I2C - SPI

The Inter-Integrated Circuit (I2C) is a multi-slave, half-duplex, single-ended 8-bit oriented serial bus specification, which uses only two wires to interconnect a given number of slave devices to a master. The Serial Peripheral Interface (SPI) is a specification about serial, synchronous and full-duplex communications between a master controller and several slave devices.

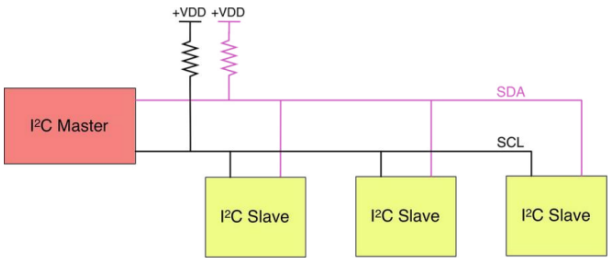
#arm #stm32 #i2c #spi

Last update: 2021-07-06 14:41:43

Table of Content

1. I2C

The Inter-Integrated Circuit (I2C) is a multi-slave, half-duplex, single-ended 8-bit oriented serial bus specification, which uses only two wires to interconnect a given number of slave devices to a master.



An I2C bus

Like SPI, I2C is synchronous, so the output of bits is synchronized to the sampling of bits by a clock signal shared between the master and the slave. The clock signal is always controlled by the master.

The two wires forming an I2C bus are bidirectional open-drain lines, named Serial Data Line (`SDA`) and Serial Clock Line (`SCL`) respectively. The I2C protocol specifies that these two lines need to be pulled up with resistors. It is quite common to use resistors with a value close to 4.7 KΩ. Refer to this guide to calculate the resistors in different use cases.

Modern microcontrollers, like STM32 ones, allow to configure GPIO lines as *open-drain pull-up*, enabling internal pull-up resistors. However, the internal pull-up resistors have a value close to 20 KΩ to avoid unwanted power leaks. Such a value increases the time needed by the bus to reach the HIGH state, reducing the transmission speed. Therefore it is strongly suggested to use external and dedicated pull-up resistors and disable the internal ones.

Being a protocol based on just two wires, there should be a way to address an individual slave device on the same bus. For this reason, I2C defines that each slave device provides a unique slave address for the given bus. The address may be 7- or 10-bit wide (this last option is quite uncommon).

I2C works on different speed rate as below:

- Standard : 100 Kbps
- Fast mode : 400 Kbps
- Fast mode plus : 1 Mbps
- High speed mode : 3.4 Mbps
- Ultra fast mode : 5 Mbps

In the I2C protocol all transactions are always initiated and completed by the master. This is one of the few rules of this communication protocol to keep in mind while programming (and,

especially, debugging) I2C devices. All messages exchanged over the I2C bus are broken up into two types of frame:

- an address frame, where the master indicates to which slave the message is being sent, and
- one or more data frames, which are 8-bit data messages passed from master to slave or vice versa.

Data is placed on the `SDA` line after `SCL` goes low, and it is sampled after the `SCL` line goes high. The time between clock edges and data read/write is defined by devices on the bus and it vary from chip to chip.

An I2C Message

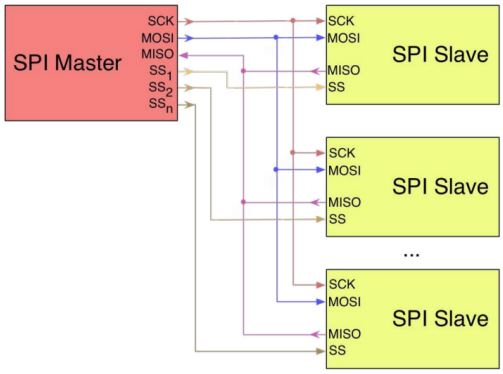
The least significant bit (LSB) in the Address byte is the Read/Write mode. `1` means `Read`, while `0` means `Write`. Therefore, when refer to the address of I2C device, usually, it is known as the Write Address (8-bit address mode). Sometimes, it is written as 7-bit and needed to add one bit for Read/Write mode.

