Introduction to I²C and SPI protocols

I²C vs SPI

Today, at the low end of the communication protocols, we find I²C (for 'Inter-Integrated Circuit', protocol) and SPI (for 'Serial Peripheral Interface'). Both protocols are well-suited for communications between integrated circuits, for slow communication with on-board peripherals. At the roots of these two popular protocols we find two major companies – Philips for I²C and Motorola for SPI – and two different histories about why, when and how the protocols were created.

The I²C bus was developed in 1982; its original purpose was to provide an easy way to connect a CPU to peripherals chips in a TV set. Peripheral devices in embedded systems are often connected to the microcontroller as memory-mapped I/O devices. One common way to do this is connecting the peripherals to the microcontroller parallel address and data busses. This results in lots of wiring on the PCB (printed circuit board) and additional 'glue logic' to decode the address bus on which all the peripherals are connected. In order to spare microcontroller pins, additional logic and make the PCBs simpler – in order words, to lower the costs – Philips labs in Eindhoven (The Netherlands) invented the 'Inter-Integrated Circuit', IIC or I²C protocol that only requires 2 wires for connecting all the peripheral to a microcontroller. The original specification defined a bus speed of 100 kbps (kilo bits per second). The specification was reviewed several times, notably introducing the 400 kbps speed in 1995 and – since 1998, 3.4 Mbps for even faster peripherals.

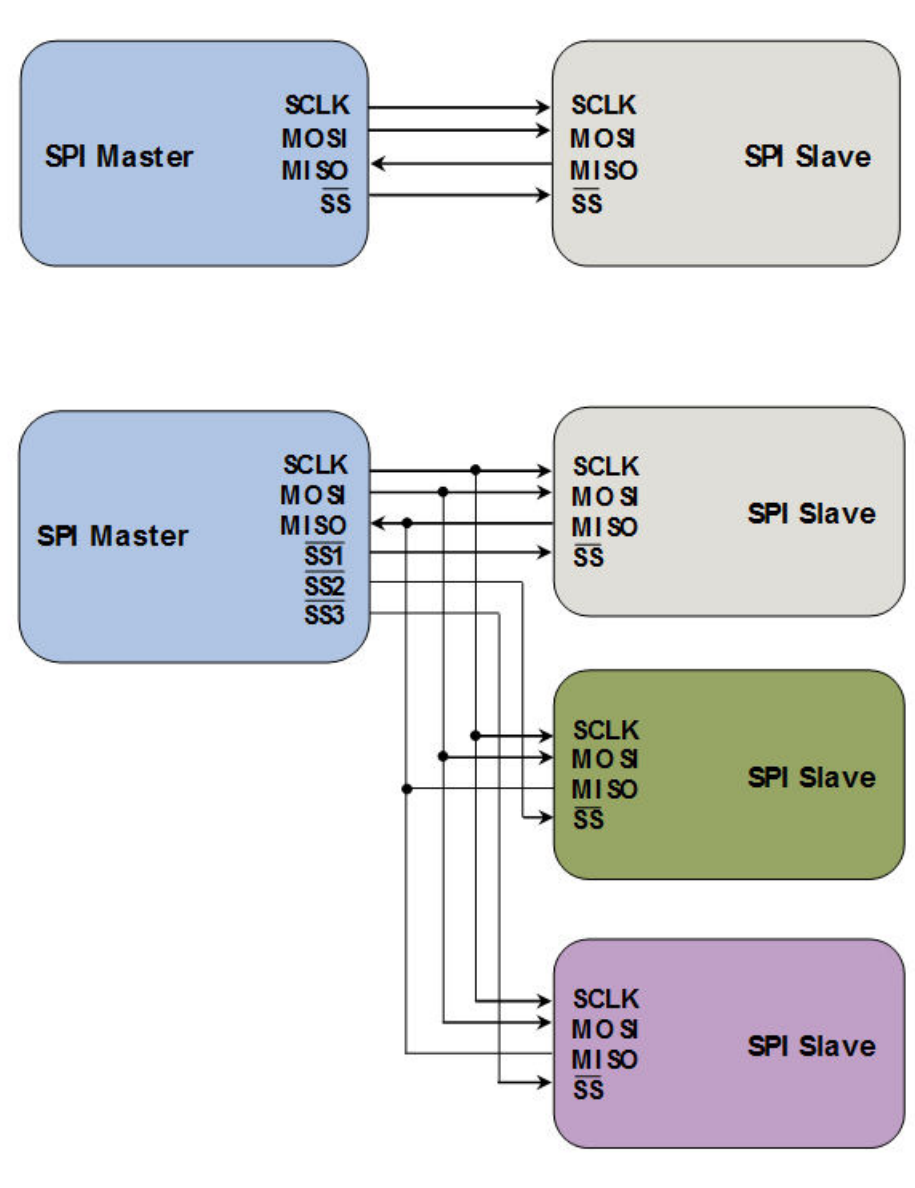
It seems the Serial Peripheral Protocol (SPI) was first introduced with the first microcontroller deriving from the same architecture as the popular Motorola 68000 microprocessor, announced in 1979. SPI defined the external microcontroller bus, used to connect the microcontroller peripherals with 4 wires. Unlike I²C, it is hard to find a formal separate 'specification' of the SPI bus – for a detailed 'official' description, one has to read the microcontrollers data sheets and associated application notes.

