Chapiter V : Communication Buses

VI Communication Buses

In an embedded system, the various components — processor, memory, and peripherals — must exchange information smoothly and in a coordinated manner. This exchange relies on a set of electrical lines called communication buses, which act as data transport networks within the system.

A bus is therefore a mechanism for transferring information between the different elements of a system. It generally consists of multiple wires (or PCB traces) carrying address, data, and control signals. Each bus is characterized by its width (number of lines), the type of information it transmits, and its mode of operation — parallel or serial, synchronous or asynchronous.

Choosing the appropriate bus is essential, as it determines the speed of communication, energy consumption, and system stability. In modern embedded architectures, the communication bus directly influences the overall performance of the device.

Communication between a processor and a peripheral is based on the coordinated exchange of three main types of signals: address, data, and control. These signals circulate through the system's buses, as illustrated below.

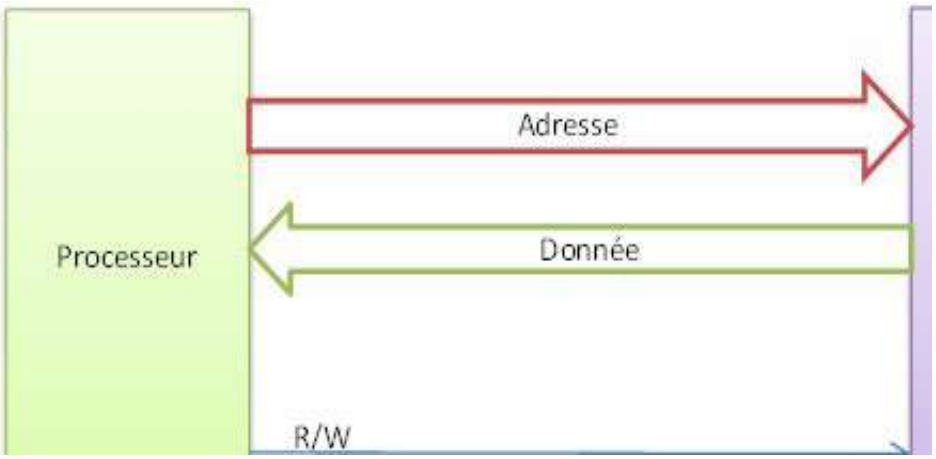

Figure V. 1 - Principle of processor–peripheral communication

The processor first sends an **address** to identify the resource or peripheral involved. This address locates the register or memory zone where the operation will take place. Depending on the type of operation, the processor may then:

- **Write data** (transfer from processor to peripheral), or
- **Read data** (transfer from peripheral to processor).

The R/W (Read/Write) control line indicates the direction of the transfer:

- **R (Read) → data are read from the peripheral.**
- **W (Write) → data are written to the peripheral.**

This tripartite organization — **address bus**, **data bus**, and **control bus** — ensures clear, reliable, and synchronized communication among system components. It forms the foundation of all interactions between the microprocessor, memory, and I/O interfaces, both in simple architectures and in complex embedded systems.

VI.1 Communication Principle – Read Cycle

When the processor needs to read data from memory or a peripheral, the operation follows a precise temporal sequence coordinated by the address, data, and control signals.

1. The processor first places a valid address on the address bus to indicate the data location.
2. The $R\overline{D}$ (Read) control signal is then asserted low, indicating that a read operation is underway. This signal enables the addressed memory or peripheral to place the requested data on the data bus.

3. Once the data become valid, the processor reads it before the $R\bar{D}$ signal returns high. When $R\bar{D}$ is deasserted, the peripheral releases the data bus, completing the read cycle.

All these steps are synchronized with the system clock, which defines the timing intervals of the transfer — known as machine cycles (T1, T2, T3, etc.).

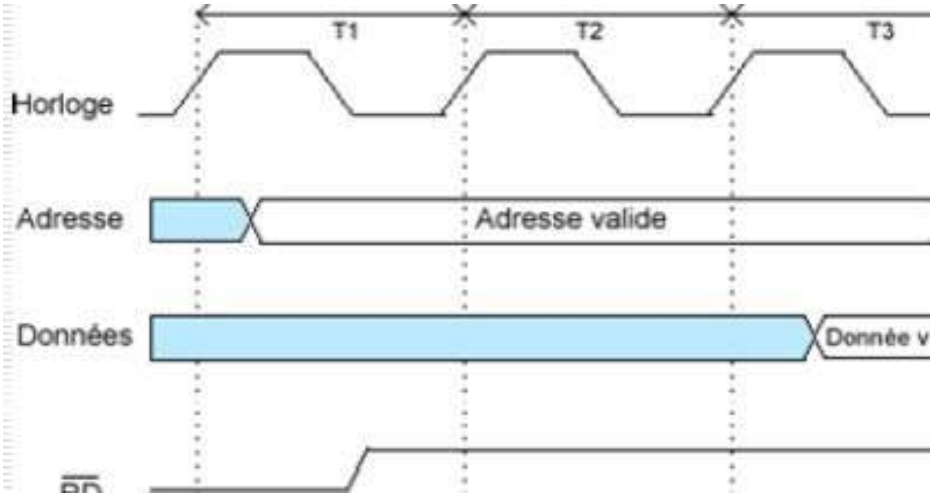

Figure V. 2 - Read cycle timing diagram

During T1, the processor outputs the address.