> For example:Device Address = `0xEE` means Write address is `0xEE` and Read Address is `0xEF`.
> Some people will write Device Address = `0x77` (in 7-bit mode), then do calculation `(0x77 << 1) + 0` = `0xEE` for write mode, and do `(0x77 << 1) + 1` = `0xEF` for read mode.

The number of bytes that can be transmitted per transfer is unrestricted. Each byte must be followed by an Acknowledge (`ACK`) bit. Data is transferred with the Most Significant Bit (MSB) first. The ACK takes place after every byte. The ACK bit allows the receiver to signal the transmitter that the byte was successfully received and another byte may be sent. The master generates all clock pulses over the `SCL` line, including the ACK ninth clock pulse.

2. SPI

The Serial Peripheral Interface (SPI) is a specification about serial, synchronous and full-duplex communications between a master controller and several slave devices. The nature of the SPI interface allows full duplex as well as half duplex communications over the same bus. Different from the I2C protocol, the SPI specification does not force a given message protocol over its bus, but it is limited to bus signaling giving to slave devices total freedom about the structure of exchanged messages.

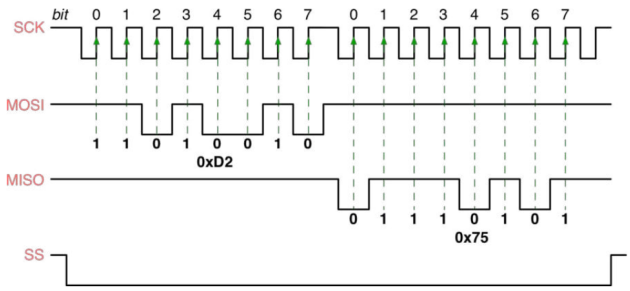


A SPI bus

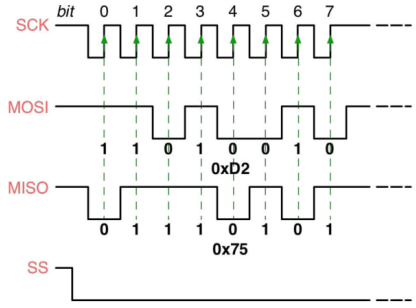
In SPI, only one side generates the clock signal (usually called `CLK` or `SCK` for Serial ClocK). The side that generates the clock is called the "master", and the other side is called the "slave". There is always only one master (which is almost always a microcontroller), but there can be multiple slaves (more on this in a bit).

When data is sent from the master to a slave, it's sent on a data line called `MOSI`, for "Master Out / Slave In". If the slave needs to send a response back to the master, the master will continue to generate a prearranged number of clock cycles, and the slave will put the data onto a third data line called `MISO`, for "Master In / Slave Out".

- `SCK` : this signal I/O is used to generate the clock to synchronize data transfer over the SPI bus. It is generated by the master device, and this means that in an SPI bus every transfer is always started by the master. Different from the I2C specification, the SPI is intrinsically faster and the SPI clock speed is usually several MHz. Nowadays is quite common to find SPI devices able to exchange data at a rate up to 100MHz. Moreover, the SPI protocol allows to devices with different communication speeds to coexist over the same bus.
- `MOSI` : the name of this signal I/O stands for *Master Output Slave Input*, and it is used to send data from the master device to a slave one. Different from the I2C bus, where just one wire is used to exchange data both the ways, the SPI protocol defines two distinct lines to exchange data between master and slaves.
- `MISO` : it stands for *Master Input Slave Output* and it corresponds to the I/O line used to send data from a slave device to the master.
- `SSn` : it stands for *Slave Select* and in a typical SPI bus there exist `n` separated lines used to address the specific SPI devices involved in a transaction. Different from the I2C protocol, the SPI does not use slave addresses to select devices, but it demands this operation to a physical line that is asserted LOW to perform a selection. In a typical SPI bus only one slave device can be active at same time by asserting low its SS line. This is the reason why devices with different communication speed can coexist on the same bus



Half-duplex data exchange on SPI bus



Full-duplex data exchange on SPI bus

> **Reading data**
>
> Notice it's said "prearranged" in the above description. Because the master always generates the clock signal, it must know in advance when a slave needs to return data and how much data will be returned. This is very different than asynchronous serial, where random amounts of data can be sent in either direction at any time. In practice this isn't a problem, as SPI is generally used to talk to sensors that have a very specific command structure. In cases of a variable amount of data, slave could always return one or two bytes specifying the length of the data and then have the master retrieve the full amount after that.