SPI

SPI is quite straightforward – it defines features any digital electronic engineer would think of if it were to quickly define a way to communicate between 2 digital devices. SPI is a protocol on 4 signal lines (please refer to figure 1):

– A clock signal named SCLK, sent from the bus master to all slaves; all the SPI signals are synchronous to this clock signal;
– A slave select signal for each slave, SSn, used to select the slave the master communicates with;
– A data line from the master to the slaves, named MOSI (Master Out-Slave In)
– A data line from the slaves to the master, named MISO (Master In-Slave Out).

Figure 1 : Two SPI busses topologies. The upper figure shows a SPI master connected to a single slave (point-to-point topology). The lower figure shows a SPI master connected to multiple slaves.



SPI is a single-master communication protocol. This means that one central device initiates all the communications with the slaves. When the SPI master wishes to send data to a slave and/or request information from it, it selects slave by pulling the corresponding SS line low and it activates the clock signal at a clock frequency usable by the master and the slave. The master generates information onto MOSI line while it samples the MISO line (refer to figure 2).

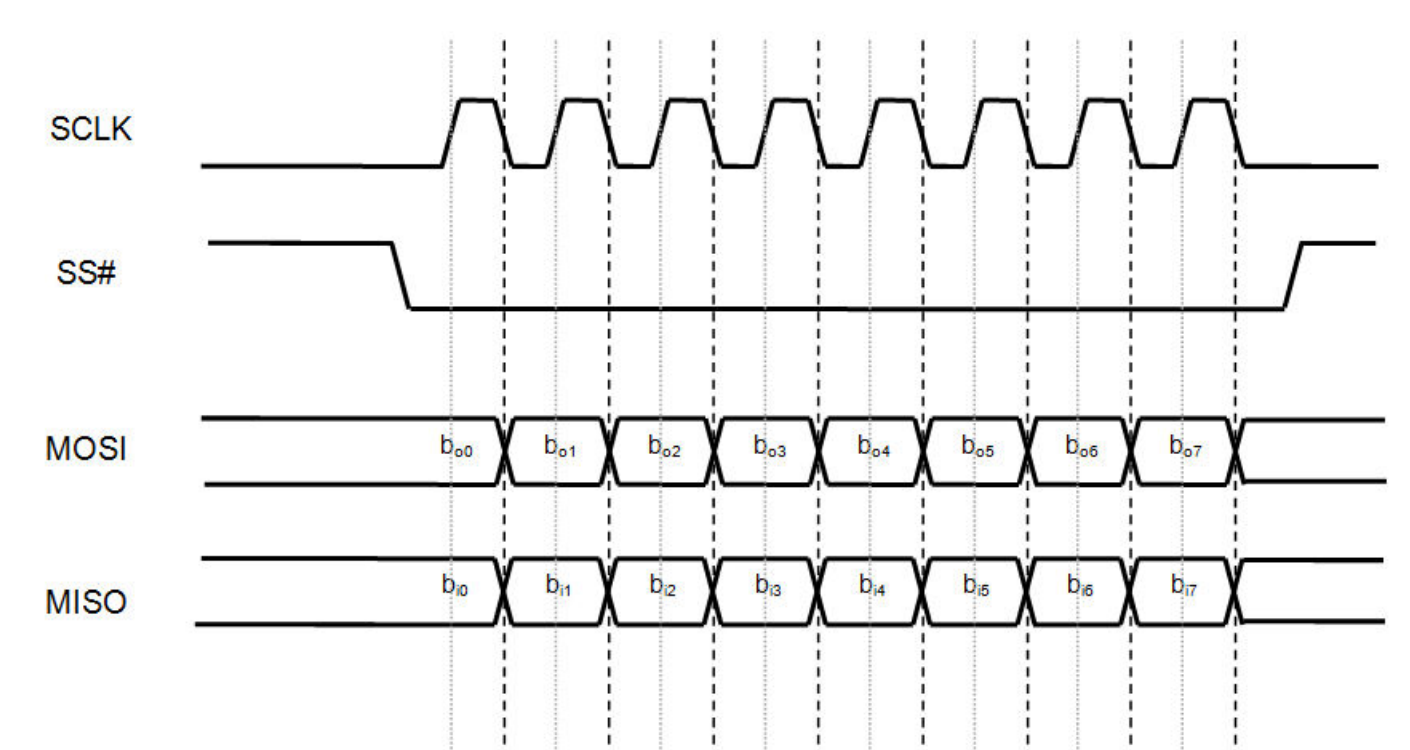


Figure 2 : A simple SPI communication. Data bits on MOSI and MISO toggle on the SCLK falling edge and are sampled on the SCLK rising edge. The SPI mode defines which SCLK edge is used for toggling data and which SCLK edge is used for sampling data.

Four communication modes are available (MODE 0, 1, 2, 3) – that basically define the SCLK edge on which the MOSI line toggles, the SCLK edge on which the master samples the MISO line and the SCLK signal steady level (that is the clock level, high or low, when the clock is not active). Each mode is formally defined with a pair of parameters called 'clock polarity' (CPOL) and 'clock phase' (CPHA).