During T2, the data become available on the bus.

During T3, the processor reads the data before the bus is released.

This sequence ensures reliable, orderly, and synchronized data transfer within the embedded system.

VI.2 Communication Principle – Write Cycle

The write cycle corresponds to the process by which the processor sends data to a memory or peripheral for storage.

4. The processor outputs a **valid address** on the address bus to identify the target register or memory cell.
5. The data to be written are placed on the data bus.
6. The **R/$\bar{W}$** control signal is driven low to indicate a write operation. The data must remain stable during the active period of this signal to ensure proper storage.
7. Once the operation is completed, **R/$\bar{W}$** returns high, marking the end of the cycle.

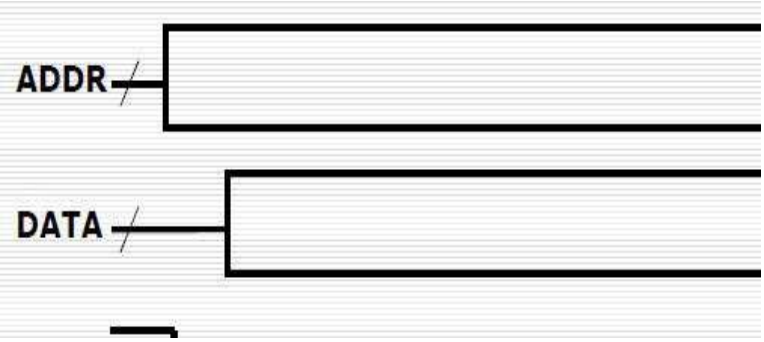

Figure V. 3 - Write cycle timing diagram

This communication process is fundamental in embedded systems, as it allows the processor to update registers, save variables, and configure external peripherals. A precise understanding of the read and write cycles is essential for ensuring both reliability and performance of the overall system.

VI.3 Types of Communication Buses

Buses are the essential communication pathways between the various components of an embedded system. They ensure the orderly transfer of data, addresses, and control signals between the processor, memory, and peripherals.

There are two main categories of buses:

- Internal buses, used within the microprocessor or microcontroller;
- External buses, which connect the processor to peripheral devices and external memories.

VI.3.1 Internal Bus

The internal bus connects the functional units of the processor: the Arithmetic and Logic Unit (ALU), registers, cache memory, and the control unit.
It enables very high-speed exchanges, ensuring consistency between the instructions and data processed by the CPU.

One of the most common internal buses in modern ARM architectures is the AHB (Advanced High-performance Bus), defined in the AMBA standard (Advanced Microcontroller Bus Architecture).

A) The AHB Bus (Advanced High-performance Bus)

The AHB bus is designed to provide fast and efficient transfers between the processor core and internal components such as memory and high-performance peripherals. It serves as the backbone of communication inside the microcontroller.

Main characteristics of the AHB bus:

- High-performance transfers: supports high data rates through a *pipelined* operation, allowing a new transaction to begin before the previous one finishes.
- Variable width: typically 32 or 64 bits, enabling multiple bytes to be transferred per clock cycle.
- Centralized arbitration: an AHB controller grants the bus to only one master (CPU, DMA, etc.) at a time, preventing conflicts.
- Burst mode: allows sequential block transfers to increase bandwidth.
- Unified interface: all connected peripherals follow a single standardized protocol, simplifying integration.

Typical applications in embedded systems:

- Communication between an ARM Cortex-M processor and the internal Flash memory for code execution.
- Direct data transfer between SRAM and a DMA controller, reducing CPU workload.
- Data exchange between a graphic controller and video memory in embedded systems with displays.

Thus, the AHB bus ensures fast and stable internal communication, which is essential for real-time embedded systems.

VI.3.2 External Bus

The external bus enables communication between the microcontroller and external peripherals such as sensors, actuators, external memories, or communication modules (Wi-Fi, GPS, Bluetooth, etc.). It can operate in parallel mode (multiple bits transmitted simultaneously) or serial mode (data transmitted bit by bit). In modern embedded systems, serial buses are preferred for their simplicity and low power consumption.

A) SPI Bus (Serial Peripheral Interface)

The SPI bus is a synchronous serial communication protocol widely used to connect a microcontroller to high-speed external peripherals. It operates in a master–slave configuration, where the microcontroller (master) manages communication with one or more peripherals (slaves).

Main lines:

- **MOSI (Master Out Slave In):** data sent from the master to the slave.
- **MISO (Master In Slave Out):** data sent from the slave to the master.
- **SCK (Serial Clock):** clock signal generated by the master to synchronize communication.
- **SS (Slave Select):** selection line that activates the target peripheral.

