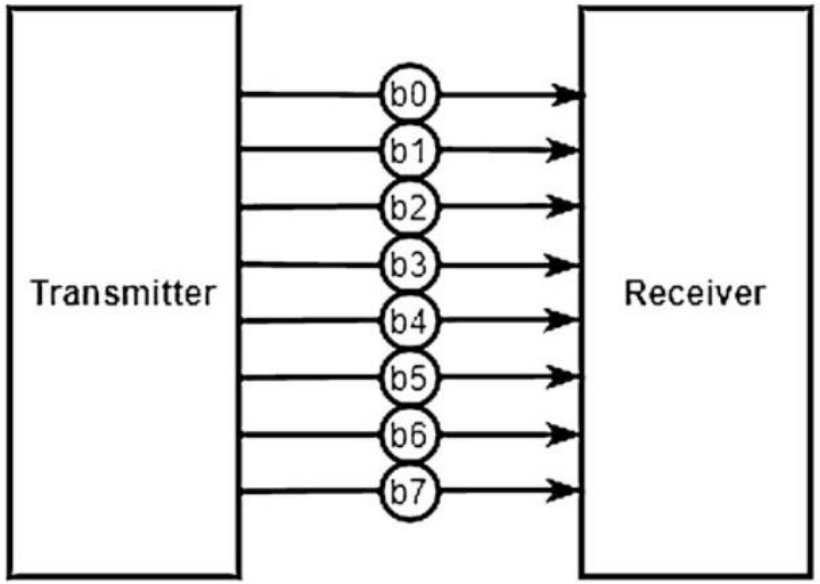
SERIAL COMMUNICATION

Shadi Daana

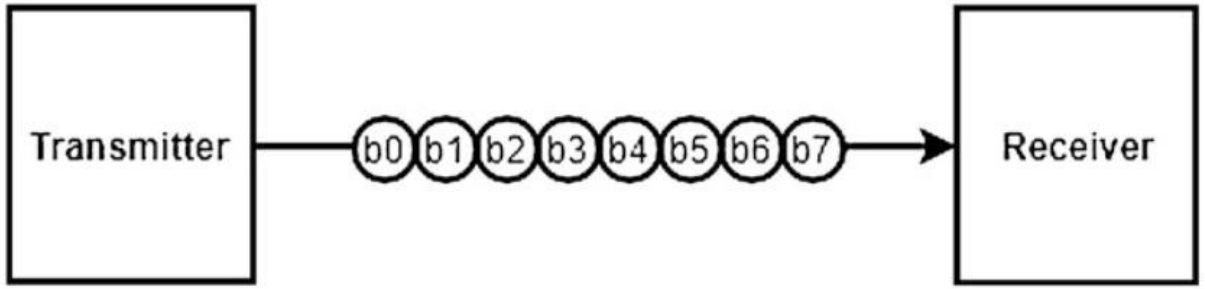
Serial and Parallel Data Transfer

- There are two different data transfer types: **parallel** and **serial**.
- Parallel data transfer is used to achieve high transfer speeds.
- One such application is receiving data from a digital camera or feeding data to an LCD
- The disadvantage of parallel data transfer is that it requires several pins to operate

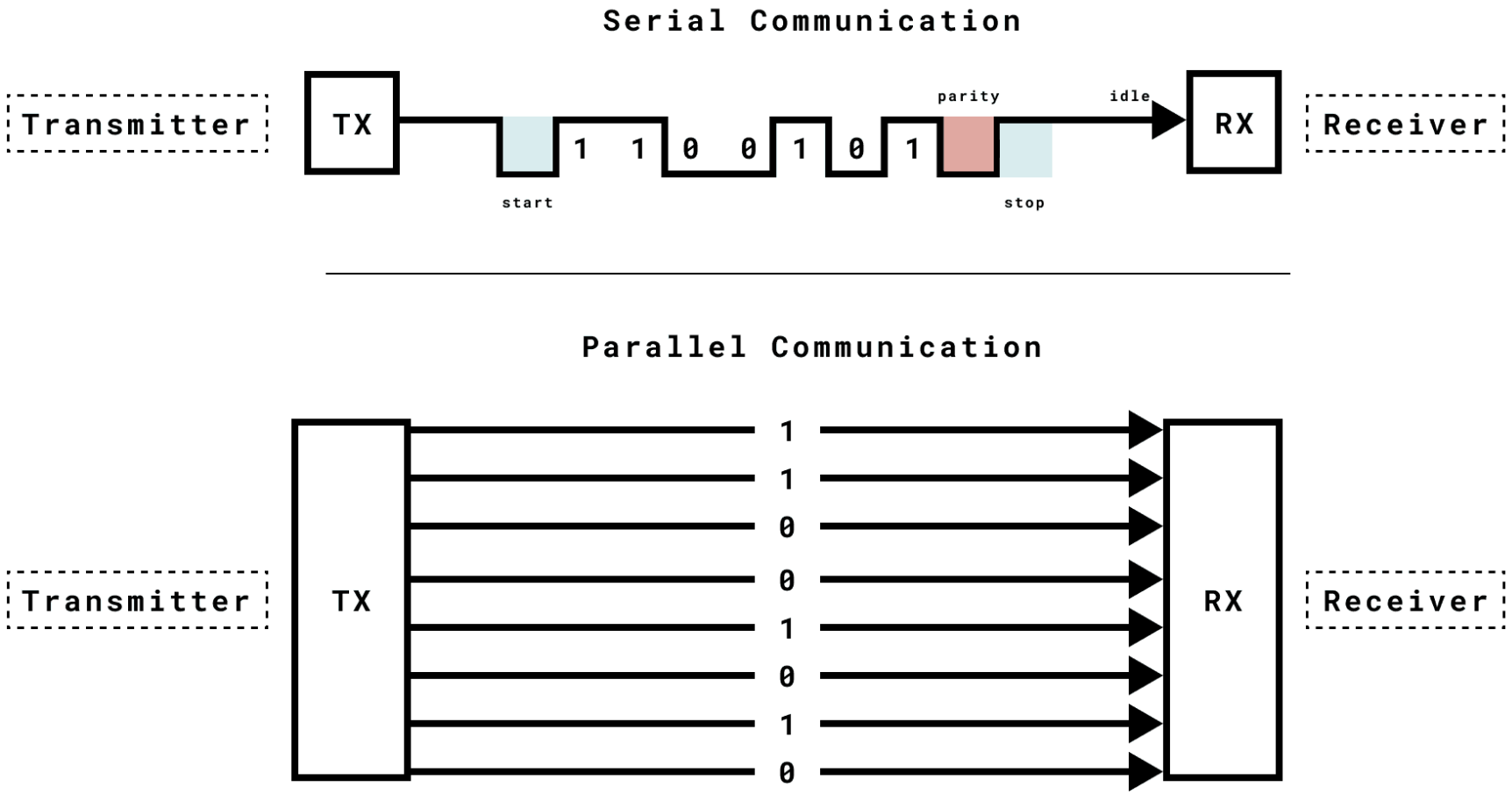


Serial and Parallel Data Transfer

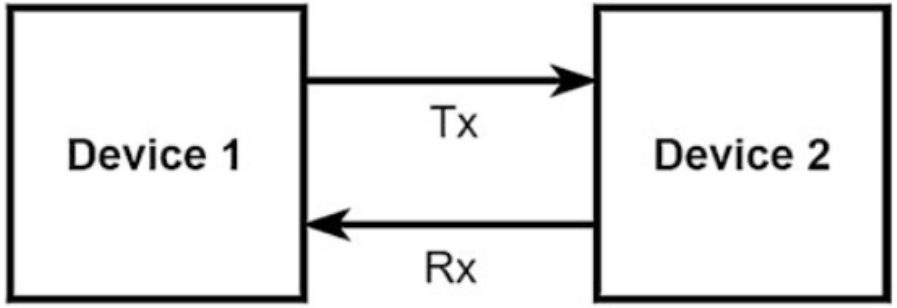
- Serial data transfer provides an alternative such that it only needs one or two pins.
- This allows communicating with more than one device at the same time.
- The disadvantage of serial data transfer is that it is slow since data should be transmitted sequentially