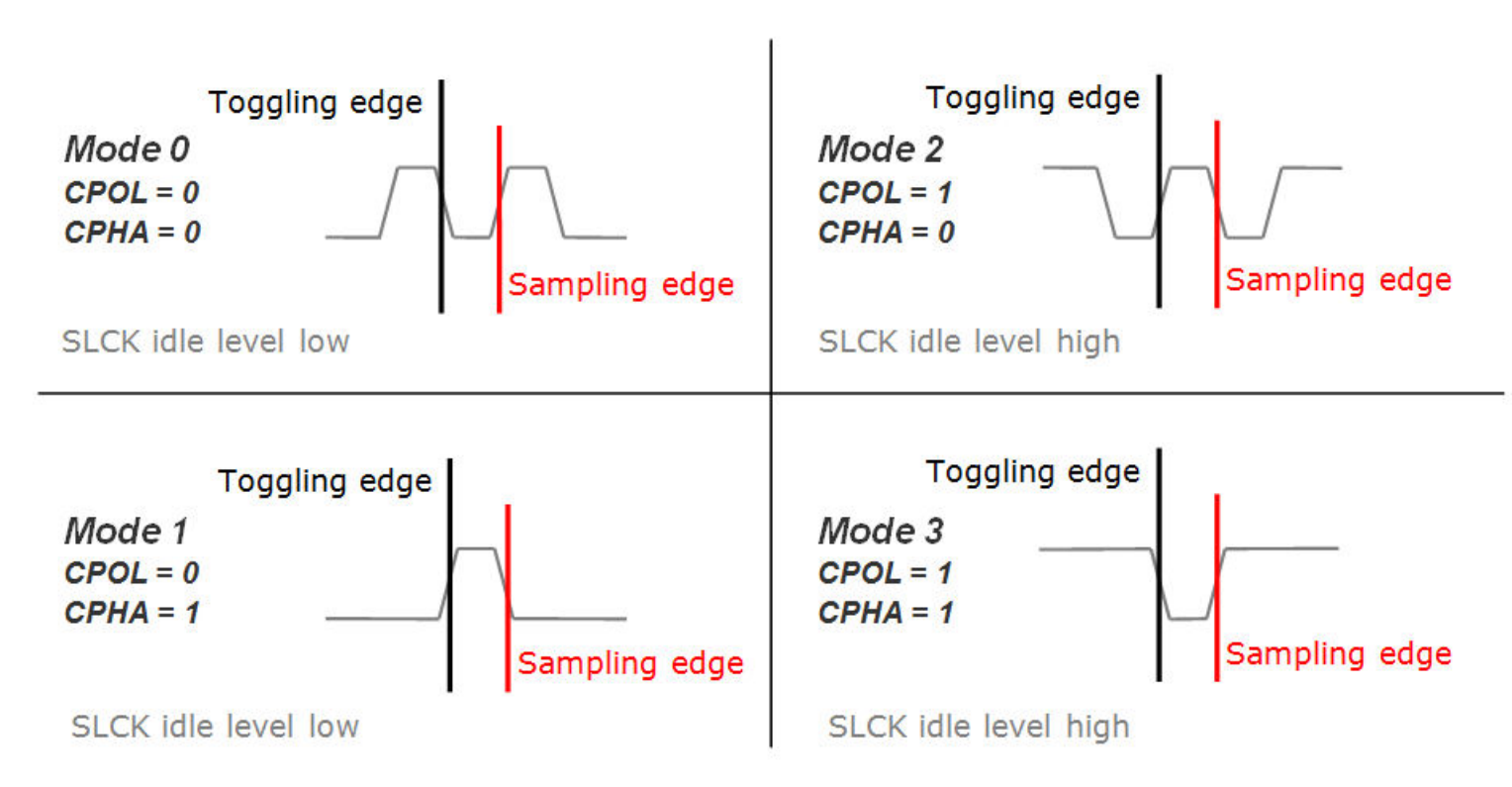


Figure 3 : SPI modes are defined with the parameters 'CPOL' – clock polarity and 'CPHA' – clock phase, which explicitly define 3 parameters: the edges used for data sampling and data toggling and the SCL clock signal idle level – that is the conventional level SCLK is set at when the bus is not in communication.

A master/slave pair must use the same set of parameters – SCLK frequency, CPOL, and CPHA for a communication to be possible. If multiple slaves are used, that are fixed in different configurations, the master will have to reconfigure itself each time it needs to communicate with a different slave.

This is basically all what is defined for the SPI protocol. SPI does not define any maximum data rate, not any particular addressing scheme; it does not have a acknowledgement mechanism to confirm receipt of data and does not offer any flow control. Actually, the SPI master has no knowledge of whether a slave exists, unless 'something' additional is done outside the SPI protocol. For example a simple codec won't need more than SPI, while a command-response type of control would need a higher-level protocol built on top of the SPI interface. SPI does not care about the physical interface characteristics like the I/O voltages and standard used between the devices. Initially, most SPI implementation used a non-continuous clock and byte-by-byte scheme. But many variants of the protocol now exist, that use a continuous clock signal and an arbitrary transfer length.

Back to Top

I²C

I²C is a multi-master protocol that uses 2 signal lines. The two I²C signals are called 'serial data' (SDA) and 'serial clock' (SCL). There is no need of chip select (slave select) or arbitration logic. Virtually any number of slaves and any number of masters can be connected onto these 2 signal lines and communicate between each other using a protocol that defines:

– 7-bits slave addresses: each device connected to the bus has got such a unique address;
– data divided into 8-bit bytes
– a few control bits for controlling the communication start, end, direction and for an acknowledgment mechanism.

The data rate has to be chosen between 100 kbps, 400 kbps and 3.4 Mbps, respectively called standard mode, fast mode and high speed mode. Some I²C variants include 10 kbps (low speed mode) and 1 Mbps (fast mode +) as valid speeds.

Physically, the I²C bus consists of the 2 active wires SDA and SCL and a ground connection (refer to figure 4). The active wires are both bi-directional. The I2C protocol specification states that the IC that initiates a data transfer on the bus is considered the Bus Master. Consequently, at that time, all the other ICs are regarded to be Bus Slaves.

