuration tables to enable plug and play operation, such as in 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 backlight, contrast, hue, color balance settings etc in monitors (via Display Data Channel).
- Changing sound volume in intelligent speakers.
- Controlling small (e.g. feature phone) LCD or OLED displays.
- Reading hardware monitors and diagnostic sensors, e.g. a fan's speed.
- Turning on and off the power supply of system components.


Microchip MCP23008 8-bit I^2C I/O expander in DIP-18 package[2]

A particular strength of I^2C 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.

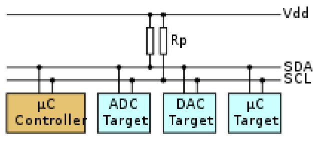
Revisions

History of I^2C specification releases

Year	Version	Notes	Refs
1981	Patent	U.S. Patent 4,689,740 filed on November 2, 1981 by U.S. Philips Corporation.	[3][4]
1982	Original	The 100 kbit/s I^2C system was created as a simple internal bus system for building control electronics with various Philips chips.	—
1992	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.	—
1998	2	Added 3.4 Mbit/s *High-speed mode (Hs)* with power-saving requirements for electric voltage and current.	[5]
2000	2.1	Clarified version 2, without significant functional changes.	[6]
2007	3	Added 1 Mbit/s *Fast-mode plus (Fm+)* (using 20 mA drivers), and a device ID mechanism.	[7]
2012	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.	[8]
2012	5	Corrected mistakes.	[9]
2014	6	Corrected two graphs.	[10]
2021	7	Changed terms "master/slave" to "controller/target" to align with I3C bus specification. Updated Table 5 assigned manufacturer IDs. Added Section 9 overview of I3C bus. This is the current standard (login required).	[11]

Design

I^2C uses only two bidirectional open-collector or open-drain lines: serial data line (SDA) and serial clock line (SCL), pulled up with resistors.[11] Typical voltages used are +5 V or +3.3 V, although systems with other voltages are permitted.



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

The I^2C reference design has a 7-bit address space, with a rarely used 10-bit extension.[12] Common I^2C 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. Later revisions of I^2C 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 that the bit rates are quoted for the transfers between controller and target without clock stretching or other hardware overhead. Protocol overheads include a target address and perhaps a register address within the target 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 target 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 I^2C 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.

I^2C modes

Mode[11]	Maximum speed	Maximum capacitance	Drive	Direction
Standard mode (Sm)	100 kbit/s	400 pF	Open drain*	Bidirectional
Fast mode (Fm)	400 kbit/s	400 pF	Open drain*	Bidirectional
Fast mode plus (Fm+)	1 Mbit/s	550 pF	Open drain*	Bidirectional
High-speed mode (Hs)	1.7 Mbit/s	400 pF	Open drain*	Bidirectional
High-speed mode (Hs)	3.4 Mbit/s	100 pF	Open drain*	Bidirectional
Ultra-fast mode (UFm)	5 Mbit/s	?	Push–pull	Unidirectional

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, either controller or target:

- Controller node: Node that generates the clock and initiates communication with targets.
- Target node: Node that receives the clock and responds when addressed by the controller.

The bus is a multi-controller bus, which means that any number of controller nodes can be present. Additionally, controller and target 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:

- Controller transmit: Controller node is sending data to a target.
- Controller receive: Controller node is receiving data from a target.
- Target transmit: Target node is sending data to the controller.
- Target receive: Target node is receiving data from the controller.

In addition to 0 and 1 data bits, the I^2C 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 controller is initially in controller transmit mode by sending a START followed by the 7-bit address of the target 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 target.

If the target exists on the bus then it will respond with an ACK bit (active low for acknowledged) for that address. The controller then continues in either transmit or receive mode (according to the read/write bit it sent), and the target 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 controller wishes to write to the target, then it repeatedly sends a byte with the target sending an ACK bit. (In this situation, the controller is in controller transmit mode, and the target is in target receive mode.)