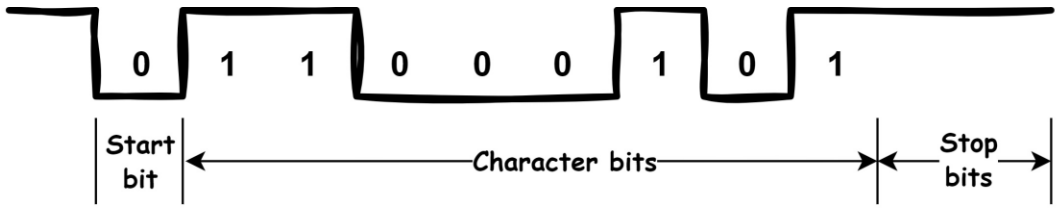


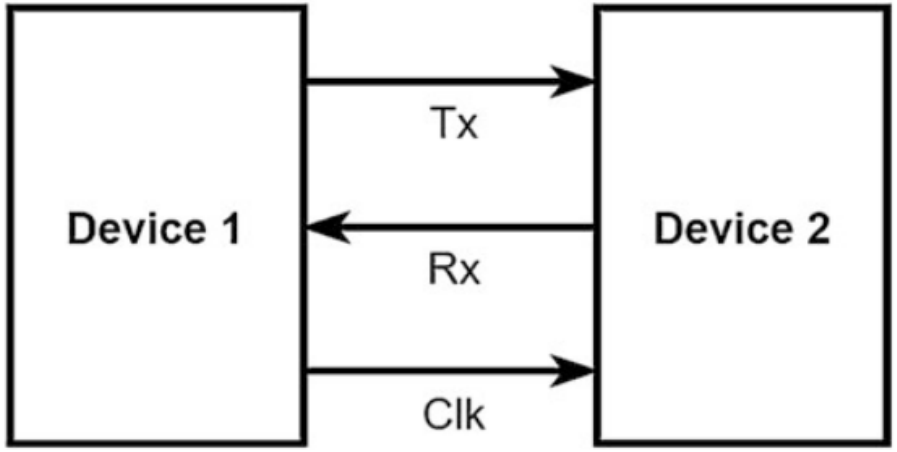
Serial and Parallel Data Transfer

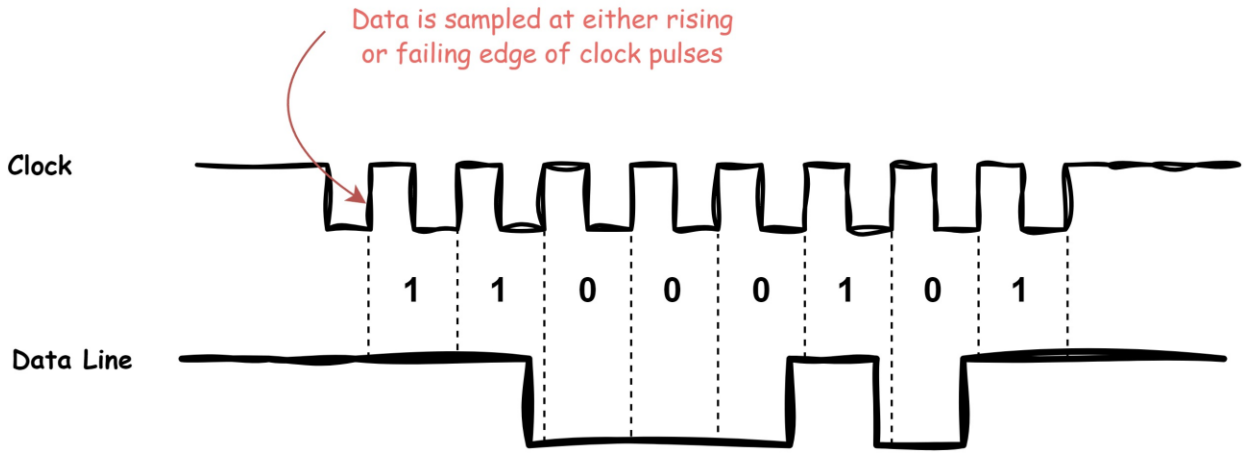


Synchronous and Asynchronous Data Transfer







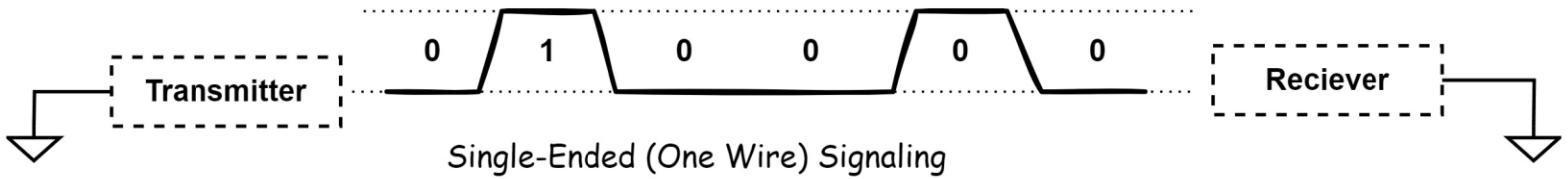


Synchronous and Asynchronous Data Transfer

- Data transfer in digital communication can be **synchronous** or **asynchronous**.
- In synchronous data transfer, the transmitter and receiver are synchronized by a common clock signal.
- This synchronization signal can be sent along with data.
- The asynchronous data transfer does not need such a synchronization signal. Instead, start and stop bits for each data packet are sent. However, the transmitter and receiver should know and agree upon the transfer speed beforehand.

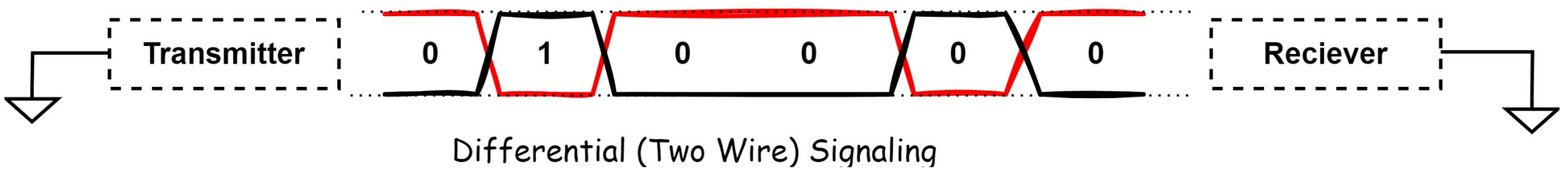
Signal Representation and Line Formations

- We introduced how a bit is represented in voltage levels (HIGH and LOW).
- In serial communication, there are two different line formations called **single-ended** and **differential** to transfer data.
- **Single-ended** signaling is the simplest method of transmitting digital data. One wire carries a varying (0s and 1s) that represents data
- The transferred signal is represented as a changing voltage value with respect to ground in the single-ended line



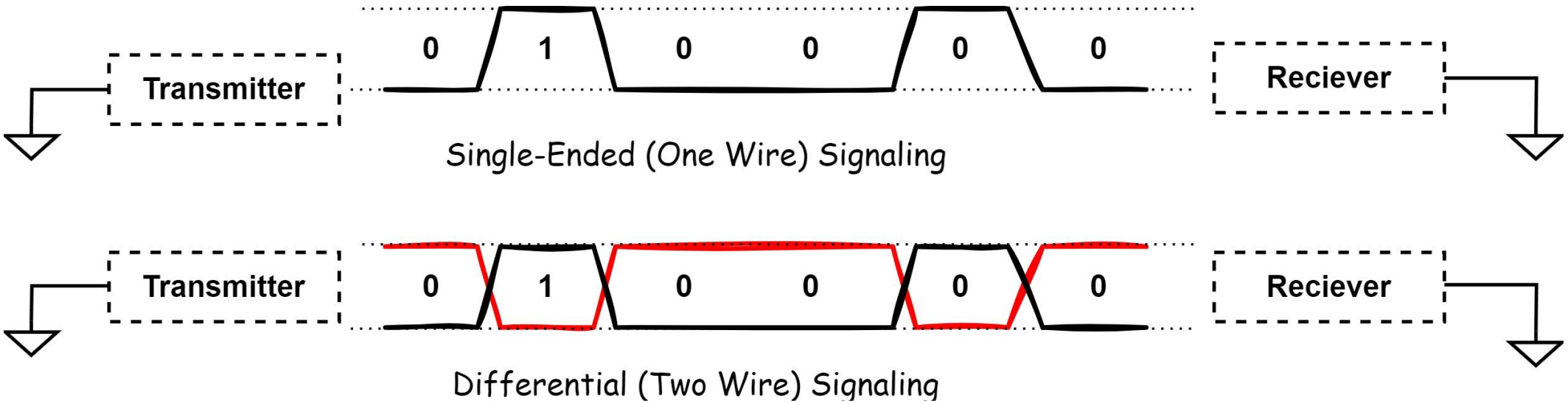
Signal Representation and Line Formations

- **Differential signaling**, employs two complementary voltage signals in order to transmit one information signal.
- One information signal requires a pair of conductors; one carries the signal and the other carries the inverted signal.
- The transferred signal is represented by the difference of two opposite voltage values in these lines



Differential (Two Wire) Signaling

Signal Representation and Line Formations



Single-Ended Vs Differential

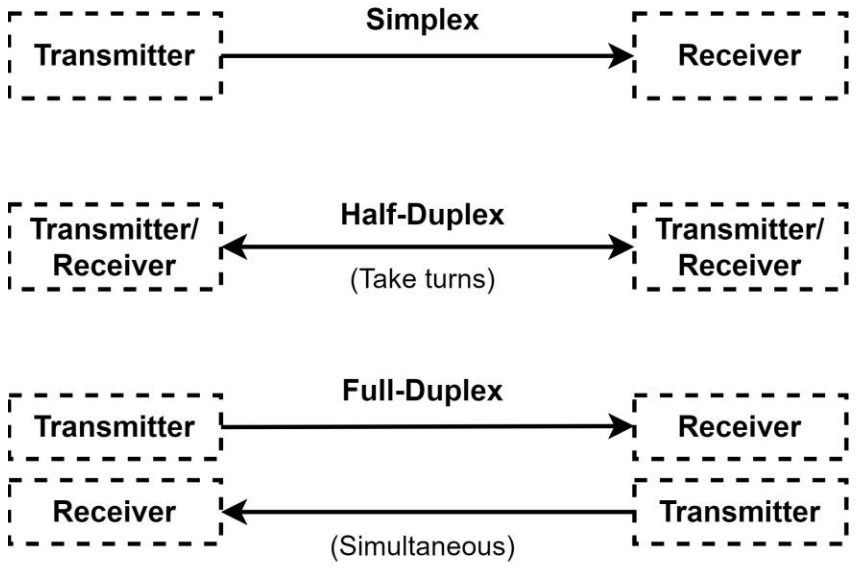
- The main advantage of single-ended over differential signaling is that fewer wires are needed to transmit multiple signals.
- If there are n signals, then there are n+1 wires, one for each signal and one for ground, while differential signaling uses at least 2n wires.
- The disadvantage of the single-ended line is that the transferred signal is affected by noise and line length in long-range

Single-Ended Vs Differential

- The advantage of the differential line usage is that it is not affected much by noise or range as in single-ended line.
- Hence, the differential line is used in long-range data transfer most of the time.
- Besides, the voltage levels used in operation exceed the ones available in the GPIO pins of the STM32F4 microcontroller. Hence, extra circuitry may be needed to use the differential line signal transfer.

Simplex, Half, and Full Duplex Communication

- There are three communication modes as simplex, half, and full duplex to handle communication direction preferences
- The **simplex mode** has one-way (unidirectional) communication.
- In **half duplex mode**, two-way (bidirectional) communication is possible. However, only one-way communication can be done at once.
- In **full duplex mode**, two-way (bidirectional) communication is possible. Moreover, the two-way communication can be done at the same time



Master and Slave Modes

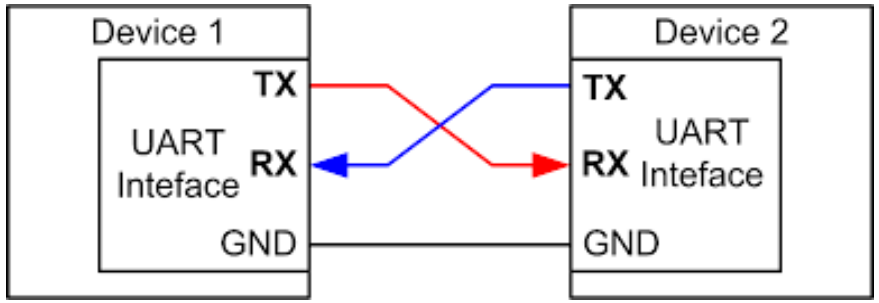
- During data transfer between two or more devices, one can act as master and the other as slave
- The master device generally starts the communication, asks for information from slave devices, and controls them
- the master device is the one generating the clock signal in synchronous communication
- The slave device is the one responding to the master or acting in accordance with the master device commands

Baud Rate

- The **baud rate** indicates how many bits are transmitted in 1 second for digital systems having logic levels 0 and 1
- This also corresponds to the **bit rate** definition
- It indicates the speed of data transfer
- the higher the baud rate, the faster the communication

UART Protocol

- **UART** stands for **Universal Asynchronous Receiver /Transmitter** and defines a protocol, or set of rules, for exchanging serial data between two devices
- UART is very simple and only uses two wires between transmitter and receiver to transmit and receive in both directions
- There is also **Universal Synchronous/Asynchronous Receiver/Transmitter (USART)**
- **USART** works in synchronous mode with an extra clock pin

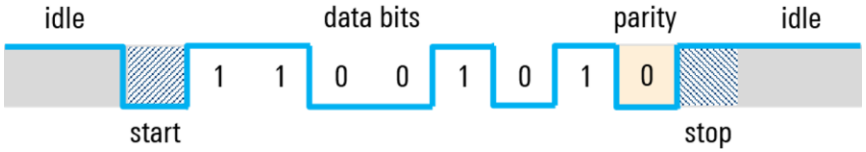


UART Protocol

- UART is generally preferred when fast data transfer speed is not needed. Hence, it is most of the times used for debugging and programming a microcontroller.
- Besides, it is used to communicate two microcontrollers.
- If two microcontrollers on different boards need to communicate with UART, a physical layer should be available such as RS232 or RS485.
- These will eliminate noise and power loss due to long-range signal transfer.