Advantages of the SPI bus:

- Very high data rate (tens of MHz).
- Simple interface to implement and low power consumption.
- Ideal for short-distance communication, making it perfect for compact embedded systems.

Typical applications:

- Communication between a microcontroller and an SPI Flash memory for data storage.
- Connection of an inertial sensor (IMU) or accelerometer in mobile robots.
- Interface with TFT or OLED displays in embedded visualization systems.

VI.3.3 Serial Bus

A serial bus transmits data bit by bit over one or more lines, synchronized by a clock or a defined timing protocol. This transmission mode significantly reduces the number of wires, thereby lowering cost, size, and electromagnetic interference.

In a serial communication, binary information is transmitted sequentially on the same line, with synchronization ensured either by a shared clock or timing rules. This makes serial buses particularly suitable for compact and reliable embedded systems.

They are used to connect the microcontroller to external peripherals such as sensors, memories, displays, or communication modules.

A serial bus can be:

- Synchronous, such as SPI (Serial Peripheral Interface) or I²C (Inter-Integrated Circuit); or
- Asynchronous, such as UART (Universal Asynchronous Receiver-Transmitter).

Examples of serial buses in embedded systems:

- SPI is widely used to connect a microcontroller to an external Flash memory, TFT display, or analog-to-digital converter.
- I²C connects multiple sensors (temperature, pressure, humidity) using only two lines (SDA and SCL), reducing pin usage.
- CAN (Controller Area Network) is common in automotive systems, ensuring reliable communication between ECUs with high noise immunity.
- UART interfaces are used for communication with GPS, GSM, or Bluetooth modules.

Due to its reliability, compactness, and low wiring complexity, the serial bus is the preferred solution for external communication in most modern embedded systems, especially when the required transfer rate remains below a few hundred megahertz.

VI.3.4 Parallel Bus

A parallel bus allows simultaneous transfer of multiple bits across several distinct lines. Each bit is transmitted on a separate wire, enabling high data throughput per clock cycle. This architecture makes the parallel bus ideal for high-speed internal communication between the processor and memory, as well as certain external interfaces.

A parallel bus typically includes data, address, and control lines.
At each clock pulse, all bits of a word are transmitted simultaneously, ensuring fast and efficient communication. However, this method requires a large number of physical connections, which increases wiring complexity and susceptibility to noise.
At higher frequencies, synchronization among all lines becomes increasingly difficult.

In embedded systems, parallel buses are commonly used to connect:

- Microcontrollers to external SRAM or Flash memory, or
- Microcontrollers to parallel LCD displays.

For example, some ARM-based embedded boards use a 16- or 32-bit external parallel bus to interface with external memory, extending system storage capacity. This configuration is common in data loggers, industrial display panels, and embedded graphics controllers.

In systems with graphical displays, the parallel LCD bus (FSMC – *Flexible Static Memory Controller*) is often used to transfer images or display data at high speed between the microcontroller and the screen module. This bus supports 8-, 16-, or 32-bit transfers, ensuring high display responsiveness, which is crucial for Human–Machine Interfaces (HMI).

While parallel buses offer very high throughput, they come with higher power consumption and routing complexity. In modern embedded architectures, they are often replaced or complemented by high-speed serial buses such as SPI or QSPI, which reduce wiring while maintaining satisfactory transfer rates.

VI.4 Bus Characteristics

Communication buses form the foundation of data exchange in embedded systems. Their performance mainly depends on three key parameters: bus width, transfer rate, and communication protocol. These characteristics directly affect the speed, reliability, and energy efficiency of the overall system.

VI.4.1 Bus Width

The bus width corresponds to the number of physical lines used to transmit data simultaneously.
The greater the width, the larger the amount of data transferred per clock cycle. For example, an 8-bit bus transfers one byte per cycle, while a 32-bit bus transfers four bytes.

In modern microcontrollers, such as those based on ARM Cortex-M cores, the internal bus width is typically 32 bits, offering a good balance between speed and power consumption. However, increasing the number of lines makes the wiring more complex and raises energy consumption.

Therefore, low-power systems, such as autonomous sensors or IoT devices, tend to use narrower buses (8 or 16 bits), whereas high-performance systems, such as industrial control processors, adopt 64-bit buses to maximize bandwidth.

VI.4.2 Transfer Rate

The transfer rate represents the amount of data that can be exchanged per unit of time, expressed in bits per second (bps). It depends on the clock frequency, bus width, and the nature of communication (serial or parallel).

In an embedded system, this rate determines memory access speed, communication fluidity with peripherals, and the overall responsiveness of the device.

For instance:

- An SPI bus can reach tens of megahertz,
- Whereas an I²C bus is often limited to a few hundred kilohertz.
- Internal buses such as AHB or AXI, used in ARM architectures, can achieve hundreds of megahertz, which is essential for fast data processing.

An appropriate transfer rate ensures effective synchronization between modules and minimizes communication delays.

