An Introduction to I²C and SPI Protocols

Frédéric Leens

Today, at the low end of the communication protocols we find the inter-integrated circuit (I²C) and the serial peripheral interface (SPI) protocols. Both protocols are well suited for communications between integrated circuits for slow communication with on-board peripherals. The two protocols coexist in modern digital electronics systems, and they probably will continue to compete in the future, as both I²C and SPI are actually quite complementary for this kind of communication [1].

At the roots of these popular protocols, we find two major companies—Motorola for SPI and Philips for I²C—and two different histories about why, when, and how the protocols were created. The SPI was introduced with the first microcontroller that derived from the same architecture as the popular Motorola 68000 microprocessor announced in 1979. For those who do not remember, the 16-bit 68000 microprocessor was at that time indisputably the best microprocessor on the market and was used in the Apple Macintosh computer in 1984. The microcontroller series was the 68xx. One of the first microcontrollers with an SPI bus was the M68HC03. The Motorola M68HC11 microcontroller, still available today, is also derived from this architecture. SPI was defined as the external microcontroller bus and was used to connect the microcontroller peripherals with four 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 microcontroller data sheets and associated application notes [2], [3].

The I²C bus was developed in 1982; its original purpose was to provide an easy way to connect a CPU to peripheral chips in a television 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 printed circuit board (PCB) 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 to make the PCBs simpler (in other words, to lower the costs), Philips labs in Eindhoven, The Netherlands, invented the "inter-integrated circuit," IIC or I²C protocol that only requires two wires for connecting all the peripherals to a microcontroller. The original specification defined a bus speed of 100 kb/s. The specification was reviewed several times, notably introducing the 400 kb/s speed in 1995 and, since 1998, 3.4 Mb/s for even faster peripherals. As of October 1, 2006, no licensing fees are required to implement the I²C protocol, making official a type of "fair open policy" that Philips Semiconductor has always demonstrated with I²C [4]. Strictly speaking, fees are still required to "officially" allocate slave I²C addresses.

NXP (formerly Philips Semiconductor) and Freescale (formerly the semiconductor branch of Motorola) are today in charge for the I²C and SPI "official" specifications, although the protocols have become so popular that we could wonder if they are not rather de facto public protocols. Let's review each of these protocols.

1094-6969/09/$25.00©2009IEEE

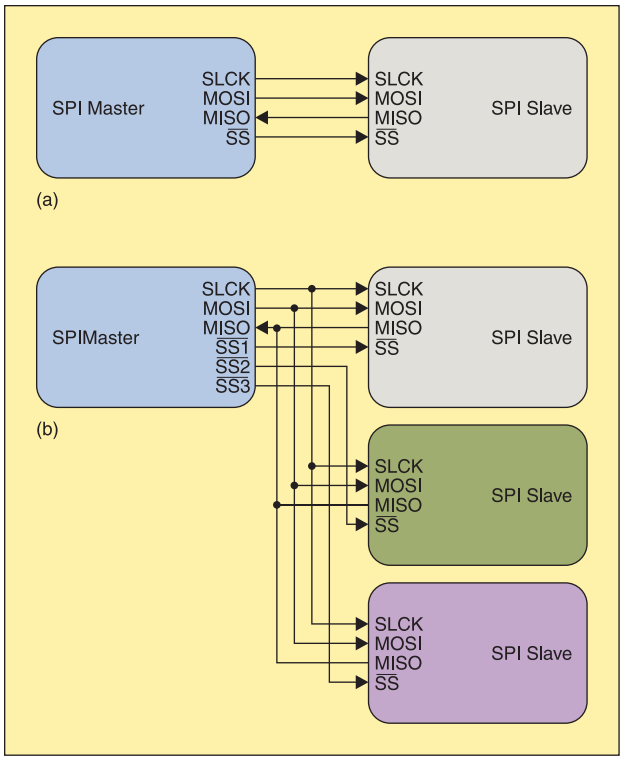

Fig. 1. Two SPI bus topologies. (a) shows an SPI master connected to a single slave (point-to-point topology). (b) shows an SPI master connected to multiple slaves.

Serial Peripheral Interface

SPI is quite straightforward—it defines features any digital electronic engineer would think of if it were necessary to quickly define a way to communicate between two digital devices. SPI is a protocol on four signal lines (Figure 1):

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

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 a 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 the MOSI line while it samples the MISO line (Figure 2).