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

An I^2C transaction may consist of multiple messages. The controller 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

I^2C defines basic types of transactions, each of which begins with a START and ends with a STOP:

- Single message where a controller writes data to a target.
- Single message where a controller reads data from a target.
- Combined format, where a controller issues at least two reads or writes to one or more targets.

In a combined transaction, each read or write begins with a START and the target address. The START conditions after the first are also called *repeated START* bits. Repeated STARTs are not preceded by STOP conditions, which is how targets know that the next message is part of the same transaction.

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

Pure I^2C 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 target. 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 I^2C target addresses), and their new configurations would take effect at the same time: when they receive that STOP.

With only a few exceptions, neither I^2C 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 I^2C *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 targets 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 target responds on subsequent reads. Most SMBus operations involve single-byte commands.

Messaging example: 24C32 EEPROM

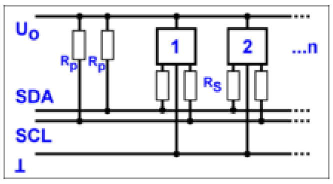
STMicroelectronics 24C08: serial EEPROM with I^2C bus[13]

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 support only 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 data to and reading from 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. I^2C 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 controller 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 I^2C 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 controller 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 controller 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



I^2C bus: R_p are pull-up resistors, R_s are optional series resistors.[11]

At the physical layer, both SCL and SDA lines are an open-drain (MOSFET) or open-collector (BJT) bus design, thus a pull-up resistor is needed for each line. 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 targets. 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[11]: 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 controller) sets SDA to the desired value and (after a small delay to let the value propagate) lets SCL float high. The controller 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 target's clock stretching.

Once SCL is high, the controller waits a minimum time (4 μs for standard-speed I^2C) 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 controller transmitting to target) The target is unable to accept the data. No such target, command not understood, or unable to accept any more data.
- (If target transmitting to controller) The controller 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 controller.

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

- Begin transferring another byte of data: the transmitter sets SDA, and the controller pulses SCL high.
- Send a "Stop": Set SDA low, let SCL go high, then let SDA go high. This releases the I^{2}C bus.
- Send a "Repeated start": Set SDA high, let SCL go high, then pull SDA low again. This starts a new I^{2}C bus message without releasing the bus.

Clock stretching using SCL

One of the more significant features of the I^{2}C protocol is clock stretching. An addressed target 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 controller that is communicating with the target may not finish the transmission of the current bit, but must wait until the clock line actually goes high. If the target is clock-stretching, the clock line will still be low (because the connections are open-drain). The same is true if a second, slower, controller tries to drive the clock at the same time. (If there is more than one controller, all but one of them will normally lose arbitration.)

The controller must wait until it observes the clock line going high, and an additional minimal time (4 µs for standard 100 kbit/s I^{2}C) before pulling the clock low again.

Although the controller may also hold the SCL line low for as long as it desires (this is not allowed since Rev. 6 of the protocol – subsection 3.1.1), the term "clock stretching" is normally used only when targets 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 target is a microcontroller, its I^{2}C 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 I^{2}C where the target drives SCL. Many targets do not need to clock stretch and thus treat SCL as strictly an input with no circuitry to drive it. Some controllers, 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 I^{2}C.

To ensure a minimal bus throughput, SMBus places limits on how far clocks may be stretched. Hosts and targets adhering to those limits cannot block access to the bus for more than a short time, which is not a guarantee made by pure I^{2}C systems.

Arbitration using SDA

Every controller monitors the bus for start and stop bits and does not start a message while another controller is keeping the bus busy. However, two controllers may start transmission at about the same time; in this case, arbitration occurs. Target transmit mode can also be arbitrated, when a controller addresses multiple targets, but this is less common. In contrast to protocols (such as Ethernet) that use random back-off delays before issuing a retry, I^{2}C 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 controller, 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 controllers are sending a message to two different targets, the one sending the lower target address always "wins" arbitration in the address stage. Since the two controllers may send messages to the same target address, and addresses sometimes refer to multiple targets, arbitration must sometimes continue into the data stages.

