I²C

Not to be confused with I²S.

I²C (**Inter-Integrated Circuit**), pronounced *I-squared-C*, is a multi-master, multi-slave, single-ended, serial computer bus invented by Philips Semiconductor (now NXP Semiconductors). It is typically 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 still required to obtain I²C slave addresses allocated by NXP.[1]

Several competitors, such as Siemens AG (later Infineon Technologies AG, 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.

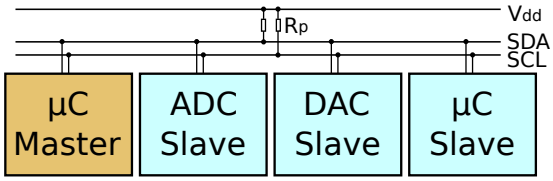
SMBus, defined by Intel in 1995, is a subset of I²C that defines the protocol use more strictly. One purpose of SMBus is to promote robustness and interoperability. Accordingly, modern I²C systems incorporate some policies and rules from SMBus, sometimes supporting both I²C and SMBus, requiring only minimal reconfiguration either by commanding or output pin use.

1 Revisions

The history of I²C specification releases:

- In 1982, the original 100 kHz I²C system was created as a simple internal bus system for building control electronics with various Philips chips.
- In 1992, Version 1 added 400 kHz *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 MHz *High-speed mode (Hs)* with power-saving requirements for electric voltage and current.
- In 2000, Version 2.1 introduced a minor cleanup of version 2.
- In 2007, Version 3 added 1 MHz *Fast-mode plus (Fm+)*, and a device ID mechanism.
- In 2012, Version 4 added 5 MHz *Ultra Fast-mode (UFm)* for new USDA (data) and USCL (clock) lines using push-pull logic without pull-up resistors, and added assigned manufacturer ID table. It is only an **Unidirectional** bus.
- In 2012, Version 5 corrected mistakes.
- In 2014, Version 6 corrected two graphs. This is the most recent standard.[3]

2 Design



A sample schematic with one master (a microcontroller), three slave nodes (an ADC, a DAC, and a microcontroller), and pull-up resistors Rp

I²C uses only two bidirectional open-drain lines, Serial Data Line (SDA) and Serial Clock Line (SCL), pulled up with resistors. 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 or a 10-bit (depending on the device used) address space.[4] Common I²C bus speeds are the 100 kbit/s *standard mode* and the 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* or Fm+, and 3.4 Mbit/s *High Speed mode*). These speeds are more widely used on embedded systems than on PCs. There are also other features, such as 16-bit addressing.

Note the bit rates are quoted for the transactions 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 maximum number of nodes 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.

2.1 Reference design

The before mentioned 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 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

The master is initially in master transmit mode by sending a start bit 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 its complementary mode (receive or transmit, respectively).

The address and the data bytes are sent most significant bit first. The start bit is indicated by a high-to-low transition of SDA with SCL high; the stop bit 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 but the last one. (In this situation, the master is in master receive mode and the slave is in slave transmit mode.)

The master then either ends transmission with a stop bit, or it may send another START bit if it wishes to retain control of the bus for another transfer (a “combined message”).

2.2 Message protocols

I^2C defines basic types of messages, 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 messages, where a master issues at least two reads and/or writes to one or more slaves.

In a combined message, each read or write begins with a START and the slave address. After the first START in a combined message these are also called *repeated START* bits. Repeated START bits are not preceded by STOP bits, which is how slaves know the next transfer is part of the same message.

Any given slave 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 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 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 slaves adopt request/response control models, where one or more bytes following a write com-

mand are treated as a command or address. Those bytes determine how subsequent written bytes are treated and/or how the slave responds on subsequent reads. Most SMBus operations involve single byte commands.

2.3 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 to address bytes within the 32 kbit (4 kB) supported by that EEPROM; the same two byte addressing is also used by larger EEPROMs, such as 24c512 ones storing 512 kbits (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. (That data transfer part of the protocol also makes trouble for 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 kbits, such as 2 kbit 24c02 ones, are often used on SMBus with inefficient single byte data transfers.)

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 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 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.

2.4 Physical layer

At the physical layer, both SCL and SDA lines are of open-drain design, thus, pull-up resistors are needed. Pulling the line to ground is considered a logical zero while letting the line float is a logical one. This is used as a channel access method. High speed systems (and some others) also add a current source pull up, at least on SCL; this accommodates higher bus capacitance and enables 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 see this, and thereby know that another node is active at the same time.