VI.4.3 Communication Protocols

Communication protocols define the rules governing exchanges between components connected to a bus. They specify how data are encoded, addressed, synchronized, and validated to ensure reliable transmission. Each bus employs a specific protocol suited to its functional requirements.

For example:

- The SPI (Serial Peripheral Interface) protocol enables high-speed communication between a microcontroller and external devices such as memories or displays.
- The I²C (Inter-Integrated Circuit) protocol, using only two wires, is widely employed to connect multiple sensors on the same bus, each identified by a distinct address.
- The CAN (Controller Area Network) protocol is robust and error-tolerant, making it ideal for critical automotive communications.
- Finally, internal buses standardized by ARM, such as AHB and AXI, rely on optimized protocols that guarantee fast and orderly data transfers between the processor, memory, and peripherals.

VI.5 Communication Protocols

VI.5.1 UART (Universal Asynchronous Receiver-Transmitter)

UART is a widely used asynchronous serial communication protocol in embedded systems for data exchange between a microcontroller and a peripheral (computer, Bluetooth module, GPS, sensor, etc.). Unlike synchronous protocols such as SPI or I²C, UART does not require any shared clock signal — bit synchronization is achieved through start and stop signals transmitted within each frame.

UART communication is based on two main lines:

- **TX (Transmit):** line used to send data.
- **RX (Receive):** line used to receive data.

A common ground (GND) connects both devices to ensure a shared electrical reference.

Communication is generally point-to-point, meaning between only two elements: a transmitter and a receiver. Each bit is transmitted sequentially on the TX line, while the RX line of the receiving device captures the data.

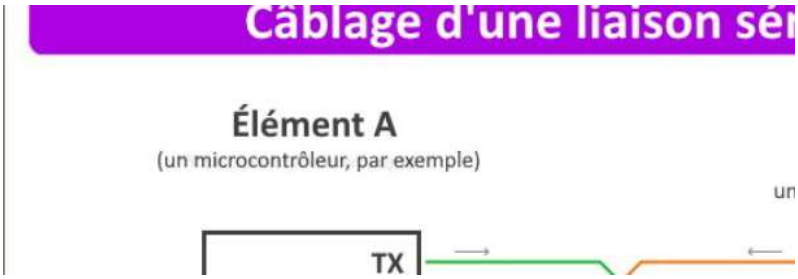

Figure V. 4 - UART connection diagram between two elements

This figure illustrates the typical **crossed connection** between two UART devices:

- The TX output of element A (for example, a microcontroller) is connected to the RX input of element B (such as a GPS module).
- Conversely, the TX output of element B is connected to the RX input of element A.

This crossed configuration enables bidirectional communication between the two devices.

A UART communication frame consists of several successive parts allowing the reliable transmission of one byte of data:

- 1 start bit: indicates the beginning of transmission (transition from high to low).
- 5 to 9 data bits: contain the useful information, sent from the least significant bit (LSB) to the most significant bit (MSB).
- 1 parity bit (optional): used for simple error detection.
- 1 or 2 stop bits: indicate the end of transmission (return to the high level).

During idle periods, the TX line remains at a logical high state.

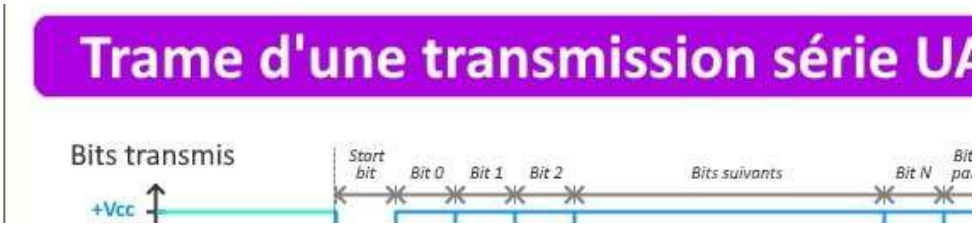

Figure V. 5 - Frame of a UART serial transmission (sequence)

This figure illustrates the typical timing diagram of a UART frame. It shows the succession of the different bits composing a frame:

- The start bit drives the line low, triggering message reception.
- Data bits (D0 to Dn) are then transmitted one after another.
- The parity bit, when enabled, allows simple error detection during transmission.
- Finally, the stop bit returns the line to the high state, marking the end of the message.

Transmission speeds, expressed in baud rates, vary depending on the system — from 1200 to 115200 baud (and even higher in modern systems). Both devices must be configured with identical parameters (speed, number of bits, parity, etc.) to ensure reliable communication.

VI.5.2 SPI (Serial Peripheral Interface)

The SPI (Serial Peripheral Interface) is a synchronous serial communication protocol widely used in embedded systems to connect a master microcontroller to multiple slave devices, such as sensors, memories, displays, or analog-to-digital converters.

It is based on a master-slave architecture, where the master controls synchronization and peripheral selection via a shared clock and control lines.