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

One case which must be handled carefully in multi-controller I^{2}C implementations is that of the controllers talking to each other. One controller may lose arbitration to an incoming message, and must change its role from controller to target in time to acknowledge its own address.

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

Arbitration in SMBus

While I^{2}C only arbitrates between controllers, SMBus uses arbitration in three additional contexts, where multiple targets respond to the controller, and one gets its message through.

- Although conceptually a single-controller bus, a target device that supports the "host notify protocol" acts as a controller 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 targets try to notify the host at the same time, one of them will lose arbitration and need to retry.
- An alternative target 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 targets respond with a data bytes containing their own address. When the target 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 target'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.[14] Targets may be grouped into "zones", and all targets 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 controller.

PMBus reserves I^{2}C addresses 0x28 and 0x37 for zone reads and writes, respectively.

Differences between modes

There are several possible operating modes for I^{2}C 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:

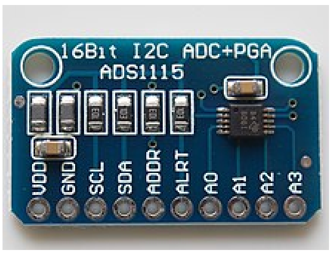
- *Fast mode* is highly compatible and simply tightens several of the timing parameters to achieve 400 kbit/s speed. *Fast mode* is widely supported by I^{2}C target devices, so a controller may use it as long as it knows that the bus capacitance and pull-up strength allow it.
- *Fast mode plus* achieves 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 I^{2}C devices on the same bus, but requires the controller 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 controller drives the clock high at the appropriate time and the target may not stretch it. All high-speed transfers are preceded by a single-byte "controller code" at fast or standard speed. This code serves three purposes:
  1. it tells high-speed target 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 controller (there are eight controller codes, and each controller 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 I^{2}C subset, which is incompatible with other modes except in that it is easy to add support for it to an existing I^{2}C interface hardware design. Only one controller is permitted, and it actively drives 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 I^{2}C bus modes is limited to:
  - the start and stop conditions are used to delimit transfers,
  - I^{2}C addressing allows multiple target devices to share the bus without SPI bus style target 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 controller(s), and a longer-than-normal bus may be operated at a slower-than-nominal speed by underclocking.

Circuit interconnections

I^2C is popular for interfacing peripheral circuits to prototyping systems, such as the Arduino and Raspberry Pi. I^2C does not employ a standardized connector, however, board designers have created various wiring schemes for I^2C 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).[15]



A 16-bit ADC board with I^2C interface

The vast majority of applications use I^2C 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 I^2C can communicate up to 20 meters (possibly over 100 meters) over CAT5 or other cable.[16][17]

Several standard connectors carry I^2C signals. For example, the UEXT connector carries I^2C; the 10-pin iPack connector carries I^2C;[18] the 6P6C Lego Mindstorms NXT connector carries I^2C;[19][20][21][22] a few people use the 8P8C connectors and CAT5 cable normally used for Ethernet physical layer to instead carry differential-encoded I^2C signals[23] or boosted single-ended I^2C signals;[24] and every HDMI and most DVI and VGA connectors carry DDC2 data over I^2C.

Buffering and multiplexing

When there are many I^2C 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 I^2C 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 I^2C to be sent over longer cables or traces. Buffers for bi-directional lines such as I^2C must use one of several schemes for preventing latch-up. I^2C 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 I^2C acknowledgement).

Sharing SCL between multiple buses

When having a single controller, it is possible to have multiple I^2C buses share the same SCL line.[25][26] 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 targets 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 I^2C message.