When used on SCL, this is called *clock stretching* and gives slaves a flow control mechanism. When used on SDA, this is called arbitration and ensures 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. Releasing SDA to float high again would be a stop marker, signaling the end of a bus transaction. Although legal, this is typically pointless immediately after a start, 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 I^2C) to ensure 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.

During the acknowledgment, SCL is always controlled by the master.

After the acknowledge bit, the master may do one of three things:

- Prepare to transfer another byte of data: the transmitter set 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, and pull SDA low again. This starts a new I²C bus transaction without releasing the bus.

2.4.1 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 minimum 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 minimum 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.

2.4.2 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’s 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.

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. Slaves that can be accessed by multiple masters must have commands that are idempotent for this reason.

2.4.3 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.

2.5 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, and 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).[5]

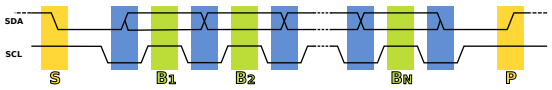
2.6 Buffering and multiplexing

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).

2.7 Timing diagram



Data transfer sequence

1. Data Transfer is initiated with a START bit (S) signaled by SDA being pulled low while SCL stays high.
2. SDA sets the 1st data bit level while keeping SCL low (during blue bar time.)
3. The data is sampled (received) when SCL rises (green) for the first bit (B1).
4. This process repeats, SDA transitioning while SCL is low, and the data being read while SCL is high (B2, Bn).
5. A STOP bit (P) is signaled when SDA is pulled high while SCL is high.

In order to avoid false marker detection, SDA is changed on the SCL falling edge and is sampled and captured on the rising edge of SCL.

2.8 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).[6]

// Hardware-specific support functions that MUST be customized: #define I2CSPEED 100 void I2C_delay(void); bool read_SCL(void); // Set SCL as input and return current level of line, 0 or 1 bool read_SDA(void); // Set SDA as input and return current level of line, 0 or 1 void set_SCL(void); // Actively drive SCL signal high void clear_SCL(void); // Actively drive SCL signal low void set_SDA(void); // Actively drive SDA signal high 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 cond // 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(); } I2C_delay(); 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 slave, 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 slave drive data set_SDA(); // Wait for SDA value to be written by slave, 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 slave, 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 slave. 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; } }

3 Applications

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

- Reading configuration data from SPD EEPROMs on SDRAM, DDR SDRAM, DDR2 SDRAM memory sticks (DIMM) and other stacked PC boards
- Supporting systems management for PCI cards, through an SMBus 2.0 connection.
- Accessing NVRAM chips that keep user settings.
- Accessing low speed DACs and ADCs.
- Changing contrast, hue, and color balance settings in monitors (Display Data Channel).
- Changing sound volume in intelligent speakers.
- Controlling OLED/LCD displays, like in a cellphone.
- Reading hardware monitors and diagnostic sensors, like a CPU thermistor or fan speed.[7]
- Reading real-time clocks.
- Turning on and turning off the power supply of system components.[8]

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, require more pins and signals to connect devices.

4 Operating system support

- In AmigaOS one can use the i2c.resource component[9] 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.
- 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 which 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 via a set of extensions provided by TF Services.

5 Development tools

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

5.1 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 Universal Serial Bus (USB) to I²C adapters. Not all of them require proprietary drivers or APIs.

5.2 I²C protocol analyzers

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

5.3 Logic analyzers

When developing and/or troubleshooting the I²C bus, examination of hardware signals can be very important. Logic analyzers are tools which collect, analyze, decode, and store signals so people can view the high-speed waveforms at their leisure. Logic analyzers display timestamps 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.

6 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, and manufacturers rarely dedicate enough pins to configure the full slave address used on a given board. Three pins is typical, giving only eight choices of slave address. While some devices can set multiple address bits per pin,[10][11] e.g., by using a spare internal ADC channel to sense one of eight ranges set by an external voltage divider, usually each pin controls one address bit. Manufacturers may provide pins to configure a few low order bits of the address and arbitrarily set the higher order bits to some value based on the model. This limits the number of devices of that model which may be present on the same bus to some low number, typically between two and eight. That partially addresses the issue of address collisions between different vendors. Ten-bit I²C addresses are not yet widely used, and many host operating systems do not support them.[12] 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+ one-megabit 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).

7 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.[13] Vendors use the name TWI, even though I²C is not a registered trademark. 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, that is still useful for a single master communicating with simple slaves that never stretch the clock.