UART Working Principles

- Data transfer in UART can be done in frames.
- Each frame is composed of the start bit and stop bit, data bits , and an optional parity bit.



- There can be 5 to 9 user data bits, although 7 or 8 bits is most common. These data bits are usually transmitted with the least significant bit first.
- Start and stop bits are available in each frame.
- However, the **parity bit** can be used if needed. UART uses NRZ data encoding.
- In general, UART uses the TTL voltage levels to represent signals.

UART Working Principles

Parity bit

- A UART frame can also contain an optional parity bit that can be used for error detection.
- This bit is inserted between the end of the data bits and the stop bit.
- The value of the parity bit depends on the type of parity being used (even or odd):
  - In even parity, the bits whose value is 1 are counted. If that count is odd, the parity bit value is set to 1
  - In odd parity, the bits whose value is 1 are counted. If that count is even, the parity bit value is set to 1

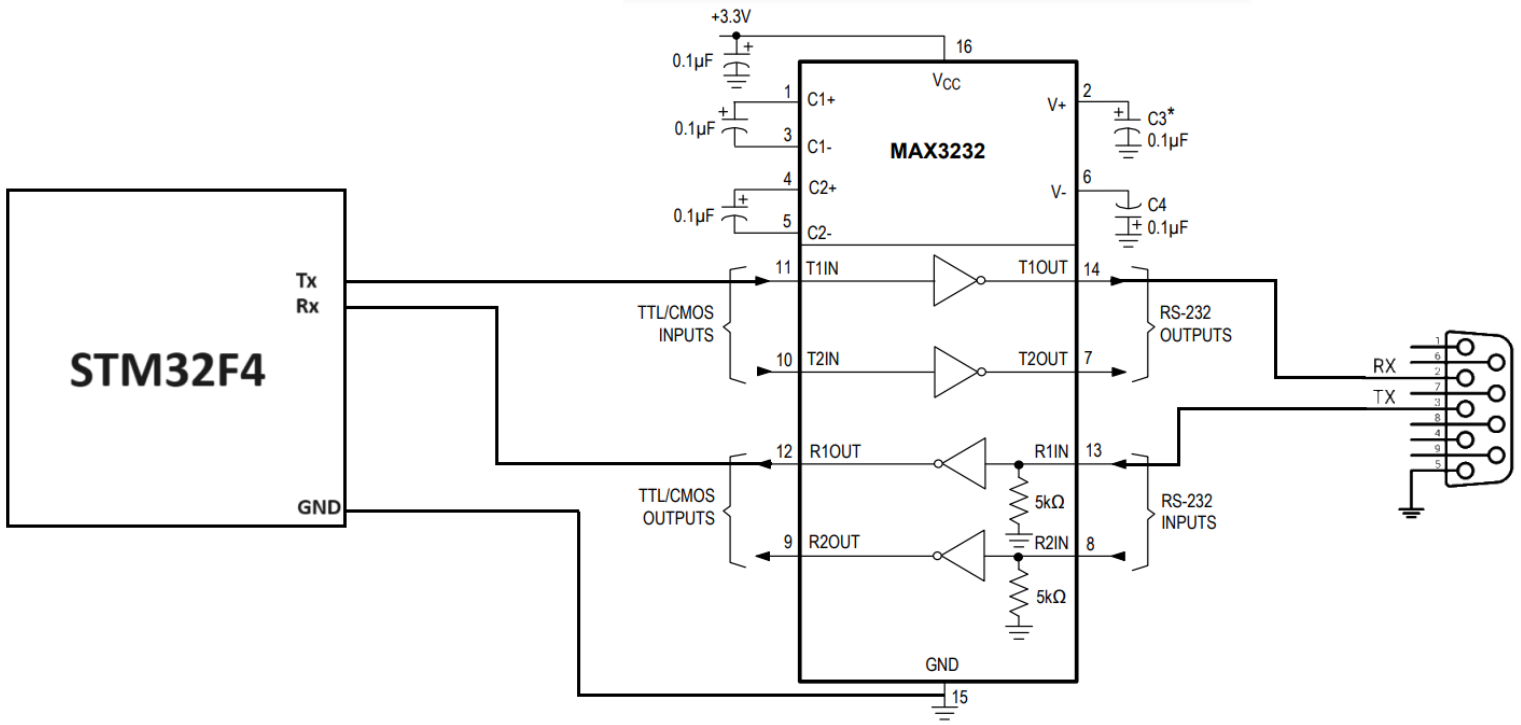
Data(7-bit)	Count of 1-bits	Even parity	Odd parity
0000000	0	0	1
1010001	3	1	0
1101001	4	0	1
1111111	7	1	0

UART Working Principles

- **UART** can work in either **full duplex, half duplex, or simplex modes**.
- In the **full duplex** mode, there are two pins called Rx (receive) and Tx (transmit).
- In the **half duplex** mode, the Tx pin is used for both as Tx and Rx.
- In the simplex mode, Tx pin of one device is connected to Rx pin of the other device.
- The remaining pins are not connected.
- The master and slave definitions do not apply to UART.
- The most common UART baud rates in use today are 4800, 9600, 19.2K, 57.6K, and 115.2K

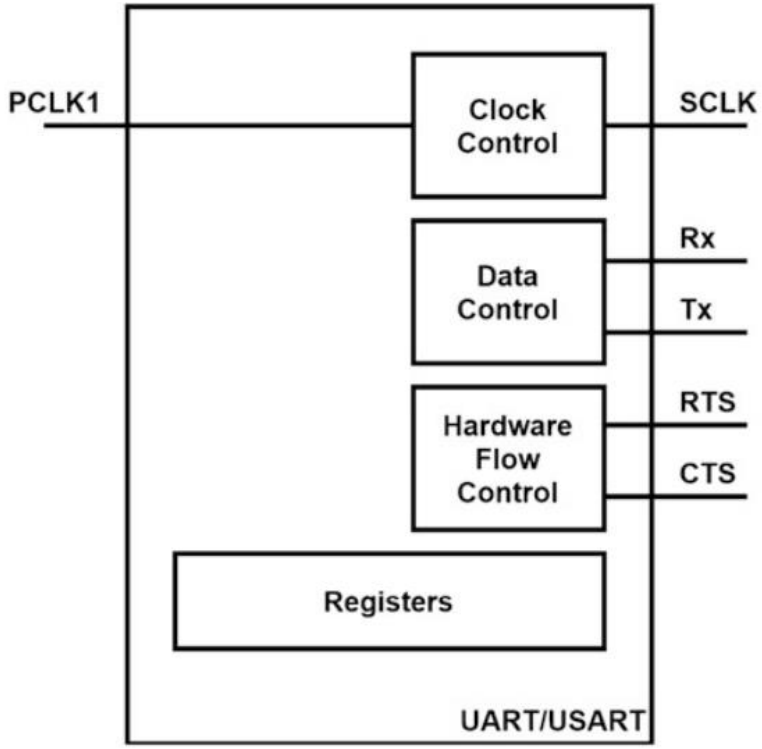
UART Working Principles

- If two devices need to communicate with UART, a physical layer should be available such as RS232 or RS485.
- These will eliminate noise and power loss due to long-range signal transfer



UART Modules in the STM32F4 Microcontroller

- The STM32F4 microcontroller has four UART modules called UART4, UART5, UART7, and UART8.
- It also has four USART modules USART1, USART2, USART3, and USART6. These modules can be used in **UART mode**.
- The clock pin (SCLK) and HW flow control pins (RTS and CTS) will not be used when UART is active



UART Modules in the STM32F4 Microcontroller

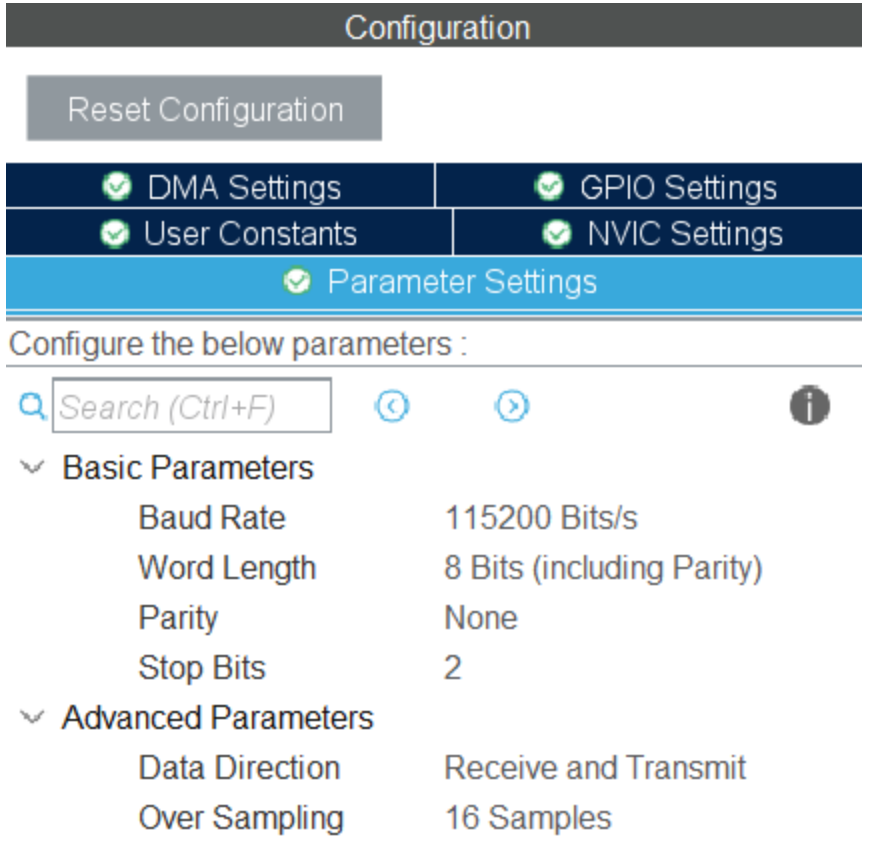
- The USART1 and USART6 can reach baud rates up to 11.25 Mbits/s.
- The other modules can reach a maximum baud rate of 5.625 Mbits/s.
- All supported baud rate values for UART and USART modules are given in the datasheet

Setup UART via CubeIDE

- **Create a New Project:**
  - Open CubeIDE and create a new STM32 project.
  - Select the appropriate STM32 microcontroller model for your project.
- **Configure System Clock:**
  - In the "Pinout & Configuration" tab, configure the system clock settings. This is essential for the proper functioning of the UART.
- **Configure UART Pins:**
  - In the "Pinout & Configuration" tab, configure the GPIO pins that are connected to Rx and Tx pins
  - Map these pins to the **UART** peripheral.
- **Select UART Mode:**
  - Under the “**Connectivity**” section, select the “Asynchronous” option for UART mode

Setup UART via CubeIDE – Cont.

