

PART-1

# Mastering Microcontroller with Embedded Driver Development

FastBit Embedded Brain Academy  
Check all online courses at  
[www.fastbitlab.com](http://www.fastbitlab.com)

# About FastBit EBA

FastBit Embedded Brain Academy is an online training wing of Bharati Software.

We leverage the power of internet to bring online courses at your fingertip in the domain of embedded systems and programming, microcontrollers, real-time operating systems, firmware development, Embedded Linux.

All our online video courses are hosted in Udemy E-learning platform which enables you to exercise 30 days no questions asked money back guarantee.

For more information please visit : [www.fastbitlab.com](http://www.fastbitlab.com)

Email : [contact@fastbitlab.com](mailto:contact@fastbitlab.com)

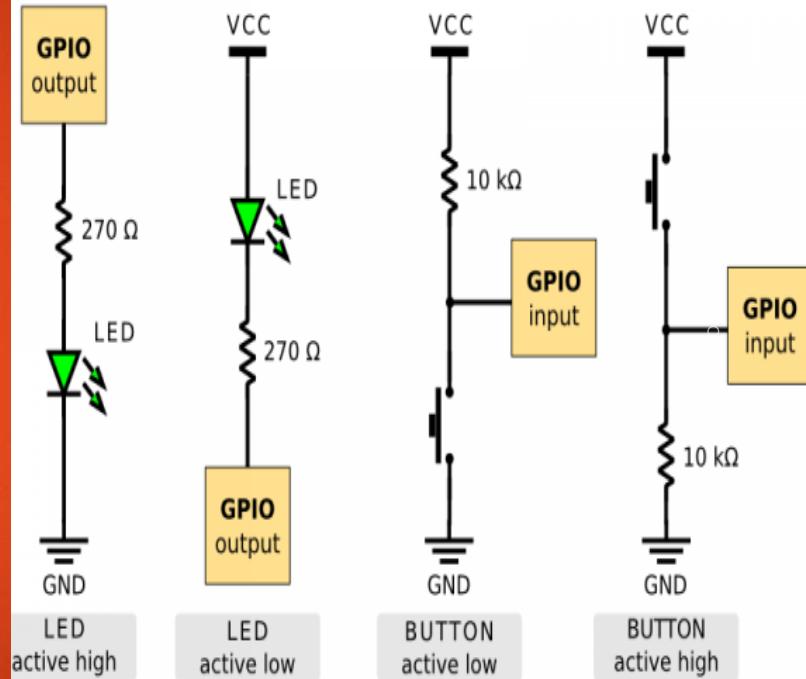
# GPIO Must Know Concepts !

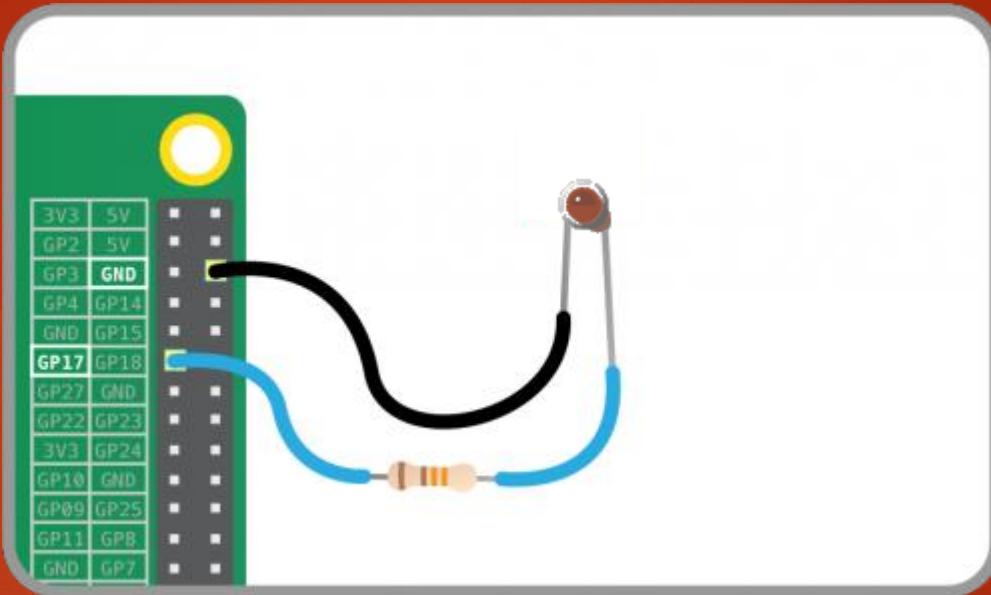
# Section Takeaways !

- ▶ This section has totally 7 lectures
- ▶ You will learn about behind the scene implementation about GPIO pin and GPIO port
- ▶ GPIO Input mode configurations like HIGH-Z , pull-up,pull-down state
- ▶ You will learn about different GPIO output Configurations like Open drain, pushpull , etc.
- ▶ You will learn about I/O power optimization

# GPIO Pin and GPIO Port

# General Purpose Input Output





Copyright © 2019 Bharati Software

# GPIOs typically used for

Reading digital signal  
Issuing interrupts

Generating triggers for  
external components

Waking up the processor  
and many

Let's begin with some of the **must know concepts** in GPIO. These concepts are **generic** and can be applied to any microcontroller you have !



Port 'A' having many *pins*



Port 'Soller Town' having many *boats*

Intel  
8051



Each port have  
8 i/o pins

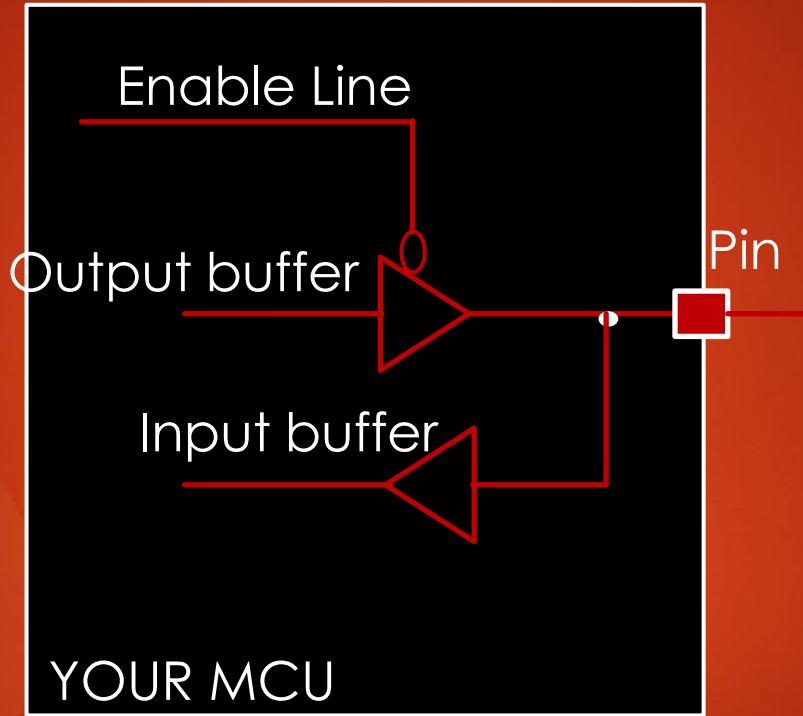
Each port have  
32 i/o pins

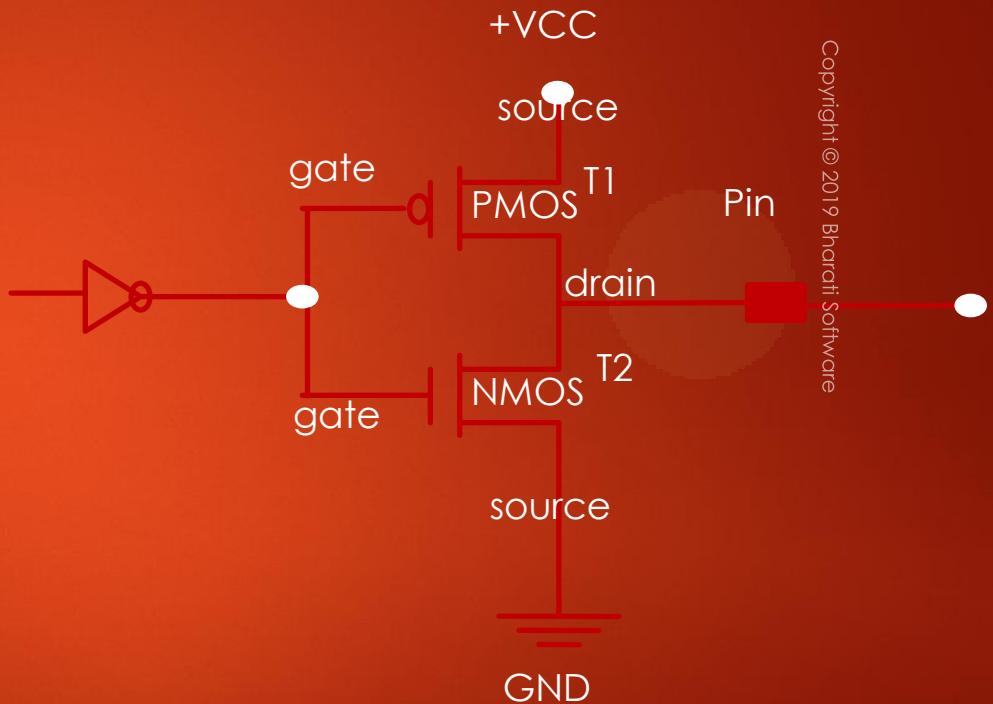
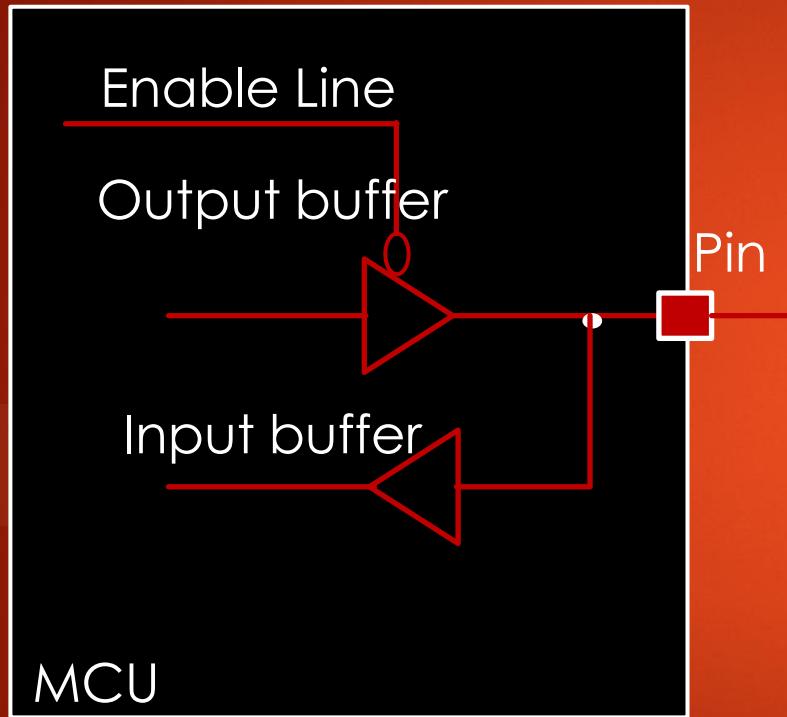
Each port have  
16 i/o pins

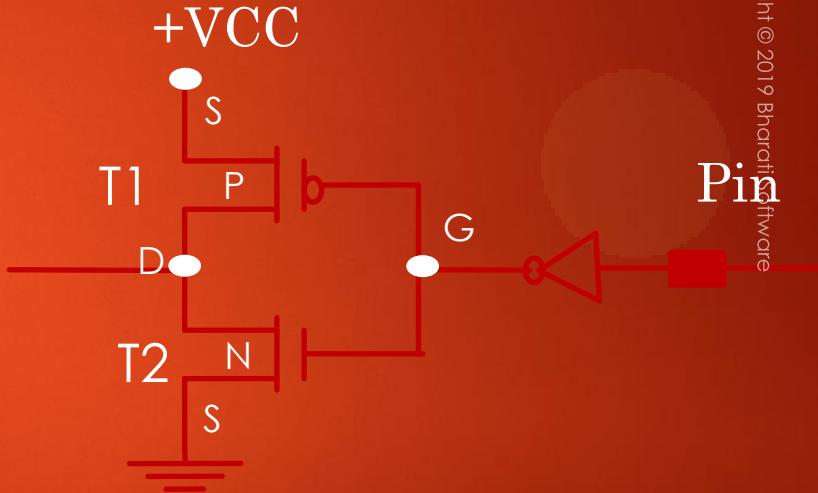
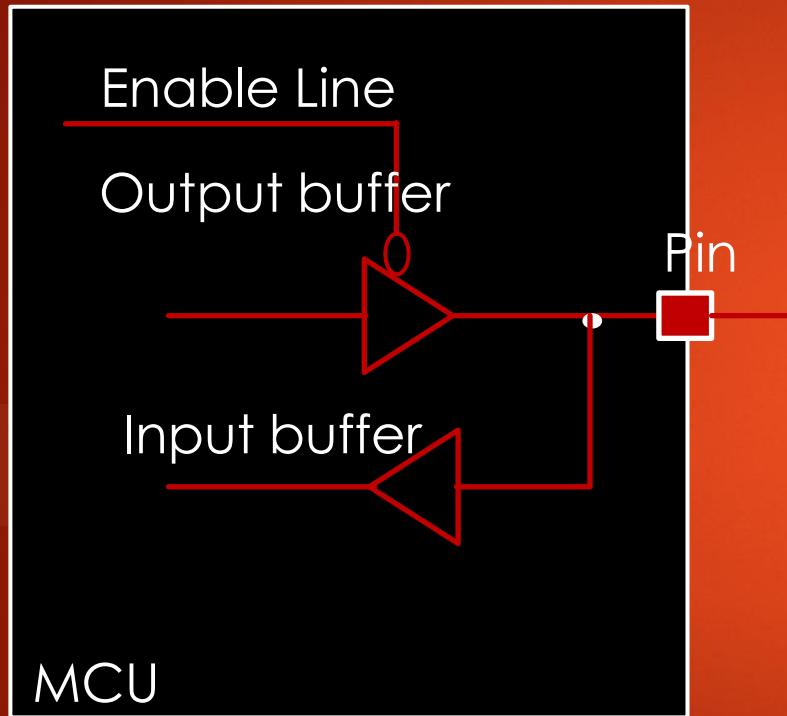


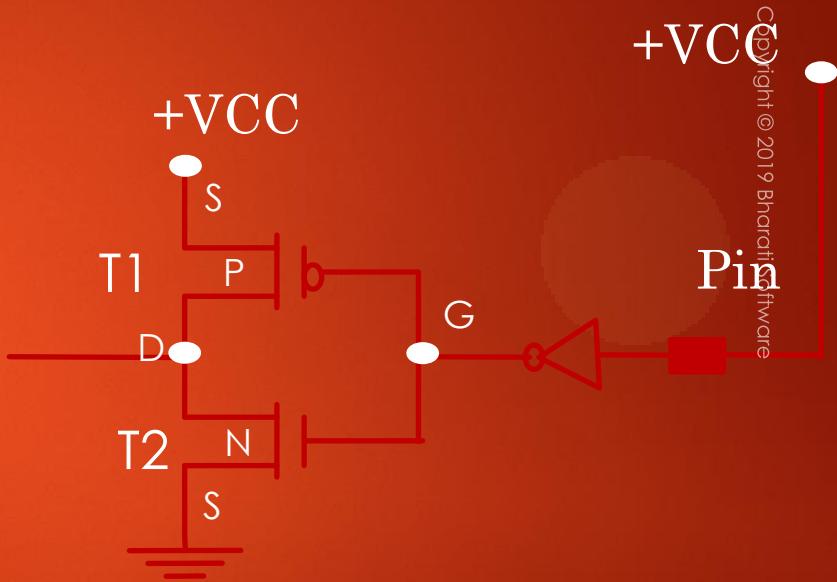
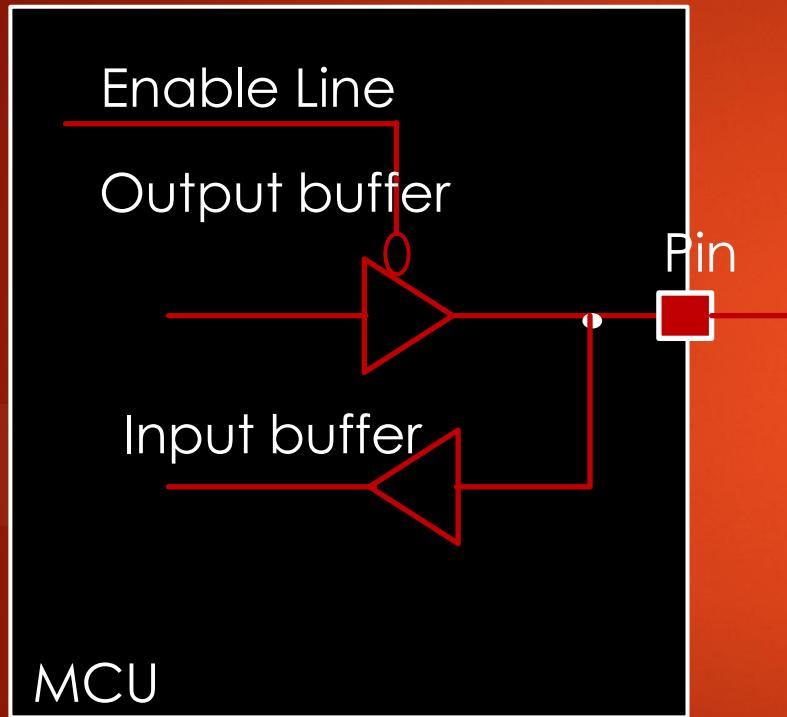
# GPIO Pin Behind the Scene

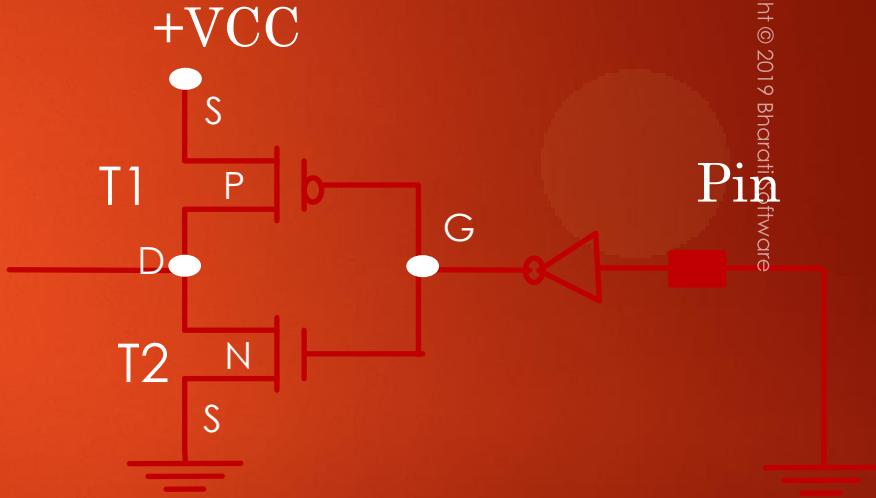
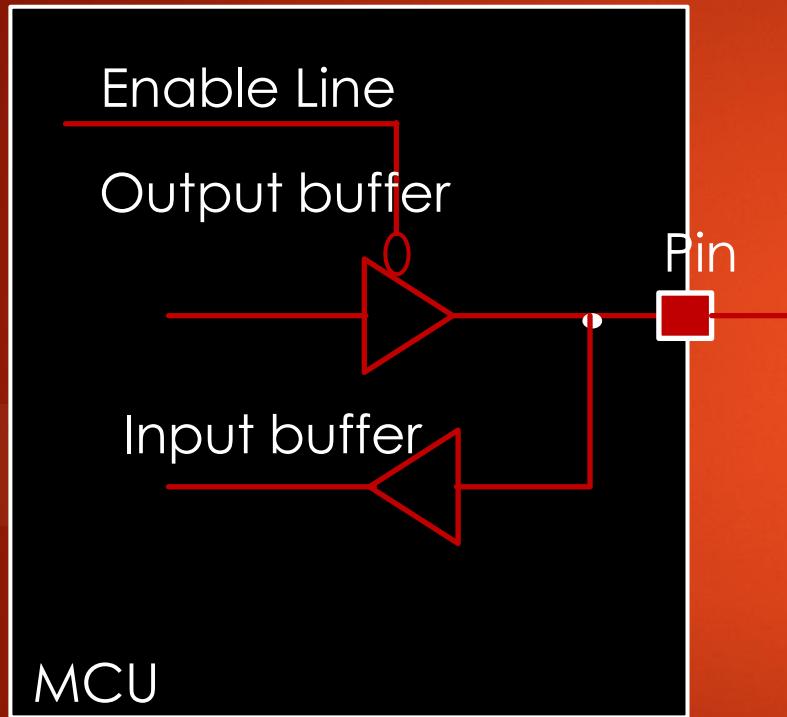
# How does a GPIO pin is actually implemented inside the MCU?