Main SPI bus lines:

- SCLK (Serial Clock): clock generated by the master to synchronize communication.
- MOSI (Master Out Slave In): line used to send data from the master to the slave.
- MISO (Master In Slave Out): line used by the slave to send data to the master.
- SS (Slave Select): signal activated by the master (low level) to select the desired slave.

When the SS line is set low, the corresponding slave becomes active, while the other peripherals remain inactive. Data transfer occurs bit by bit with each clock pulse, making communication fast, reliable, and full-duplex (simultaneous transmission and reception).

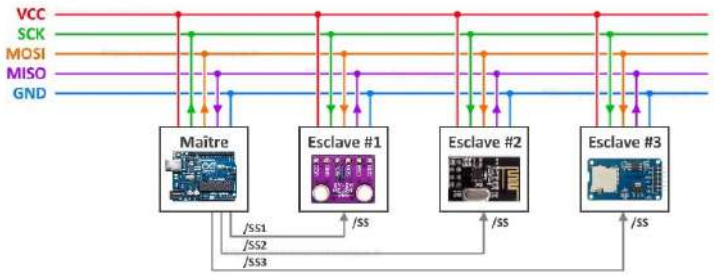

Figure V. 6 - SPI communication architecture between master and slave

This figure illustrates the fundamental structure of the SPI bus connecting a master to several slaves through the four communication lines. The master generates the clock (SCK) and controls the MOSI and MISO lines, while each slave has an independent SS line. This configuration allows selective communication: only one slave is active at a time, thus avoiding data collisions.

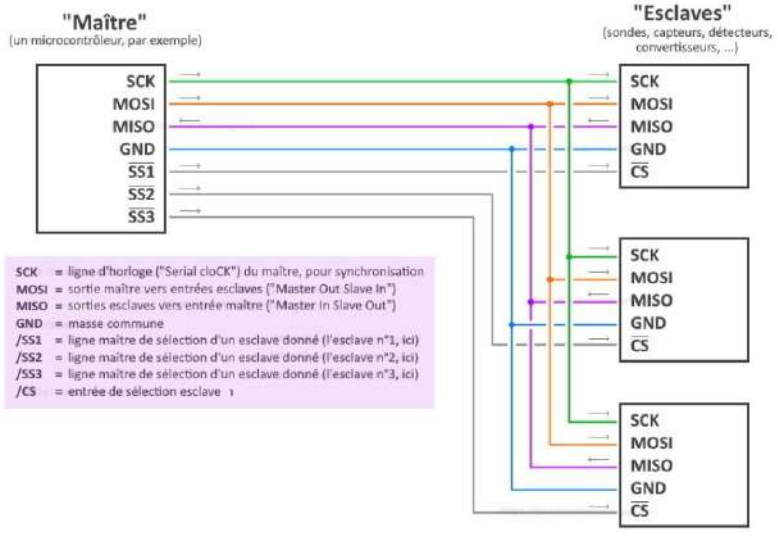

Figure V. 7 - Example of SPI wiring (with 1 master and 3 slaves)

This figure shows the practical implementation of the SPI bus with a master microcontroller and three slave peripherals. The lines SCK, MOSI, MISO, and GND are common to all peripherals, simplifying wiring. However, each slave has a dedicated SS line (/SS1, /SS2, /SS3).

When communication is established, the master pulls down the SS line corresponding to the desired slave, while the other SS lines remain high, deactivating unrelated slaves. This mechanism allows precise control and prevents interference between peripherals, at the cost of having one SS line per slave.

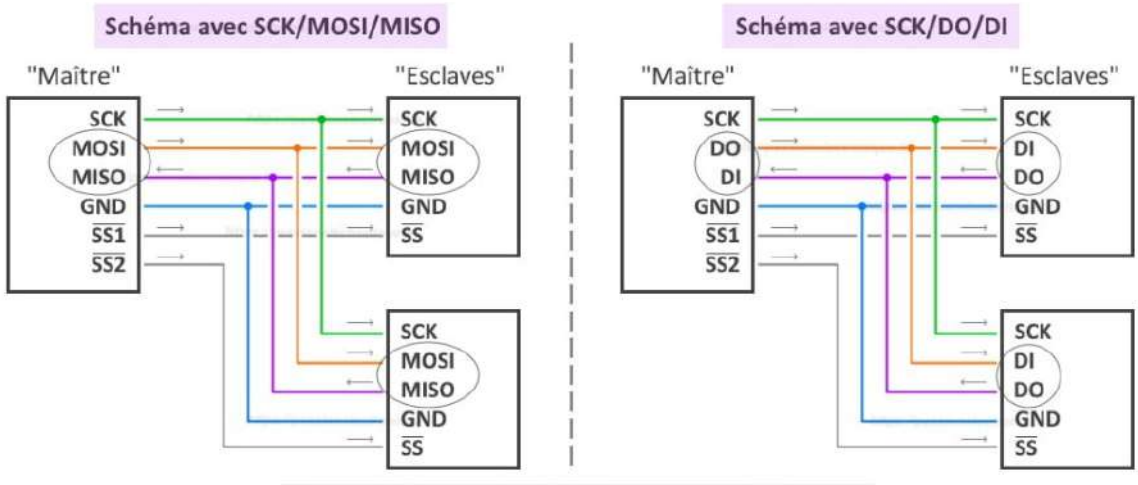

Figure V. 8 - Different SPI input/output notations

This figure compares two naming conventions used by different manufacturers:

- SCK/MOSI/MISO (the most common notation),
- SCK/DO/DI (an alternative but equivalent notation).