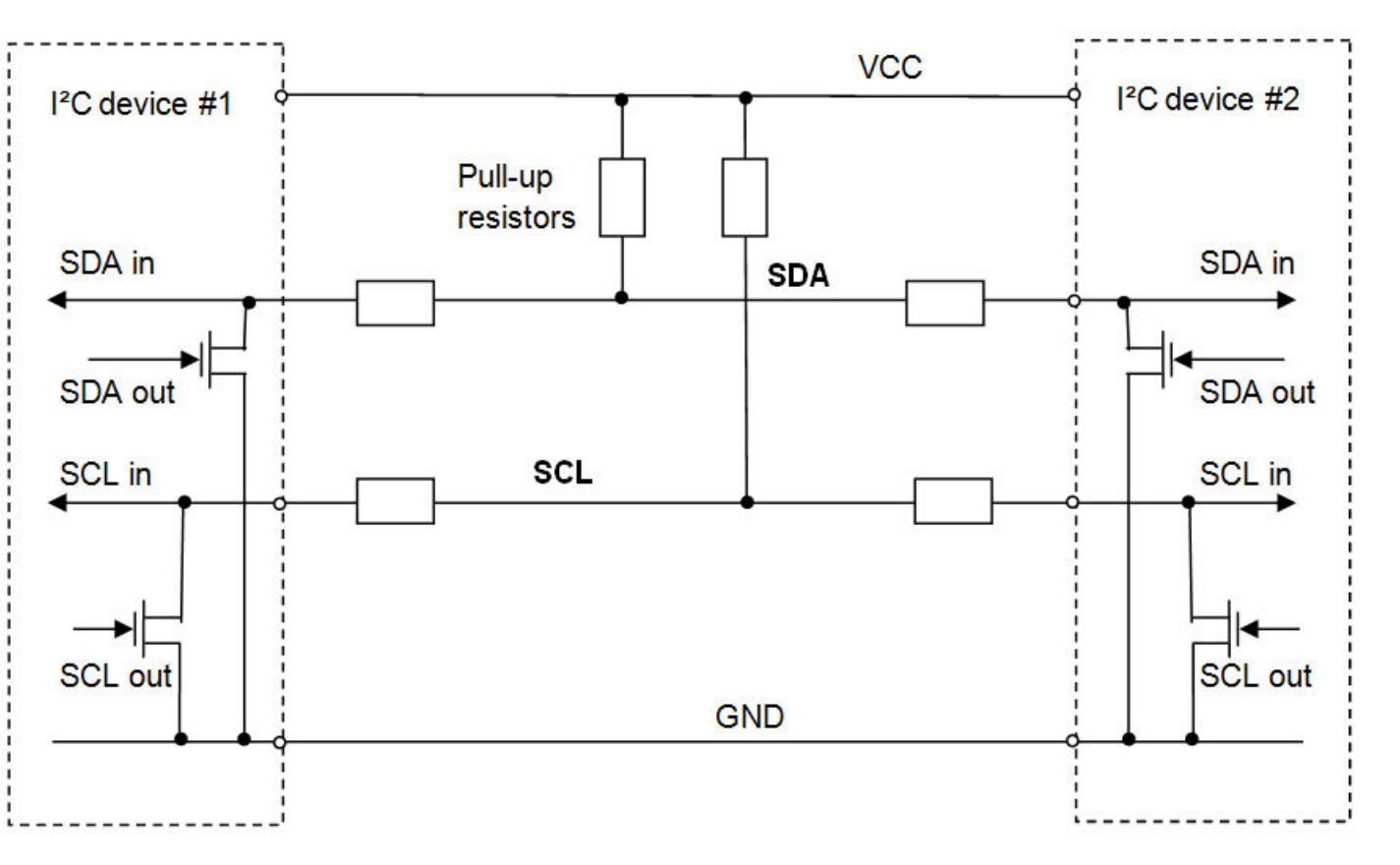


Figure 4: I²C bus with 2 devices connected. SDA and SCL are connected to VCC through pull-up resistors. Each device controls the bus lines outputs with open drain buffers.