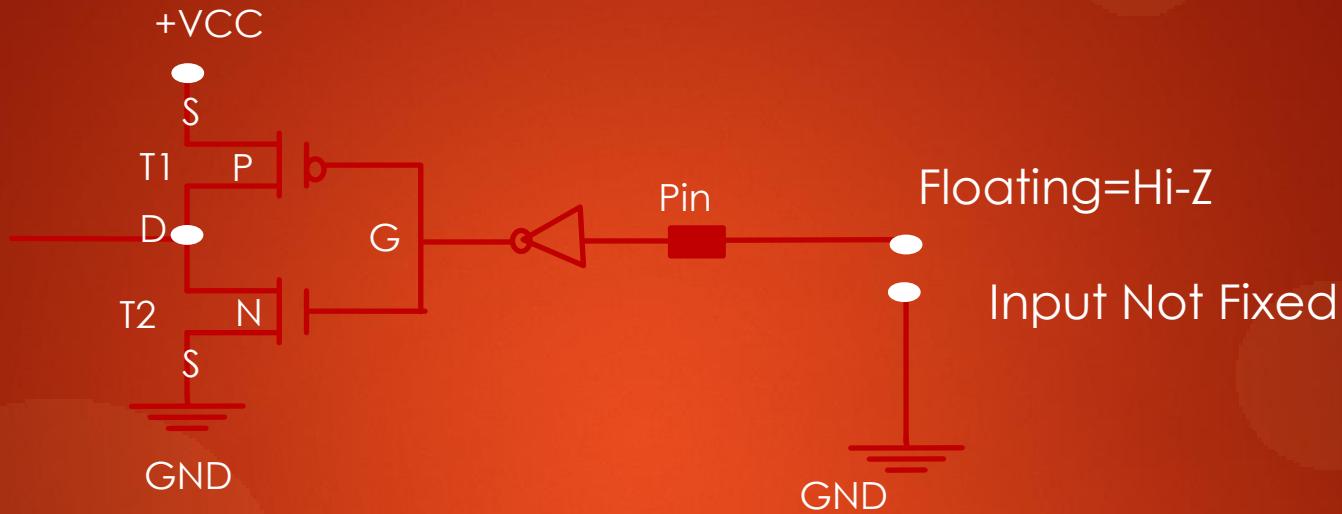


# GPIO INPUT MODE

High Impedance(HI-Z) State

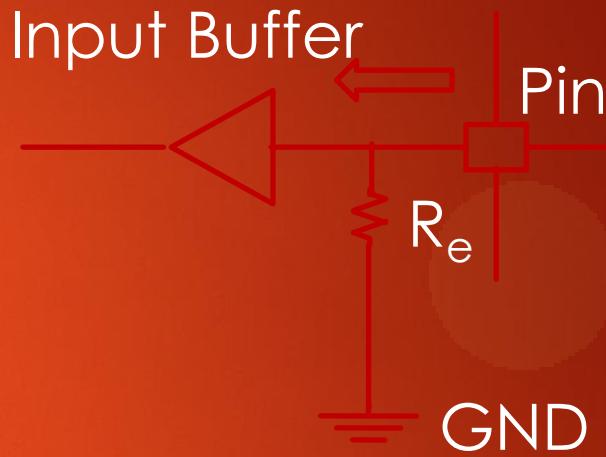
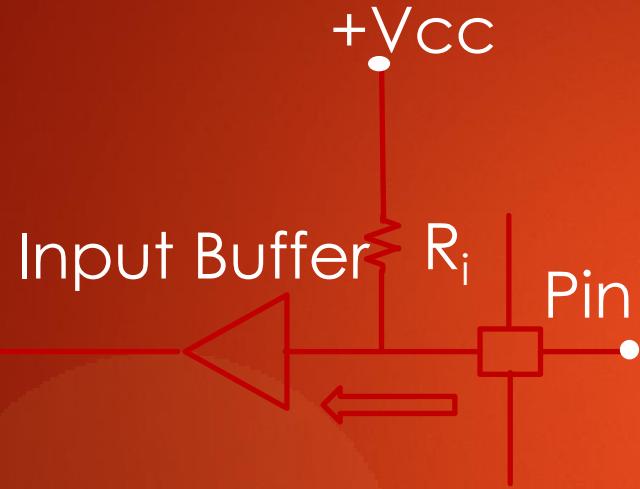
Let's understand, what exactly is this High Impedance state, that people talk about when the pin is in input mode ??

High impedance is also called as HI-Z state



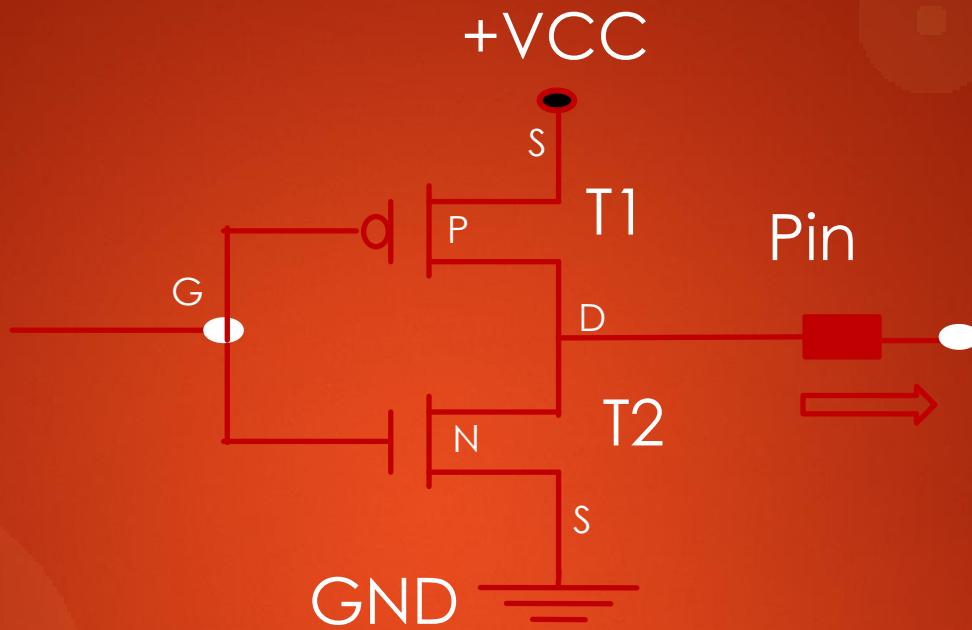
# GPIO INPUT MODE

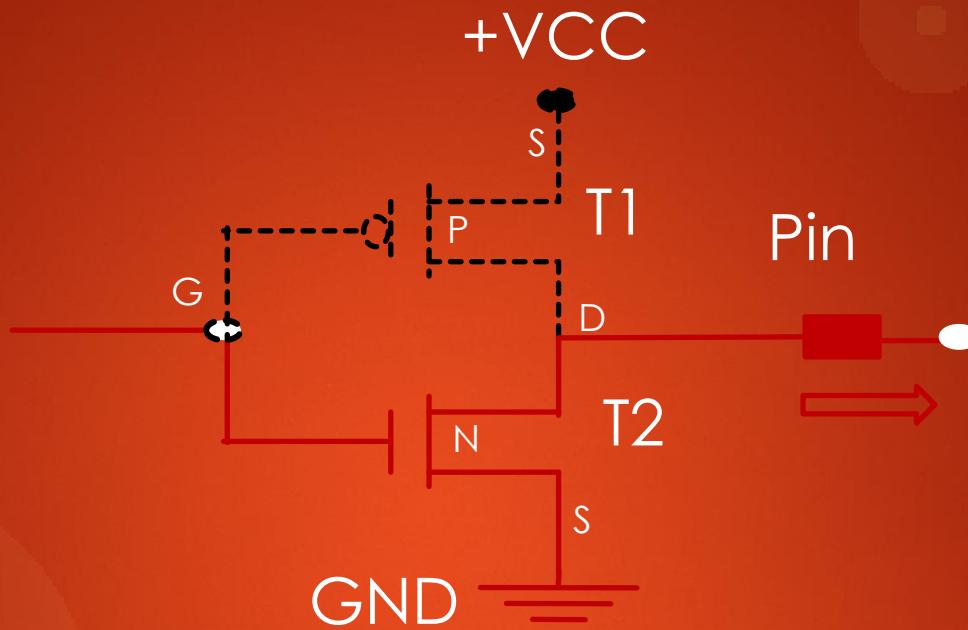
Pull-Up/Pull-Down State



# GPIO OUTPUT MODE

Open-drain State





# GPIO OUTPUT MODE

Open drain with Pull-Up

+VCC

R<sub>i</sub>

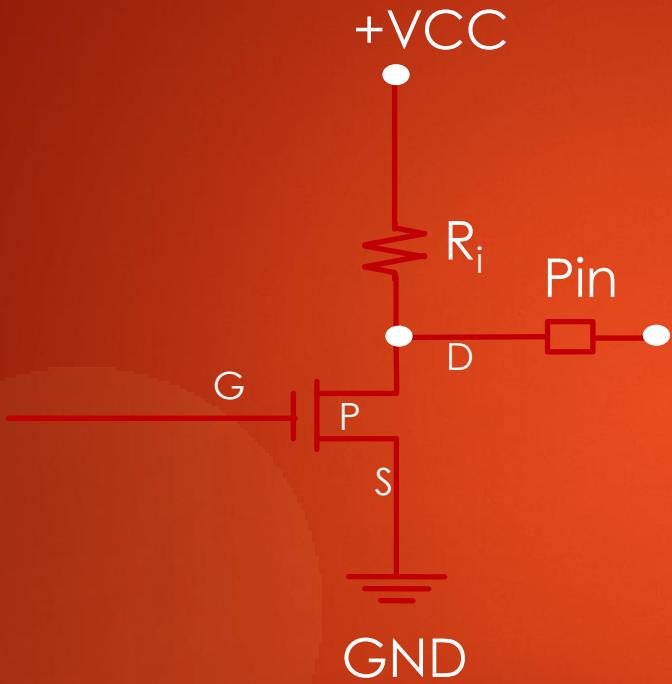
Pin

G

P

S

GND



Open drain with internal pull up

+VCC

R<sub>i</sub>

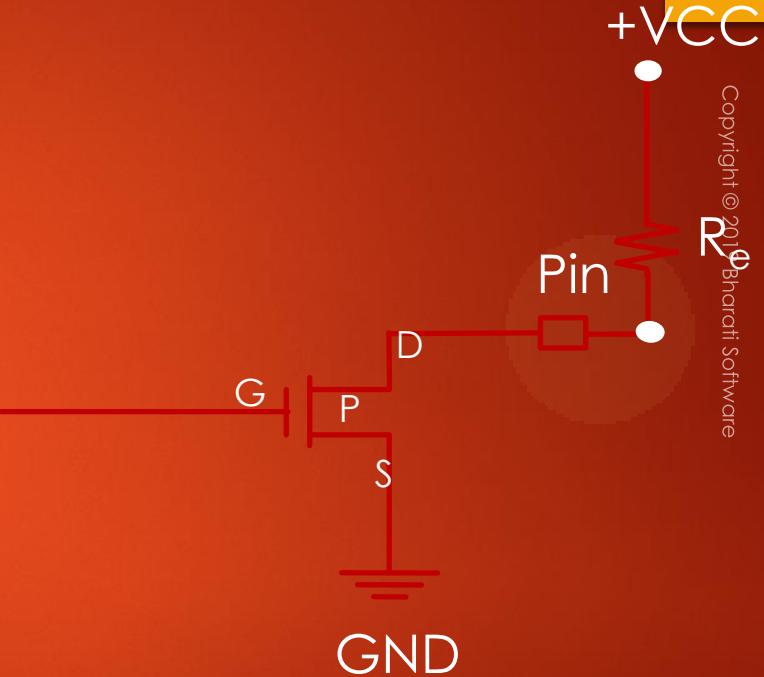
Pin

G

P

S

GND



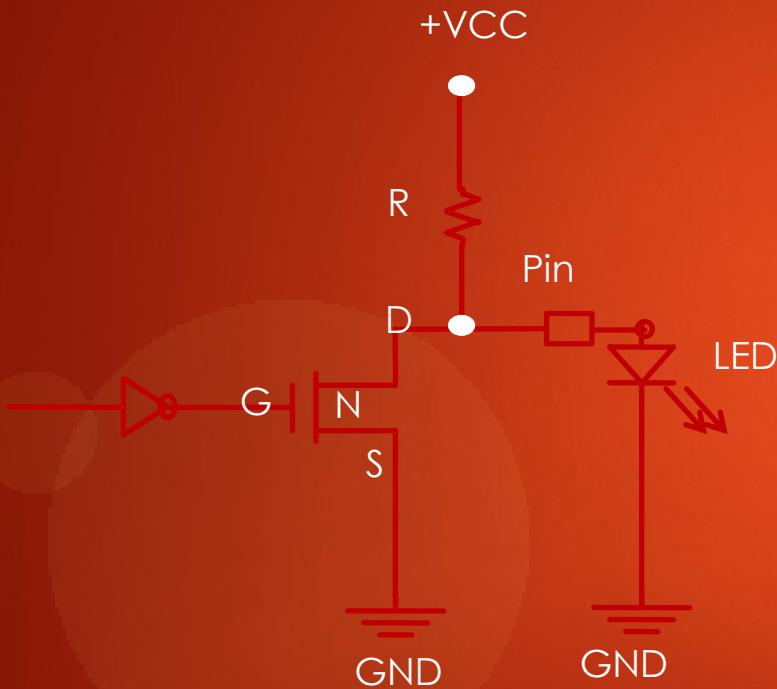
Open drain with External pull up

# Practical Usage -1

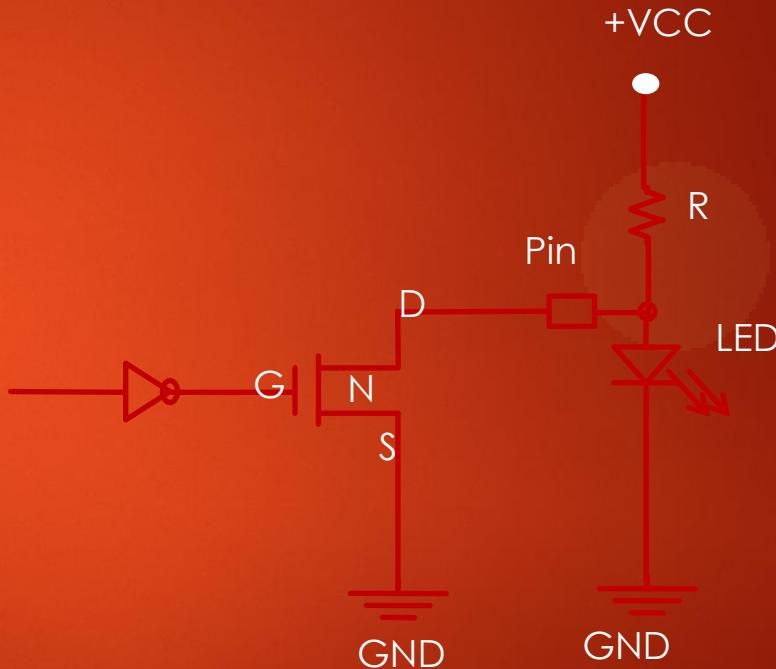
How to drive a LED from  
Open drain GPIO Pin ??