Line state

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

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 (controller/target)	Receiver sample bit (controller/target)	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 (controller)**	**Rising edge (controller)**	**Falling edge (controller)**	**Rising edge (controller)**	**Falling edge (controller)**	**Rising edge (controller)**

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 (controller)**
SCL	**Falling edge (controller)**	Held low	**Rising edge (controller)**	Passive high	Passive pullup

Addressing structure

7-bit addressing

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

10-bit addressing

Field:	**S**	10-bit mode indicator					Upper addr		R/W'	A	Lower address field								I^{2}C 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**		
10-bit address pos	**Start**						**10**	**9**		**ACK**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**Rest of the read or write message goes here**	**Stop**
Note		Indicates 10-bit mode					MSB		1 = Read									LSB		
									0 = Write											

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 controller code
7	1111	1XX	1	Device ID
8	1111	0XX	X	10-bit target addressing

SMBus reserves some additional addresses. In particular, `0001 000` is reserved for the SMBus host, which may be used by controller-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.

This leaves a total of 107 unreserved 7-bit addresses in common between I^{2}C, SMBus, and PMBus.

Non-reserved addresses in 7-bit address space

MSB (4-bit)	Typical usage[27][28][29][30][31]
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 target addressing, it is also used by VESA DDC display dependent devices such as pointing devices.[30]

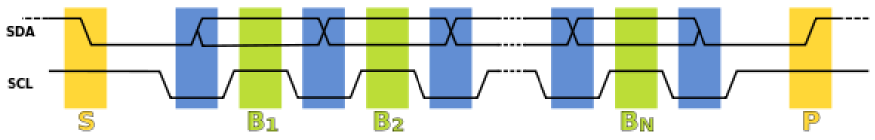
Transaction format

An I^{2}C *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 I^{2}C 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; targets are forbidden from responding if they observe a stop symbol. Configuration, calibration or self-test modes which cause the target 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) signalled 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 is 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 through 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 signalled 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 I^2C message containing n data bits (including acknowledges) contains $n + 1$ clock pulses.

Software Design

I^2C lends itself to a "bus driver" software design. Software for attached devices is written to call a "bus driver" that handles the actual low-level I^2C hardware. This permits the driver code for attached devices to port easily to other hardware, including a bit-banging design.

