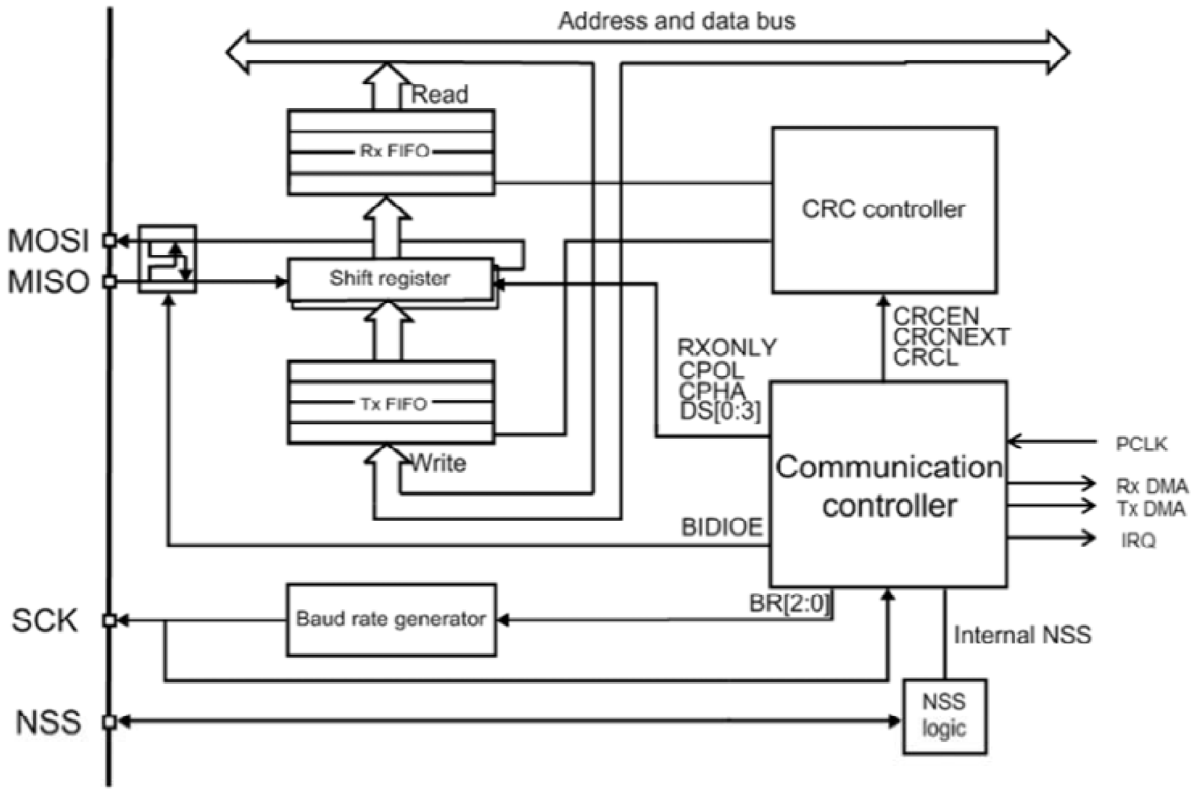
Communication Protocols in STM32

SPI IN STM32

The internal standard peripheral interface or spi provides a simple communication interface allowing the microcontroller to communicate with external devices. This interface is highly configurable to support many standard protocols. applications benefit from the simple and direct connection to components which requires a few pins. Thanks to the high configuration capabilities of the spi, many devices can be simply accommodated in the existing project.

The communication speed can't exceed half of the internal bus frequency, and a minimum of two wires is required to provide the serial data flow synchronized by clock signal in a single direction. An optional hardware slave select control signal can be added.

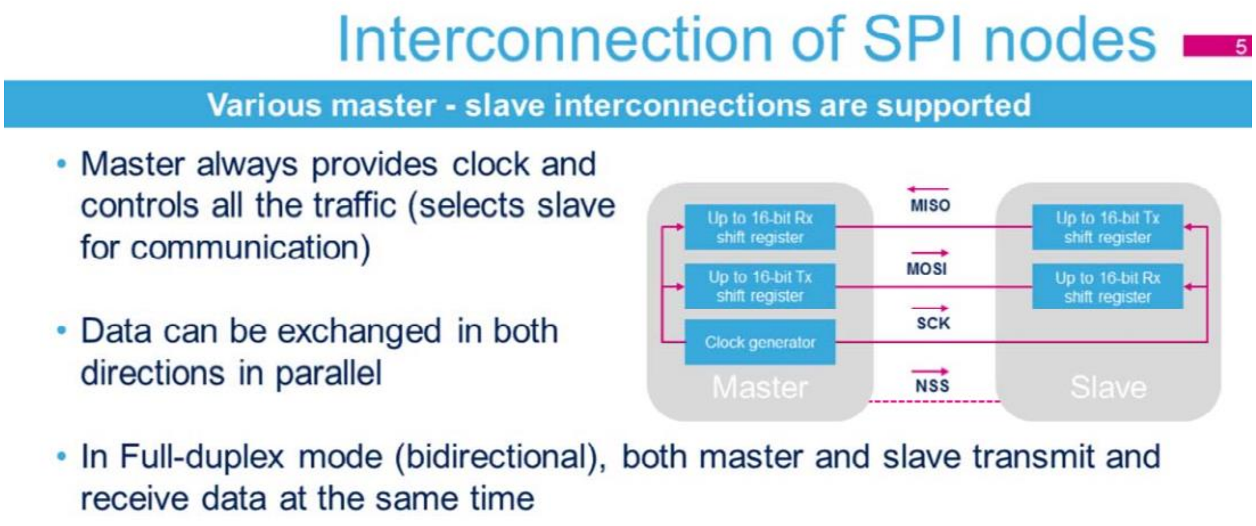
At the protocol level, the user can use specific data buffers with an optional automatic cyclic redundancy check or CRC calculation, and transfers through the DMA controller. There are a wide range of SPI events that can generate interrupt requests.



The simplified SPI block diagram shows the basic control mechanisms and functions. There are 4 I/O signals associated with the SPI peripheral. All the data passes through receive and transmit buffers via their specific interfaces. The control block features are enabled or disabled depending on the configuration.