# Driving LED's

Copyright © 2019 Bharati Software



Using Internal Pull-up

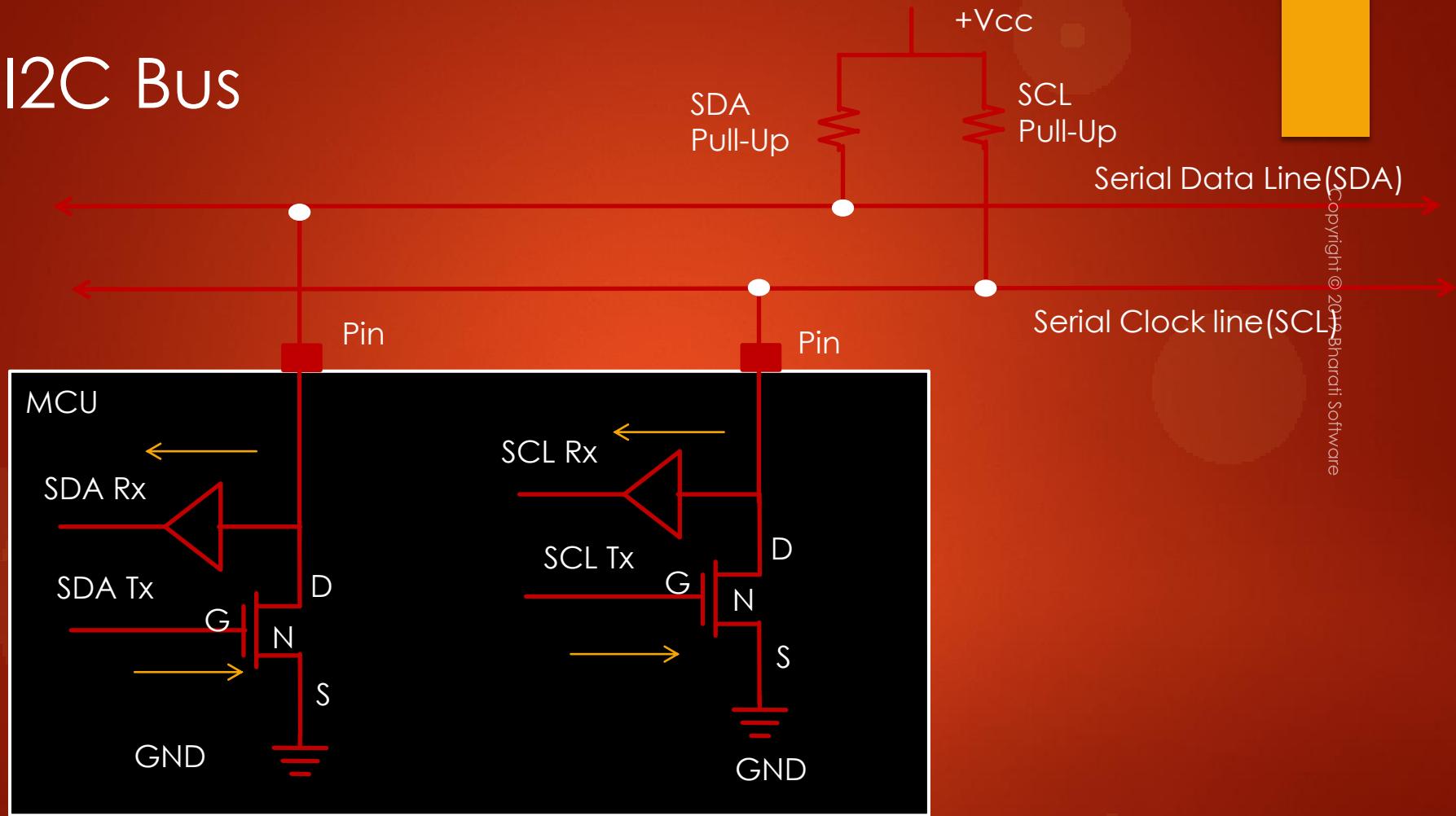


Using External Pull-up

# Practical Usage -2

Driving I2C bus

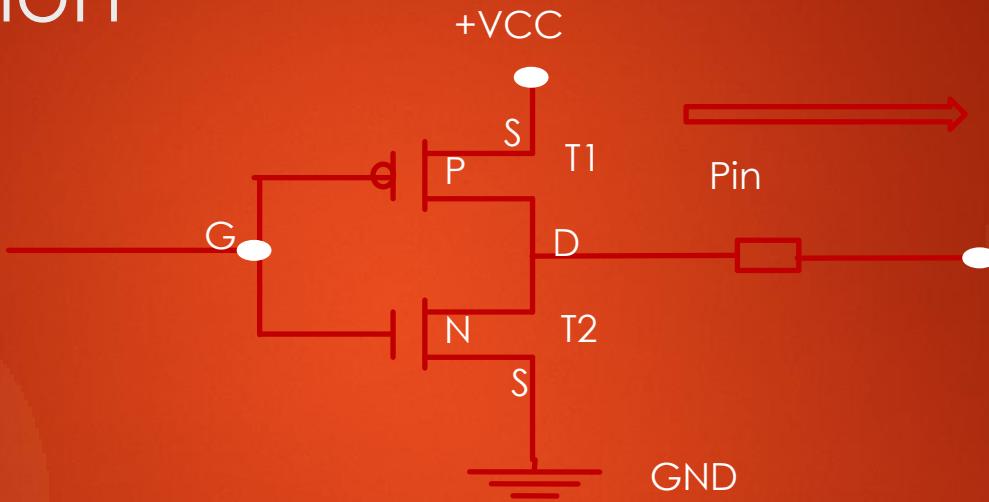
# I2C Bus



# GPIO OUTPUT MODE

Push-Pull State

# Output Mode with Push-Pull Configuration



+VCC

R<sub>i</sub>

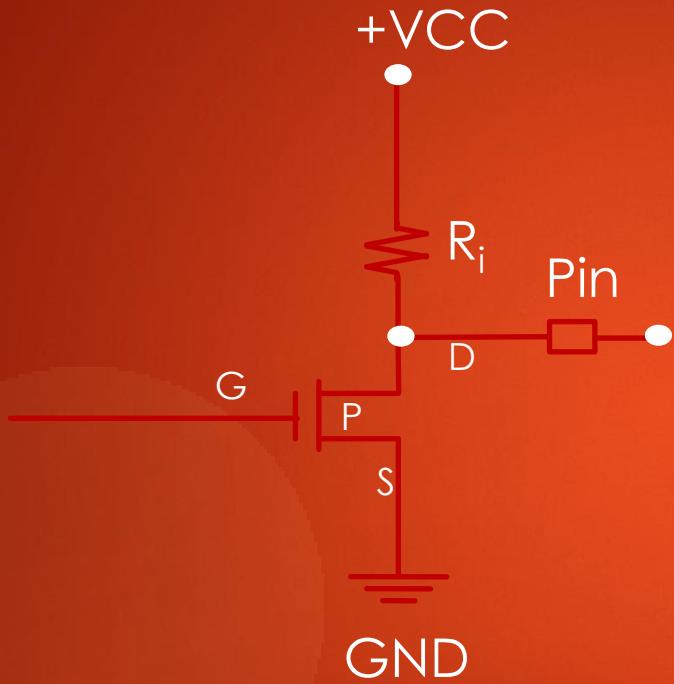
Pin

G

P

S

GND



Open drain with internal pull up

+VCC

R<sub>i</sub>

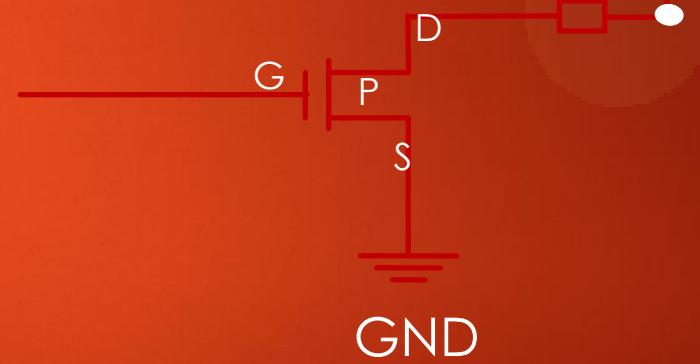
Pin

G

P

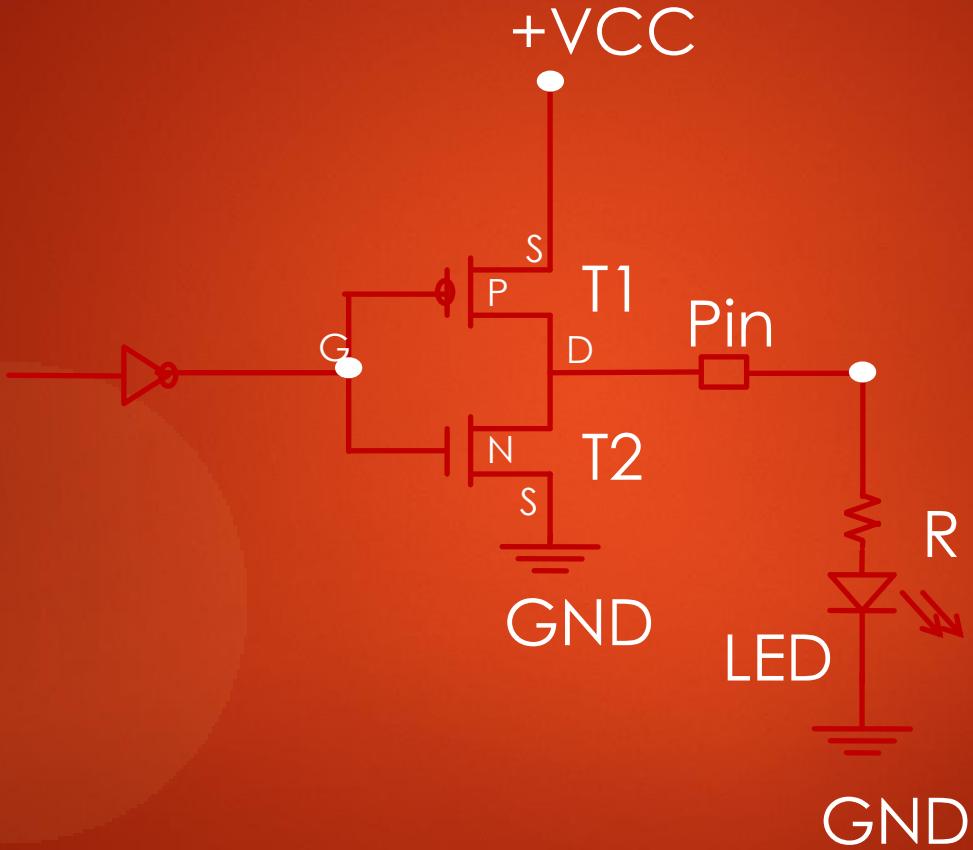
S

GND



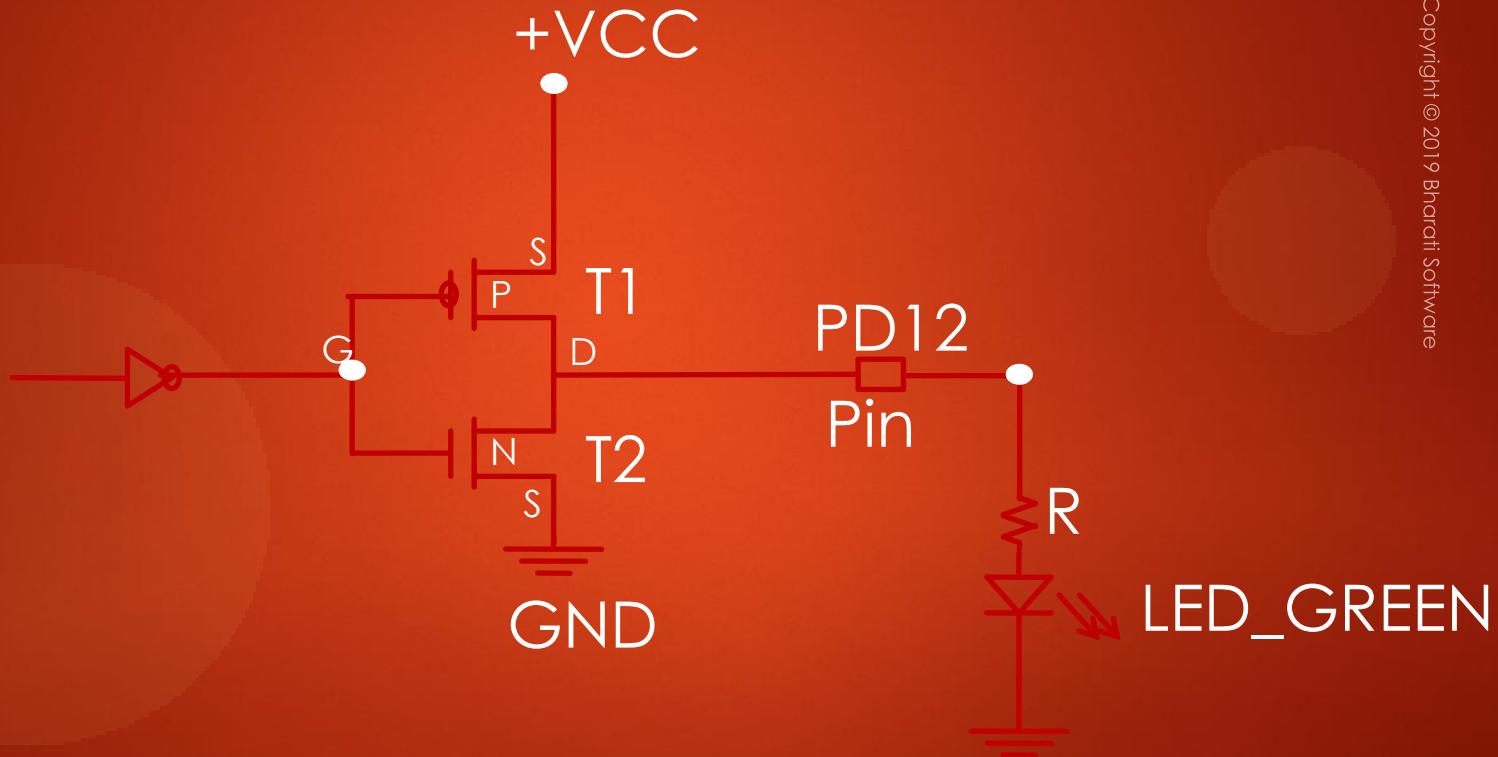
Open drain with External pull up

# How to drive a LED from Push-Pull GPIO Pin ??



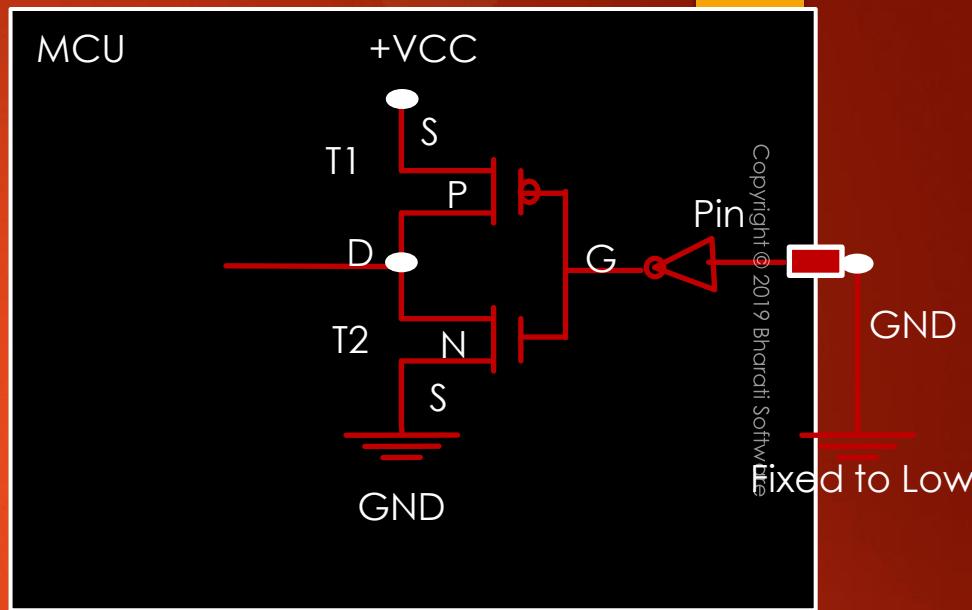
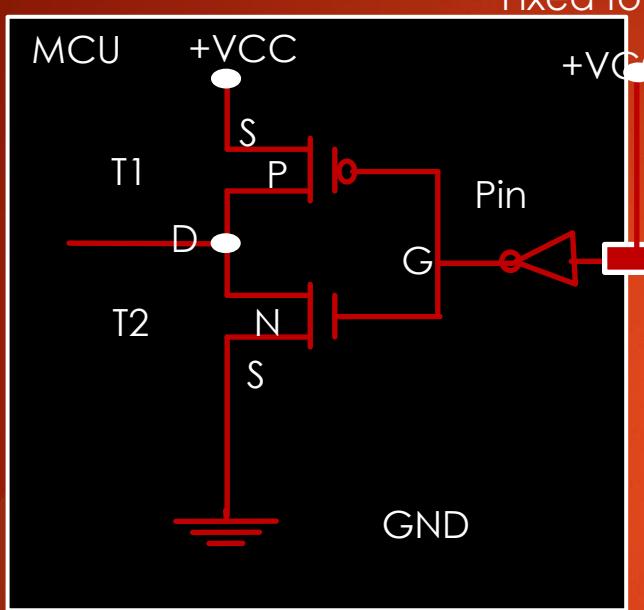
# Discovery Board LED Connection