Thus:

- MOSI = DO (Data Out) corresponds to the master's data output.
- MISO = DI (Data In) corresponds to the master's data input.
- SCK remains the common clock line for all implementations.
- SS (or CS) designates the slave select line.

This figure highlights the compatibility of terminologies used by various manufacturers (Microchip, Atmel, STMicroelectronics, etc.), showing that despite naming differences, the logical structure and operating principles remain identical.

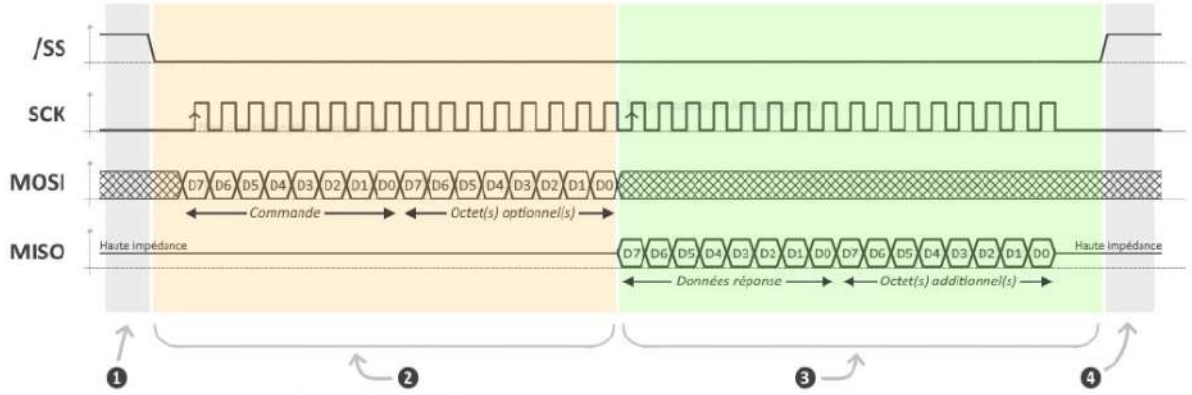

Figure V. 9 - Example of reading on the SPI bus

This figure illustrates the timing sequence of an SPI read operation:

- Initialization: the master pulls the SS line low to select a specific slave.
- Command transmission: the master sends an instruction via MOSI (e.g., a request to read a memory register).
- Reception: after receiving the command, the slave sends the requested data on the MISO line, bit by bit, synchronized with SCK pulses.
- End of communication: when the read operation is complete, the master sets the SS line high to deactivate the slave.

This figure clearly illustrates the synchronous and bidirectional nature of SPI: each data bit is transferred in phase with the clock, ensuring fast and error-free synchronization.

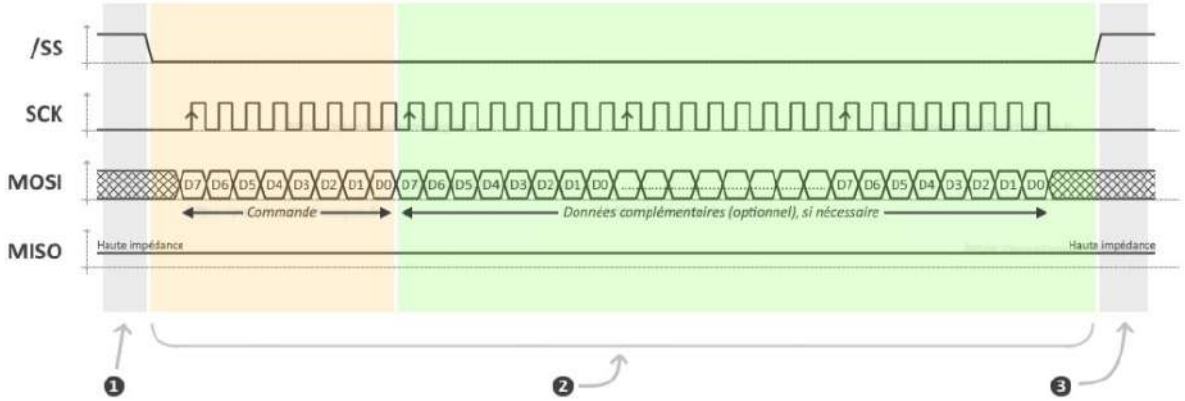

Figure V. 10 - Example of writing on the SPI bus

The SPI write operation is similar to reading, but this time the master sends data to the slave:

- The master first pulls the SS line low, thus selecting the target slave.
- It then sends one or more data packets (typically 8 bits each) on MOSI, possibly accompanied by additional parameters.
- Each bit is sent with every clock pulse on the SCK line.
- At the end of the transmission, the master sets the SS line high, indicating the end of communication.