- **Configure UART Settings:**
  - In the "Parameter Settings" tab, configure the UART parameters such as baud rate, word length, etc.
- **Configure UART working mode:**
  - The UART module can operate in two modes as blocking (polling) or nonblocking (interrupt).
  - If an interrupt is not required, then this step can be skipped. Otherwise, the UART interrupt should be enabled from the NVIC tab in the configuration menu



Setup UART via CubeIDE

- **Write UART Code:**
  - Open the generated code files, typically in the "Src" folder.
  - Locate the file named **`main.c`** or similar, and add your UART code.
  - There are two functions for blocking (polling) mode operation of the UART module.
    - HAL_UART_Transmit can be used to send data.
    - HAL_UART_Receive can be used to receive data in blocking mode.

```
HAL_UART_Transmit(UART_HandleTypeDef *huart , uint8_t *pData
,
uint16_t Size , uint32_t Timeout)
/*
huart: pointer to the UART_HandleTypeDef struct
pData: pointer to data buffer
Size: size of data elements to be sent
Timeout: timeout duration
*/
HAL_UART_Receive(UART_HandleTypeDef *huart , uint8_t *pData
,
uint16_t Size , uint32_t Timeout)
/*
Size: size of data elements to be received
*/
```

Setup UART via CubeIDE

- **Write UART Code – cont.**
  - There are two functions for non-blocking (interrupt) mode operation of UART module:
    - HAL_UART_Transmit_IT can be used to send data in nonblocking mode.
    - HAL_UART_Receive_IT can be used to receive data in non-blocking mode.
    - If the UART interrupt has been enabled and data is sent or received, then the following callback function is called:
      - HAL_UART_TxCpltCallback or
      - HAL_UART_RxCpltCallback is called.
    - The user can define his function, and the predefined weak functions are ignored

```
HAL_UART_Transmit_IT(UART_HandleTypeDef *huart , uint8_t *pData ,
uint16_t Size)
/*
huart: pointer to the UART_HandleTypeDef struct
pData: pointer to data buffer
Size: size of data elements to be sent
*/
HAL_UART_Receive_IT(UART_HandleTypeDef *huart , uint8_t *pData ,
uint16_t Size)
/*
Size: size of data elements to be received
*/
void HAL_UART_TxCpltCallback (UART_HandleTypeDef *huart)
void HAL_UART_RxCpltCallback (UART_HandleTypeDef *huart)
```