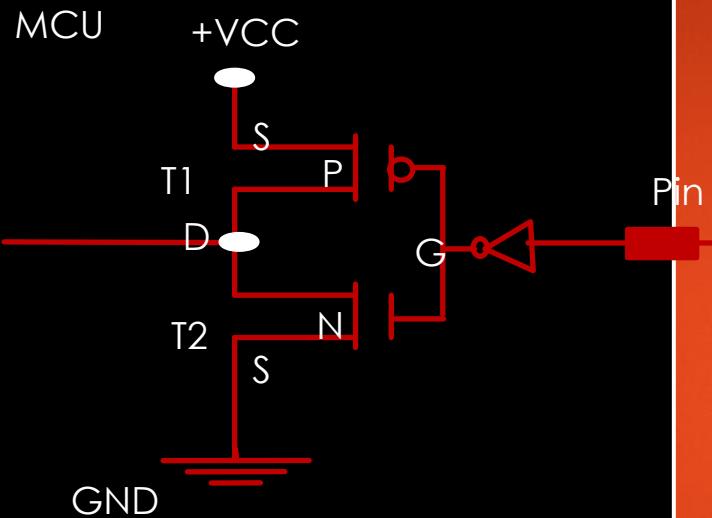
Copyright © 2019 Bharati Software



# Optimizing IO Power Consumption

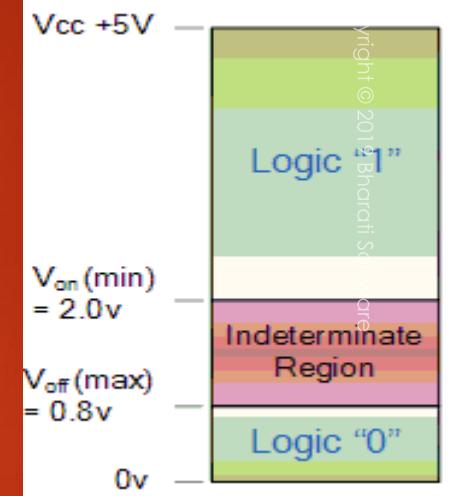
Leakage Mechanism By Input Pin Floating





Voltage of input  
is not fixed  
May be 0.5Vcc  
or 0.3Vcc

### Input Logic Level



Valve Closed



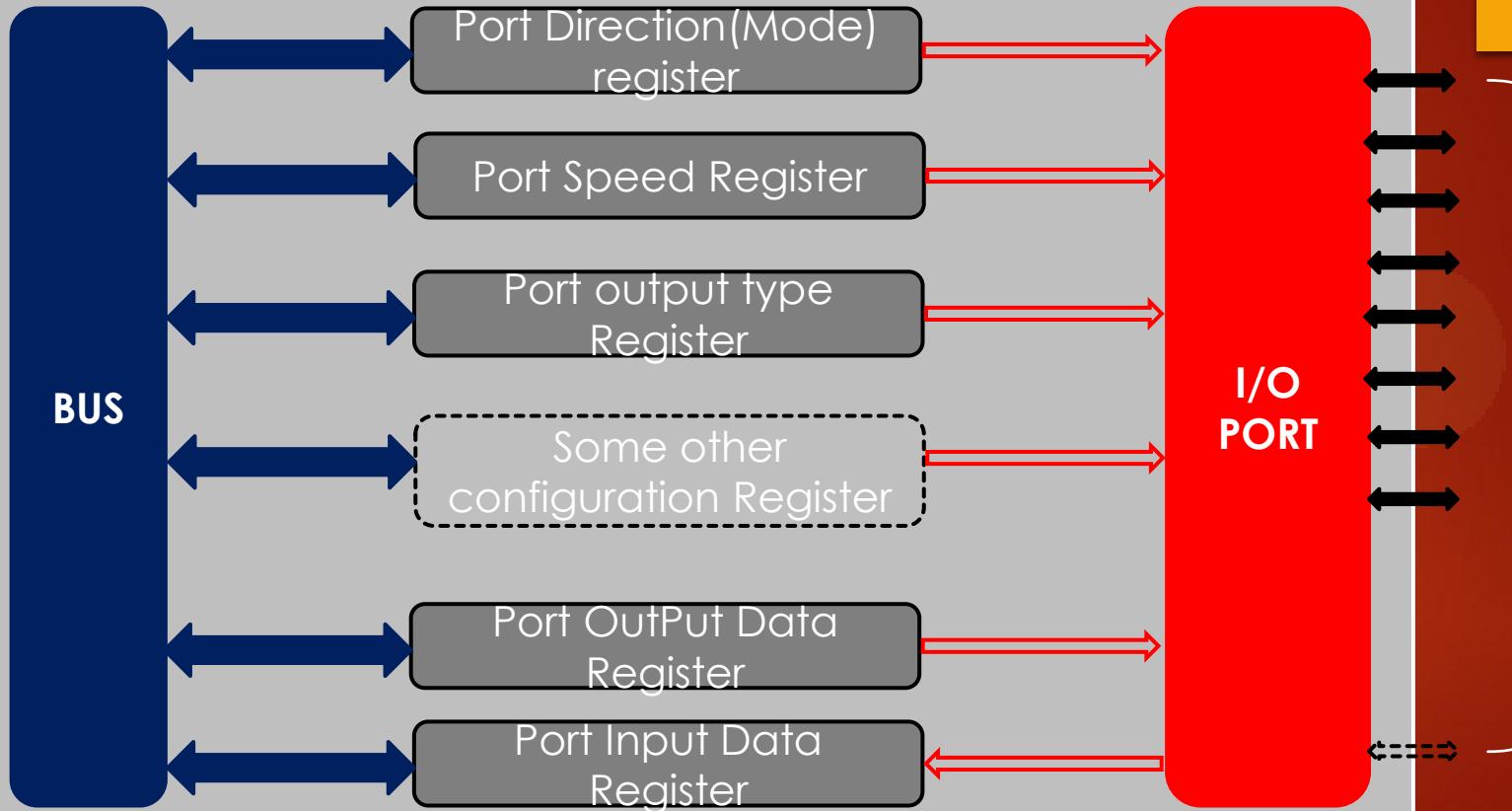
No water flows

Valve opened  
partially



Water Tickles out

# GPIO Programming Structure





In STM32F4xx series of microcontrollers, each GPIO port is governed by many configuration registers.

# Exploring GPIO Port and Pins On the Discovery board



**STM32F407VG**

**GPIOA**

**GPIOB**

**GPIOC**

**GPIOD**

**GPIOE**

**GPIOF**

**GPIOG**

**GPIOH**

**GPIOI**

Each port will have its  
Own set of configuration  
registers



right © 2019 Bharati Software

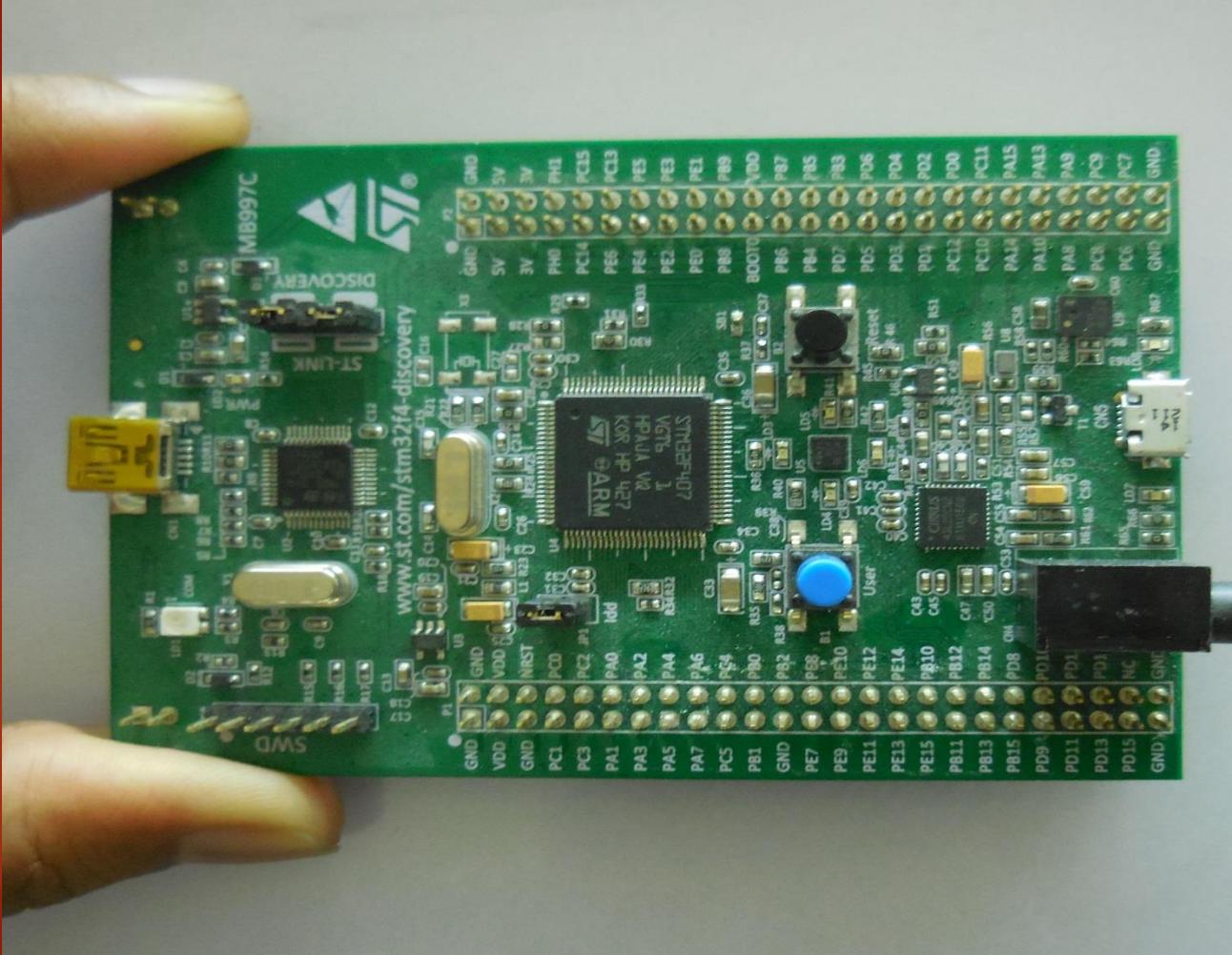
**GPIOA**

**GPIOB**

**GPIOC**

**GPIOD**

**GPIOE**



# GPIO Driver development

# GPIO Port MODE Register

32 bit reg.

**GPIOA\_MODER**

32 bit reg.

**GPIOD\_MODER**

32 bit reg.

**GPIOG\_MODER**

32 bit reg.

**GPIOB\_MODER**

32 bit reg.

**GPIOE\_MODER**

32 bit reg.

**GPIOH\_MODER**

32 bit reg.

**GPIOC\_MODER**

32 bit reg.

**GPIOF\_MODER**

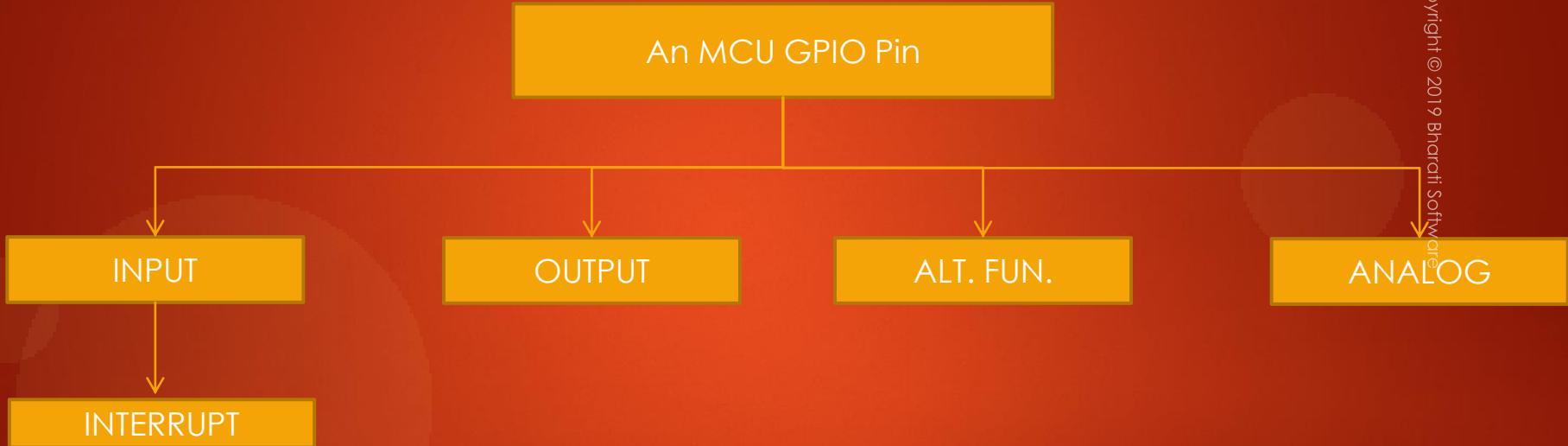
32 bit reg.

**GPIOI\_MODER**

Before using any GPIO, you must decide its MODE.  
Whether you want to use it as an Input, Output or  
for any analog purposes

**A GPIO Pin can be used for many purposes as shown here . That's why it is called as “General” Purpose.**

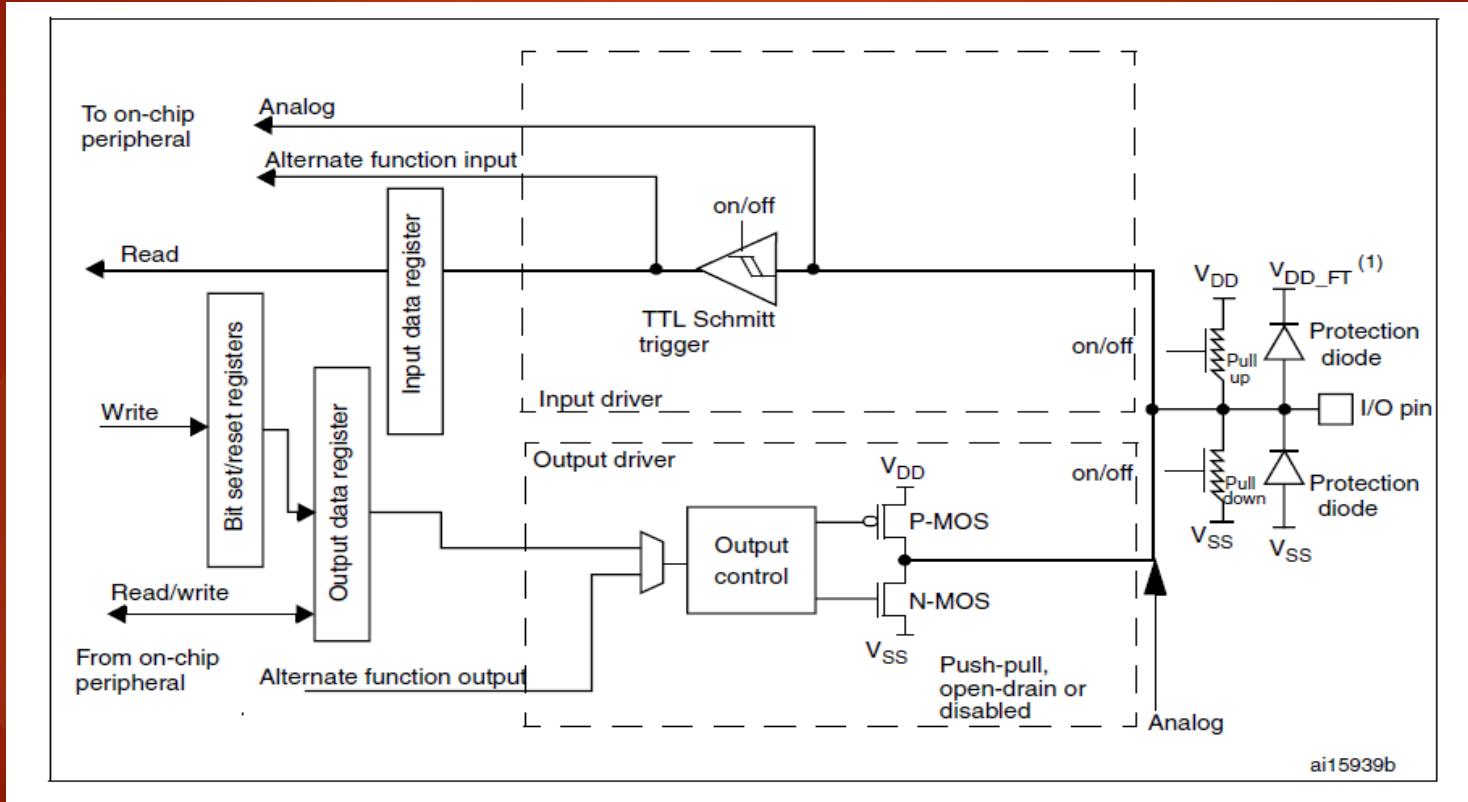
**Some pins of the MCU can not be used for all these purposes. So those are called as just pins but not GPIOs**



When MCU pin is in INPUT mode it can be configured to issue an interrupt to the processor