Example of bit-banging the I^2C protocol

Below is an example of bit-banging the I^2C protocol as an I^2C controller. The example is written in pseudo C. It illustrates all of the I^2C features described before (clock stretching, arbitration, start/stop bit, ack/nack).[32]

```c
// Hardware-specific support functions that MUST be customized:
#define I2CSPEED 100
void I2C_delay(void);
bool read_SCL(void);  // Return current level of SCL line, 0 or 1
bool read_SDA(void);  // Return current level of SDA line, 0 or 1
void set_SCL(void);   // Do not drive SCL (set pin high-impedance)
void clear_SCL(void); // Actively drive SCL signal low
void set_SDA(void);   // Do not drive SDA (set pin high-impedance)
void clear_SDA(void); // Actively drive SDA signal low
void arbitration_lost(void);

bool started = false; // global data

void i2c_start_cond(void) {
  if (started) {
    // if started, do a restart condition
    // set SDA to 1
    set_SDA();
    I2C_delay();
    set_SCL();
    while (read_SCL() == 0) { // Clock stretching
      // You should add timeout to this loop
    }

    // Repeated start setup time, minimum 4.7us
    I2C_delay();
  }

  if (read_SDA() == 0) {
    arbitration_lost();
  }

  // SCL is high, set SDA from 1 to 0.
  clear_SDA();
  I2C_delay();
  clear_SCL();
  started = true;
}

void i2c_stop_cond(void) {
  // set SDA to 0
  clear_SDA();
  I2C_delay();

  set_SCL();
  // Clock stretching
  while (read_SCL() == 0) {
    // add timeout to this loop.
  }

  // Stop bit setup time, minimum 4us
  I2C_delay();

  // SCL is high, set SDA from 0 to 1
  set_SDA();
  I2C_delay();

  if (read_SDA() == 0) {
    arbitration_lost();
  }

  started = false;
}

// Write a bit to I2C bus
void i2c_write_bit(bool bit) {
  if (bit) {
    set_SDA();
  } else {
    clear_SDA();
  }

  // SDA change propagation delay
  I2C_delay();

  // Set SCL high to indicate a new valid SDA value is available
  set_SCL();

  // Wait for SDA value to be read by target, minimum of 4us for standard mode
  I2C_delay();

  while (read_SCL() == 0) { // Clock stretching
    // You should add timeout to this loop
  }

  // SCL is high, now data is valid
  // If SDA is high, check that nobody else is driving SDA
  if (bit && (read_SDA() == 0)) {
    arbitration_lost();
  }

  // Clear the SCL to low in preparation for next change
  clear_SCL();
}

// Read a bit from I2C bus
bool i2c_read_bit(void) {
  bool bit;

  // Let the target drive data
  set_SDA();

  // Wait for SDA value to be written by target, minimum of 4us for standard mode
  I2C_delay();

  // Set SCL high to indicate a new valid SDA value is available
  set_SCL();

  while (read_SCL() == 0) { // Clock stretching
    // You should add timeout to this loop
  }

  // Wait for SDA value to be written by target, minimum of 4us for standard mode
  I2C_delay();

  // SCL is high, read out bit
  bit = read_SDA();

  // Set SCL low in preparation for next operation
  clear_SCL();

  return bit;
}

// Write a byte to I2C bus. Return 0 if ack by the target.
bool i2c_write_byte(bool send_start,
                    bool send_stop,
                    unsigned char byte) {
  unsigned bit;
  bool     nack;

  if (send_start) {
    i2c_start_cond();
  }

  for (bit = 0; bit < 8; ++bit) {
    i2c_write_bit((byte & 0x80) != 0);
    byte <<= 1;
  }

  nack = i2c_read_bit();

  if (send_stop) {
    i2c_stop_cond();
  }

  return nack;
}

// Read a byte from I2C bus
unsigned char i2c_read_byte(bool nack, bool send_stop) {
  unsigned char byte = 0;
  unsigned char bit;

  for (bit = 0; bit < 8; ++bit) {
    byte = (byte << 1) | i2c_read_bit();
  }

  i2c_write_bit(nack);

  if (send_stop) {
    i2c_stop_cond();
  }

  return byte;
}

void I2C_delay(void) {
  volatile int v;
  int i;

  for (i = 0; i < I2CSPEED / 2; ++i) {
    v;
  }
}
```

Operating system support

- In AmigaOS one can use the i2c.resource component[33] 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 I^{2}C communications natively as part of its MMBasic.
- PICAXE uses the i2c and hi2c commands.
- eCos supports I^{2}C for several hardware architectures.
- ChibiOS/RT supports I^{2}C for several hardware architectures.
- FreeBSD, NetBSD and OpenBSD also provide an I^{2}C framework, with support for a number of common 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 I^{2}C targets;[34] 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;[35] as a result, most I^{2}C sensor drivers are automatically enabled by default in applicable architectures without ill effects on stability; individual sensors, both I^{2}C 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 I^{2}C 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 I^{2}C target 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 I^{2}C targets are disabled by default in NetBSD in `GENERIC` i386/amd64 builds.
- In Linux, I^{2}C is handled with a device driver for the specific device, and another for the I^{2}C (or SMBus) adapter to which it is connected. Hundreds of such drivers are part of current Linux kernel releases.
- In Mac OS X, there are about two dozen I^{2}C kernel extensions that communicate with sensors for reading voltage, current, temperature, motion, and other physical status.
- In Microsoft Windows, I^{2}C is implemented by the respective device drivers of much of the industry's available hardware. For HID embedded/SoC devices, Windows 8 and later have an integrated I²C bus driver.[36]
- In Windows CE, I^{2}C is implemented by the respective device drivers of much of the industry's available hardware.
- Unison OS, a POSIX RTOS for IoT, supports I^{2}C for several MCU and MPU hardware architectures.
- In RISC OS, I^{2}C is provided with a generic I^{2}C interface from the IO controller and supported from the OS module system
- In Sinclair QDOS and Minerva QL operating systems I^{2}C is supported by a set of extensions provided by TF Services.