This figure also shows that the slave maintains the MISO line in high impedance during writing, so as not to interfere with the signal transmitted on MOSI.

This sequential and strictly clocked operation ensures reliable exchanges, even at speeds reaching several tens of megahertz.

Here is the complete and precise English translation of your section on I²C (Inter-Integrated Circuit) — preserving all technical meaning, structure, and formatting exactly as in your French text:

VI.5.3 I²C (Inter-Integrated Circuit)

The I²C (Inter-Integrated Circuit) protocol, developed by Philips (now NXP), is a synchronous serial communication interface widely used in embedded systems to interconnect multiple integrated circuits on the same board. It is characterized by its simple wiring and its ability to manage multiple slave devices using only two lines:

- **SDA (Serial Data):** bidirectional data transfer line.
- **SCL (Serial Clock):** clock line generated by the master to synchronize communication.

Each connected peripheral has a unique address, allowing the master to communicate individually with it. The I²C bus is therefore multi-master / multi-slave, although in practice only one master is active at a time.

Operating Principle

I²C communication is based on a master–slave exchange:

- The master initiates transactions (by sending a START and STOP bit).
- The slaves respond according to their assigned address.

- Data are exchanged bit by bit on the SDA line, synchronized by the SCL clock. Each byte sent is followed by an acknowledgment bit (ACK/NACK) confirming receipt.

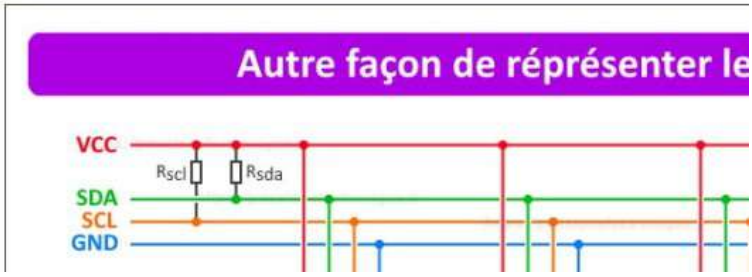

Figure V. 11 - Alternative representation of the I²C bus

This figure illustrates a typical I²C bus topology connecting a master microcontroller (e.g., Arduino) to several slave devices: a temperature sensor, an OLED display, and a digital-to-analog converter (DAC). The SDA and SCL lines are shared by all modules, while the Rsda and Rscl resistors ensure logical pull-up to VCC. These resistors are essential to maintain correct logic levels because I²C peripheral outputs use open-drain configurations.

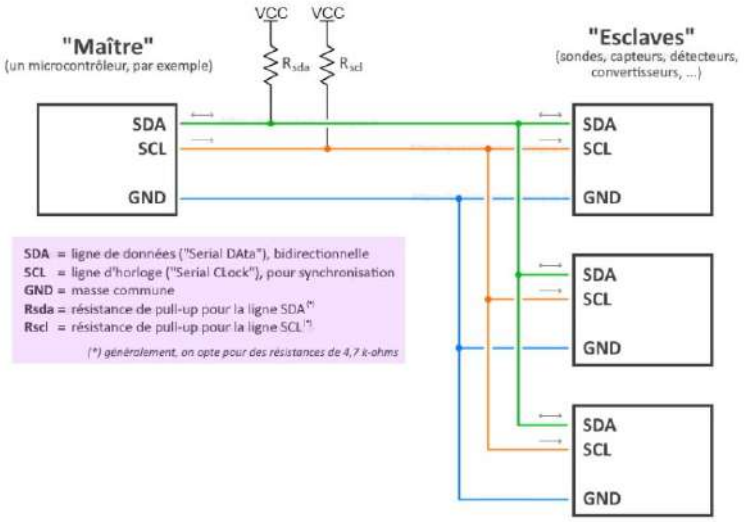

Figure V. 12 - Example of wiring for a multi-point I²C link

This figure shows the complete electrical structure of the bus: the SDA and SCL lines are common to all devices, and two pull-up resistors (typically 4.7 kΩ) are connected to the VCC power supply. The master (microcontroller) and the slaves (sensors or modules) share the same GND, ensuring proper signal synchronization. This schematic illustrates both the bidirectional nature of the SDA bus and the centralized synchronization via SCL.