Note that SPI is "full duplex" (has separate send and receive lines), and, thus, in certain situations, SPI can transmit and receive data at the same time (for example, requesting a new sensor reading while retrieving the data from the previous one).

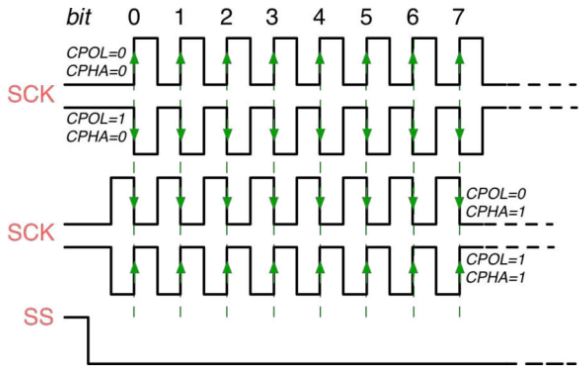
In addition to setting the bus clock frequency, the master and slaves must also agree on the **clock polarity and phase** with respect to the data exchanged over `MOSI` and `MISO` lines.

At `CPOL=0` the base value of the clock is zero, i.e. the active state is 1 and idle state is 0.

- For `CPHA=0`, data is captured on the SCK rising edge (LOW → HIGH transition) and data is output on a falling edge (HIGH → LOW clock transition).
- For `CPHA=1`, data is captured on the SCK falling edge and data is output on a rising edge.

At `CPOL=1` the base value of the clock is one (inversion of CPOL=0), i.e. the active state is 0 and idle state is 1.

- For `CPHA=0`, data is captured on SCK falling edge and data is output on a rising edge.
- For `CPHA=1`, data is captured on SCK rising edge and data is output on a falling edge.



Clock timing for data capture in difference of CPOL and CPHA

3. I2C vs SPI

Here are important differences between I2C and SPI protocols. Advantages are in bold text.

I2C	SPI
Multi-master and multi-slave	Single-master and multi-slave
Two wires	At least 4 wires
Half-duplex	**Full-duplex**
Clock stretching If the slave cannot able to send data fast enough then it suppresses the clock to stop the communication	N/A
Slow speed	**High speed**
Less susceptible to noise	More susceptible to noise
Acknowledgment bit after each byte → More reliable	N/A
Extra overhead due to start and stop bits	**No overhead**, no start-stop bits
Long distance	Short distance

Why is I2C slower than SPI?

Firstly, at the same bit rate, I2C has overhead of extra bits for the address byte, start and stop bit, and an ack for every data byte. SPI does not have those information.

Secondary, a hardware problem, I2C use open-collector lines which means the transmitter only drives the line to low. The speed on the I2C lines depends on the capacitance and the pull-up resistor.

4. Working modes

Like other peripheral, both I2C and SPI support Polling, Interrupt and DMA mode. In high bit rate, DMA should be used to avoid overlapped interrupts.

SPI is dropped when using Interrupt at high bit rate

Take an example of SPI on 10 Mbps, which will generate 1.25 millions interrupts per second (!) in case of 8-bit transfer. An STM32L4xx running at 80 MHz will have a room of only 64 cycles to process an Interrupt. However, calling an interrupt takes 12 cycle, exiting an interrupt costs 10 cycles (ideal state without waiting). Therefore, an interrupt code must be served in only 42 cycles !

Should use DMA when the bit rate is high, because DMA only causes 2 interrupts: “Half-transfer” and “Transfer-Complete” in a routine of transfer a large data buffer.