In typical SPI communication, there should be at least 2 devices attached to the SPI bus. One of them should be the master and the other will essentially be a slave. The master initiates communication by generating a serial clock signal to shift a data frame out, at the same time serial data is being shifted-in to the master. This process is the same whether it's a read or write operation. There are 4 major pins that dictate the SPI connection b/w two devices: -

- MOSI -> Master output slave input (DOUT From Master).
- MISO -> Master input slave output (DOUT From Slave).
- SCLK -> Serial Clock, generated by the master and goes to the slave.
- SS -> Slave Select. Generated by the master to control which slave to talk to. It's usually an active-low signal.



Interconnection of SPI nodes

5

Various master - slave interconnections are supported

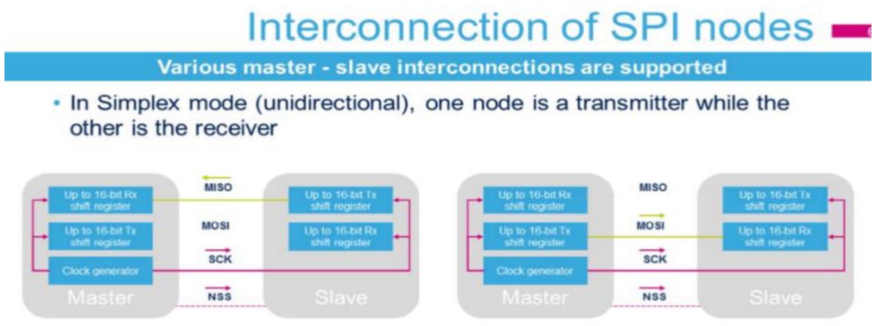
- Master always provides clock and controls all the traffic (selects slave for communication)
- Data can be exchanged in both directions in parallel
- In Full-duplex mode (bidirectional), both master and slave transmit and receive data at the same time

Up to 16-bit Rx shift register
Up to 16-bit Tx shift register
Clock generator
Master
MISO
MOSI
SCK
NSS
Up to 16-bit Tx shift register
Up to 16-bit Rx shift register
Slave

The SPI master always controls the bus traffic and provides the clock signal to the dedicated slave through the SCK line. The master can select the slave it wants to communicate with through the optional Slave Select or NSS signal. Data stored in the dedicated shift registers can be exchanged synchronously between the master and slave through the MOSI (Master Output, Slave Input) and the MISO (Master Input, Slave Output) data lines. In Full-duplex mode, both data lines are used and synchronous data flows in both directions.

Here, SPI can be configured to be in three different modes: - Simplex, Half-Duplex and Full Duplex

1. Simplex Communication



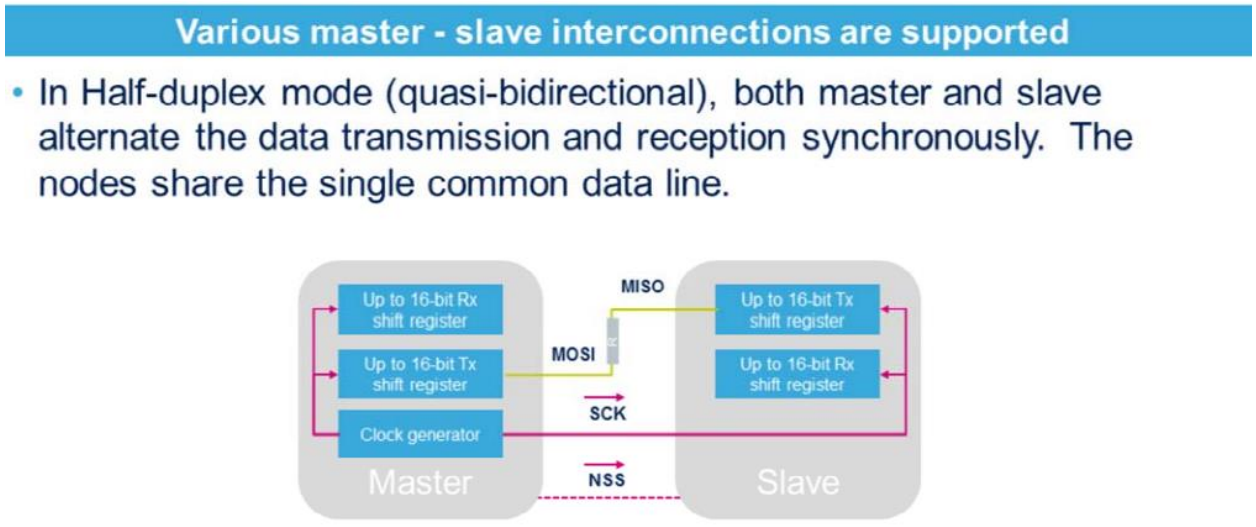
Interconnection of SPI nodes

Various master - slave interconnections are supported

- In Simplex mode (unidirectional), one node is a transmitter while the other is the receiver

Up to 16-bit Rx shift register
Up to 16-bit Tx shift register
Clock generator
Master
MISO
MOSI
SCK
NSS
Up to 16-bit Tx shift register
Up to 16-bit Rx shift register
Slave
Up to 16-bit Rx shift register
Up to 16-bit Tx shift register
Clock generator
Master
MISO
MOSI
SCK
NSS
Up to 16-bit Tx shift register
Up to 16-bit Rx shift register
Slave

2. Half-Duplex Communication



Various master - slave interconnections are supported

- In Half-duplex mode (quasi-bidirectional), both master and slave alternate the data transmission and reception synchronously. The nodes share the single common data line.