Development tools

When developing or troubleshooting systems using I^{2}C, visibility at the level of hardware signals can be important.

Host adapters

There are a number of I^{2}C host adapter hardware solutions for making a I^{2}C controller or target connection to host computers, running Linux, Mac or Windows. Most options are USB-to-I^{2}C adapters. Not all of them require proprietary drivers or APIs.

Protocol analyzers

I^{2}C protocol analyzers are tools that sample an I^{2}C 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 I^{2}C 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

On low-power systems, the pull-up resistors can use more power than the entire rest of the design combined. On these, the resistors are often powered by a switchable voltage source, such as a DIO from a microcontroller. The pull-ups also limit the speed of the bus and have a small additional cost. Therefore, some designers are turning to other serial buses, e.g. I3C or SPI, that do not need pull-ups.

The assignment of target addresses is a weakness of I^{2}C. 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 target 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.[37][38][39]

10-bit I^{2}C addresses are not yet widely used, and many host operating systems do not support them.[40] 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. 3 version of the I^{2}C specification adds a device ID mechanism.

I^{2}C 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). I^{2}C 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 I^{2}C 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 controller 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 I^{2}C 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

I^{2}C 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 I^{2}C for shelf management. Multi-controller I^{2}C 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.[41] Vendors use the name TWI, even though I^{2}C is not a registered trademark as of 2014-11-07.[42] Trademark protection only exists for the respective logo (see upper right corner), and patents on I^{2}C have now lapsed. According to Microchip Technology, TWI and I2C have a few differences. One of them is that TWI does not support START byte.[43]

In some cases, use of the term "two-wire interface" indicates incomplete implementation of the I^{2}C specification. Not supporting arbitration or clock stretching is one common limitation, which is still useful for a single controller communicating with simple targets that never stretch the clock.

MIPI I3C sensor interface standard (I3C) is a development of I^{2}C, under development in 2017.[44]

See also

- List of network buses

- ACCESS.bus
- I3C
- Power Management Bus
- System Management Bus
- UEXT Connector
- VESA Display Data Channel