8 See also

- DVI
- DisplayPort
- HDMI
- List of network buses
- UEXT Connector
- VGA
- System Management Bus

9 References

[1] I²C Licensing Information

[2] Freescale merger with NXP

[3] Official I2C Specification Version 6

[4] 7-bit, 8-bit, and 10-bit I2C Slave Addressing

[5] Is there any definitive I2C pin-out guidance out there? Not looking for a "STANDARD"; StackExchange.

[6] TWI Master Bit Band Driver; Atmel; July 2012.

[7] Speedfan - reading computer hardware monitoring chips Archived August 16, 2015, at the Wayback Machine.

[8] "Benefits of Power Supplies Equipped with I2C Ethernet Communications". *Aegis Power Systems, Inc.* Aegis Power Systems, Inc. Retrieved 21 December 2015.

[9] i2c.resource component for AmigaOS 4.x

[10] Maxim's MAX7314 uses a common purely digital low/high/SDA/SCL scheme to configure four addresses per address pin.

[11] TI's UCD9112 uses two ADC channels to select any valid 7-bit address.

[12] https://lkml.org/lkml/2005/8/16/156

[13] avr-libc: Example using the two-wire interface (TWI)

10 Further reading

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

11 External links

11.1 Official

- Official I2C specification (free), NXP
- List of assigned NXP / Philips I2C addresses (free), NXP

11.2 Other

- Detailed Introduction, Primer
- Tackling I2C Development Complexities Using Innovative Tools
- Advanced I^2C Bus Analysis Webinar
- I^2C Background
- I^2C Bus / Access Bus
- Using the I^2C Bus with Linux
- Proven Implementations of the I^2C Bus
- OpenBSD iic(4) manual page
- lm-sensors, Linux-monitoring sensors package which supports sensors using the I^2C bus, among others
- I^2C Tools for Linux, Tools to access I^2C and SMBus devices
- massmind I^2C page Source code, samples and technical information for using I^2C with PC, PIC and SX microcontrollers.
- Serial buses information page
- I^2C Bus Technical Overview and Frequently Asked Questions
- The Bus Buffer Resource. For 2-wire buses such as I^2C, SMBus, PMBus, IPMB & IPMI
- I^2C Protocol
- I^2C logic microchips from Texas Instruments
- A commercial hardware implementation of I^2C written in VHDL
- OpenCores open source hardware implementation, in Verilog and VHDL
- Introduction to SPI and I2C protocols
- I2C Protocol Decode software **(unavailable since July 7, 2014; not temporary?)** Archived March 11, 2012, at the Wayback Machine.
- Beginner's guide to using Arduino with I^2C devices, including worked examples
- Effects of Varying I^2C Pull-up Resistors
- I2C Tutorial
- How I2C Communication Works