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

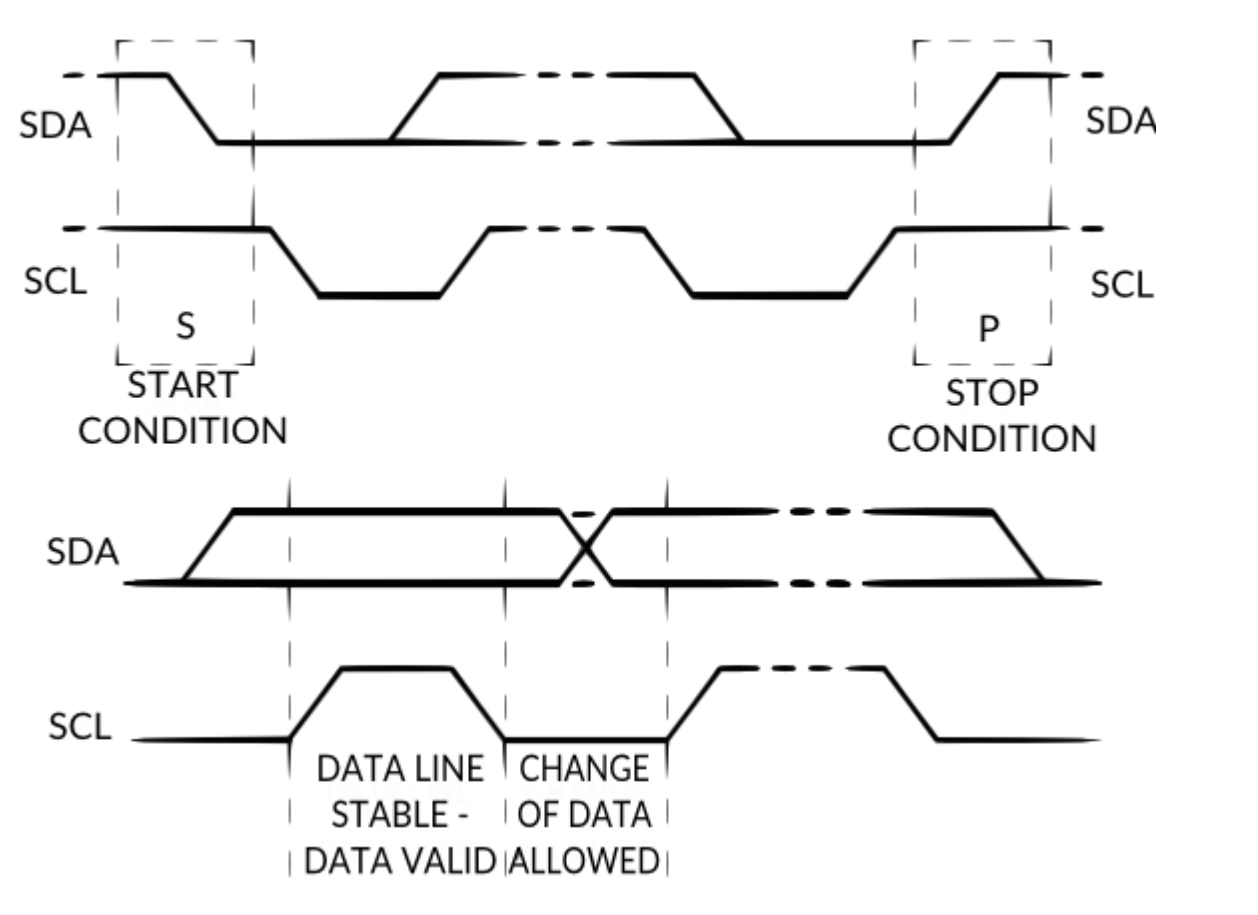
Then the master sends the ADDRESS of the device it wants to access, along with an indication whether the access is a Read or Write operation (Write in our example). Having received the address, all IC's will compare it with their own address. If it doesn't match, they simply wait until the bus is released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

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

When a master wants to receive data from a slave, it proceeds the same way, but sets the RD/nWR bit at a logical one. Once the slave has acknowledged the address, it starts sending the requested data, byte by byte. After each data byte, it is up to the master to acknowledge the received data (refer to figure 5).