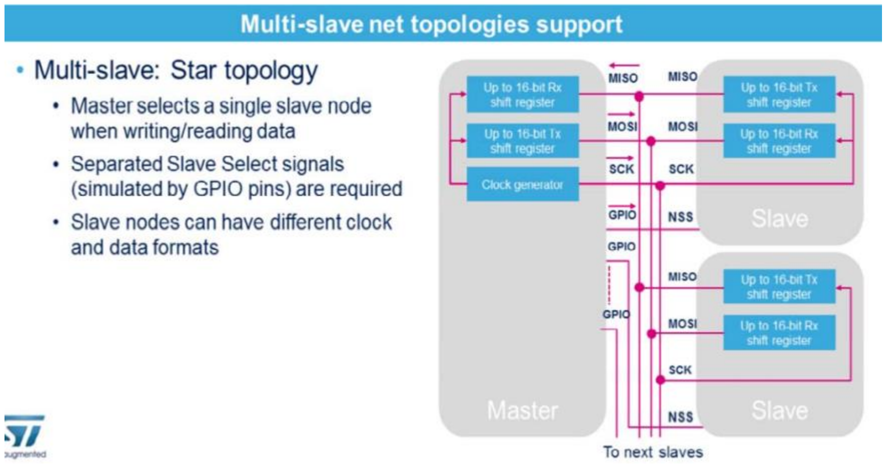
Up to 16-bit Rx shift register
Up to 16-bit Tx shift register
Clock generator
Master
MISO
MOSI
SCK
NSS
Up to 16-bit Tx shift register
Up to 16-bit Rx shift register
Slave

There is a cross connection between the master MOSI and the slave MISO pins in this mode. The master and slave must alternate their transmitter and receiver roles synchronously when having a common data line. It is common to add a serial resistor on the half duplex data line to prevent possible temporary short-circuit connection, since master and slave nodes are not usually synchronized.

In STM32, various slave master interconnections are supported, either there is a system of multiple slaves connected to one master or multiple master system.

In the case of multiple slaves, there are generally two topologies to consider: -

- Star Topology
  The master communicates with one slave at a time, since you can only have one slave transmit data back to the master through the common MISO pin. In this topology, a separated Slave Select signal from the master has to be provided to each slave node, so the master can select which slave to communicate with. Thanks to separate Slave Select signals (NSS), SPI data and clock format can be adapted for each slave, if the multiple slave nodes do not have a common configuration.

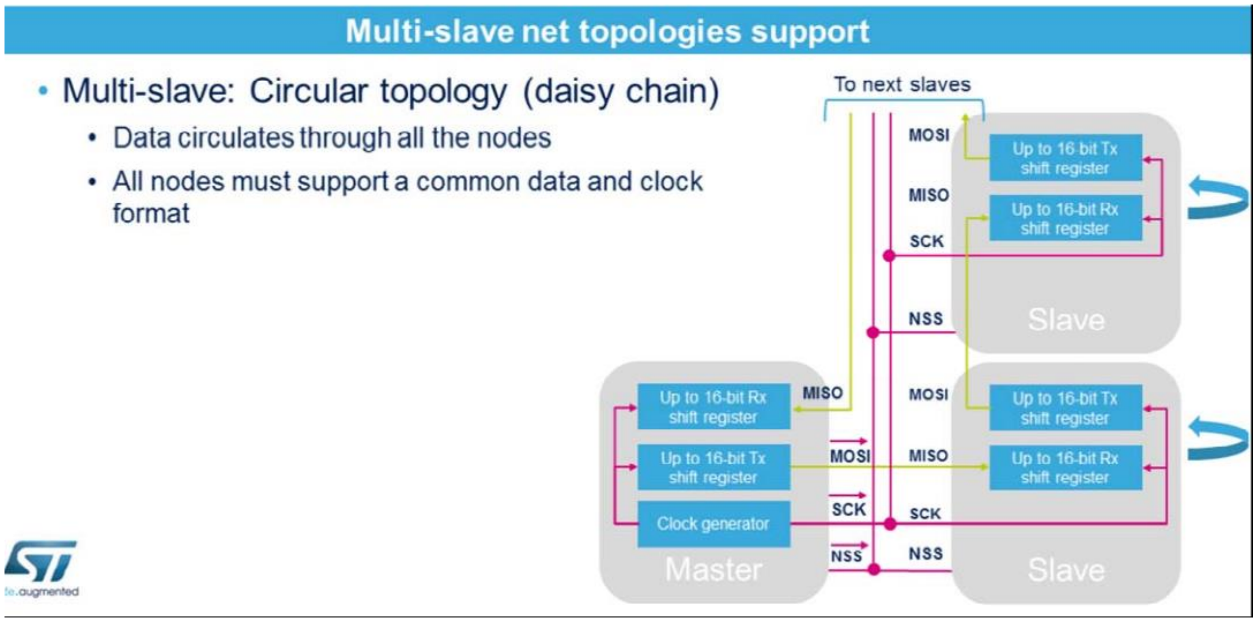


Multi-slave net topologies support

- Multi-slave: Star topology
  - Master selects a single slave node when writing/reading data
  - Separated Slave Select signals (simulated by GPIO pins) are required
  - Slave nodes can have different clock and data formats

Up to 16-bit Rx shift register
Up to 16-bit Tx shift register
Clock generator
Master
MISO
MOSI
SCK
GPIO
GPIO
GPIO
MISO
MOSI
SCK
NSS
Up to 16-bit Tx shift register
Up to 16-bit Rx shift register
Slave
MISO
MOSI
SCK
NSS
Up to 16-bit Tx shift register
Up to 16-bit Rx shift register
Slave
To next slaves

- Circular Topology
  Another multi-slave configuration is the circular topology where the inputs and outputs of all the nodes are connected in a closed serial chain. A common Slave Select signal is used for all the nodes as communication occurs at the same time. All nodes must have the same data and clock format configuration. Microcontroller SPI nodes typically use separate internal transmit and receive shift registers, so the data transferred between them must be handled by software in a circular mode.