# Input configuration

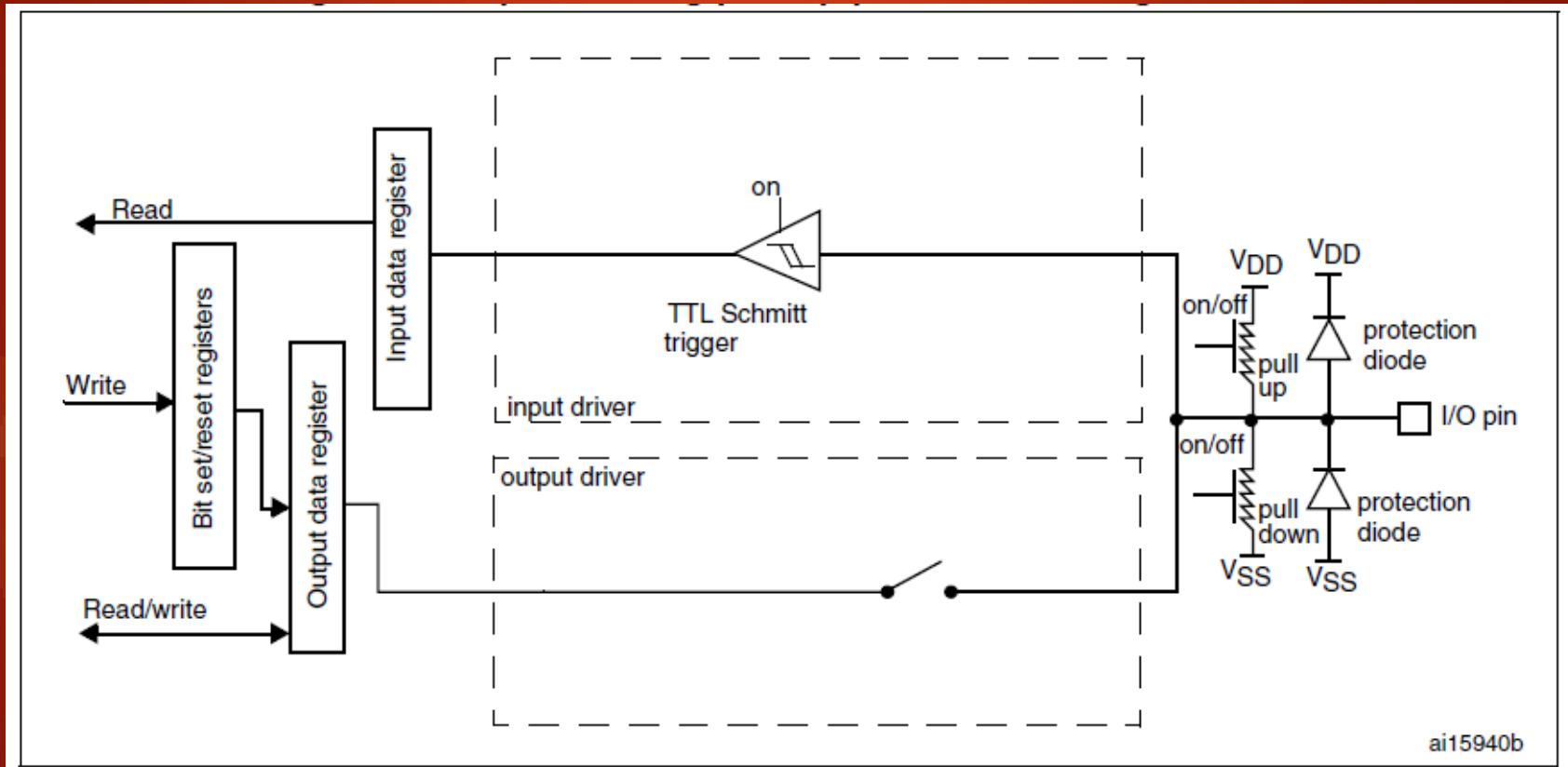
Copyright © 2019 Bharati Software

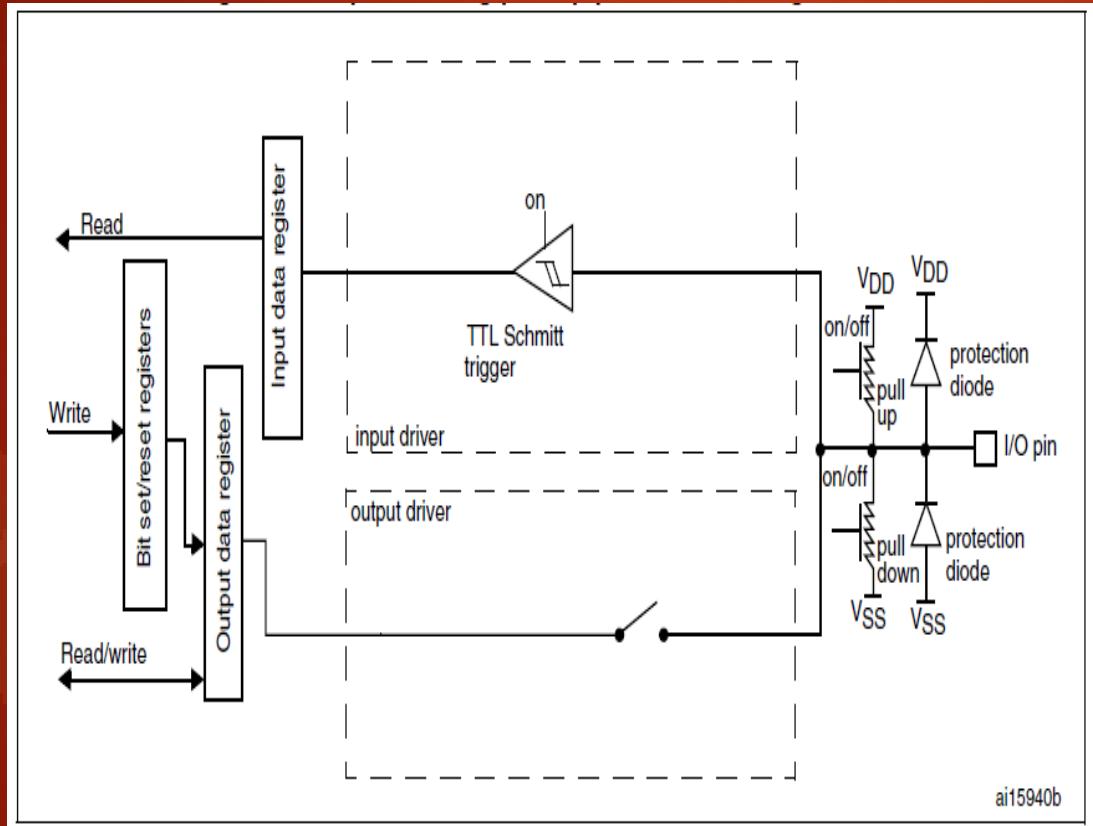


# Input mode : Summary

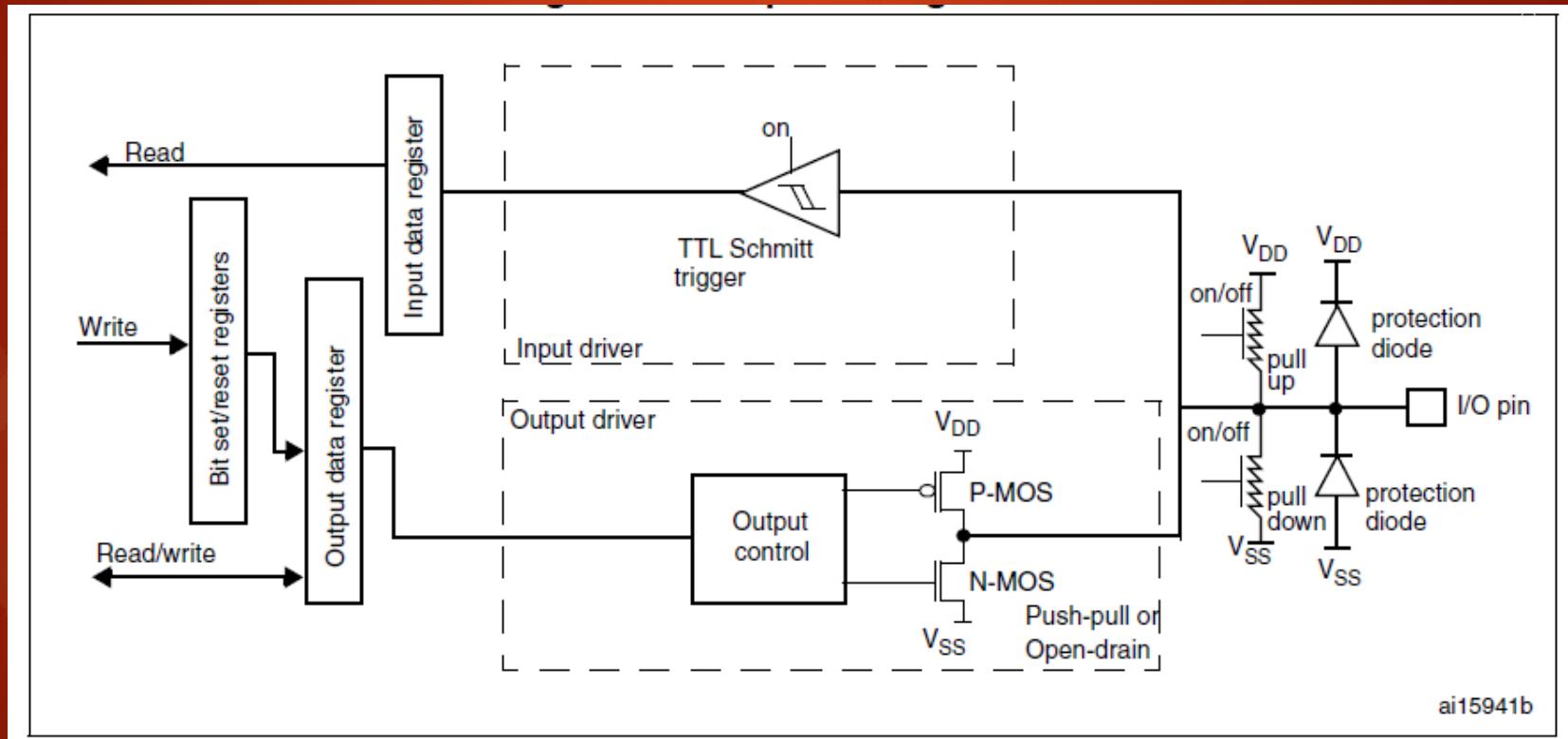
- ▶ When the I/O port is programmed as Input:
  - ▶ the output buffer is disabled
  - ▶ the Schmitt trigger input is activated
  - ▶ the pull-up and pull-down resistors are activated depending on the value in the GPIOx\_PUPDR register
  - ▶ The data present on the I/O pin are sampled into the input data register every AHB1 clock cycle
  - ▶ A read access to the input data register provides the I/O State

# Floating input



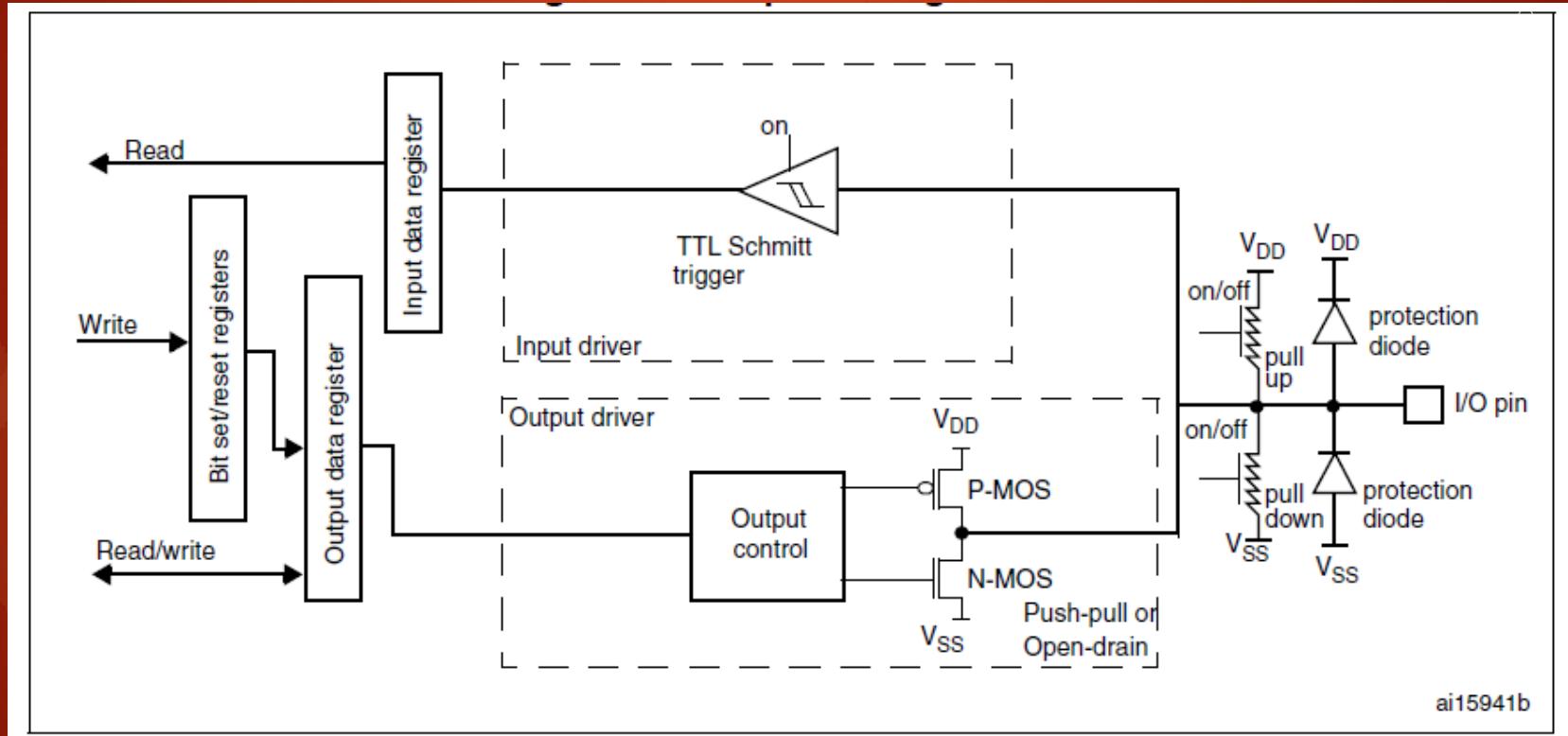


# Output configuration



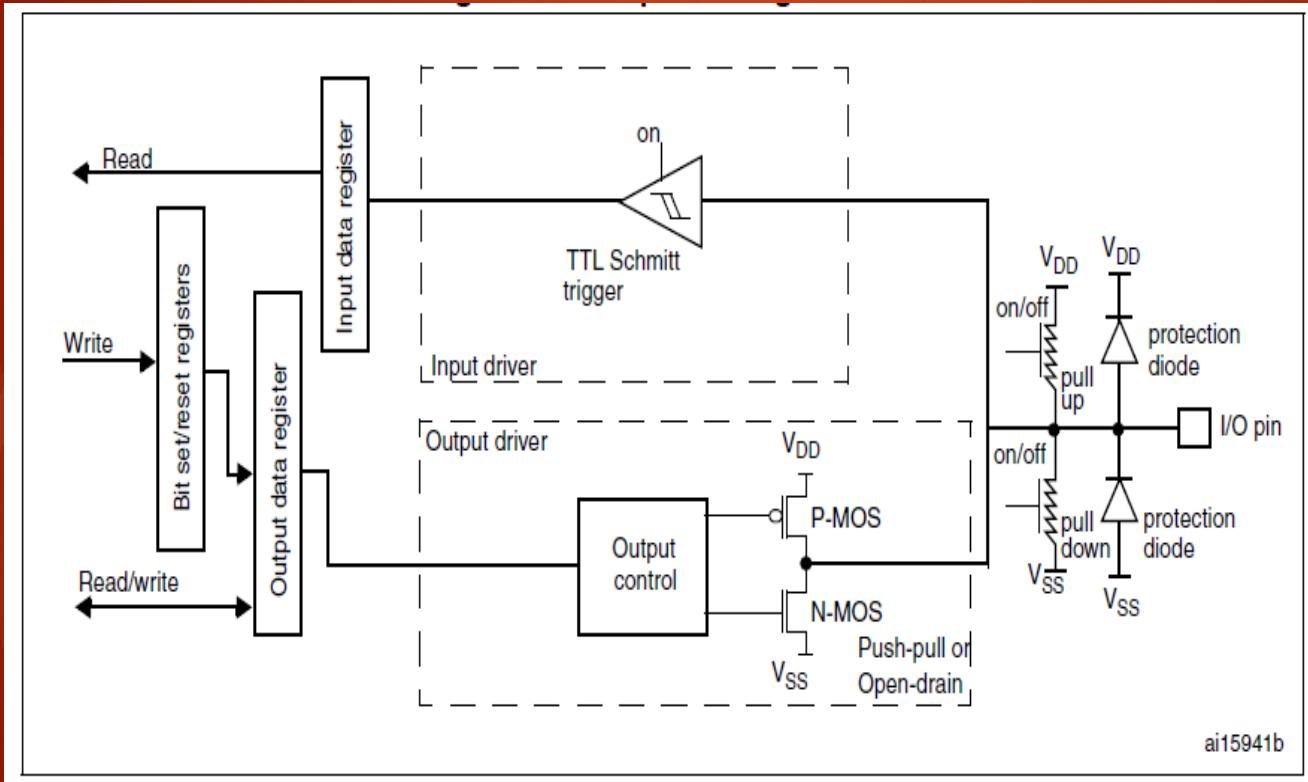
ai15941b

# Output configuration ( Push pull)



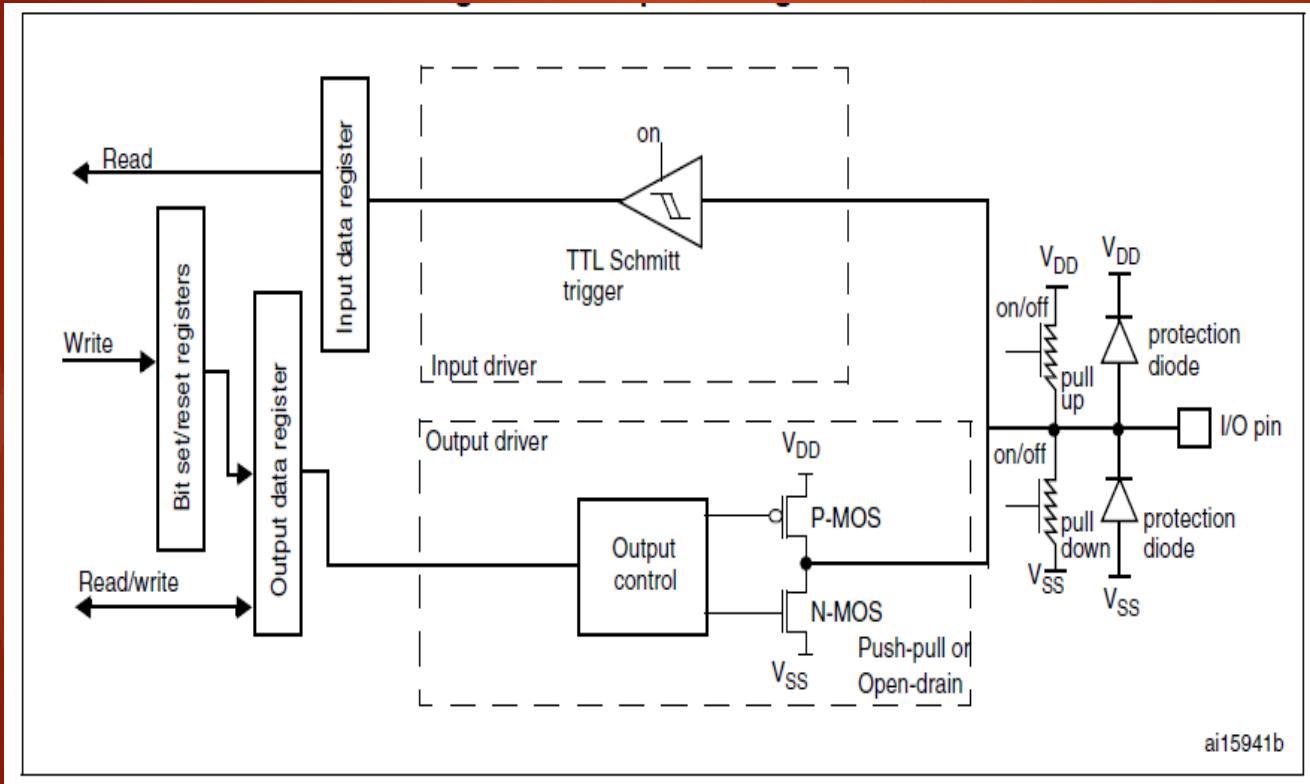
# Output configuration ( Push pull)

Copyright © 2019 Bharati Software

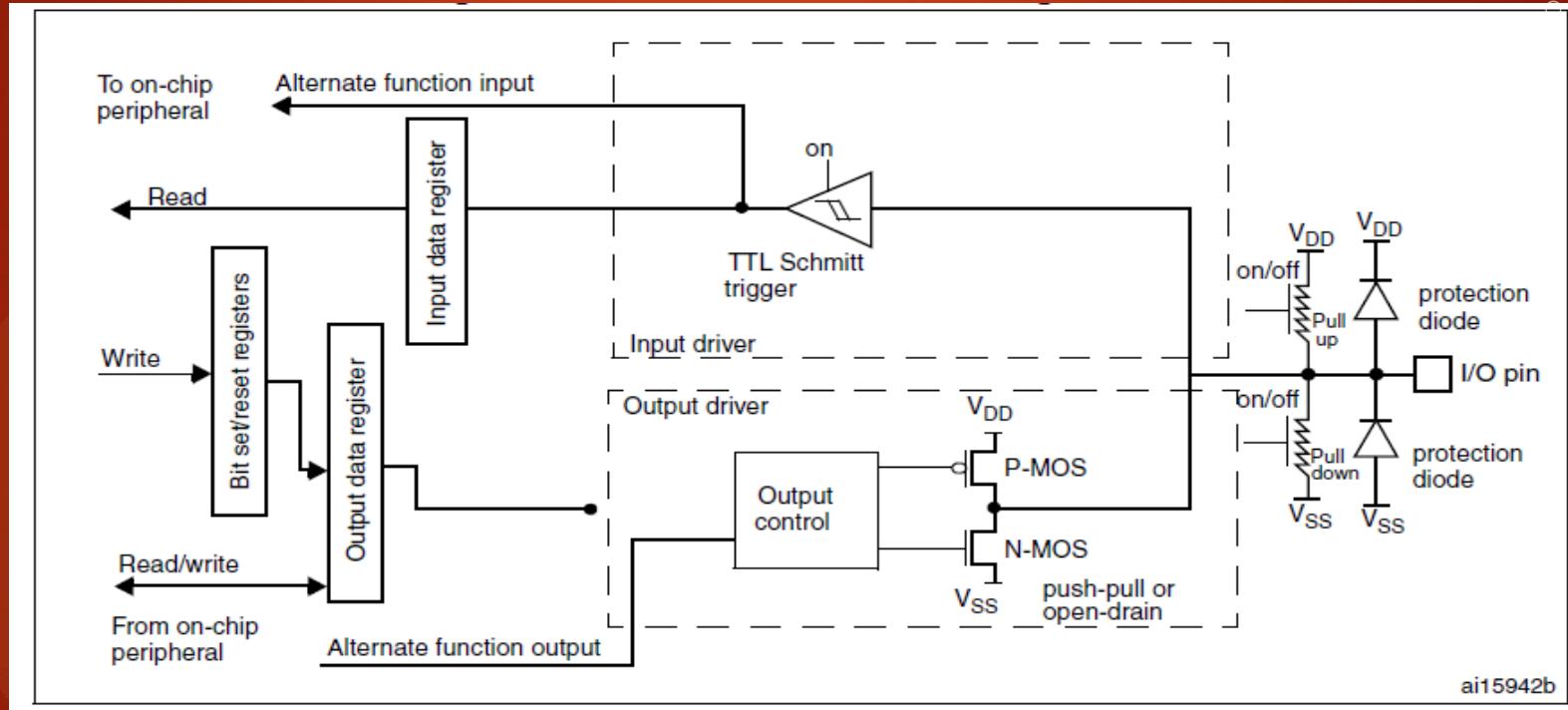


# Output configuration ( Open Drain)

Copyright © 2019 Bharati Software



# Alternate function configuration



# GPIO Port Output Type Register

When a GPIO pin is in the output mode, this register is used to select the output type of that pin