Details on I2C protocol

START and STOP are unique conditions on the bus that are closely dependent of the I²C bus physical structure. Moreover, the I²C specification states that data may only change on the SDA line if the SCL clock signal is at low level; conversely, the data on the SDA line is considered as stable when SCL is in high state (refer to figure 6 hereafter).



At the physical layer, both SCL and SDA lines are open-drain I/Os with pull-up resistors (refer to figure 4). Pulling such a line to ground is decoded as a logical zero, while releasing the line and letting it flow is a logical one. Actually, a device on a I²C bus 'only drives zeros'.

Here we come to where I²C is truly elegant. Associating the physical layer and the protocol described above allow flawless communication between any number of devices, on just 2 physical wires.

For example, what happens if 2 devices are simultaneously trying to put information on the SDA and / or SCL lines?

At electrical level, there is actually no conflict at all if multiple devices try to put any logic level on the I²C bus lines simultaneously. If one of the drivers tries to write a logical zero and the other a logical one, then the open-drain and pull-up structure ensures that there will be no shortcut and the bus will actually see a logical zero transiting on the bus. In other words, in any conflict, a logic zero always 'wins'.

The bus physical implementation also allows the master devices to simultaneously write and listen to the bus lines. This way, any device is able to detect collisions. In case of a conflict between two masters (one of them trying to write a zero and the other one a one), the master that gains the arbitration on the bus will even not be aware there has been a conflict: only the master that looses will know – since it intends to write a logic one and reads a logic zero. As a result, a master that looses arbitration on a I²C will stop trying to access the bus. In most cases, it will just delay its access and try the same access later.

Moreover, the I²C protocol also helps at dealing with communication problems. Any device present on the I²C listens to it permanently. Potential masters on the I²C detecting a START condition will wait until a STOP is detected to attempt a new bus access. Slaves on the I²C bus will decode the device address that follows the START condition and check if it matches theirs. All the slaves that are not addressed will wait until a STOP condition is issued before listening again to the bus. Similarly, since the I²C protocol foresees active-low acknowledge bit after each byte, the master / slave couple is able to detect their counterpart presence. Ultimately, if anything else goes wrong, this would mean that the device 'talking on the bus' (master or slave) would know it by simply comparing what it sends with what is seen on the bus. If a difference is detected, a STOP condition must be issued, which releases the bus.

Additionally, I²C has got some advanced features, like extended bus addressing, clock stretching and the very specific 3.4 Mbps high speed mode.

Back to Top

– 10-bits device addressing

Any I²C device must have a built-in 7 bits address. In theory, this means that there would be only 128 different I²C devices types in the world. Practically, there are much more different I²C devices and it is a high probability that 2 devices have the same address on a I²C bus. To overcome this limitation, devices often have multiple built-in addresses that the engineer can chose by though external configuration pins on the device. The I²C specification also foresees a 10-bits addressing scheme in order to extend the range of available devices address.

Practically, this has got the following impact on the I²C protocol (refer to figure 7):

– Two address words are used for device addressing instead of one.
– The first address word MSBs are conventionally coded as "11110" so any device on the bus is aware the master sends a 10 bits device address.

START	1	1	1	1	0	A9	A8	Rd/ nWr	ACK	A7	A6	A5	A4	A3	A2	A1	A0	ACK	...
	First address word									Second address word									

10 bits address:

A9	A8	A7	A6	A5	A4	A3	A2	A1	A0

Figure 7: I²C 10-bits addressing. A 10-bits address is split into 2 words. The first word contains a conventional code on its 5 most significant bits to mark a 10-bits address, followed by the 2 MSBs of the 10-bits address and the Rd/nWR bit. The second address word contains the 8 least significant bits of the 10-bits address. This addition ensures backward compatibility with the 7-bits addressing scheme.

Actually, there are other reserved address codes for specific types of accesses (refer to table 1). For details about them, please refer to the I²C specification.