Maximum SPI frequency in different modes

> Excerpt from source code `stm32xxxx_hal_spi.c`

```
DataSize = SPI_DATASIZE_8BIT:
+-----------------------------------------------------------------------------------------+
|         |                | 2Lines Fullduplex   |     2Lines RxOnly    |         1Line        |
| Process | Transfer mode  |---------------------|----------------------|----------------------|
|         |                |  Master  |  Slave   |  Master   |  Slave   |  Master   |  Slave   |
|=========================================================================================|
|    T    |     Polling    | Fpclk/4  | Fpclk/8  |    NA     |    NA    |    NA     |   NA     |
|    X    |----------------|----------|----------|-----------|----------|-----------|----------|
|    /    |     Interrupt  | Fpclk/4  | Fpclk/16 |    NA     |    NA    |    NA     |   NA     |
|    R    |----------------|----------|----------|-----------|----------|-----------|----------|
|    X    |       DMA      | Fpclk/2  | Fpclk/2  |    NA     |    NA    |    NA     |   NA     |
|=========|================|==========|==========|===========|==========|===========|==========|
|         |     Polling    | Fpclk/4  | Fpclk/8  | Fpclk/16  | Fpclk/8  | Fpclk/8   | Fpclk/8  |
|         |----------------|----------|----------|-----------|----------|-----------|----------|
|    R    |     Interrupt  | Fpclk/8  | Fpclk/16 | Fpclk/8   | Fpclk/8  | Fpclk/8   | Fpclk/4  |
|    X    |----------------|----------|----------|-----------|----------|-----------|----------|
|         |       DMA      | Fpclk/4  | Fpclk/2  | Fpclk/2   | Fpclk/16 | Fpclk/2   | Fpclk/16 |
|=========|================|==========|==========|===========|==========|===========|==========|
|         |     Polling    | Fpclk/8  | Fpclk/2  |    NA     |    NA    | Fpclk/8   | Fpclk/8  |
|         |----------------|----------|----------|-----------|----------|-----------|----------|
|    T    |     Interrupt  | Fpclk/2  | Fpclk/4  |    NA     |    NA    | Fpclk/16  | Fpclk/8  |
|    X    |----------------|----------|----------|-----------|----------|-----------|----------|
|         |       DMA      | Fpclk/2  | Fpclk/2  |    NA     |    NA    | Fpclk/8   | Fpclk/16 |
+-----------------------------------------------------------------------------------------+
```

```
DataSize = SPI_DATASIZE_16BIT:
+-----------------------------------------------------------------------------------------------+
|         |                | 2Lines Fullduplex   |     2Lines RxOnly    |         1Line        |
| Process | Transfer mode  |---------------------|----------------------|----------------------|
|         |                |  Master  |  Slave   |  Master   |  Slave   |  Master   |  Slave   |
|===============================================================================================|
|    T    |     Polling    | Fpclk/4  | Fpclk/8  |    NA     |    NA    |    NA     |   NA     |
|    X    |----------------|----------|----------|-----------|----------|-----------|----------|
|    /    |     Interrupt  | Fpclk/4  | Fpclk/16 |    NA     |    NA    |    NA     |   NA     |
|    R    |----------------|----------|----------|-----------|----------|-----------|----------|
|    X    |       DMA      | Fpclk/2  | Fpclk/2  |    NA     |    NA    |    NA     |   NA     |
|=========|================|==========|==========|===========|==========|===========|==========|
|         |     Polling    | Fpclk/4  | Fpclk/8  | Fpclk/16  | Fpclk/8  | Fpclk/8   | Fpclk/8  |
|         |----------------|----------|----------|-----------|----------|-----------|----------|
|    R    |     Interrupt  | Fpclk/8  | Fpclk/16 | Fpclk/8   | Fpclk/8  | Fpclk/8   | Fpclk/4  |
|    X    |----------------|----------|----------|-----------|----------|-----------|----------|
|         |       DMA      | Fpclk/4  | Fpclk/2  | Fpclk/2   | Fpclk/16 | Fpclk/2   | Fpclk/16 |
|=========|================|==========|==========|===========|==========|===========|==========|
|         |     Polling    | Fpclk/8  | Fpclk/2  |    NA     |    NA    | Fpclk/8   | Fpclk/8  |
|         |----------------|----------|----------|-----------|----------|-----------|----------|
|    T    |     Interrupt  | Fpclk/2  | Fpclk/4  |    NA     |    NA    | Fpclk/16  | Fpclk/8  |
|    X    |----------------|----------|----------|-----------|----------|-----------|----------|
|         |       DMA      | Fpclk/2  | Fpclk/2  |    NA     |    NA    | Fpclk/8   | Fpclk/16 |
+-----------------------------------------------------------------------------------------------+
```

5. STM32Cube HAL Usage

The Hardware Abstract Layer (HAL) is designed so that it abstracts from the specific peripheral memory mapping. But, it also provides a general and more user-friendly way to configure the peripheral, without forcing the programmers to now how to configure its registers in detail.