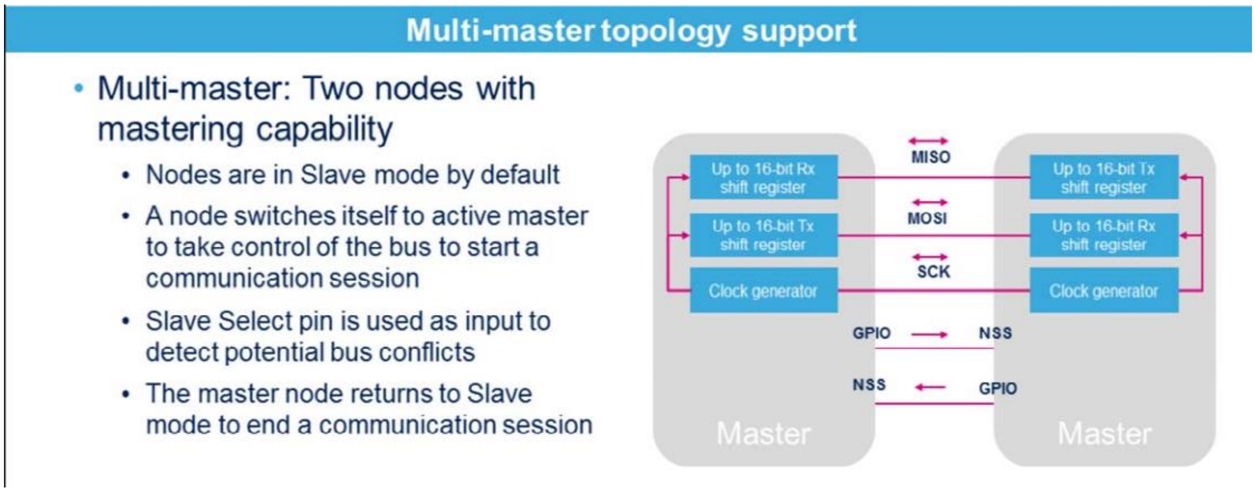


Multi-slave net topologies support

- Multi-slave: Circular topology (daisy chain)
  - Data circulates through all the nodes
  - All nodes must support a common data and clock format

To next slaves
MOSI
MISO
SCK
NSS
Up to 16 bit Tx shift register
Up to 16-bit Rx shift register
Slave
Up to 16-bit Rx shift register
Up to 16-bit Tx shift register
Clock generator
Master
Slave
life.augmented

The Multi Master Environment mostly includes switching of two devices between master mode and slave mode. This mode is used to connect two master nodes exclusively. When either node is not active, they are by default in a slave mode. When one node wants to take control of the bus, it switches itself into Master mode and asserts the Slave Select signal on the other node through a GPIO pin. Both Slave Select NSS pins work as a hardware input to detect potential bus collisions between nodes as only one can master the SPI bus at a single time. After the session is done, the active node master releases the Slave Select signal and returns to passive slave mode waiting for the next session to start.



Multi-master topology support

- Multi-master: Two nodes with mastering capability
  - Nodes are in Slave mode by default
  - A node switches itself to active master to take control of the bus to start a communication session
  - Slave Select pin is used as input to detect potential bus conflicts
  - The master node returns to Slave mode to end a communication session

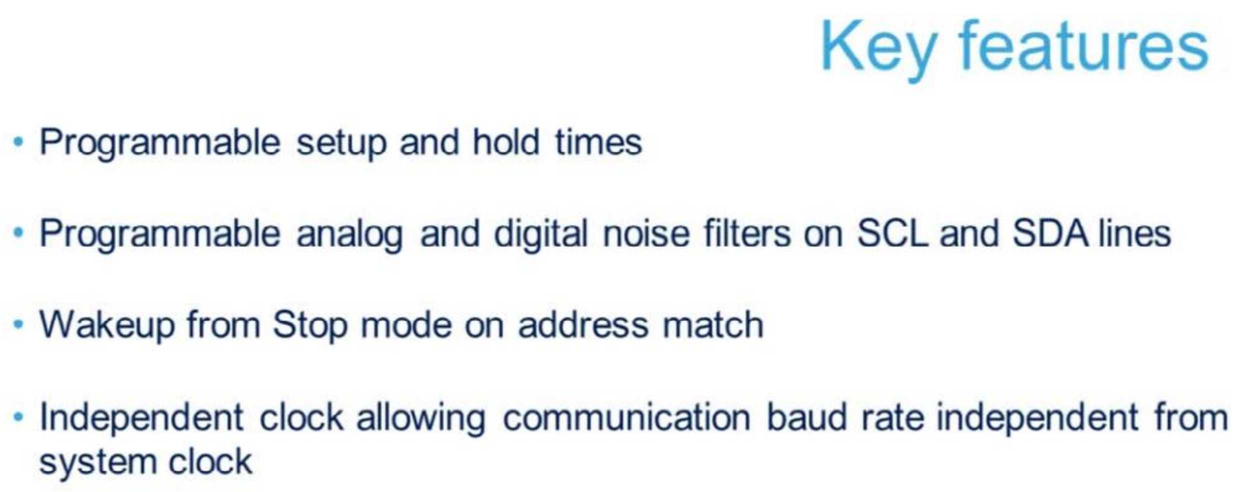
Up to 16-bit Rx shift register
Up to 16-bit Tx shift register
Clock generator
MISO
MOSI
SCK
GPIO
NSS
NSS
GPIO
Master
Master

I2C IN STM32

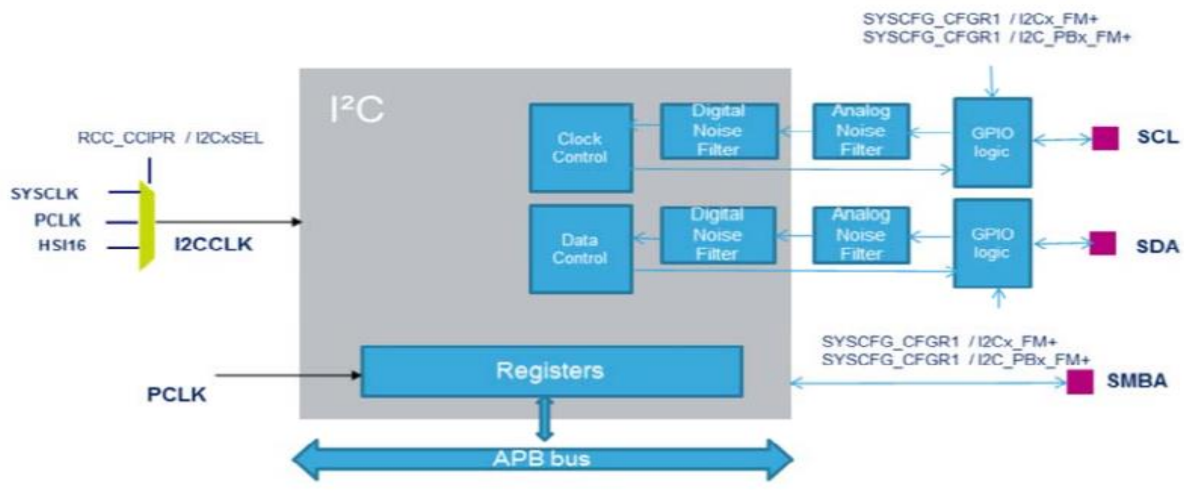
I2C (i-square-c) is an acronym for “Inter-Integrated-Circuit” which was originally created by Philips Semiconductors (now NXP) back in 1982. The I2C is a multi-master, multi-slave, synchronous, bidirectional, half-duplex serial communication bus. It’s widely used for attaching lower-speed peripheral ICs to processors and microcontrollers in short-distance, intra-board communication.