UART Example- Blocking mode

- In this example, the UART module is configured in blocking mode
  - A string and counter value is sent to PC from the STM32F4 microcontroller via UART1.
  - The green LED connected to pin PA13 of the STM32F4 board is toggled once data is transmitted.
  - On the PC side, we should use an application, such as PuTTY or Tera Term, to receive data sent from the STM32F4 microcontroller

```
/* USER CODE BEGIN Includes */
#include "stdio.h"
/* USER CODE END Includes */
/* USER CODE BEGIN PV */
uint8_t serData[] = " STM32F4 UART PC Example\r\n";
uint8_t decNum[6];
uint16_t counter = 0;
/* USER CODE END PV */
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    sprintf(decNum , "%d", counter++);
    HAL_UART_Transmit(&huart1 , decNum ,
(uint16_t)sizeof(decNum),
    1000);
    HAL_UART_Transmit(&huart1 , serData ,
(uint16_t)sizeof(serData),
    1000);
    HAL_GPIO_TogglePin(GPIOA , GPIO_PIN_13);
    HAL_Delay(2000);
/* USER CODE END WHILE */
/* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

UART Example – Non blocking mode

- In this example, the UART module is configured in blocking mode
  - This time when the character ’g’ is received, the green LED connected to pin PA13 of the STM32F4 is toggled.
  - When the character ’r’ is received, the red LED connected to pin PA14 of the STM32F4 is toggled.
  - To test the code, send the characters ’r’ and ’g’ from the host PC to the STM32F4 microcontroller. Hence, we can observe that the red and green LEDs toggle.

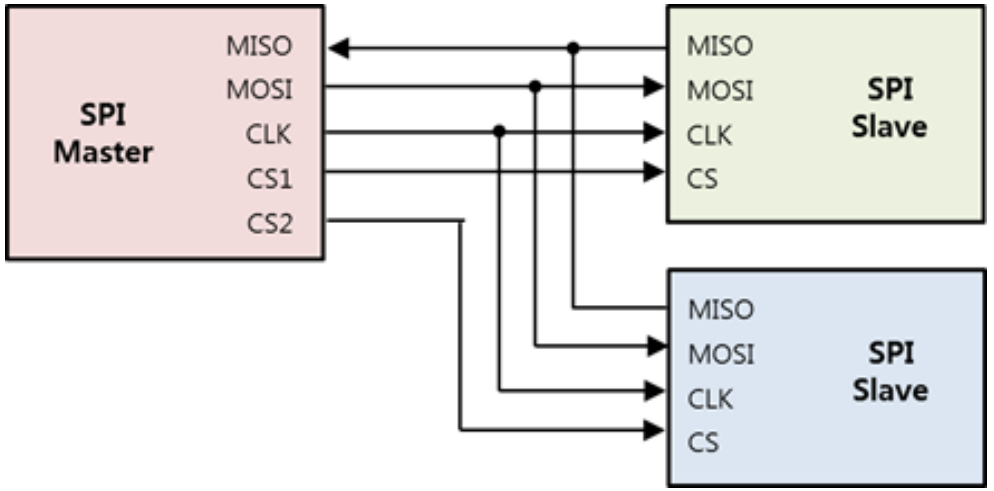
```
/* USER CODE BEGIN PV */
uint8_t serRxData;
/* USER CODE END PV */
/* USER CODE BEGIN 0 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef
*huart)
{
if (serRxData == ’g’)
    HAL_GPIO_TogglePin(GPIOA , GPIO_PIN_13);
    else if (serRxData == ’r’)
    HAL_GPIO_TogglePin(GPIOA , GPIO_PIN_14);
    __NOP();
    HAL_UART_Receive_IT(&huart1 , &serRxData , 1);
}
/* USER CODE END 0 */
/* USER CODE BEGIN 2 */
HAL_UART_Receive_IT(&huart1 , &serRxData , 1);
/* USER CODE END 2 */
```

SPI Protocol

- The second digital communication type we will be introducing is **the serial peripheral interface (SPI)**
- SPI has an 8 or 16-bit frame consisting of only data. Hence, no start, stop, or parity bits are used in the frame.
- SPI can work in half or full duplex mode. Moreover, if we do not receive data from the slave in the full-duplex mode it works in simplex mode.
- SPI can work in either master or slave mode based on the application.
- The master device generates the clock signal.

SPI Protocol

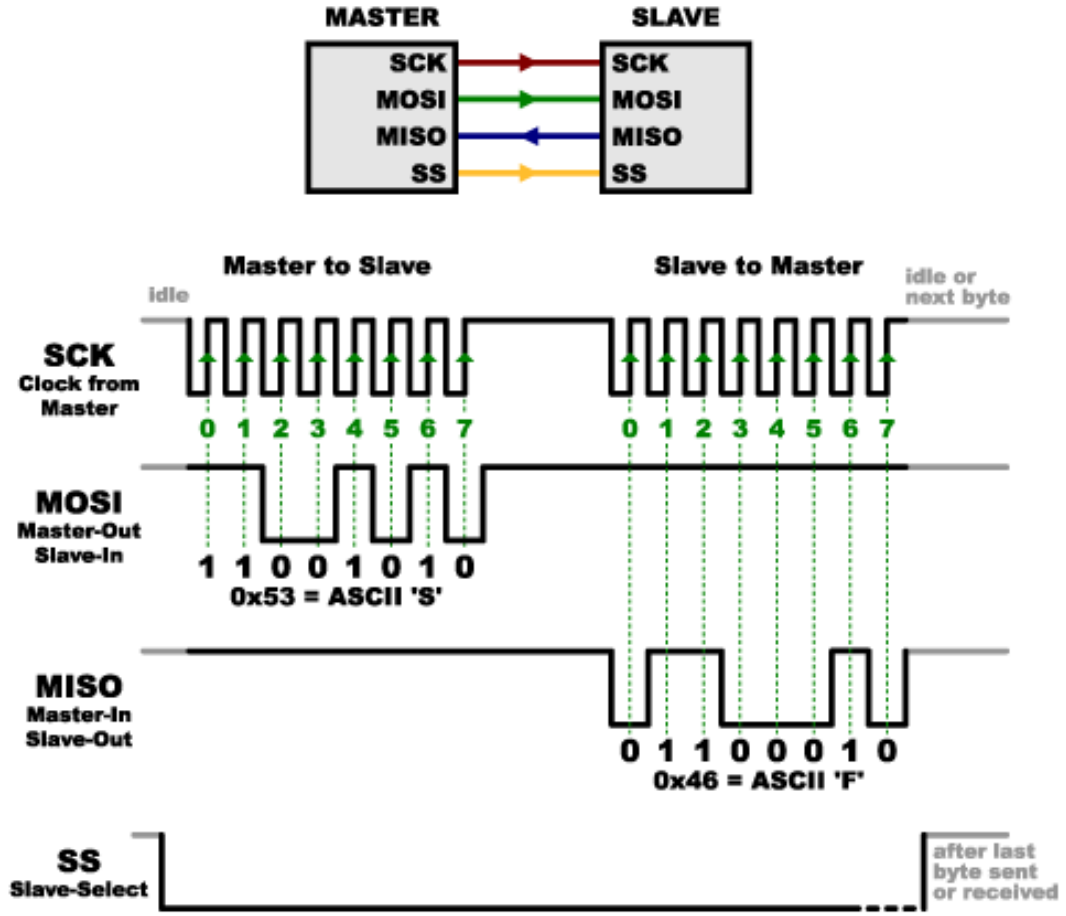
- More than one slave device can be connected to the communication line.
- In this setting, the SPI master **chip select (CS)** pins should be used to select which slave to be communicated to.
- In this setup, one dedicated CS pin should be used for each slave device.



SPI Protocol

- There are four pins in operation
  - (MISO) as master-in/ slave-out
  - (MOSI) master-out/ slave-in
  - (SCLK) serial clock
  - (NSS) Active-low slave select or (CS) user-specific pin
- When the device is in **master mode**, MISO works for data input, MOSI for data output, SCLK as clock output, and NSS as slave select output (in active low mode)
- When the device is in **slave mode**, MISO works for data output, MOSI for data input, SCLK as clock input, and NSS as slave select input (in active low mode).

SPI Protocol