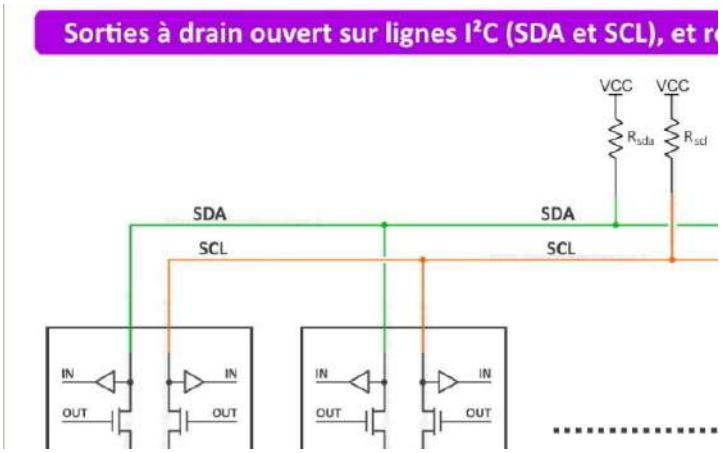

Figure V. 13 - Open-drain outputs on SDA and SCL lines

This diagram highlights how open-drain outputs operate on I²C lines. Each connected element (master or slave) includes a transistor output that can only pull the line low. The high level is achieved through the pull-up resistors (Rsda and Rscl). This mechanism allows multiple devices to share the bus safely without short-circuiting, since none actively drives a high level — the lines remain "free" when no device pulls them down. This principle enables multi-master operation and bus arbitration on I²C networks.

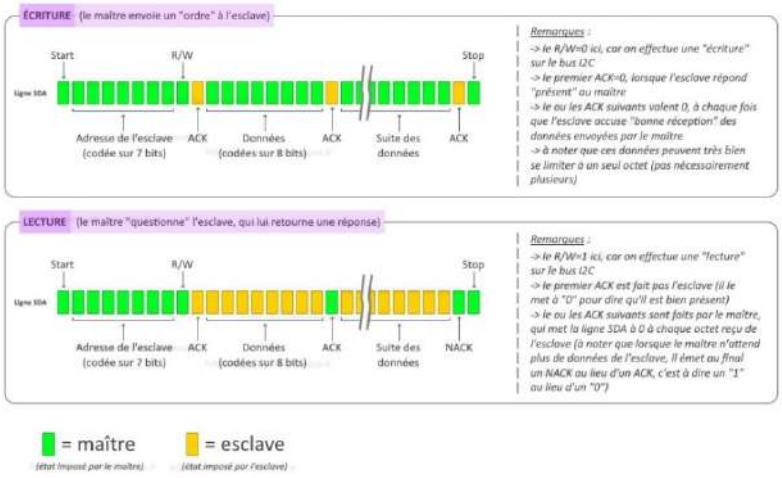

Figure V. 14 - Reading and writing on the I²C bus

This figure explains the two main phases of the protocol:

- Write Operation: The master sends a command to the slave, composed of its 7-bit address, a R/W bit (set to 0 for write), followed by the data to be transferred. The slave confirms receipt with an ACK bit.
- Read Operation: The master queries a slave (R/W bit = 1) and then reads the data transmitted by the slave.

Each step is precisely synchronized by the SCL clock line. The figure also highlights the role of the master (in green) and slaves (in yellow), as well as the bidirectional data flow depending on the operation type.

Tabel V. 1 - Comparison Table: UART – I²C – SPI

Criterion	UART	I²C (Inter-Integrated Circuit)	SPI (Serial Peripheral Interface)
Number of wires (excluding ground/power)	2 wires: RX and TX	2 wires: SDA (data) and SCL (clock)	3 wires (SCK, MOSI, MISO) + 1 additional SS line per slave
Data format	5 to 9 bits	8 bits	Typically 8 bits
Addressing	None (point-to-point link between two elements)	7-bit (or 10-bit) address encoded in the message	Physical: one SS line per slave
Transmission speed	1200 to 115200 bits/s (up to several Mbit/s depending on configuration)	100 kbit/s (Standard Mode)400 kbit/s (Fast Mode)1 Mbit/s (Fast Mode Plus)up to 3.4 Mbit/s (High-Speed Mode)	1 to 20 MHz depending on clock frequency (often ~4 MHz on microcontrollers such as Arduino)
Maximum transmission distance	A few meters (up to several tens with RS-232)	A few meters (reduced at high speed)	A few meters (limited by electromagnetic interference)
Acknowledgment (ACK)	None	1 ACK or NACK bit sent after each packet	None
Pull-up resistors	None required	Mandatory on SDA and SCL (typically 4.7 kΩ)	None required
Communication type	Asynchronous (no shared clock)	Synchronous (master/clock shared)	Synchronous, full duplex
Main advantages	- Simple to implement- Few wires required- Supported by most MCUs	- Multiple slaves on the same bus- Built-in addressing- Good compromise between complexity and performance	- Very fast- Full-duplex communication- Low protocol overhead
Main drawbacks	- No multi-slave support- Limited speed- No native error checking	- Requires external resistors- Lower speed compared to SPI- Sensitive to noise at high frequencies	- Requires one SS line per slave- No addressing or ACK mechanism- Short communication distance
Typical embedded applications	Communication between microcontrollers and GPS, Bluetooth, GSM modules	Environmental sensors, EEPROM memories, RTC circuits	Flash memories, high-speed sensors, OLED/TFT displays, radio modules