Four communication modes are available (MODE 0, 1, 2, 3) that define

- the SCLK edge on which the MOSI line toggles
- the SCLK edge on which the master samples the MISO line
- 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).

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 that is defined for the SPI protocol. SPI does not define any maximum data rate nor any particular addressing scheme; it does not have an 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 will not need more than SPI, whereas 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

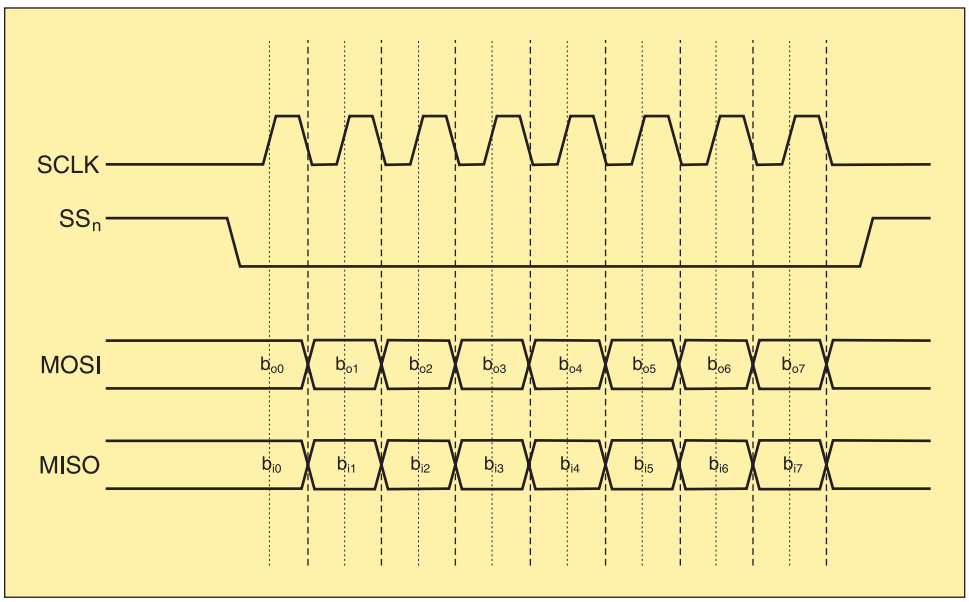

Fig. 2. A simple SPI communication. Data bits on MOSI and MISO toggle on the SCLK falling edge and are sampled on the SCLK rising edge. The SPI mode defines which SCLK edge is used for toggling data and which SCLK edge is used for sampling data.

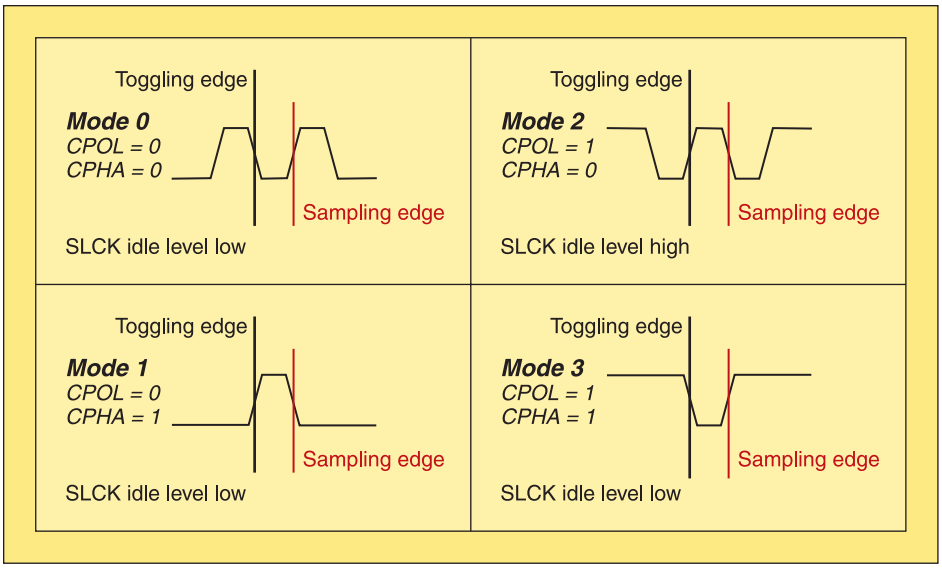

Fig. 3. SPI modes are defined with the parameters clock polarity (CPOL) and clock phase (CPHA), which explicitly define three parameters: the edges used for data sampling and data toggling and the SCL clock signal idle level—that is, the conventional level setting SCLK when the bus is not in communication.