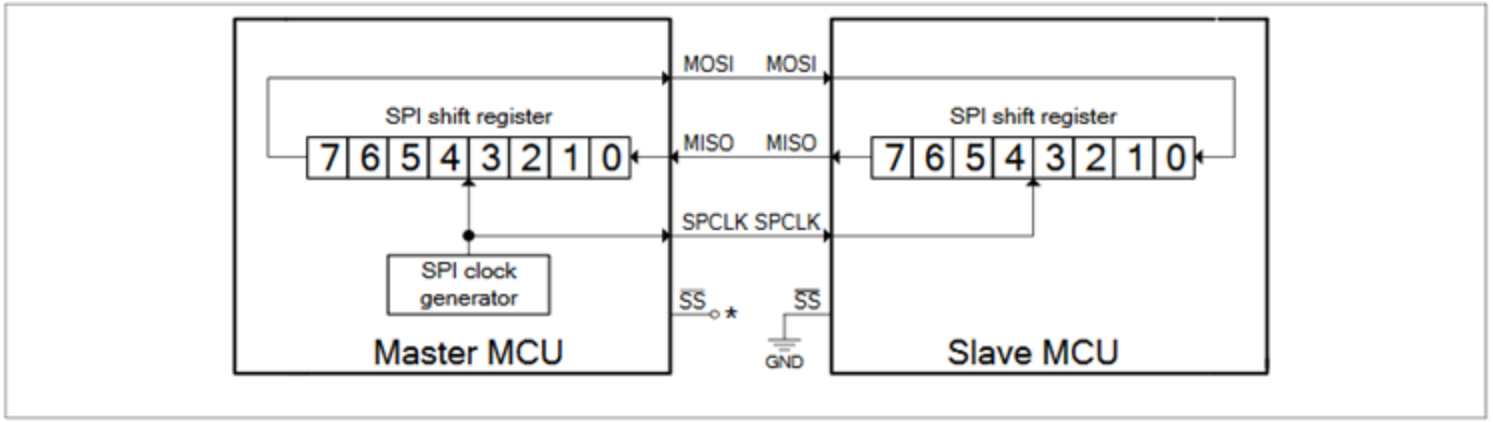


SPI Working Principles

- Data transfer between the master and slave devices occurs in serial mode between MOSI and MISO pins.
- The most significant bit is sent first in data transfer. Moreover, communication is initiated by the master device.
- When the master device transmits data to a slave device via the MOSI pin, the slave device responds via the MISO pin. This implies full duplex communication with both data out and data in, synchronized with the same clock signal (which is provided by the master device via the SCK pin).

SPI Working Principles

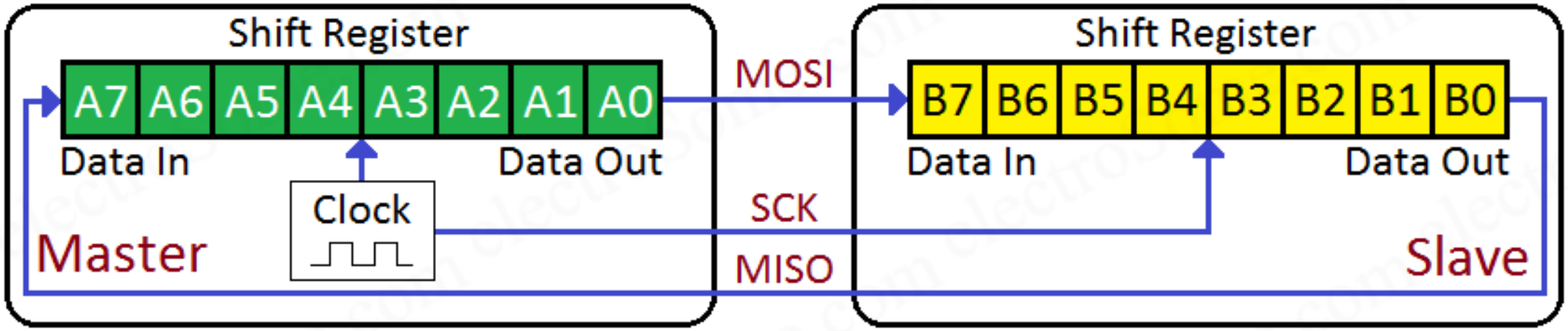
- SPI operation is based on shift registers (8-bit or 16-bit).
- Each devices (master and slave) uses shift registers to shift data in and out of the system.



SPI Working Principles

- Shift registers in the master and slave devices feed 1 bit to their output at **each clock pulse** generated by the master device.
- Since the master output pin is connected to the slave input pin, the bit fed from the master device shift register is received by the shift register in the slave device.
- Likewise, the master input is connected to the slave output. Hence, the bit fed from the shift register of the slave device is fed to the shift register in the master device.
- As the master device generates eight clock pulses, 1 data byte is transferred between the master and slave devices.

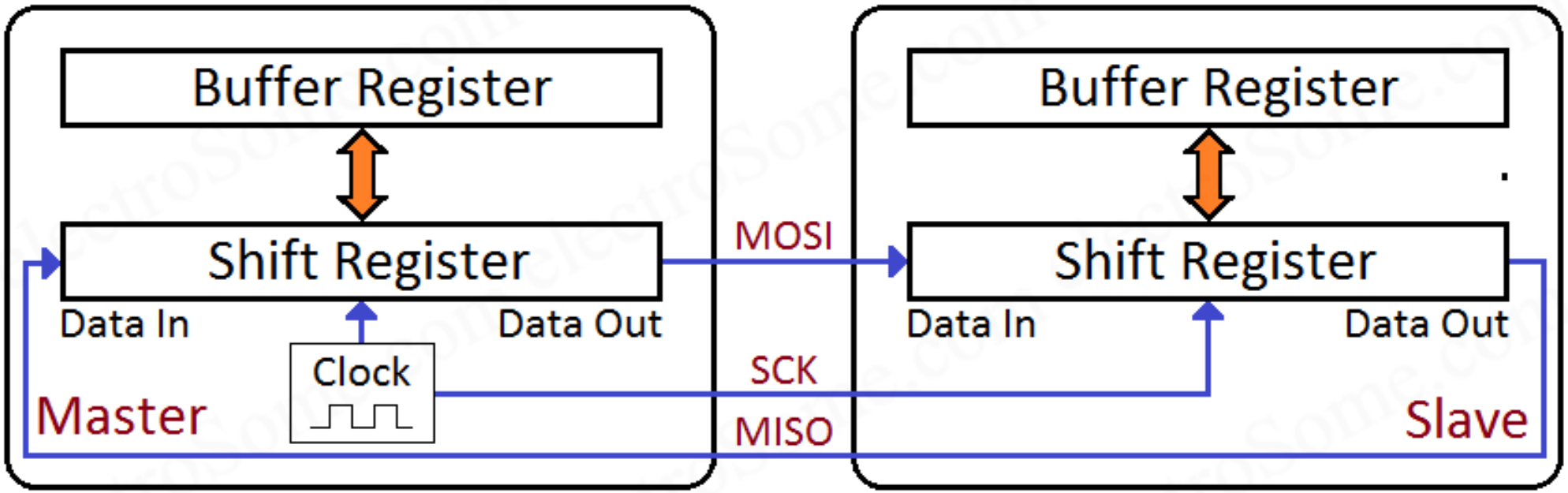
SPI Modules in the STM32F4 Microcontroller



SPI Working Principles

- Usually shift register won't be directly accessible
- If we need to transmit data, we will write it to the buffer register.
- If the master or slave device will transfer new data, it is first written to the Tx FIFO buffer.
- This enables the device to automatically write to the shift register when it is free, then the transmission will start.
- Similarly, data is received in the shift register is automatically transferred to a buffer register once the reception is complete.
- Using buffer registers avoid all glitches that can happen if we try to read or write to the shift register directly while transmission is taking place

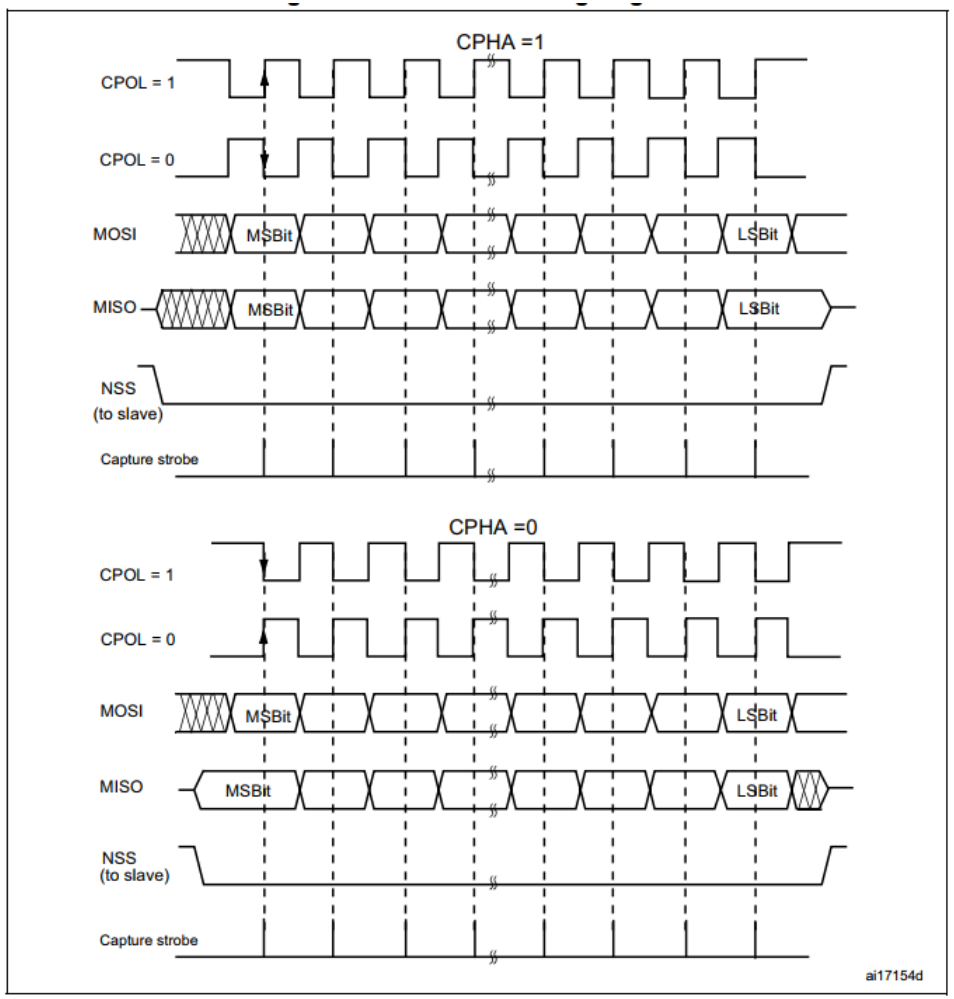
SPI Working Principles



SPI Working Principles

- In SPI, the main can select the clock polarity and clock phase.
- The **CPOL** bit sets the **polarity** of the clock signal during the **idle state**.
- The **idle state** is defined as the period when CS is high and transitioning to low at the start of the transmission and when CS is low and transitioning to high at the end of the transmission.
- The **CPHA** bit selects the **clock phase**. Depending on the CPHA bit, the rising or falling clock edge is used to sample and/or shift the data.

SPI Working Principles



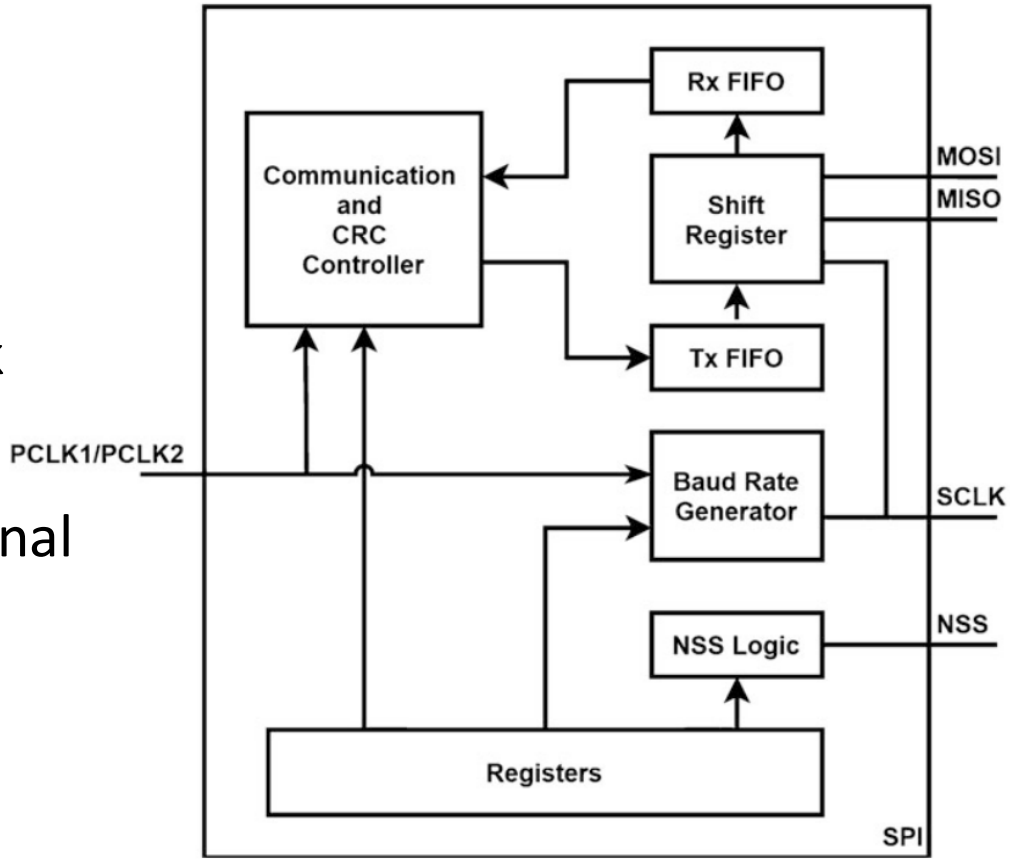
SPI Working Principles

- The combinations of polarity and phases are referred to by these "SPI mode" numbers with CPOL as the high order bit and CPHA as the low order bit:

SPI Mode	Clock polarity (CPOL)	Clock phase (CPHA)	Data is sampled on	Data is shifted out on
0	0	0	Rising	Falling
1	0	1	Falling	Rising
2	1	0	Falling	Rising
3	1	1	Rising	Falling

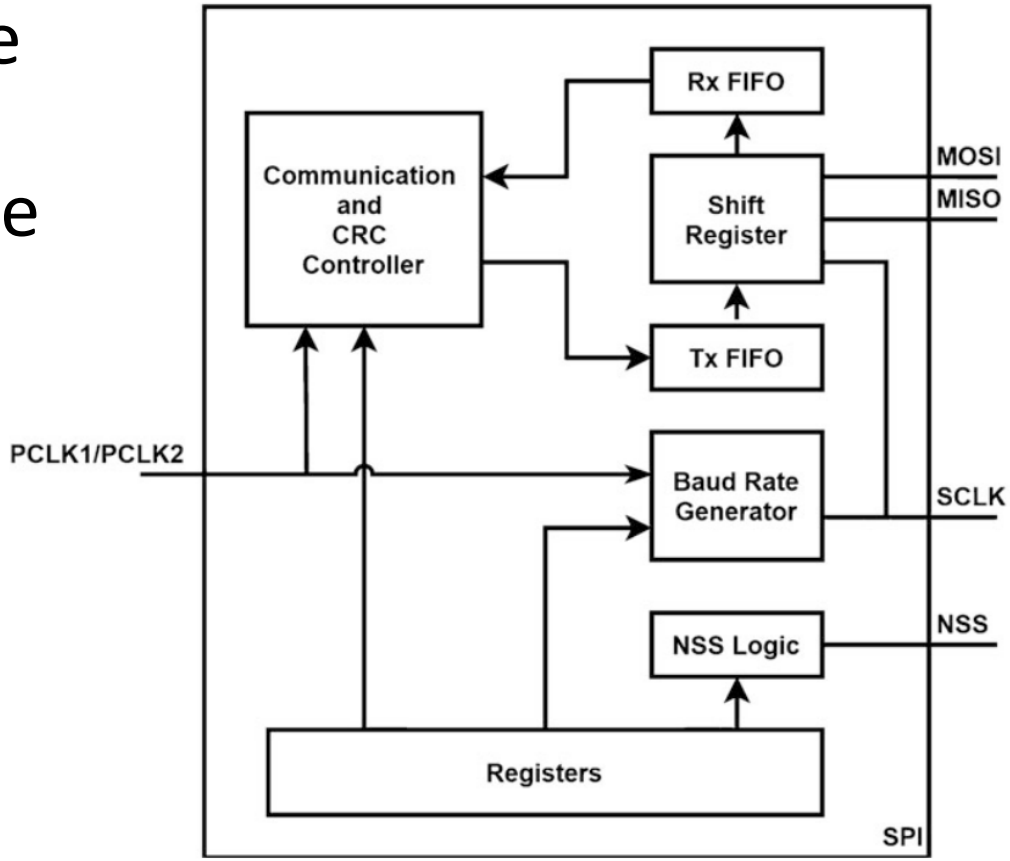
SPI Modules in the STM32F4 Microcontroller

- The STM32F4 microcontroller has up to six SPI modules as SPI1, SPI2, SPI3, SPI4, SPI5, and SPI6.
- Each module can work either in master or slave mode.