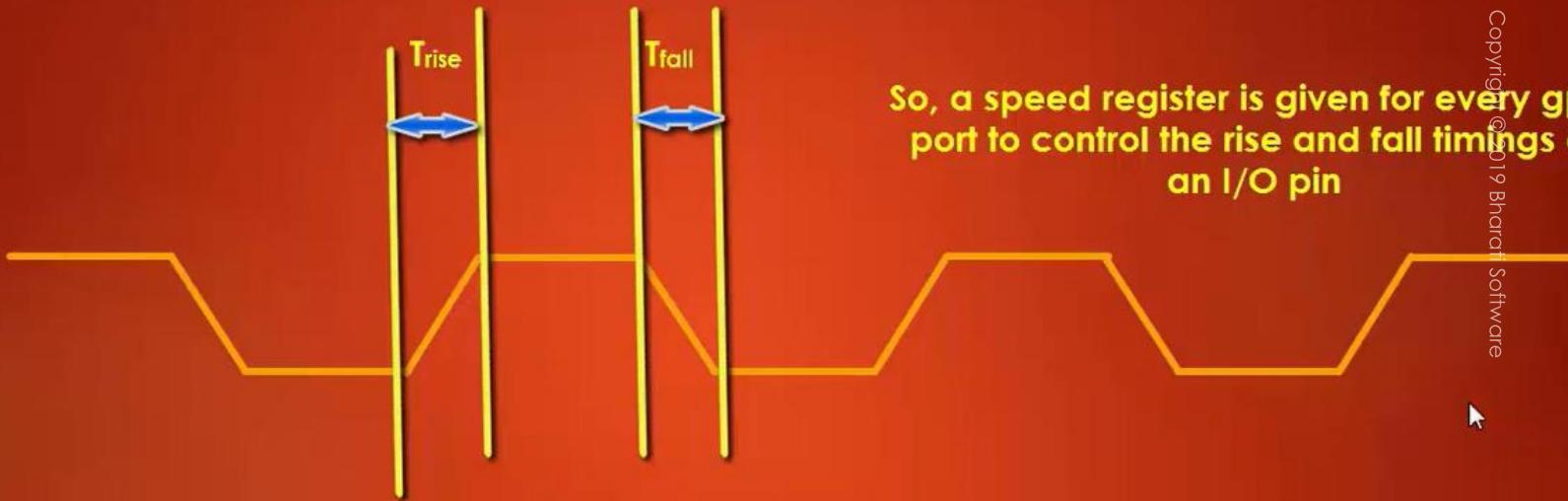
# GPIO Port Speed Register

By using speed register you can  
configure “**how quick the GPIO transitions  
from H to L and L to H**”

In other words you can control the slew rate of a pin

low speed gpios will have larger  $T_{rise}$  and  $T_{fall}$  timings.

Slew rate



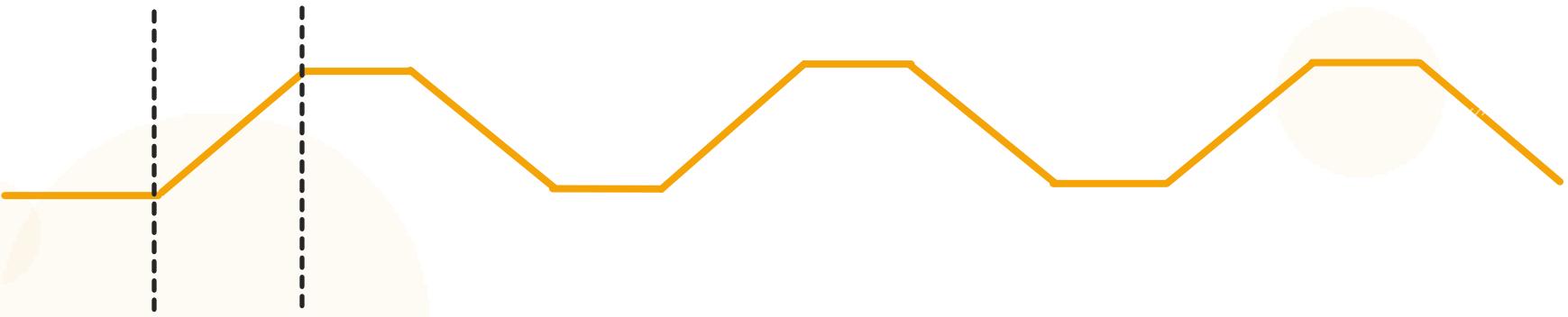
So, a speed register is given for every gpio port to control the rise and fall timings of an I/O pin





LOW

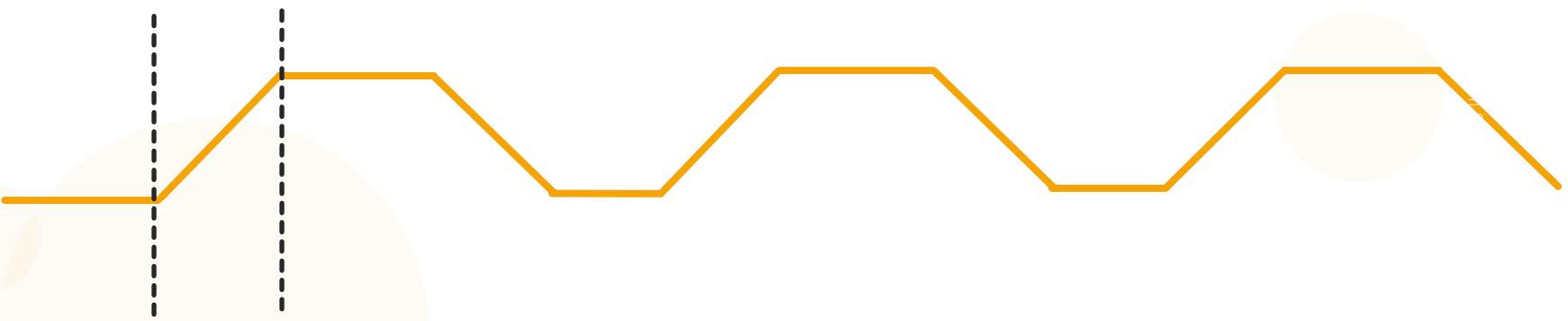
Slew rate





MEDIUM

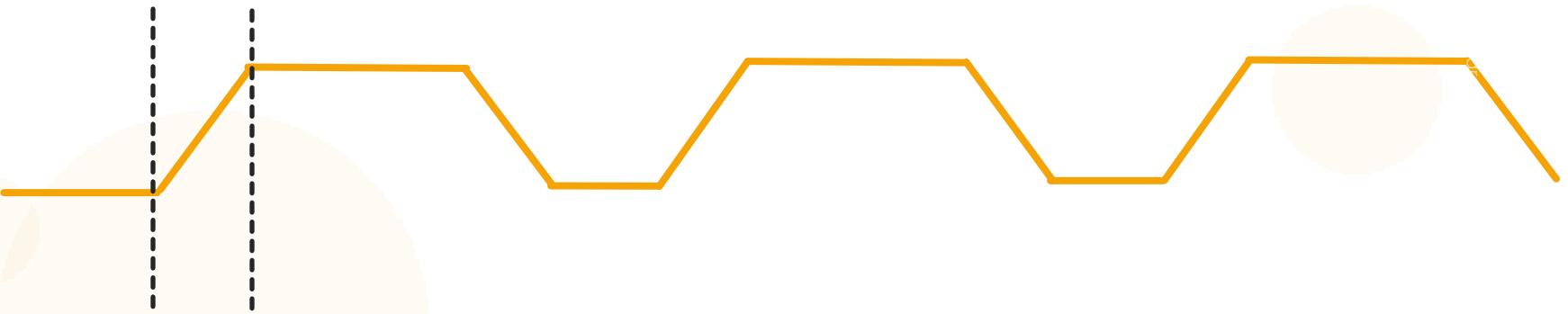
Slew rate





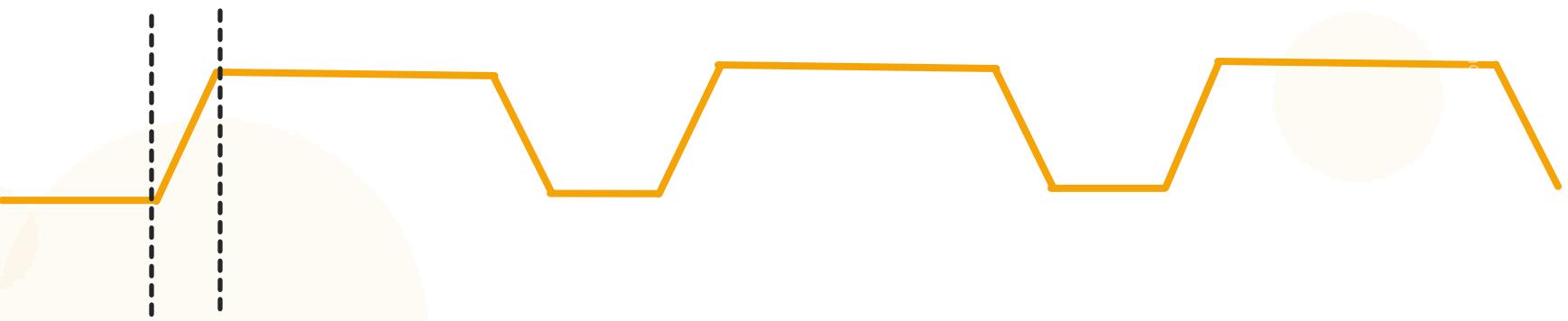
HIGH

Slew rate



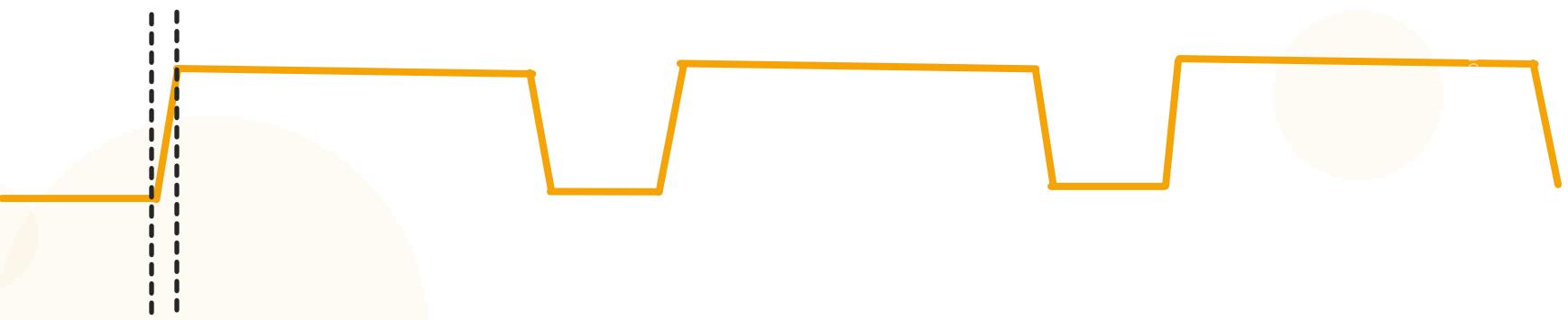
**VERY HIGH**

Slew rate



**VERY HIGH**

**Slew rate**



# Applicability of SPEED register configuration

Copyright

MODER(i)[1:0]	OTYPER(i)	OSPEEDR(i)[B:A]	PUPDR(i)[1:0]		I/O configuration	
01 <b>Pin Mode = OUTPUT</b>	0	SPEED[B:A] <b>Applicable</b>	0	0	GP output	PP
	0		0	1	GP output	PP + PU
	0		1	0	GP output	PP + PD
	0		1	1	Reserved	
	1		0	0	GP output	OD
	1		0	1	GP output	OD + PU
	1		1	0	GP output	OD + PD
	1		1	1	Reserved (GP output OD)	

# Applicability of SPEED register configuration

MODER(i)[1:0]	OTYPER(i)	OSPEEDR(i)[B:A]	PUPDR(i)[1:0]	I/O configuration	
00 <b>Pin Mode = INPUT</b>	x	x	x	0	0
	x	x	x	0	1
	x	x	x	1	0
	x	x	x	1	1
				Reserved (input floating)	

So, Speed setting is only applicable when the pin is in output mode

# GPIO Port Pull-Up/Pull-Down Register

# GPIO Port Input Data Register

# GPIO Port Output Data Register

# GPIO Functional Summary

By taking various modes and pull up /pull down resistors combinations below configurations can be obtained for a GPIO pin

Input floating

Input pull-up

Input-pull-down

Analog

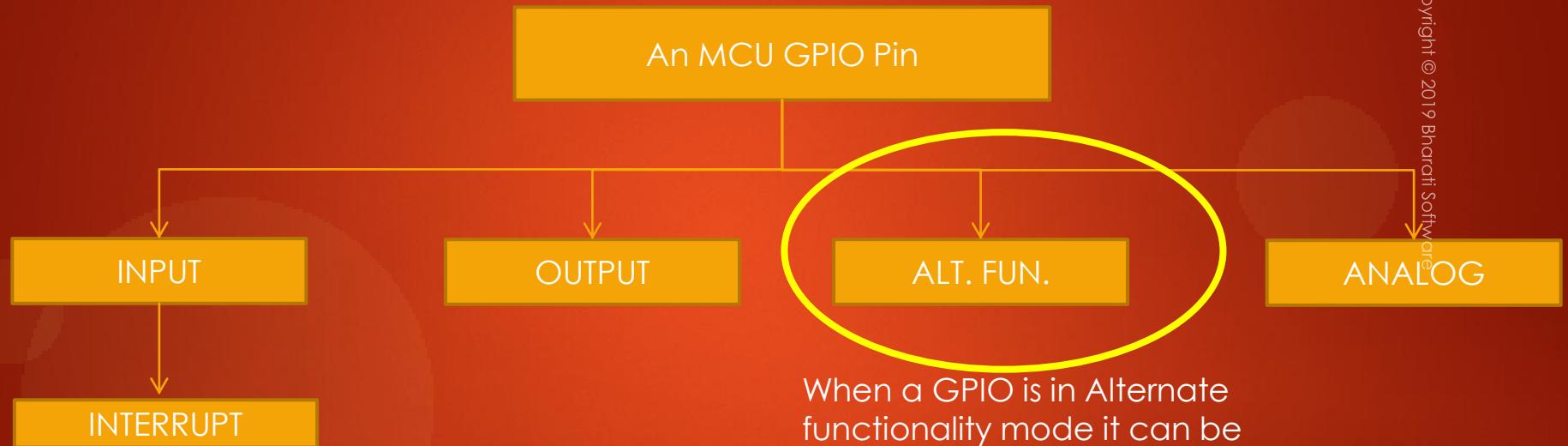
Output open-drain with pull-up or pull-down capability

Output push-pull with pull-up or pull-down capability

Alternate function push-pull with pull-up or pull-down capability

Alternate function open-drain with pull-up or pull-down capability

# Alternate Functionality configuration of a GPIO pin



When a GPIO is in Alternate functionality mode it can be used for 16 different functionalities .

# A GPIO Pin's 16 possible alternate functionalities

- AF0 (system)
- AF1 (TIM1/TIM2)
- AF2 (TIM3..5)
- AF3 (TIM8..11, CEC)
- AF4 (I2C1..4, CEC)
- AF5 (SPI1/2/3/4)
- AF6 (SPI2/3/4, SAI1)
- AF7 (SPI2/3, USART1..3, UART5, SPDIF-IN)
- AF8 (SPI2/3, USART1..3, UART5, SPDIF-IN)
- AF9 (CAN1/2, TIM12..14, QUADSPI)
- AF10 (SAI2, QUADSPI, OTG\_HS, OTG\_FS)
- AF11
- AF12 (FMC, SDIO, OTG\_HS<sup>(1)</sup>)
- AF13 (DCMI)
- AF14
- AF15 (EVENTOUT)

# Exercise :

List out all the 16 possible alternation functionalities supported by **GPIO port 'A' pin number 8 (GPIOA.8)**

MODE(Afx)	Functionality
AF0	MCO1
AF1	TIM1_CH1
AF2	Not Supported
AF3	Not Supported
AF4	I2C3_SCL
AF5	Not Supported
AF6	Not Supported
AF7	USART1_ CK

# Exercise :

List out all the 16 possible alternation functionalities supported by **GPIO port 'A' pin number 8 (GPIOA.8)**

MODE(Afx)	Functionality
AF8	Not Supported
AF9	Not Supported
AF10	OTG_FS_SOF
AF11	Not Supported
AF12	Not Supported
AF13	Not Supported
AF14	Not Supported
AF15	EVENT OUT

# Exercise :

List out all the alternation functionalities supported by **GPIO port 'C' pin number 6 (GPIOC.6)**

# Solution :

List out all the alternation functionalities supported by **GPIO port 'C' pin number 6 (GPIOC.6)**

MODE(Afx)	Functionality
AF0	Not Supported
AF1	Not Supported
AF2	TIM3_CH1
AF3	TIM8_CH1
AF4	Not Supported
AF5	I2S2_MCK
AF6	Not Supported
AF7	Not Supported

# Solution :

List out all the alternation functionalities supported by **GPIO port 'C' pin number 6 (GPIOC.6)**

MODE(Afx)	Functionality
AF8	USART6_TX
AF9	Not Supported
AF10	Not Supported
AF11	Not Supported
AF12	SDIO_D6
AF13	DCMI_D0
AF14	Not Supported
AF15	EVENT OUT

# GPIO Alternate Function Register

# Example

Find out the alternate functionally mode(AFx) and AFR(Alternate Function Register) settings to make

PA0 as UART4\_TX

PA1 as UART4\_RX

PA10 as TIM1\_CH3

# Solution: PA0 as UART4\_TX

AFx = AF8

( This info you can only get from datasheet of the MCU not from RM in the case of ST's MCUs)

# Solution: PA0 as UART4\_TX

AFR settings

Copyright © 2019

## GPIOA\_AFRL

GPIO alternate function low register (GPIOx\_AFRL) (x = A..I/J/K)

Pin 7				Pin 6				Pin 5				Pin 4			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw												
Pin 3				Pin 2				Pin 1				Pin 0			

# Solution: PA1 as UART4\_RX

AFx = AF8

Copyright © 2019

## GPIOA\_AFRL

GPIO alternate function low register (GPIOx\_AFRL) (x = A..I/J/K)

Pin 7

Pin 6

Pin 5

Pin 4

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw												

Pin 3

Pin 2

Pin 1

Pin 0

# Solution: PA10 as TIM1\_CH3

AFx = AF1

Copyright © 20

**GPIO alternate function high register (GPIOx\_AFRH)**      **GPIOA\_AFRH**  
**(x = A..I/J)**

**Pin 15**

**Pin 14**

**Pin 13**

**Pin 12**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRH15[3:0]				AFRH14[3:0]				AFRH13[3:0]				AFRH12[3:0]			
rw	rw	rw	rw												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRH11[3:0]				AFRH10[3:0]				AFRH9[3:0]				AFRH8[3:0]			
rw	rw	rw	rw												

**Pin 11**

**Pin 10**

**Pin 9**

**Pin 8**

**GPIO alternate function high register (GPIOx\_AFRH)**  
(x = A..I/J)

Pin 15

Pin 14

Pin 13

Pin 12

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRH15[3:0]				AFRH14[3:0]				AFRH13[3:0]				AFRH12[3:0]			
rw	rw	rw	rw												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRH11[3:0]				AFRH10[3:0]				AFRH9[3:0]				AFRH8[3:0]			
rw	rw	rw	rw												

Pin 11

Pin 10

Pin 9

Pin 8

**GPIO alternate function low register (GPIOx\_AFRL) (x = A..I/J/K)**

Pin 7

Pin 6

Pin 5

Pin 4

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRLO[3:0]			
rw	rw	rw	rw												

Pin 3

Pin 2

Pin 1

Pin 0

#### 8.4.5 GPIO port input data register (GPIOx\_IDR) (x = A..I/J/K)

Address offset: 0x10

Reset value: 0x0000 XXXX (where X means undefined)

# Enabling /Disabling GPIO Port Peripheral Clock

## RCC Peri. Clock Enable Registers

**RCC\_AHB1ENR**

**RCC\_AHB2ENR**

**RCC\_AHB3ENR**

**RCC\_APB1ENR**

**RCC\_APB2ENR**

## RCC AHB1 peripheral clock register (RCC\_AHB1ENR)

Address offset: 0x30

Reset value: 0x0010 0000

Access: no wait state, word, half-word and byte access.