The I²C peripheral supports multi-master and slave modes. The I²C IO pins must be configured in open-drain mode. The logic high level is driven by an external pull-up. The I²C alternate functions are available on IO pins supplied by VDD, which can be from 1.71 to 3.6 volts, and on IO pins supplied by VDDIO2, which can be from 1.08 to 3.6 volts. This allows communication with external chips at voltages different from the STM32L4 main power supply.

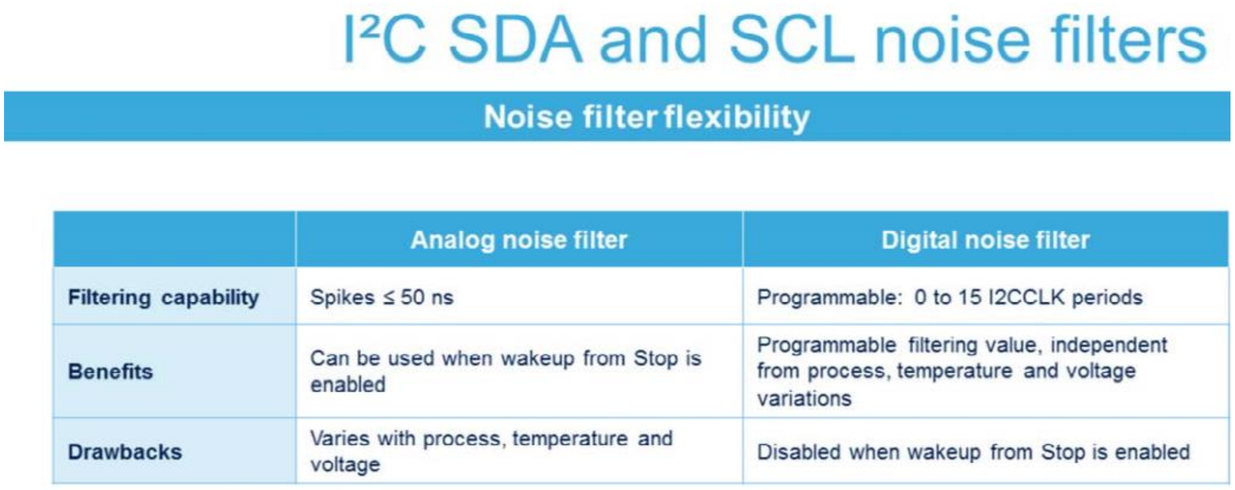
A typical use case is communication with an application processor in sensor hub applications. The IO pins support the 20-mA output drive required for Fast mode Plus. The peripheral controls all I²C bus-specific sequencing, protocol, arbitration and timing values. 7- and 10-bit addressing modes are supported, and multiple 7-bit addresses can be supported in the same application. The peripheral supports slave clock stretching and clock stretching from slave can be disabled by software.


Key features

- Programmable setup and hold times
- Programmable analog and digital noise filters on SCL and SDA lines
- Wakeup from Stop mode on address match
- Independent clock allowing communication baud rate independent from system clock



I²C block diagram. The registers are accessed through the APB bus, and the peripheral is clocked with the I2C clock, which is independent from the APB clock. The I²C clock can be selected between the system clock, APB clock and the high-speed internal 16 MHz RC oscillator. Analog and digital noise filters are present on the SCL and SDA lines. A 20-mA driving capability is enabled using the control bits in the System configuration registers. In addition, an SMBus Alert pin is available in SMBus mode.