12 Text and image sources, contributors, and licenses

12.1 Text

- **I²C** *Source:* https://en.wikipedia.org/wiki/I%C2%B2C?oldid=736260761 *Contributors:* Maury Markowitz, Heron, Nixdorf, Kku, CesarB, Ellywa, William M. Connolley, Glenn, Crissov, Zoicon5, Echoray, Omegatron, Ortonmc, Chealer, Martinwguy, Knobunc, BenFrantzDale, Ds13, Joconnor, Jason Quinn, Whitis, Mboverload, Bobblewik, OverlordQ, Kiteinthewind, Eric B. and Rakim, DanMatan, BrianWilloughby, Vijaykumar~enwiki, GreenReaper, RevRagnarok, Danh, Rich Farmbrough, Sladen, Bender235, Mykhal, Plugwash, Kwamikagami, Dajhorn, Vinsci, Jantangring, Siddharthc~enwiki, Polluks, Giraffedata, Thesmog, Hooperbloob, RyanTMulligan, Guy Harris, Atlant, Hopp, Cburnett, Dan East, Unixxx, QUILzhunter931, Poppafuze, Drichards2, Macaddct1984, Eyreland, Alecv, Yuriybrisk, Volland, Tizio, Arisa, Brighterorange, Utzig, Gavinatkinson, FlaBot, Lmatt, Chobot, YurikBot, Wavelength, Todd Vierling, Crazytales, Shaddack, PhilipO, Morcheeba, Cedar101, Orbit02, Benandorsqueaks, Benhoyt, KnightRider~enwiki, Qoqnous, SmackBot, Lordfuzzz, Eveningmist, JeffyP, Gilliam, Kazkaskazkasako, Chris the speller, Thumperward, Adpete, Kostmo, Krallja, Cantalamessa, Prattmic, Alphathon, Aggsal, Cybercobra, Luís Felipe Braga, Mingthemad, SpareHeadOne, Changtaiyeh~enwiki, RomanSpa, Beard0, Caiaffa, Iridescent, Ronaldvd, Anon user, Irwangatot, HenkeB, Phatom87, Jamie Lokier, Garrickk, Shadwstalkr, Pipatron, Thijs!bot, RLE64, Electron9, DirkHelgemo, Widefox, Ebikeguy, Behnam Ghiaseddin, JAnDbot, Deflective, Gstein, Mrwhizzard, Tyson882, Robinmholt, Midgrid, Mfloryan, S3000, Moggie2002, R'n'B, Lijojohnc, Dmitri Yuriev, Ajfweb, H1voltage, Logicalelegance, Reelrt, VolkovBot, Charleca, Turm, Wiae, Xreal, Nave.notnilc, Ulf Abrahamsson~enwiki, Thunderbird2, Kbrose, Gerakibot, Crm123, Hawk777, Nmh, Fahidka, DaBler, Faradayplank, Lightmouse, Dabbdabb, KJG2007, ClueBot, Intx13, Fyyer, Cab.jones, Mild Bill Hiccup, Auntof6, RamanGupta16, Amolhshah, Ranjithsutari, Mechanicalsoldier, DumZiBoT, Telagam.dinakar, Dgtsyb, Vianello, Addbot, Mortense, Ki162, Jelsova, MrOllie, Fiftyquid, Legobot, Yobot, Fraggle81, AnomieBOT, Materialscientist, Simonjohndoherty, Eumolpo, Xqbot, Drilnoth, Asdf39, Vaesumed, KlimkeM, SassoBot, Reply123, AnotherOnymous, Username20090319, Glider87, Mfwitten, Rhipps, DrilBot, Yupferris, MastiBot, Ksacilotto, Orenburg1, Hoptroff, Crackwitz, Vrenator, Cwavnew, Cp82, Limited Atonement, EmausBot, MarioBlunk, Dewritech, GoingBatty, Smcclos, Matthieu CASTET, Wikipelli, John Cline, Fæ, Marguel, Jwortzel, Brianetta, Encijia, Tolly4bolly, Sbmeirow, JohnBoxall, Avrguru, ClueBot NG, Widr, Viswanathamk, Helpful Pixie Bot, Strike Eagle, Gibster777, Wbm1058, DaveB549, Keshava G N, TimGremalm, Cyberbot II, Walyer Runson, عبدالرحمن مسادم, Khazar2, Vishesh3012, Nitin.goyal.gl, Faizan, PuZZleDucK, Llamawright, Bsreddyb7, Comp.arch, Spider 2004, Ginsuloft, Knivd, Vozul, TezzaC73, Lochotzke, So-retro-it-hurts, Galaviel, Nicole Miller1 and Anonymous: 345

12.2 Images

- **File:Commons-logo.svg** *Source:* https://upload.wikimedia.org/wikipedia/en/4/4a/Commons-logo.svg *License:* CC-BY-SA-3.0 *Contributors:* ? *Original artist:* ?
- **File:Folder_Hexagonal_Icon.svg** *Source:* https://upload.wikimedia.org/wikipedia/en/4/48/Folder_Hexagonal_Icon.svg *License:* Cc-by-sa-3.0 *Contributors:* ? *Original artist:* ?
- **File:I2C.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/3/3e/I2C.svg *License:* CC-BY-SA-3.0 *Contributors:* Own work made with Inkscape *Original artist:* en:user:Cburnett
- **File:I2C_data_transfer.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/6/64/I2C_data_transfer.svg *License:* Public domain *Contributors:* Own work *Original artist:* Marcin Floryan
- **File:Nuvola_apps_ksim.png** *Source:* https://upload.wikimedia.org/wikipedia/commons/8/8d/Nuvola_apps_ksim.png *License:* LGPL *Contributors:* http://icon-king.com *Original artist:* David Vignoni / ICON KING
- **File:Text_document_with_red_question_mark.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/a/a4/Text_document_with_red_question_mark.svg *License:* Public domain *Contributors:* Created by bdesham with Inkscape; based upon Text-x-generic.svg from the Tango project. *Original artist:* Benjamin D. Esham (bdesham)

12.3 Content license

- Creative Commons Attribution-Share Alike 3.0