# Peripheral Driver Development

# High level project architecture

# Sample Applications

Copyright © 2019 Bharatii Software

## Driver Layer

gpio\_driver.c , .h

i2c\_driver.c , .h

(Device header)

Stm3f407xx.h

spi\_driver.c , .h

uart\_driver.c , .h

GPIO

SPI

I2C

UART

**STM3F407x MCU**

3

## Sample Applications



### Driver Layer

gpio\_driver.c , .h

2

i2c\_driver.c , .h

spi\_driver.c , .h

uart\_driver.c , .h

(Device header)

Stm3f407xx.h

1

GPIO

SPI

I2C

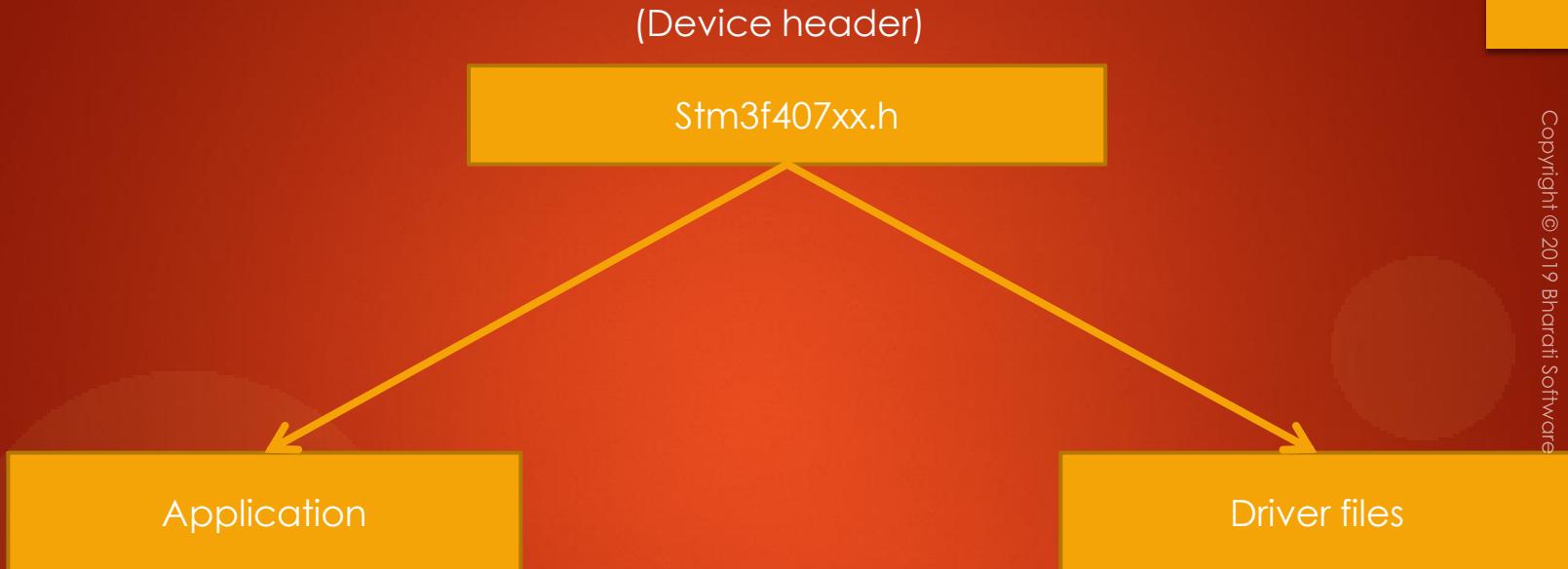
UART

STM3F407x MCU

# What is Device Header file and what it contains ?

This is a header file ('C' header file in our case) which contains Microcontroller specific details such as

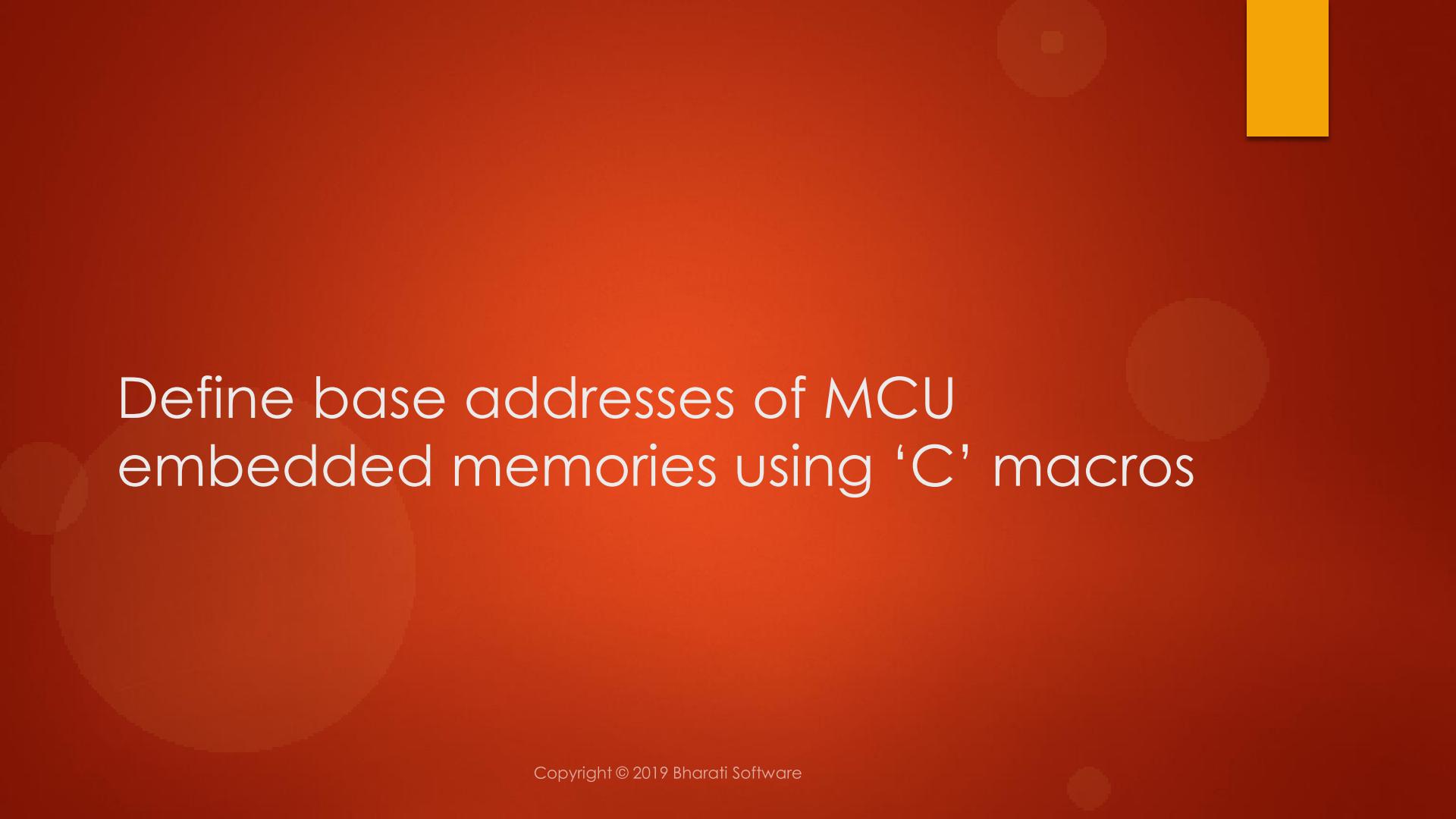
- 1) The base addresses of various memories present in the microcontroller such as (Flash, SRAM1,SRAM2,ROM,etc)
- 2) The base addresses of various bus domains such as (AHBx domain, APBx domain)
- 3) Base addresses of various peripherals present in different bus domains of the microcontroller
- 4) Clock management macros ( i.e clock enable and clock disable macros)
- 5) IRQ definitions
- 6) Peripheral Register definition structures
- 7) Peripheral register bit definitions
- 8) Other useful microcontroller configuration macros



Application and Driver source files can **#include** device specific header file to access MCU specific details

In the next lecture lets create  
MCU Device specific header file  
step by step

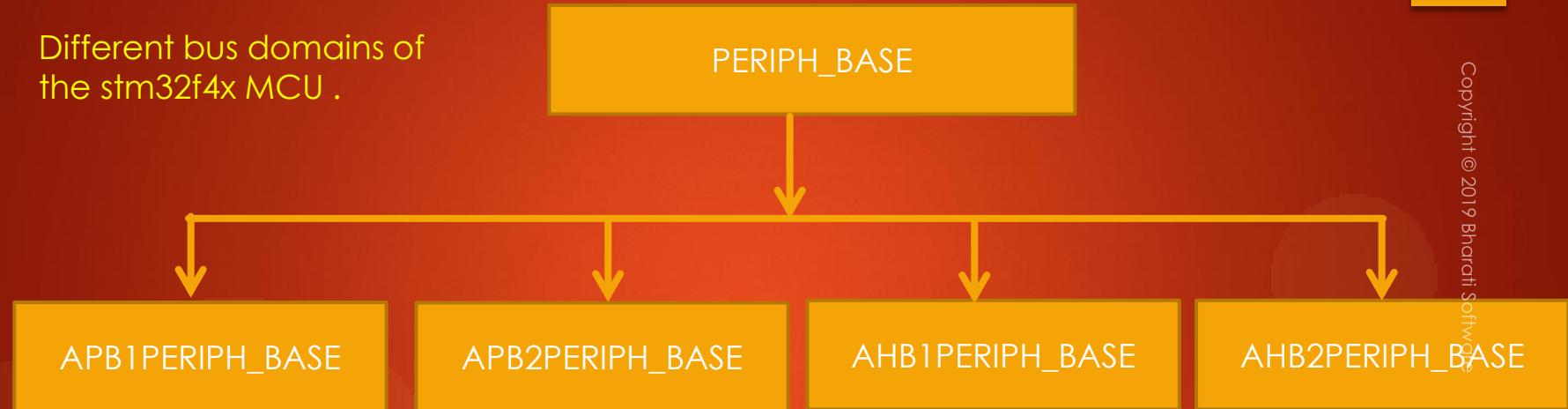
# New project creation



Define base addresses of MCU  
embedded memories using ‘C’ macros

# Defining base addresses of various bus domains (AHBx , APBx)

Different bus domains of the stm32f4x MCU .



- Different peripherals are hanging on different busses .
- AHB bus is used for those peripherals which need high speed data communication (ex. Camera interfaces, GPIOs)
- APB bus is used for those peripherals for which low speed communication would suffice .

# Full Memory map of the MCU

(APB1PERIPH\_BASE)



# Full Memory map of the MCU

Find out 0x4000\_0000 is the address of which register of which peripheral ????

(APB1PERIPH\_BASE)

32 bits wide  
32 bits wide  
32 bits wide

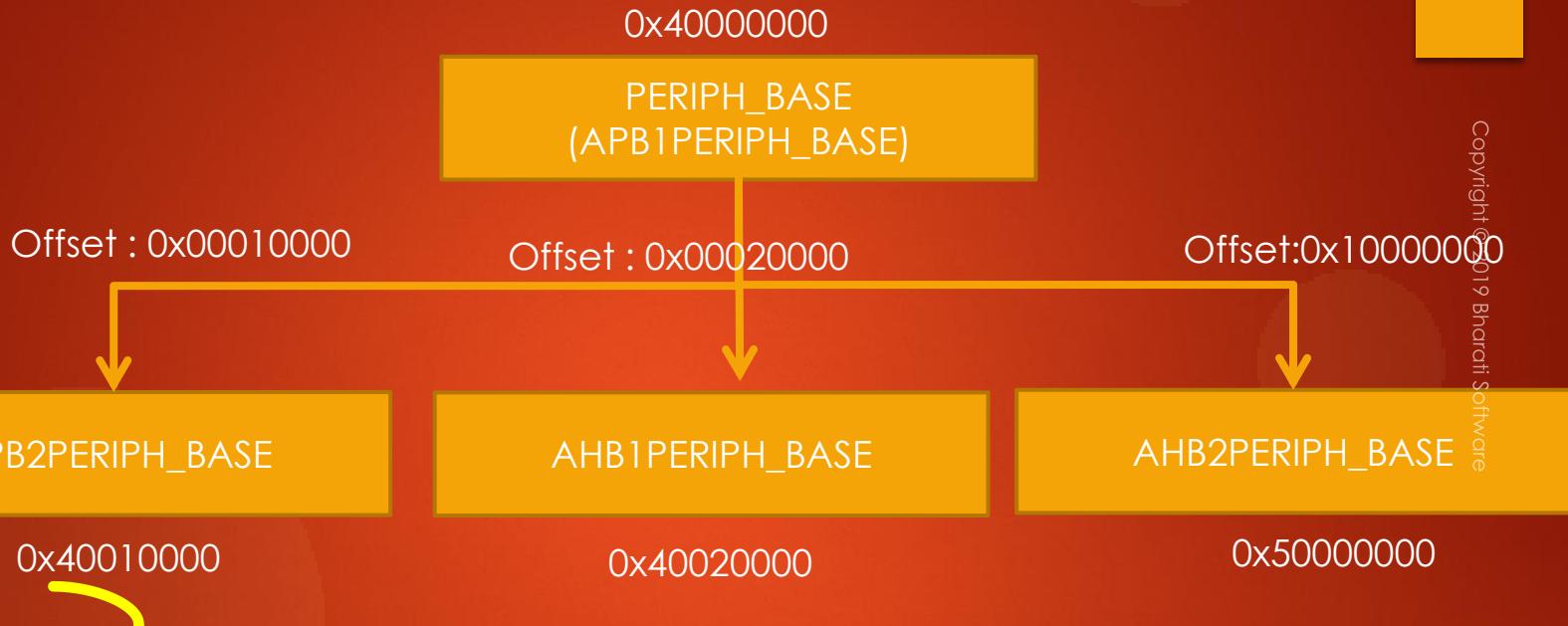
0xFFFF\_FFFF

0x40000000 (PERIPH\_BASE)

0x0000\_0004 ]  
0x0000\_0000 ]

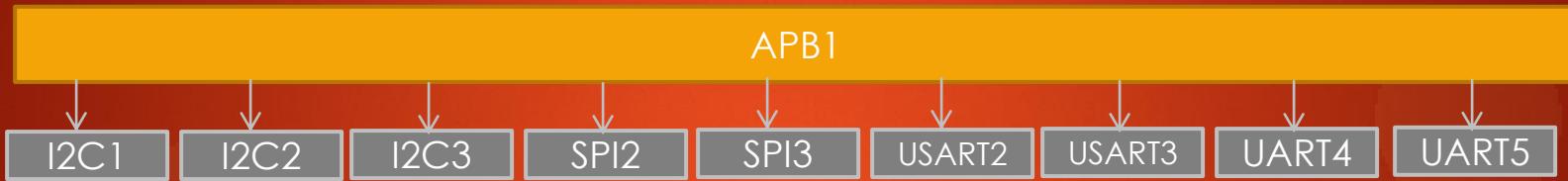
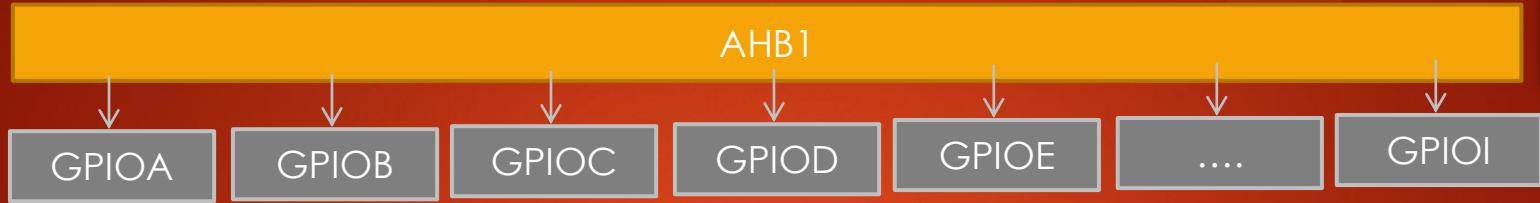
Offset between 2  
memory addresses is 4  
bytes

**Figure shows Peripheral base addresses of different bus domains  
In stm32f407xx MCU**



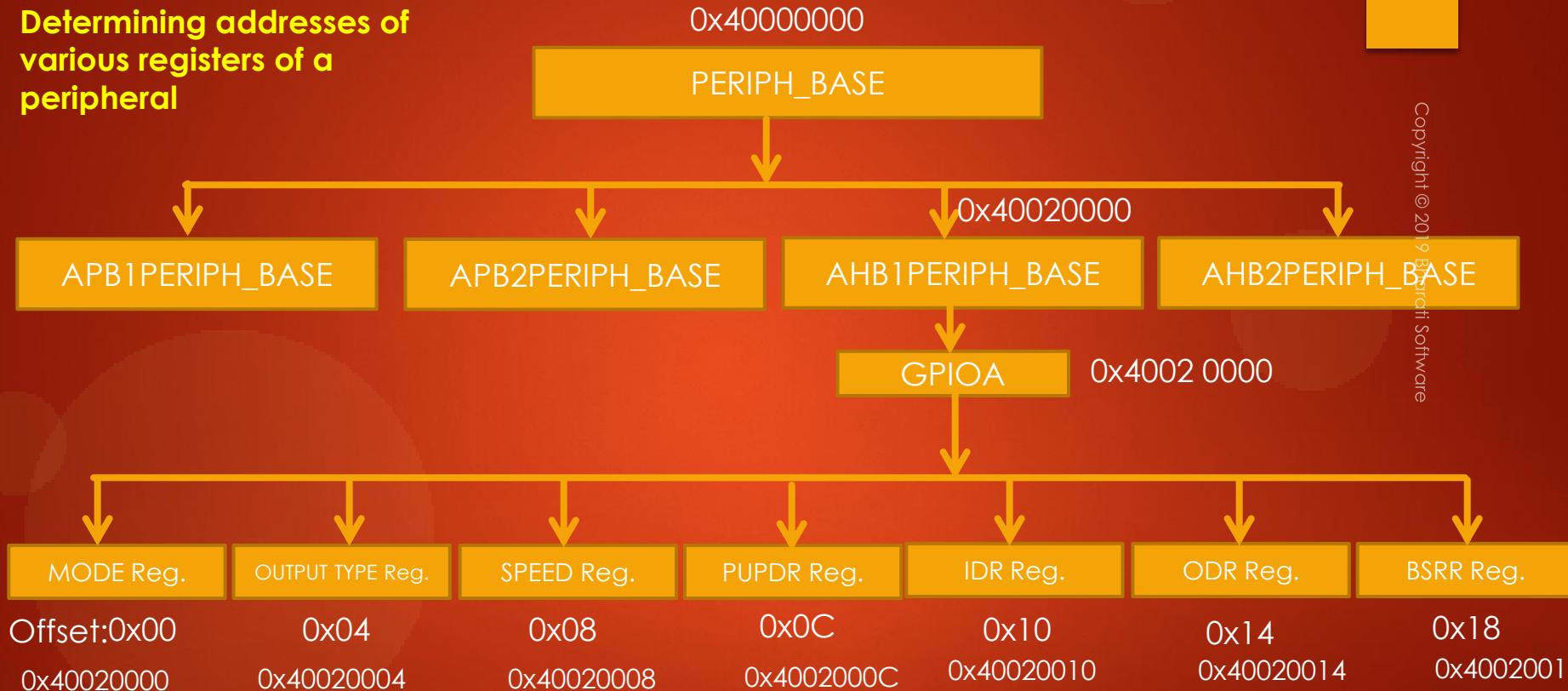
This simply means that registers of peripherals hanging on APB2 bus will appear on address 0x40010000 onwards in the memory map of the MCU.