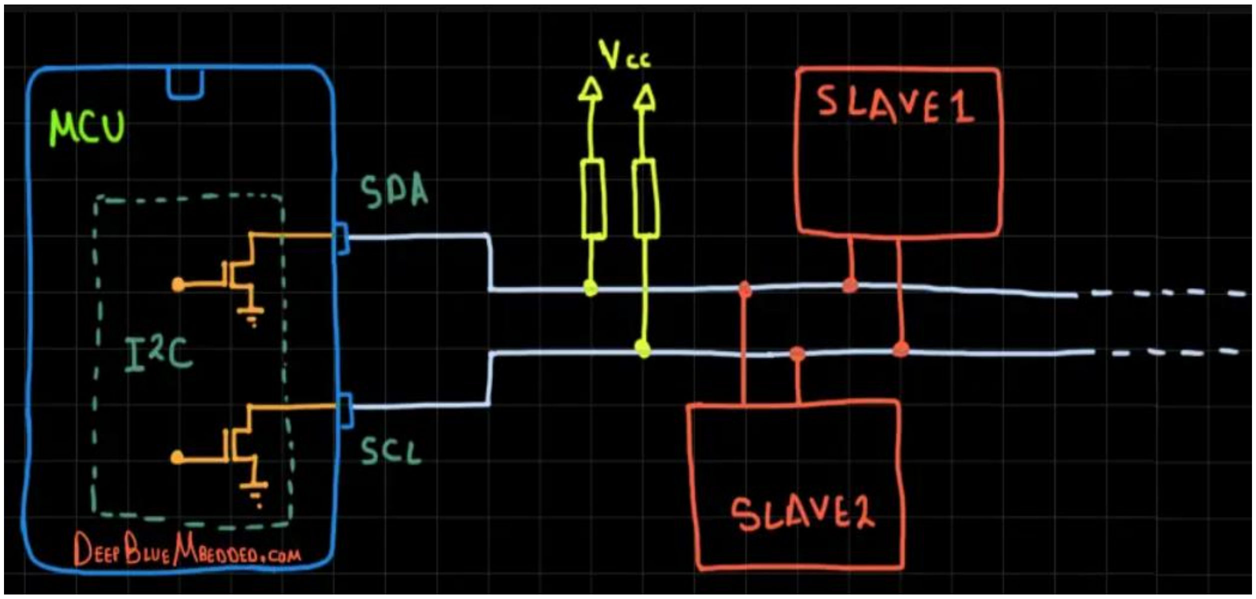
I²C SDA and SCL noise filters

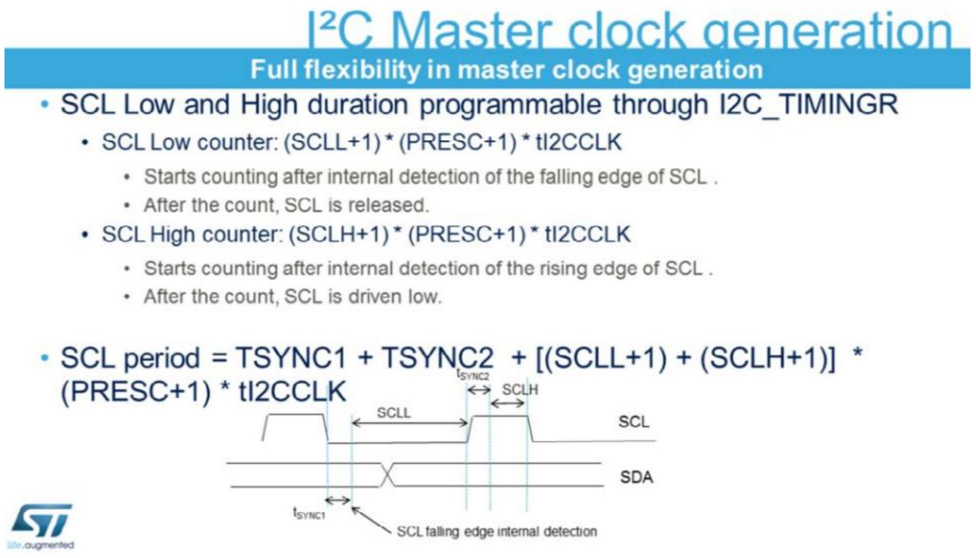
Noise filter flexibility

	Analog noise filter	Digital noise filter
Filtering capability	Spikes ≤ 50 ns	Programmable: 0 to 15 I2CCLK periods
Benefits	Can be used when wakeup from Stop is enabled	Programmable filtering value, independent from process, temperature and voltage variations
Drawbacks	Varies with process, temperature and voltage	Disabled when wakeup from Stop is enabled

The I2C bus lines being "open drain" bidirectional pins makes it perfect for multi-master multi-slave sort of communication without any risk of having collisions. And that's due to having what's called "Bus Arbitration" in case multiple masters did initiate a transaction at the exact same time.

The I2C bus uses what's known as an open-drain (open-collector) output driver for both SDA and SCL lines. Which as the name suggests is having each IO pin connected to the collector of the output driver transistor internally, while having it pulled up to Vcc with a resistor eternally. That's why the default (IDLE) state for each line is HIGH when the open-drain driver is turned OFF. However, if we turn ON the output driver, the IO pin is driven LOW to the ground by the output driver transistor

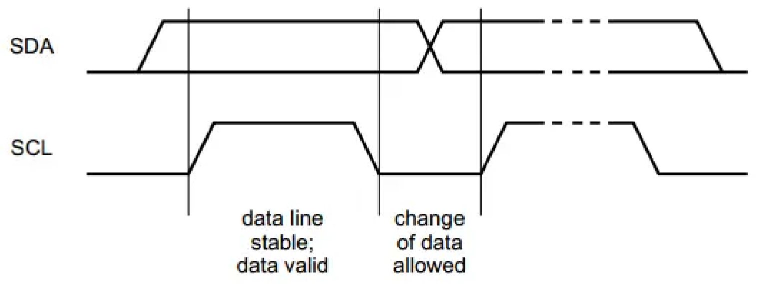




I²C Master clock generation

Full flexibility in master clock generation

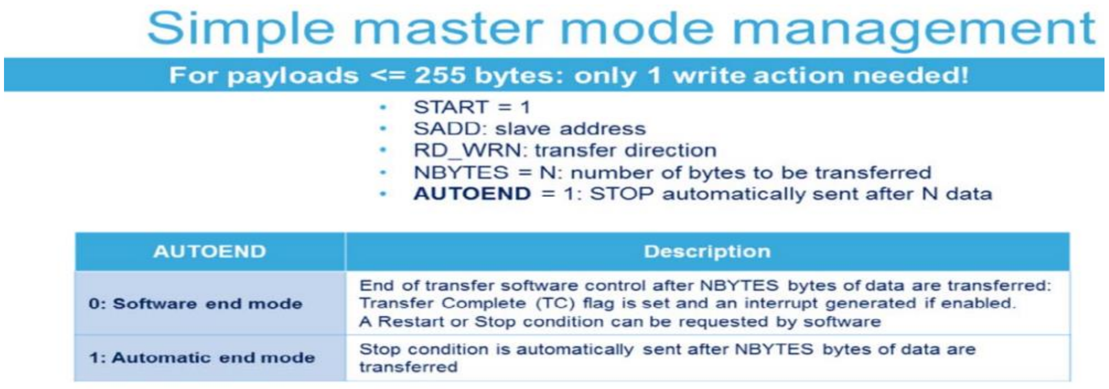
- SCL Low and High duration programmable through I2C_TIMINGR
  - SCL Low counter: (SCLL+1) * (PRESC+1) * tI2CCLK
    - Starts counting after internal detection of the falling edge of SCL .
    - After the count, SCL is released.
  - SCL High counter: (SCLH+1) * (PRESC+1) * tI2CCLK
    - Starts counting after internal detection of the rising edge of SCL .
    - After the count, SCL is driven low.
- SCL period = TSYNC1 + TSYNC2 + [(SCLL+1) + (SCLH+1)] * (PRESC+1) * tI2CCLK



Bit transfer on the I²C-bus

The SCL low- and high-level counters start after the detection of the edge of the SCL line. This implementation allows the peripheral to support the master clock synchronization mechanism in a multi-master environment as well as the slave clock stretching feature.

Therefore, the total SCL period is greater than the sum of the counters. This is linked to the added delays due to the internal detection of the SCL line edge. These delays, tSYNC1 and tSYNC2, depend on the SCL falling or rising edge, the input delay due to the filters, and the delay due to the internal SCL synchronization with the I²C clock.