- They can also work in half or full duplex modes.
- SPI modules use the peripheral clock as the clock signal.
- The master device has its own clock prescaler (inside **baud rate generator**) so that the clock signal can be divided by 2, 4, 8, 16, 32, 64, 128, or 256.
- The resulting clock signal will decide on the baud rate



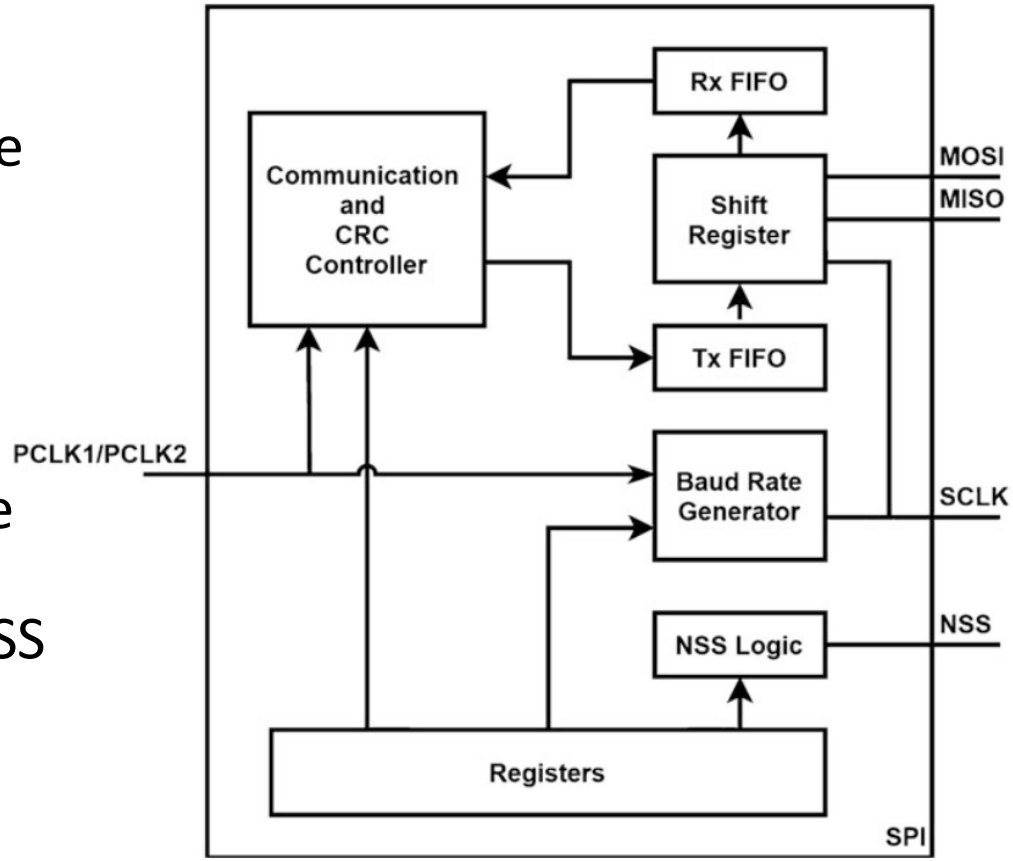
SPI Modules in the STM32F4 Microcontroller

- SPI1, SPI4, SPI5, and SPI6 modules use the APB2 peripheral clock (PCLK2). Hence, the maximum baud rate can be 45 Mbits/s.
- The SPI2 and SPI3 modules use the APB1 peripheral clock (PCLK1). This may result in the maximum baud rate of 22.5 Mbits/s



SPI Modules in the STM32F4 Microcontroller

- The shift register in the module has an essential role in data transfer.
- The communication and CRC controller blocks are responsible for reading and writing data coming from software to FIFO buffer modules.
- Moreover, they are responsible for interrupt generation.
- If the SPI module is selected as master, the NSS logic module becomes responsible for setting the output and selecting the slave device.
- If the SPI module is selected as slave, then the NSS logic module activates the read/write operation.

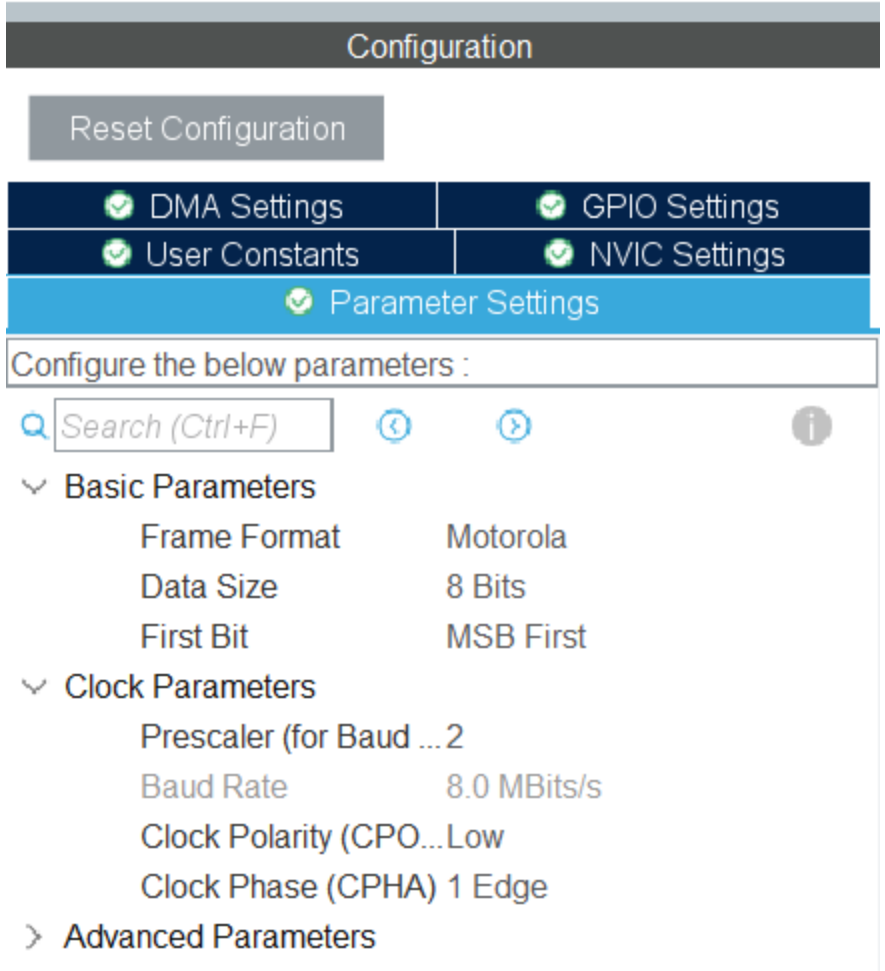


Setup SPI via CubeIDE

- **Create a New Project:**
  - Open CubeIDE and create a new STM32 project.
  - Select the appropriate STM32 microcontroller model for your project.
- **Configure System Clock:**
  - In the "Pinout & Configuration" tab, configure the system clock settings. This is essential for the proper functioning of the SPI.
- **Configure UART Pins:**
  - In the "Pinout & Configuration" tab, configure the GPIO pins that are connected to MOSI, MISO, SCLK and CS pins
  - Map these pins to the **SPI** peripheral.
- **Select UART Mode:**
  - Under the “**Connectivity**” section, select the “mode” option for SPI (Full-Duplex master, full-duplex slave, half-duplex master, etc.)

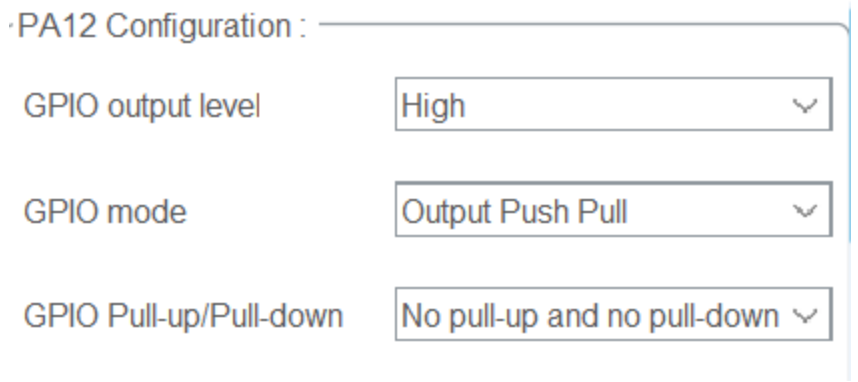
Setup SPI via CubeIDE – Cont.

- **Configure SPI Settings:**
  - In the "Parameter Settings" tab, configure the SPI parameters such as frame format, data size, etc.
- **Configure SPI working mode:**
  - The UART module can operate in two modes as blocking (polling) or nonblocking (interrupt).
  - If an interrupt is not required, then this step can be skipped. Otherwise, the UART interrupt should be enabled from the NVIC tab in the configuration menu



Setup SPI via CubeIDE – Cont.

- **Configure a GPIO Pin as a CS:**
  - In the "GPIO configuration" tab, configure the CS pin and select the "GPIO output level" to "High"
  - Whenever you want to send data using SPI, this pin should drive low.



Setup SPI via CubeIDE – Cont.

- **HAL functions in polling mode:**
  - The function HAL_SPI_Transmit can be used to send data.
  - The function HAL_SPI_Receive can be used to receive data in blocking mode.
- **HAL functions in interrupt mode:**
  - The function HAL_SPI_Transmit_IT can be used to send data.
  - The function HAL_SPI_Receive_IT can be used to receive data in non-blocking mode.
  - If the SPI interrupt is enabled, and data is sent or received, then the callback function HAL_SPI_TxCpltCallback or HAL_SPI_RxCpltCallback is called.

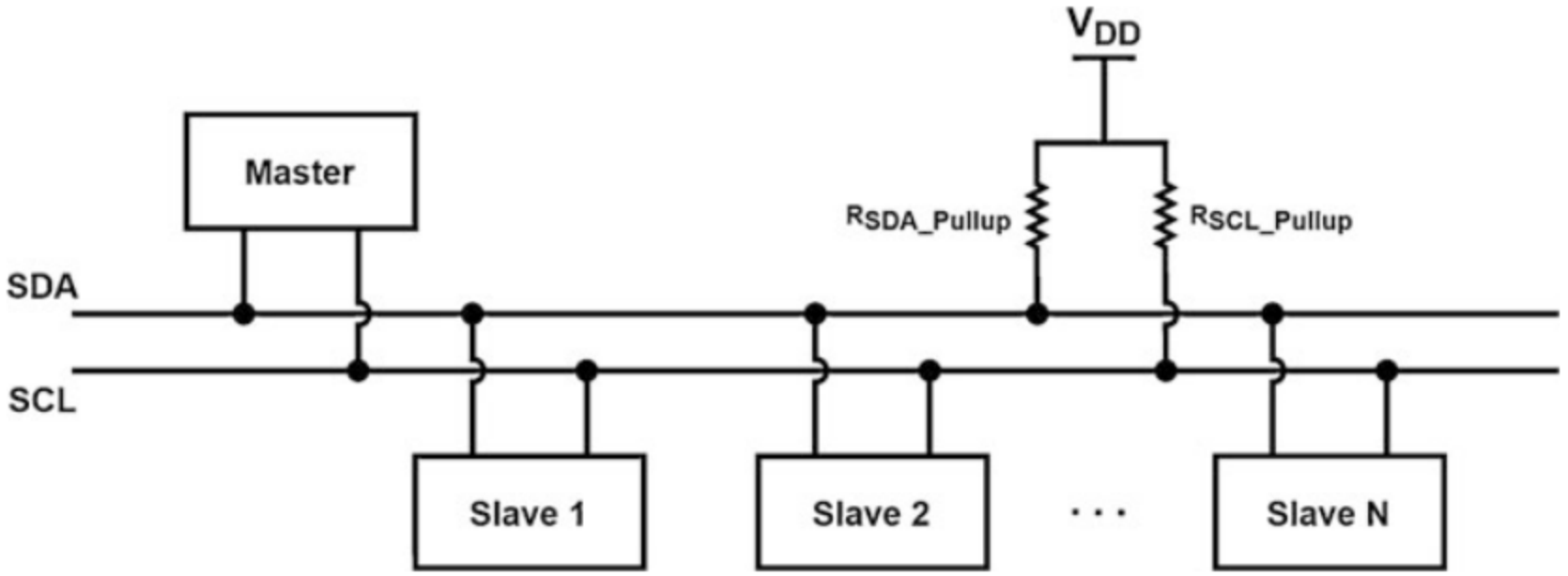
Inter-integrated Circuit (I2C)

- The third digital communication type we will be using is the inter-integrated circuit (I2C) data.