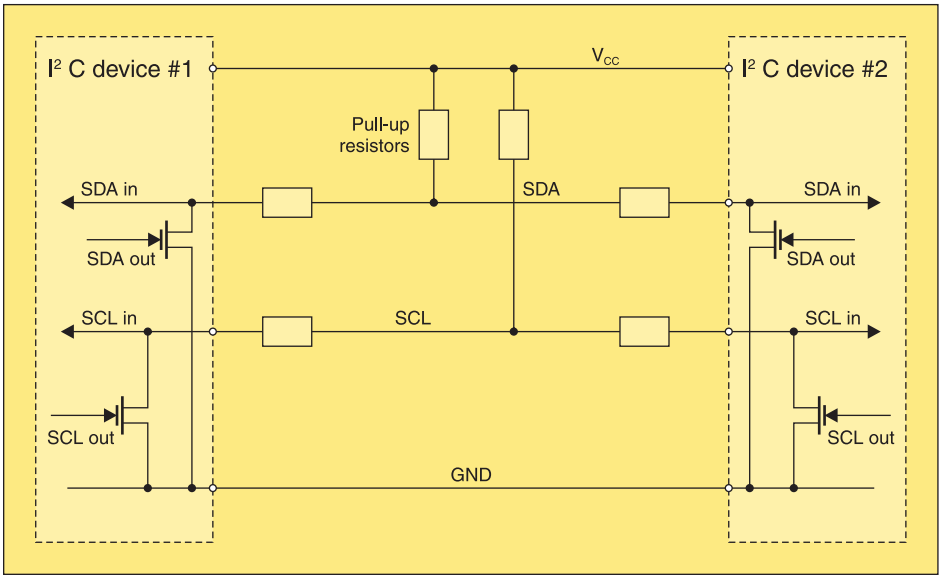

Fig. 4. I^2C bus with two devices connected. SDA and SCL are connected to V_{CC} through pull-up resistors. Each device controls the bus lines outputs with open drain buffers.

protocol now exist that use a continuous clock signal and an arbitrary transfer length.

Inter-Integrated Circuit

I^2C is a multi-master protocol that uses two signal lines. The two I^2C 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 two signal lines and communicate between each other using a protocol that defines:

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

The data rate has to be chosen between 100 kb/s, 400 kb/s, and 3.4 Mb/s, respectively called *standard mode*, *fast mode*, and *high speed mode*. Some I^2C variants include 10 kb/s (*low speed* mode) and 1 Mb/s (*fast mode +*) as valid speeds.

Physically, the I^2C bus consists of the two active wires SDA and SCL and a ground connection (Figure 4). The active wires are both bi-directional. The I^2C 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.

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 of whether the access is a Read or Write operation (Write in our example). Having received the address, all ICs will compare it with their own address. If it does not 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 that the bus has been released and that the connected ICs may expect another transmission to start at 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 logic 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 (Figure 5).

START and STOP are unique conditions on the bus that are closely dependent on 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 are considered as stable when SCL is in high state (Figure 6).

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 logic ZERO, whereas releasing the line and letting it float is a logic ONE. Actually, a device on an I²C bus "only drives zeros."

Here we come to where I²C is truly elegant. Associating the physical layer and the protocol described above enables flawless communication between any number of devices on just two physical wires. For example, what happens if two devices are simultaneously trying to put information on the SDA and/or SCL lines? At the electrical level, there is actually no conflict if multiple devices try to put any logic level on the I²C bus lines simultaneously. If one of the drivers tries to write a logic ZERO and the other a logic ONE, then the open-drain and pull-up structure ensures that there will be no short-circuit and the bus will actually see a logic ZERO transiting on the bus. In other words, in any conflict, logic ZERO always "wins."

START	Slave address	Rd/nWr	ACK	Data	ACK	Data	ACK	STOP
1 bit	7 bits	1 bit	1 bit	8 bits	1 bit	8 bits	1 bit	1 bit

Example 1: writing 2 byte to a slave. The data put on the bus by the master are shaded.

START	Slave address	0	0	Data	0	Data	0	STOP
1 bit	7 bits	1 bit	1 bit	8 bits	1 bit	8 bits	1 bit	1 bit

Example 2: reading 2 bytes from a slave. The data put on the bus by the master are shaded.