Address	Purpose
0000000 0	General Call – addresses all devices supporting the general call mode
0000000 1	Start Byte
0000001 X	CBUS addresses
0000010 X	Reserved for different bus formats
0000011 X	Reserved for future purpose
00001XX X	High-speed Master code
11110XX X	10-bits slave addressing
11111XX X	Reserved for future purposes

Table 1: I²C addresses reserved for special puposes

– Clock stretching

In an I²C communication the master device determines the clock speed. The SCL signal is an explicit clock signal on which the communication synchronizes.

However, there are situations where an I²C slave is not able to co-operate with the clock speed given by the master and needs to slow down a little. This is done by a mechanism referred to as clock stretching and is made possible by the particular open-drain / pull-up structure of a I²C bus line.

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

– High speed mode

Fundamentally, the use of pull-ups to set a logic one limits the maximum speed of the bus. This may be a limiting factor for many applications. This is why the 3.4 Mbps high speed mode was introduced. Prior to using this mode, the bus master must issue a specific 'High Speed Master' code at a lower speed mode (for example: 400 kbps Fast Mode) (refer to Table 1), which initiates a session at 3.4 Mbps. Specific I/O buffers must also be used to let the bus to shorten the signals rise time and increase the bus speed. The protocol is also somewhat adapted in such a way that no arbitration is performed during the high speed transfer. Refer to the I²C specification for more information about the high speed mode.

Back to Top

I²C vs SPI: is there a winner?

Let's compare I²C and SPI on several key protocol aspects:

– Bus topology / routing / resources:

I²C needs 2 lines and that's it, while SPI formally defines at least 4 signals and more, if you add slaves. Some unofficial SPI variants only need 3 wires, that is a SCLK, SS and a bi-directional MISO/MOSI line. Still, this implementation would require one SS line per slave. SPI requires additional work, logic and/or pins if a multi-master architecture has to be built on SPI. The only problem I²C when building a system is a limited device address space on 7 bits, overcome with the 10-bits extension.

From this point of view, I²C is a clear winner over SPI in sparing pins, board routing and how easy it is to build an I²C network.

– Throughput / Speed:

If data must be transferred at 'high speed', SPI is clearly the protocol of choice, over I²C. SPI is full-duplex; I²C is not. SPI does not define any speed limit; implementations often go over 10 Mbps. I²C is limited to 1Mbps in Fast Mode+ and to 3.4 Mbps in High Speed Mode – this last one requiring specific I/O buffers, not always easily available.

– Elegance:

It is often said that I²C is much more elegant than SPI, and that this last one is a very 'rough' (if not 'dumb') protocol. Actually, we tend to think the two protocols are equally elegant and comparable on robustness.

I²C is elegant because it offers very advanced features – such as automatic multi-master conflicts handling and built-in addressing management – on a very light infrastructure. It can be very complex, however and somewhat lacks performance.

SPI, on the other hand, is very easy to understand and to implement and offers a great deal of flexibility for extensions and variations. Simplicity is where the elegance of SPI lies. SPI should be considered as a good platform for building custom protocol stacks for communication between ICs. So, according to the engineer's need, using SPI may need more work but offers increased data transfer performance and almost total freedom.

Both SPI and I2C offer good support for communication with low-speed devices, but SPI is better suited to applications in which devices transfer data streams, while I²C is better at multi master 'register access' application.

Used properly, the two protocols offer the same level of robustness and have been equally successful among vendors. EEPROM (Electrically-Erasable Programmable Read-Only Memory), ADC (Analog-to-Digital Converter), DAC (Digital-to-Analog Converter), RTC (Real-time clocks), microcontrollers, sensors, LCD (Liquid Crystal Display) controllers are largely available with I²C, SPI or the 2 interfaces.

Conclusions.

In the world of communication protocols, I²C and SPI are often considered as 'little' communication protocols compared to Ethernet, USB, SATA, PCI-Express and others, that present throughput in the x100 megabit per second range if not gigabit per second. Though, one must not forget what each protocol is meant for. Ethernet, USB, SATA are meant for 'outside the box communications' and data exchanges between whole systems. When there is a need to implement a communication between integrated circuit such as a microcontroller and a set of relatively slow peripheral, there is no point at using any excessively complex protocols. There, I²C and SPI perfectly fit the bill and have become so popular that it is very likely that any embedded system engineer will use them during his/her career.

Back to Top

Back to Top