- I2C is generally used for communication between a microcontroller and one or more external devices.
- More specifically, I2C is suitable for setting an external device through its registers and reading data from it.
- It allows reaching a maximum of 128 such (with 7-bit address) in standard mode and 1024 devices (with 10-bit address) in extended mode.
- In these modes, I2C simplifies circuit connection and address management while reaching external devices.
- Primarily used for short distance data communication
- Two wires: serial clock (SCL) and serial data (SDA)

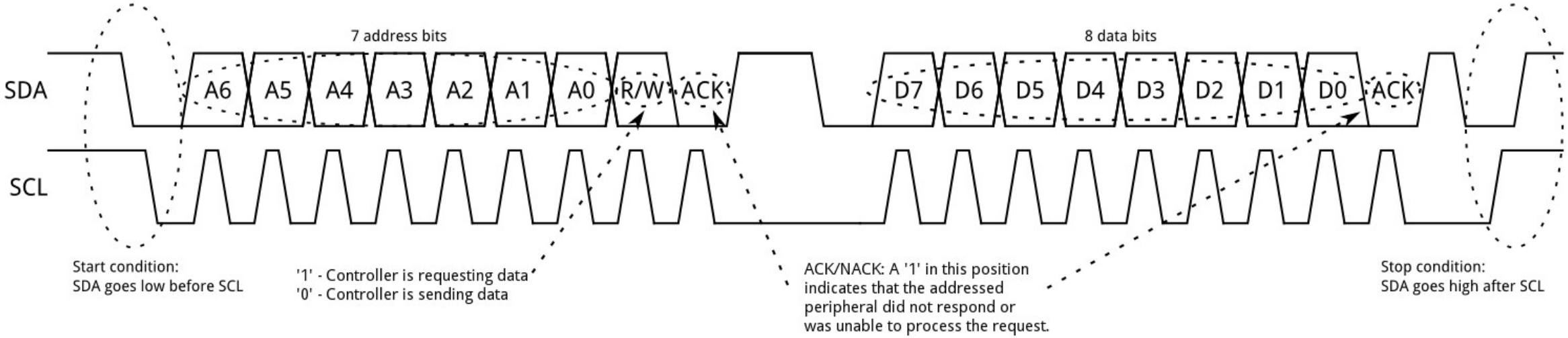
Inter-integrated Circuit (I2C)

- I2C has both the master and slave modes.
- Both master and slaves can send/ receive data
- I2C is a synchronous master-slave protocol, and it works in half-duplex mode
- The clock is generated by the master device.
- Moreover, the communication is always started by the master device. I2C has standard and fast data transfer modes
- I2C can run at different speeds

I2C basic topology



Overview of I2C frames

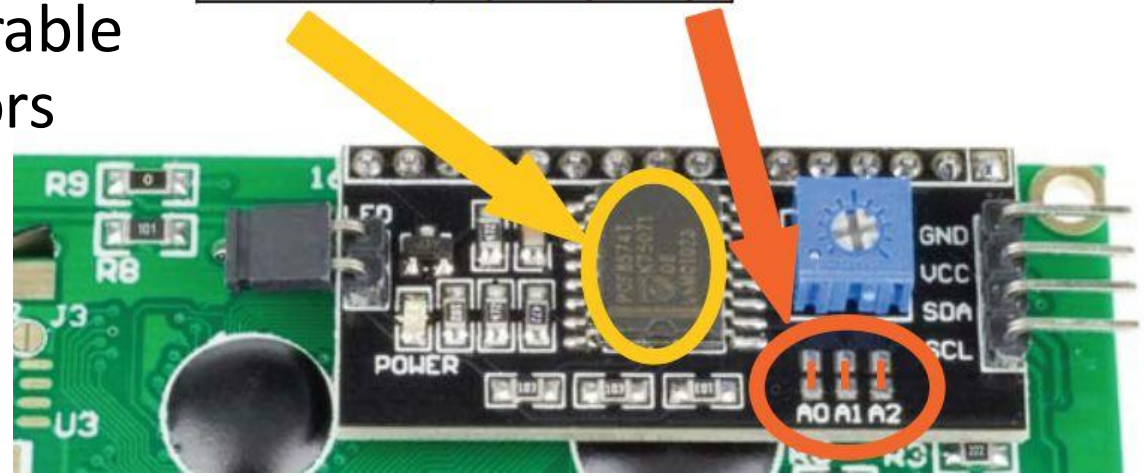


I2C working principle

- The master device sends the address of the slave device it needs to communicate with.
- Each I2C node on a bus must have a unique, fixed address (7-bit or 10-bit)
- Slave address may be hard-coded
- Slave address may be (partially) configurable via external jumpers or zero-ohm resistors

Addicore I2C Backpack Addresses

PCF8574AT	PCF8574T	Solder Jumpers		
I2C Address	I2C Address	A0	A1	A2
0x3F	0x27			
0x3E	0x26	bridged		
0x3D	0x25		bridged	
0x3C	0x24	bridged	bridged	
0x3B	0x23			bridged
0x3A	0x22	bridged		bridged
0x39	0x21		bridged	bridged
0x38	0x20	bridged	bridged	bridged

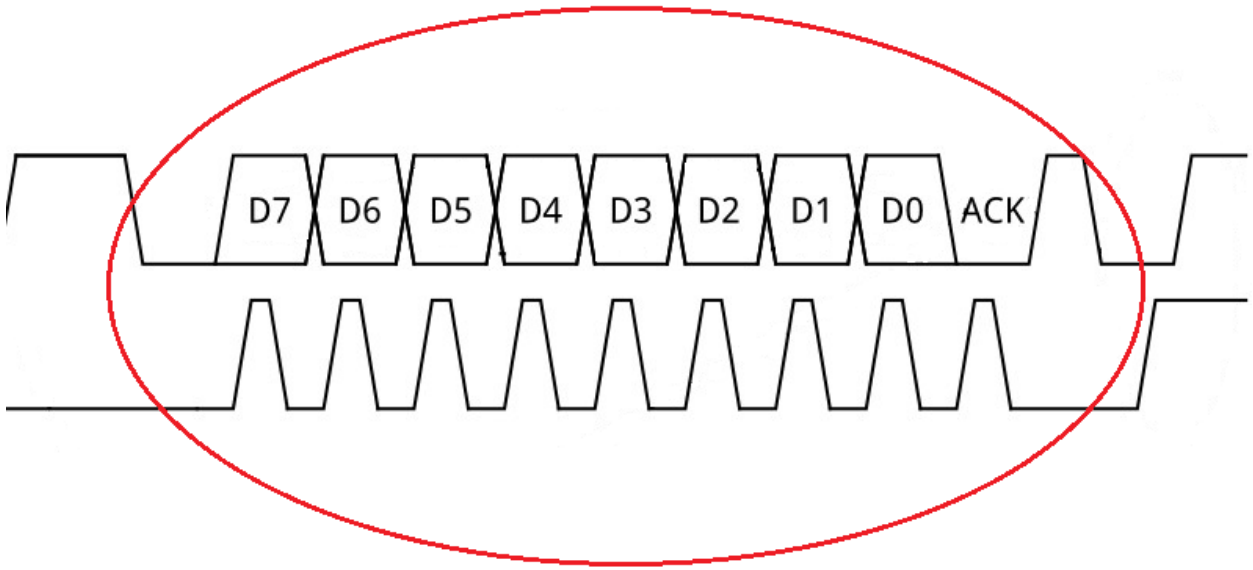


I2C working principle

- The master device can send or receive data to the slave. This is declared in the read/write (R/W) bit sent along the address.
- R/W bit is often interpreted and/or decoded as part of the address byte
- The slave device sends an acknowledge (ACK) bit each time a byte of data or after the slave address is received.
- ACK after a data byte(s) confirms receipt of data
- ACK after slave address confirms that:
  - A slave with that address is on the bus
  - The slave is ready to read/ write data (depending on R/W bit)

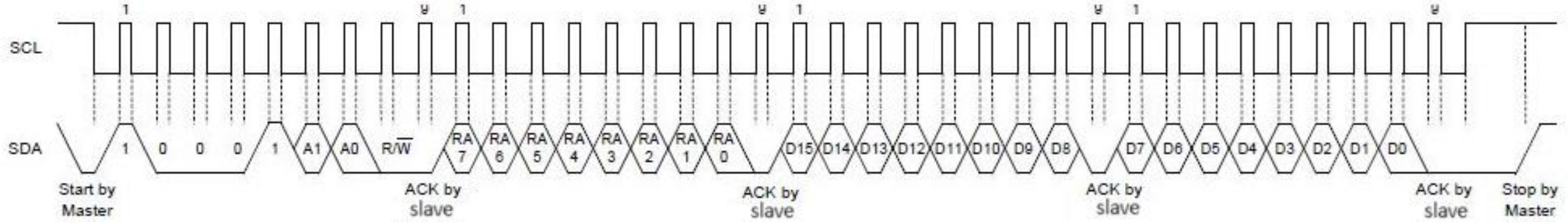
I2C working principle

- The data byte contains the information being transferred between the master and slave:
  - The data could be the content of a memory address or register in the slave device
- The data is always 8-bit long with MSB first.



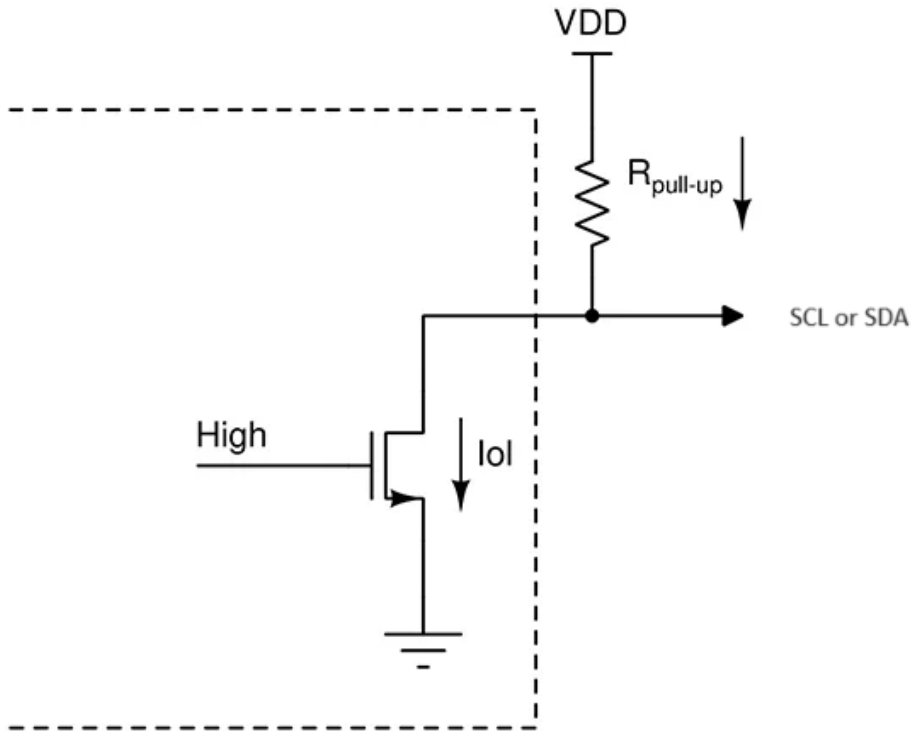
Multiple data bytes

- Multiple data bytes can be sent in one I2C frame (each byte is followed by ACK)
- Bytes could be all "data" or some may represent an internal address in the slave device



I2C working principle

- Each line (SDA and SCL) is connected to a voltage (VDD) via a pull-up resistor
- Typical pull-up resistor values are in the range of 1KΩ to 10 KΩ



I2C working principle

- I2C can operate at different bus speeds (referred as "modes")
- The STM32F4x family does not support high-speed mode

MODE	DATA RATE	NOTES
STANDARD MODE	UP TO 100kBPS	SUPPORTED BY STM32F4x
FAST MODE	UP TO 400Kbps	SUPPORTED BY STM32F4x
FAST MODE+	UP TO 1Mbps	SUPPORTED BY SOME STM32F4x MCUs
HIGH SPEED MODE	UP TO 3.4Mbps	SUPPORTED BY STM32F4x

Questions?