START	Slave address	1	0	Data	0	Data	1	STOP
1 bit	7 bits	1 bit	1 bit	8 bits	1 bit	8 bits	1 bit	1 bit

Fig. 5. Typical I²C transfer, with 2 bytes of data. The master initiates the transfer with a START condition, followed by the slave address and the transfer type (read or write) bit. The slave acknowledges its address. Each data byte is then transmitted and acknowledged by the receiver. When it receives data, the master can issue a not-acknowledge condition (NACK) when it has received enough data. The bus is released when the master issues a STOP condition.

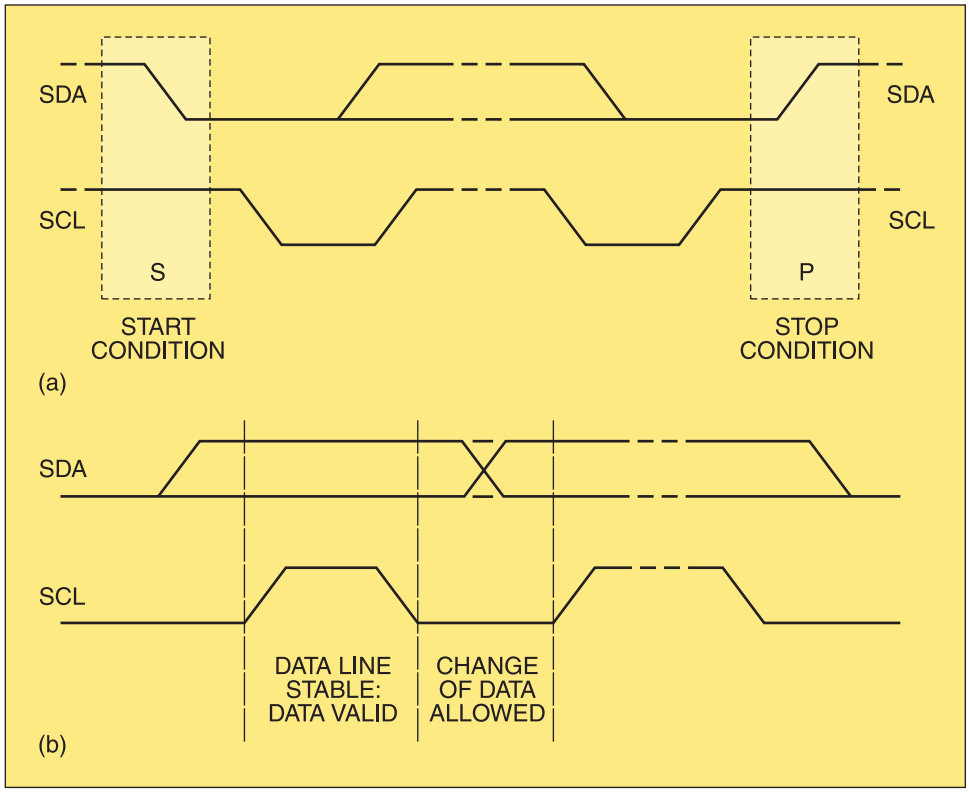

Fig. 6. When data is transmitted on the I²C bus. (a) shows the relative phase of the SCL clock signal and the SDA data signal. A data change when SCL is not stable and at low level is illegal. (b) shows the START and STOP conditions. A START condition is a high-to-low transition on SDA when SCL is high. A STOP condition is a low to high transition on SDA when SCL is high.

The bus physical implementation also enables 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 not even be aware there has been a conflict: only the master that loses will know, because it intends to write a logic ONE and reads a logic ZERO. As a result, a master that loses arbitration on an I²C will stop trying to access the bus. In most cases, it will just delay its access and try the same access later.

In addition, the I²C protocol also helps in 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, because the I²C protocol foresees active-low acknowledge bit after each byte, the master/slave couple is able to detect their counterpart's presence. Ultimately, if

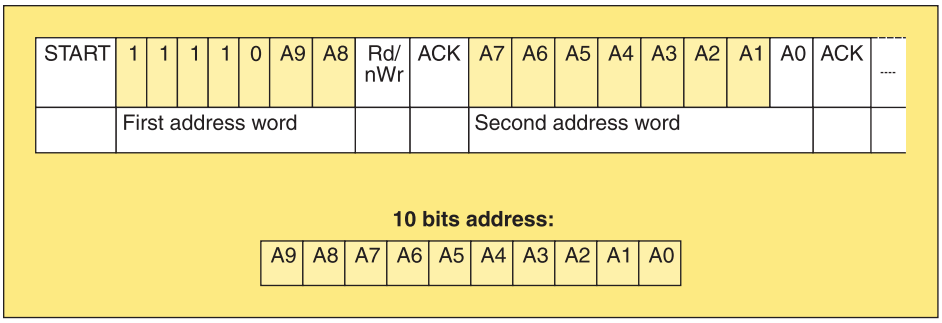

Fig. 7. I²C 10-bit addressing. A 10-bit address is split into two words. The first word contains a conventional code on its five most significant bits to mark a 10-bit address, followed by the 2 MSBs of the 10-bit address and the Rd/nWR bit. The second address word contains the eight LSBs of the 10-bit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