> excerpt from Description of STM32F0 HAL and low-layer drivers

5.1. I2C HAL

1. Declare a `I2C_HandleTypeDef` handle structure, for example: `I2C_HandleTypeDef hi2c` ;
2. Initialize the I2C low level resources by implementing the `HAL_I2C_MspInit()` API:

    a. Enable the I2Cx interface clock

    b. I2C pins configuration

    - Enable the clock for the I2C GPIOs
    - Configure I2C pins as alternate function open-drain

    c. NVIC configuration if you need to use interrupt process

    - Configure the I2Cx interrupt priority
    - Enable the NVIC I2C IRQ Channel

    d. DMA Configuration if you need to use DMA process

    - Declare a DMA_HandleTypeDef handle structure for the transmit or receive channel
    - Enable the DMAx interface clock using

- Configure the DMA handle parameters
- Configure the DMA Tx or Rx channel
- Associate the initialized DMA handle to the hi2c DMA Tx or Rx handle
- Configure the priority and enable the NVIC for the transfer complete interrupt on the DMA Tx or Rx channel

3. Configure the *Communication Clock Timing*, *Own Address1*, *Master Addressing mode*, *Dual Addressing mode*, *Own Address2*, *Own Address2 Mask*, General call and No-Stretch mode in the hi2c Init structure.
4. Initialize the I2C registers by calling the `HAL_I2C_Init()`, configures also the low level Hardware (GPIO, CLOCK, NVIC...etc) by calling the customized `HAL_I2C_MspInit(&hi2c)` API.
5. To check if target device is ready for communication, use the function `HAL_I2C_IsDeviceReady()`
6. For I2C IO and IO MEM operations, three operation modes are available within this driver:

   a. Polling mode IO operation

   - Transmit in master mode an amount of data in blocking mode using `HAL_I2C_Master_Transmit()`
   - Receive in master mode an amount of data in blocking mode using `HAL_I2C_Master_Receive()`
   - Transmit in slave mode an amount of data in blocking mode using `HAL_I2C_Slave_Transmit()`
   - Receive in slave mode an amount of data in blocking mode using `HAL_I2C_Slave_Receive()`

   b. Polling mode IO MEM operation

   - Write an amount of data in blocking mode to a specific memory address using `HAL_I2C_Mem_Write()`
   - Read an amount of data in blocking mode from a specific memory address using `HAL_I2C_Mem_Read()`

   c. Interrupt mode IO operation

   - Transmit in master mode an amount of data in non-blocking mode using `HAL_I2C_Master_Transmit_IT()`
   - At transmission end of transfer, `HAL_I2C_MasterTxCpltCallback()` is executed and user can add his own code by customization of function pointer `HAL_I2C_MasterTxCpltCallback()`
   - Receive in master mode an amount of data in non-blocking mode using `HAL_I2C_Master_Receive_IT()`

- At reception end of transfer, `HAL_I2C_MasterRxCpltCallback()` is executed and user can add his own code by customization of function pointer `HAL_I2C_MasterRxCpltCallback()`
- Transmit in slave mode an amount of data in non-blocking mode using `HAL_I2C_Slave_Transmit_IT()`
- At transmission end of transfer, `HAL_I2C_SlaveTxCpltCallback()` is executed and user can add his own code by customization of function pointer `HAL_I2C_SlaveTxCpltCallback()`
- Receive in slave mode an amount of data in non-blocking mode using `HAL_I2C_Slave_Receive_IT()`
- At reception end of transfer, `HAL_I2C_SlaveRxCpltCallback()` is executed and user can add his own code by customization of function pointer `HAL_I2C_SlaveRxCpltCallback()`
- In case of transfer Error, `HAL_I2C_ErrorCallback()` function is executed and user can add his own code by customization of function pointer `HAL_I2C_ErrorCallback()`
- Abort a master I2C process communication with Interrupt using `HAL_I2C_Master_Abort_IT()`
- End of abort process, `HAL_I2C_AbortCpltCallback()` is executed and user can add his own code by customization of function pointer `HAL_I2C_AbortCpltCallback()`
- Discard a slave I2C process communication using __HAL_I2C_GENERATE_NACK() macro. This action will inform Master to generate a Stop condition to discard the communication.