Simple master mode management

For payloads <= 255 bytes: only 1 write action needed!

- START = 1
- SADD: slave address
- RD_WRN: transfer direction
- NBYTES = N: number of bytes to be transferred
- **AUTOEND** = 1: STOP automatically sent after N data

AUTOEND	Description
0: Software end mode	End of transfer software control after NBYTES bytes of data are transferred: Transfer Complete (TC) flag is set and an interrupt generated if enabled. A Restart or Stop condition can be requested by software
1: Automatic end mode	Stop condition is automatically sent after NBYTES bytes of data are transferred

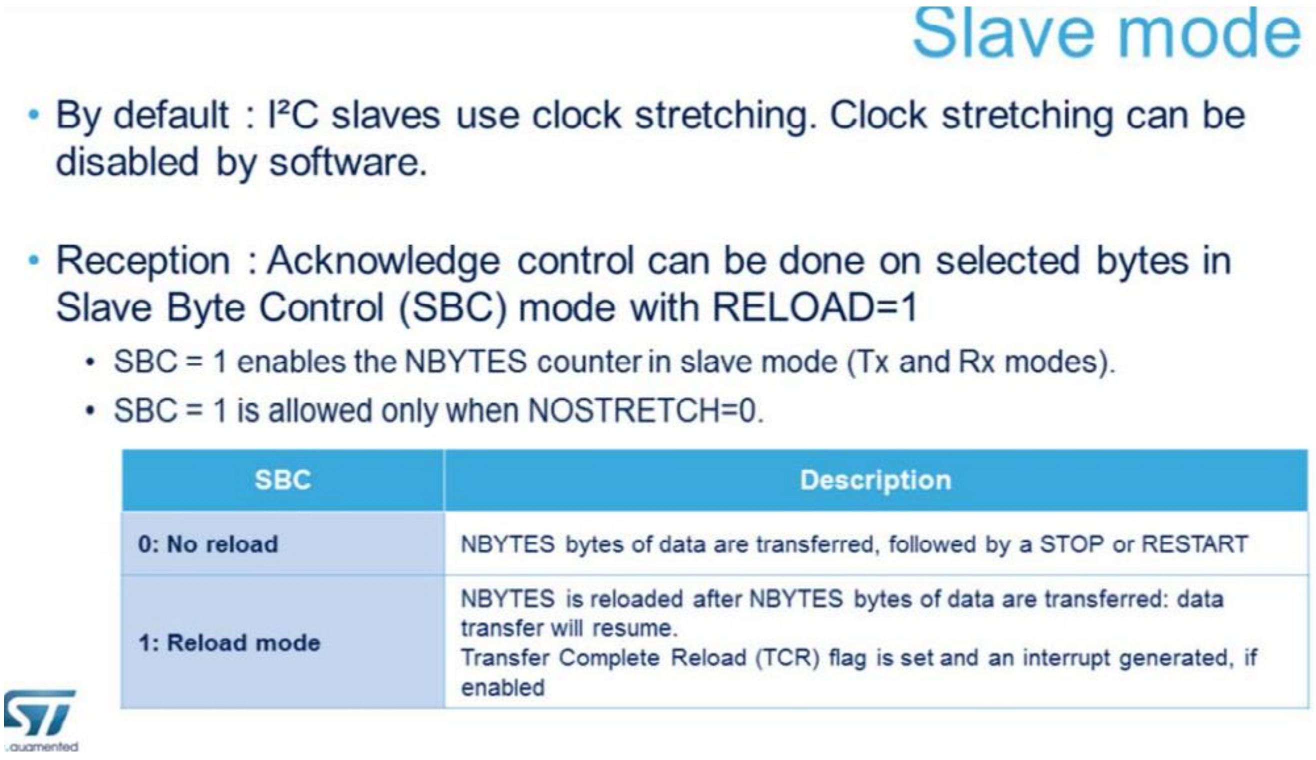
After the programmed number of bytes is transferred, the Transfer Complete (TC) flag is set and an interrupt is generated, if enabled. Then a Repeated Start or a Stop condition can be requested by software. The data transfer can be managed by interrupts or by the DMA.

Clock stretching is essentially holding the clock line SCL LOW by the slave device which prevents any master device on the I2C bus from initiating any new transaction until that slave releases the SCL back again. That's why you should be careful when using clock stretching in slave devices.

By default, the I²C slave uses clock stretching. The clock stretching feature can be disabled by software. In reception, the slave acknowledges on received byte behavior can be configured when Slave Byte Control mode is selected, together with the RELOAD bit being set. When the SBC bit is set, the number of bytes counter is enabled in Slave mode. Clock stretching must be enabled when Slave Byte Control is enabled.

In reception, when slave byte control is enabled with the RELOAD bit set and the number of bytes to be transferred is 1, the Transfer Complete Reload flag is set after each received byte and SCL is stretched. This is done after data reception and before the acknowledge pulse. The Receive Buffer Not Empty flag is also set, so the data can be read. In the TCR subroutine, an Acknowledge or NOT Acknowledge can be programmed to be sent after the byte is received.

It is recommended to clear the SBC bit in transmission, as there is no use for the byte counter in I²C Slave Transmitter mode. In SMBus mode, Slave Byte Control mode is used in transmission for sending the PEC (packet error code) byte. I2C allows several modes, Standard, Fast and High-Speed. Standard mode allows clock frequencies as high as 100kHz while Fast and High-Speed modes are faster. SMbus allows clock frequencies only as high as 100kHz, so it is not necessary to compare SMBus to any but I2C Standard mode.



Slave mode

- By default : I²C slaves use clock stretching. Clock stretching can be disabled by software.
- Reception : Acknowledge control can be done on selected bytes in Slave Byte Control (SBC) mode with RELOAD=1
  - SBC = 1 enables the NBYTES counter in slave mode (Tx and Rx modes).
  - SBC = 1 is allowed only when NOSTRETCH=0.

SBC	Description
0: No reload	NBYTES bytes of data are transferred, followed by a STOP or RESTART
1: Reload mode	NBYTES is reloaded after NBYTES bytes of data are transferred: data transfer will resume. Transfer Complete Reload (TCR) flag is set and an interrupt generated, if enabled

CAN IN STM32

The CAN-FD protocol (CAN with flexible data-rate) is an extension of the classical CAN (CAN 2.0) protocol. CAN-FD is the CAN 2.0 successor. It efficiently supports distributed real-time control with a very high-level of security. CAN-FD was developed by Bosch and standardized as ISO 11898-1:2015 (suitable for industrial, automotive and general embedded communications).