Table 1—I²C addresses reserved for special purposes. Xs are "don't care"

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-bit slave addressing
11111XX X	Reserved for future purposes

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.

I²C also has some advanced features, like extended bus addressing, clock stretching, and the very specific 3.4 Mb/s high speed mode.

10-Bit Device Addressing

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

Practically, this has the following impact on the I²C protocol (Figure 7).

- Two address words are used for device addressing instead of one
- The first address word's most significant bits (MSBs) are conventionally coded as "11110" so any device on the bus is aware the master sends a 10-bit device address
- The second address word contains the least significant bits (LSBs)

Actually, there are other reserved address codes for specific types of accesses (Table 1). For details about them, please refer to the I²C specification [4].

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 cooperate with the clock speed given by the master and needs to slow it 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 an 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 Mb/s high speed mode was introduced. Before using this mode, the bus master must issue a specific "High Speed Master" code at a lower speed mode (e.g., 400 kb/s fast mode) (refer to Table 1), which initiates a session at 3.4 Mb/s. Specific I/O buffers must also be used to let the bus 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.

I²C and SPI: Who Wins?

Let us compare I²C and SPI on several key protocol aspects.

Bus Topology/Routing/Resources

I²C needs two lines only, whereas SPI formally defines at least four signals and more if you add slaves. Some unofficial SPI variants only need three wires, i.e., an SCLK, an SS, and a bidirectional 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 with I²C when building a system is that it has a limited device address space of 7 bits that can be overcome with the 10-bit extension.

From this point of view, I²C is a clear winner over SPI in sparing pins, board routing, and ease 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 Mb/s. I²C is limited to 1Mb/s in fast mode+ and to 3.4 Mb/s in high speed mode, this last one requiring specific I/O buffers, which are not always easily available.

Elegance

It is often said that I²C is much more elegant than SPI and that SPI 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 require 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, whereas I²C is better at multi-master "register access" applications.

Used properly, the two protocols offer the same level of robustness and have been equally successful among vendors. Electrically-erasable programmable read-only memory, analog-to-digital converter, digital-to-analog converter, real-time clocks, microcontrollers, sensors, and liquid crystal display controllers are largely available with I²C, SPI, or the two 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 ×100 Mb/s range, if not Gb/s. It is important that one not forget the purpose of each protocol. Ethernet, USB, and SATA are meant for "outside the box communications" and data exchanges between whole systems. When there is a need to implement a communication between an integrated circuit such as a microcontroller and a set of relatively slow peripherals, there is no point in 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.

References

[1] S. Sarns and J. Woehr, "Exploring I2C," *Embedded Systems Programming*, vol. 4, p. 46, Sept. 1991.

[2] Freescale Semiconductor, (2008, October 13). Freescale M68HC05 Microcontrollers data sheets [Online] Available http://www.freescale.com (then go to "Get Support", "Documentation", "Data Libraries", "Data Sheets").

[3] Freescale Semiconductor, (2008, October 14). Freescale SPI Block Guide V04.01 Jul. 14 revision [Online] Available http://www.freescale.com/webapp/search/Serp.jsp?&QueryText=Freescale%20SPI%20Block%20Guide%20V03.06&SelectedAsset=Documentation&QueryText=*&&fsrch=1.

[4] NXP Semiconductors, (2008, October 13). I²C bus specification [Online] Available http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf.

Frédéric Leens (frederic.leens@byteparadigm.com) is the Sales and Marketing Manager at Byte Paradigm, a Belgian-based company he founded in 2005 with the purpose of providing PC-based instruments to test and debug embedded electronic systems. Before this position, he worked as an ASIC/FPGA and embedded system design engineer and consultant for various companies, such as Barco and NXP/Philips. He holds an M.S.E.E. from the Faculté Polytechnique de Mons, Belgium and a degree in management from the FUCAM, Mons, Belgium.