5.2. SPI HAL

1. Declare a `SPI_HandleTypeDef` handle structure, for example: `SPI_HandleTypeDef hspi;`
2. Initialize the SPI low level resources by implementing the `HAL_SPI_MspInit()` API:
   - Enable the SPIx interface clock
   - SPI pins configuration
     - Enable the clock for the SPI GPIOs
     - Configure these SPI pins as alternate function push-pull
   - NVIC configuration if you need to use interrupt process
     - Configure the SPIx interrupt priority
     - Enable the NVIC SPI IRQ handle

- DMA Configuration if you need to use DMA process
  - Declare a DMA_HandleTypeDef handle structure for the transmit or receive Stream/Channel
  - Enable the DMAx clock
  - Configure the DMA handle parameters
  - Configure the DMA Tx or Rx Stream/Channel
  - Associate the initialized hdma_tx handle to the hspi DMA Tx or Rx handle
  - Configure the priority and enable the NVIC for the transfer complete interrupt on the DMA Tx or Rx Stream/Channel

3. Program the Mode, BidirectionalMode , Data size, Baudrate Prescaler, NSS management, Clock polarity and phase, FirstBit and CRC configuration in the hspi Init structure.
4. Initialize the SPI registers by calling the `HAL_SPI_Init()` API: This API configures also the low level Hardware GPIO, CLOCK, CORTEX...etc) by calling the customized `HAL_SPI_MspInit()` API.

Circular mode restriction:

1. The DMA circular mode cannot be used when the SPI is configured in these modes:
   - Master 2 Lines Rx Only
   - Master 1 Line Rx
2. The CRC feature is not managed when the DMA circular mode is enabled
3. When the SPI DMA Pause/Stop features are used, you must use the following APIs the `HAL_SPI_DMAPause()` / `HAL_SPI_DMAStop()` only under the SPI callbacks

Master Receive mode restriction:

- In Master unidirectional receive-only mode (`MSTR=1` , `BIDIMODE=0` , `RXONLY=0`) or bidirectional receive mode (`MSTR=1` , `BIDIMODE=1` , `BIDIOE=0`), to ensure that the SPI does not initiate a new transfer the following procedure has to be respected:
  - `HAL_SPI_DeInit()`
  - `HAL_SPI_Init()`

Data buffer address alignment restriction:

1. In case more than 1 byte is requested to be transferred, the HAL SPI uses 16-bit access for data buffer. But there is no support for unaligned accesses on the Cortex-M0 processor. So, if the user wants to transfer more than 1 byte, it shall ensure that 16-bit aligned address is used for:
   - pData parameter in `HAL_SPI_Transmit()` , `HAL_SPI_Transmit_IT()` , `HAL_SPI_Receive()` and `HAL_SPI_Receive_IT()`

- pTxData and pRxData parameters in `HAL_SPI_TransmitReceive()` and `HAL_SPI_TransmitReceive_IT()`

2. There is no such restriction when going through DMA by using `HAL_SPI_Transmit_DMA()`, `HAL_SPI_Receive_DMA()` and `HAL_SPI_TransmitReceive_DMA()`.

The HAL drivers do not allow reaching all supported SPI frequencies in the different SPI modes. Refer to the source code (stm32xxxx_hal_spi.c header) to get a summary of the maximum SPI frequency that can be reached with a data size of 8 or 16 bits, depending on the APBx peripheral clock frequency (fPCLK) used by the SPI instance.

Example projects

Refer to SSD1306 OLED to see examples using I2C and SPI protocol.