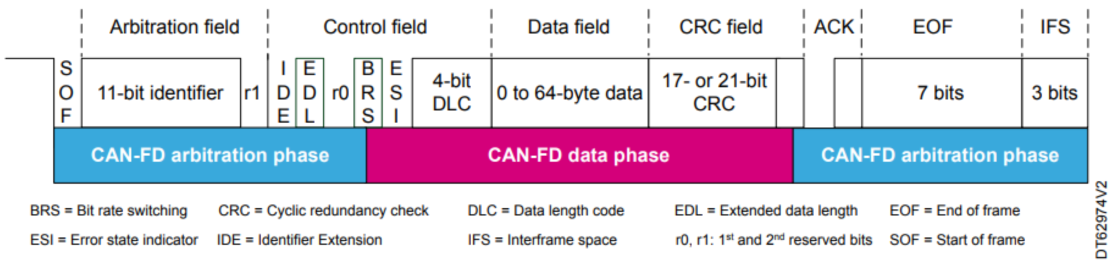
The data sent is packaged into a message as shown in the figure below. A CAN-FD message can be divided into three phases:

1. a first arbitration phase

2. a data phase

3. a second arbitration phase

Figure 1. Standard CAN-FD frame



The first arbitration phase is a message that contains:

• a start of frame (SOF)

• an ID number and other bits, that indicate the purpose of the message (supplying or requesting data), and the speed and format configuration (CAN or CAN-FD)

The data transmission phase consists of:

• the data length code (DLC), that indicates how many data bytes the message contains

• the data the user wishes to send

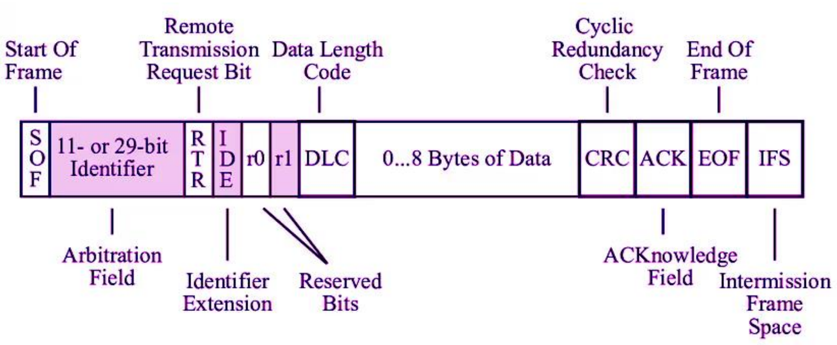
• the check cyclic redundancy sequence (CRC)

• a dominant bit

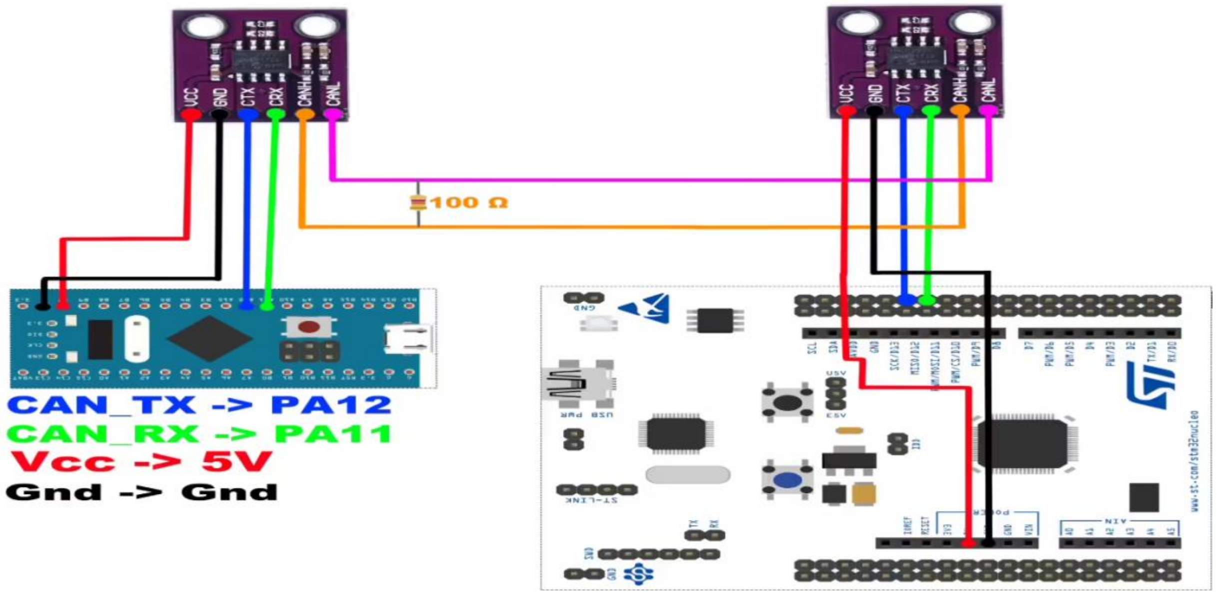
The second arbitration phase contains:

• the receiver of acknowledgment (ACK) transmitted by other nodes on the bus (if at least one has successfully received the message)

• the end of frame (EOF), No message is transmitted during the IFS: the objective is to separate the current frame with the next.



- Here, **Identifier** is the ID of the transmitting Device. It can be either **11 bits** (Standard ID) or **29 bits** (Extended ID).
- **RTR** (Remote Transmission Request) Specifies if the data is Remote frame or Data frame.
- **IDE** specifies if we are using **Standard** ID or **Extended** ID.
- **r** is the Reserved bit.
- **DLC** specifies the data length in Bytes.
- **Data Field** is where we send the actual data bytes. It can be upto 8 bytes in size.
- **CRC** is the checksum data byte.
- **ACK** is the acknowledgment bit.



VCC
GND
CTX
CRX
CANH
CANL
100 Ω
CAN_TX -> PA12
CAN_RX -> PA11
Vcc -> 5V
Gnd -> Gnd