References

1. "Financial Press Releases-NXP" (http://investors.nxp.com/phoenix.zhtml?c=209114&p=irol-newsArticle&ID=2120581). *investors.nxp.com*. Retrieved 2018-04-29.
2. "MCP23008" (https://www.microchip.com/wwwproducts/en/MCP23008). *Microchip*. May 26, 2021. Archived (https://web.archive.org/web/20210526160236/https://www.microchip.com/wwwproducts/en/MCP23008) from the original on May 26, 2021.
3. US Patent 4689740 (https://patents.google.com/patent/US4689740A/en), "Two-Wire Bus-System Comprising A Clock Wire And A Data Wire For Interconnecting A Number Of Stations", issued 1987-08-25, assigned to U.S. Philips Corporation
4. "Philips sues eight more companies for infringement of I2C bus patent" (https://www.eetimes.com/philips-sues-eight-more-companies-for-infringement-of-i2c-bus-patent/). EE Times. October 17, 2001. Archived (https://web.archive.org/web/20210402162144/https://www.eetimes.com/philips-sues-eight-more-companies-for-infringement-of-i2c-bus-patent/#) from the original on April 2, 2021.
5. I^2C-bus specification Rev 2.0; Philips Semiconductors; December 1998; Archived. (https://web.archive.org/web/20200528003930/http://esd.cs.ucr.edu/webres/i2c20.pdf)
6. I^2C-bus specification Rev 2.1; Philips Semiconductors; January 2000; Archived. (https://web.archive.org/web/20070212223025/http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf)
7. I^2C-bus specification Rev 3; NXP Semiconductors; June 19, 2007; Archived. (https://web.archive.org/web/20120207003629/http://www.nxp.com/documents/user_manual/UM10204.pdf)
8. I^2C-bus specification Rev 4; NXP Semiconductors; February 13, 2012; Archived. (https://web.archive.org/web/20120502134707/http://www.nxp.com/documents/user_manual/UM10204.pdf)
9. I^2C-bus specification Rev 5; NXP Semiconductors; October 9, 2012; Archived. (https://web.archive.org/web/20121017153327/http://www.nxp.com/documents/user_manual/UM10204.pdf)
10. "I^2C-bus specification Rev 6" (https://web.archive.org/web/20210426060837/https://www.nxp.com/docs/en/user-guide/UM10204.pdf) (PDF). NXP Semiconductors. April 4, 2014. Archived from the original (https://www.nxp.com/docs/en/user-guide/UM10204.pdf) (PDF) on April 26, 2021.
11. "I^2C-bus specification Rev 7" (https://www.nxp.com/docs/en/user-guide/UM10204.pdf) (PDF). NXP Semiconductors. October 1, 2021.
12. "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.
13. "8-Kbit serial I^2C 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.
14. *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.
15. "Is there any definitive I2C pin-out guidance out there? Not looking for a "STANDARD" " (http://electronics.stackexchange.com/a/48343). StackExchange.
16. *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
17. 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
18. *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
19. 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.
20. Gasperi, Michael; Hurbain, Philippe (2010), "Chapter 13: I^2C Bus Communication" (https://books.google.com/books?id=vtUPNDYSTssC), *Extreme NXT: Extending the LEGO MINDSTORMS NXT to the Next Level*, ISBN 9781430224549
21. 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
22. 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

23. "Sending I2C reliabily over Cat5 cables" (https://electronics.stackexchange.com/questions/107663/sending-i2c-reliabily-over-cat5-cables) Archived (https://web.archive.org/web/20170818215121/https://electronics.stackexchange.com/questions/107663/sending-i2c-reliabily-over-cat5-cables) 2017-08-18 at the Wayback Machine
24. "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
25. "Multiple I2C buses · Testato/SoftwareWire Wiki" (https://github.com/Testato/SoftwareWire/wiki/Multiple-I2C-buses). *GitHub*.
26. "Sharing I2C bus | Microchip" (https://www.microchip.com/forums/m474935.aspx).
27. "I^{2}C 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.
28. Data Handbook IC12: I2C Peripherals, Philips ordering code 9397 750 00306
29. "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.
30. "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.
31. "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"
32. 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.
33. 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.
34. 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];`"
35. Constantine A. Murenin (2010-05-21). "5.2. I^{2}C 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.
36. Introduction to HID over I2C (https://docs.microsoft.com/en-us/windows-hardware/drivers/hid/hid-over-i2c-guide)
37. 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.
38. 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.
39. 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.
40. 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). Archived (https://web.archive.org/web/20160817223546/https://lkml.org/lkml/2005/8/16/156) from the original on 2016-08-17.
41. 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.
42. "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.
43. "What is TWI? How to Configure the TWI for I2C Communication" (http://ww1.microchip.com/downloads/en/DeviceDoc/90003181A.pdf) (PDF). Microchip Technology. 2018.
44. 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 I^{2}C 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 I^{2}C Specification Rev 6 (free) (https://web.archive.org/web/20210813122132/https://www.nxp.com/docs/en/user-guide/UM10204.pdf) - NXP
- Detailed I^{2}C Introduction & Primer (https://www.i2c-bus.org)
- I^{2}C Pullup Resistor Calculation (https://www.ti.com/lit/an/slva689/slva689.pdf) - TI
- Effects of Varying I^{2}C Pullup Resistors (Scope Captures of 5V I^{2}C with 9 Different Pullup Resistances) (https://web.archive.org/web/20170730190053/http://dsscircuits.com:80/articles/effects-of-varying-i2c-pull-up-resistors)