# Defining base addresses of AHB1 peripherals, APB1 and APB2 peripherals



# Addresses of the peripheral registers

## Determining addresses of various registers of a peripheral



# Exercise:

Find out the addresses of below registers of SPI1 peripheral .

Register Name	Address
<b>control register 1</b>	??
<b>control register 2</b>	??
<b>status register</b>	??
<b>data register</b>	??
<b>CRC polynomial register</b>	??
<b>RX CRC register</b>	??
<b>TX CRC register</b>	??
<b>configuration register</b>	??
<b>prescaler register</b>	??

# Structuring Peripheral register details

# Example : 'C' Structure for registers of GPIO peripheral

Copyright © 2018

```
/*
 * peripheral register definition structure for GPIO
 */
typedef struct
{
    uint32_t MODER;           /*!< Give a short description,          Address offset: 0x00 */
    uint32_t OTYPER;          /*!< TODO,                           Address offset: 0x04 */
    uint32_t OSPEEDR;         /*!< TODO,                           Address offset: 0x08 */
    uint32_t PUPDR;           /*!< TODO,                           Address offset: 0x0C */
    uint32_t IDR;             /*!< TODO,                           Address offset: 0x10 */
    uint32_t ODR;             /*!< TODO,                           Address offset: 0x14 */
    uint32_t BSRRL;           /*!< TODO,                           Address offset: 0x18 */
    uint32_t BSRRH;           /*!< TODO,                           Address offset: 0x1A */
    uint32_t LCKR;            /*!< TODO,                           Address offset: 0x1C */
    uint32_t AFR[2];          /*!< TODO,                           Address offset: 0x20-0x24 */
} GPIO_RegDef_t;
```

Variable(Place holder) to hold values for GPIOx\_MODE register and so on

```

/*
 * peripheral register definition structure for GPIO
 */
typedef struct
{
    uint32_t MODER; /*!< Give a short description,
    uint32_t OTYPER; /*!< TODO,
    uint32_t OSPEEDR; /*!< TODO,
    uint32_t PUPDR; /*!< TODO,
    uint32_t IDR; /*!< TODO,
    uint32_t ODR; /*!< TODO,
    uint32_t BSRRL; /*!< TODO,
    uint32_t BSRRH; /*!< TODO,
    uint32_t LCKR; /*!< TODO,
    uint32_t AFR[2]; /*!< TODO,
} GPIO_RegDef_t;

GPIO_RegDef_t *pGPIOA = (GPIO_RegDef_t*)0x40020000;

```

Address of MODER variable(&MODER) will be (0x40020000+0x00)  
 Address of OTYPER variable(&OTYPER ) will be (0x40020000+0x04) and so on

Address offset: 0x00	*/
Address offset: 0x04	*/
Address offset: 0x08	*/
Address offset: 0x0C	*/
Address offset: 0x10	*/
Address offset: 0x14	*/
Address offset: 0x18	*/
Address offset: 0x1A	*/
Address offset: 0x1C	*/
Address offset: 0x20-0x24	*/

Base address of GPIOA

```

//Helps programmers for easy access to various registers of the peripherals
pGPIOA->MODER = 25; //storing value 25 in to MODER register
*(0x40020000+0x00) = 25; //this is how compiler does
pGPIOA->ODR = 44; //storing value 44 in to OD register
*(0x40020000+0x14) = 44; //this is how compiler does

```

```
/*
 * peripheral register definition structure for GPIO
 */
typedef struct
{
    uint32_t MODER;      /*!< Give a short description,
    uint32_t OTYPER;    /*!< TODO,
    uint32_t OSPEEDR;   /*!< TODO,
    uint32_t PUPDR;     /*!< TODO,
    uint32_t IDR;       /*!< TODO,
    uint32_t ODR;       /*!< TODO,
    uint32_t BSRRL;     /*!< TODO,
    uint32_t BSRRH;     /*!< TODO,
    uint32_t LCKR;      /*!< TODO,
    uint32_t AFR[2];    /*!< TODO,
} GPIO_RegDef_t;

GPIO_RegDef_t *pGPIOA = (GPIO_RegDef_t*)0x40020000;
```

Base address of GPIOA



Address offset: 0x00	*/
Address offset: 0x04	*/
Address offset: 0x08	*/
Address offset: 0x0C	*/
Address offset: 0x10	*/
Address offset: 0x14	*/
Address offset: 0x18	*/
Address offset: 0x1A	*/
Address offset: 0x1C	*/
Address offset: 0x20-0x24	*/

```
/*
 * peripheral register definition structure for GPIO
 */
typedef struct
{
    uint32_t MODER;      /*!< Give a short description,
    uint32_t OTYPER;     /*!< TODO,
    uint32_t OSPEEDR;    /*!< TODO,
    uint32_t PUPDR;      /*!< TODO,
    uint32_t IDR;        /*!< TODO,
    uint32_t ODR;        /*!< TODO,
    uint16_t BSRRL;      /*!< TODO,
    uint16_t BSRRH;      /*!< TODO,
    uint32_t LCKR;       /*!< TODO,
    uint32_t AFR[2];     /*!< TODO,
} GPIO_RegDef_t;
```

```
GPIO_RegDef_t *pGPIOA = GPIOA;
```

Base address of GPIOA

```
Address offset: 0x00      */
Address offset: 0x04      */
Address offset: 0x08      */
Address offset: 0x0C      */
Address offset: 0x10      */
Address offset: 0x14      */
Address offset: 0x18      */
Address offset: 0x1A      */
Address offset: 0x1C      */
Address offset: 0x20-0x24 */
```

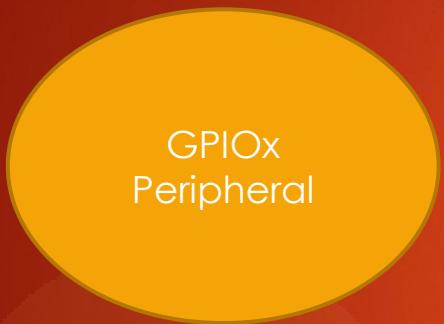
```
#define GPIOA ((GPIO_RegDef_t*)GPIOA_BASE)
```

# Peripheral clock enable and disable macros

# Macros for IRQ(Interrupt Request) Numbers of the MCU

# Creating GPIO Driver .c and .h files

# GPIO Handle and Configuration structures

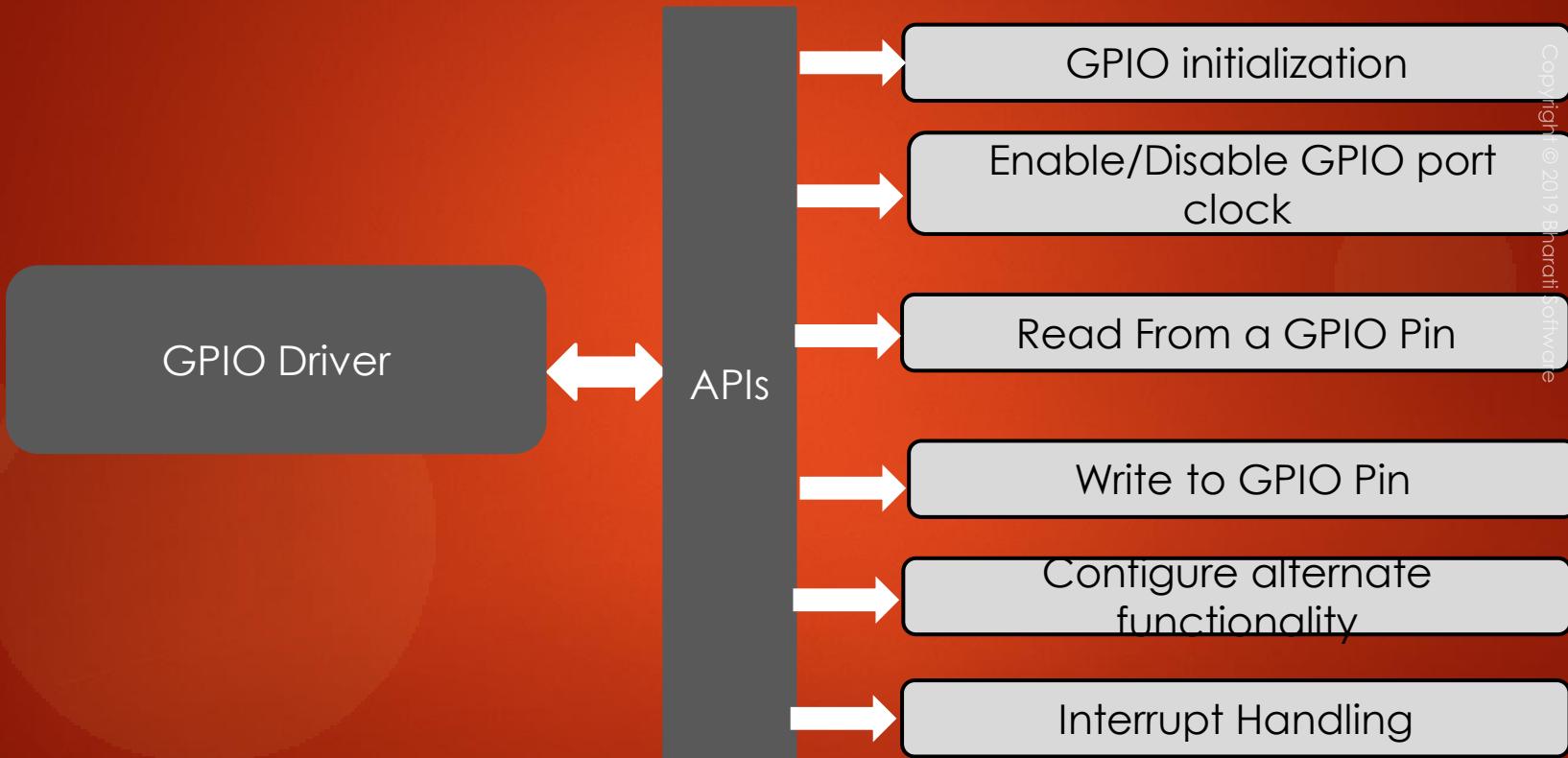


- GPIO Port Name
- GPIO Pin number
- GPIO mode
- GPIO speed
- GPIO outputtype
- GPIO Pullup-pulldown
- GPIO Alt.function mode

Configurable items  
For user application

# GPIO Driver APIs requirements

# Driver API Requirements

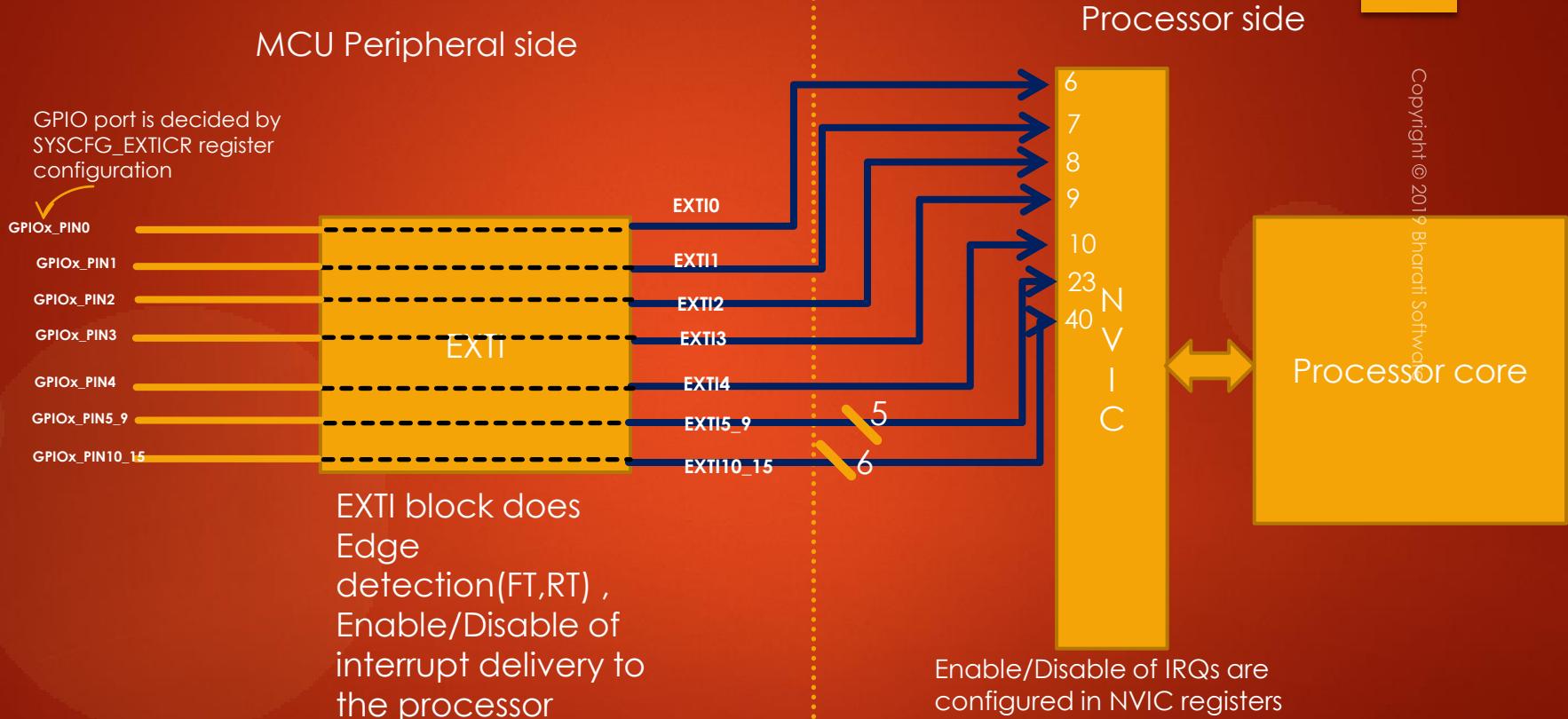


# GPIO Pin Interrupt Configuration

Copyright © 2019 Bharati Software

1. Pin must be in input configuration
2. Configure the edge trigger (RT,FT,RFT)
3. Enable interrupt delivery from peripheral to the processor (on peripheral side)
4. Identify the IRQ number on which the processor accepts the interrupt from that pin
5. Configure the IRQ priority for the identified IRQ number (Processor side)
6. Enable interrupt reception on that IRQ number (Processor side)
7. Implement IRQ handler

# STM32f4x GPIO Pins interrupt delivery to the Processor



### 9.3.3 SYSCFG external interrupt configuration register 1 (SYSCFG\_EXTICR1)

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI3[3:0]				EXTI2[3:0]				EXTI1[3:0]				EXTI0[3:0]			
rw	rw	rw	rw												

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **EXTIx[3:0]**: EXTI x configuration (x = 0 to 3)

These bits are written by software to select the source input for the EXTIx external interrupt.

0000: PA[x] pin

0001: PB[x] pin

0010: PC[x] pin

0011: PD[x] pin

0100: PE[x] pin

0101: PF[x] pin

0110: PG[x] pin

### 9.3.4 SYSCFG external interrupt configuration register 2 (SYSCFG\_EXTICR2)

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI7[3:0]				EXTI6[3:0]				EXTI5[3:0]				EXTI4[3:0]			
rw	rw	rw	rw												

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **EXTIx[3:0]**: EXTI x configuration (x = 4 to 7)

These bits are written by software to select the source input for the EXTIx external interrupt.

0000: PA[x] pin

0001: PB[x] pin

0010: PC[x] pin

0011: PD[x] pin

0100: PE[x] pin

### 9.3.6 SYSCFG external interrupt configuration register 4 (SYSCFG\_EXTICR4)

Address offset: 0x14

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI15[3:0]				EXTI14[3:0]				EXTI13[3:0]				EXTI12[3:0]			
rw	rw	rw	rw												

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **EXTIx[3:0]**: EXTI x configuration (x = 12 to 15)

These bits are written by software to select the source input for the EXTIx external interrupt.

0000: PA[x] pin

0001: PB[x] pin

0010: PC[x] pin

0011: PD[x] pin

0100: PE[x] pin

# Interrupt Set-enable Registers

Copyright ©

NVIC\_ISERO

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IRQ31															IRQ16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRQ15						IRQ9		.				IRQ2	IRQ1	IRQ0	

0 has no effect

1 enables the interrupt

# Interrupt Set-enable Registers

Copyright ©

NVIC\_ISER1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IRQ63															IRQ48
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRQ47						IRQ41		.				IRQ34	IRQ33	IRQ32	

0 has no effect

1 enables the interrupt

# Interrupt Set-enable Registers

Copyright ©

NVIC\_ISER2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IRQ95															IRQ80
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
								.					IRQ66	IRQ65	IRQ64

0 has no effect

1 enables the interrupt

# Interrupt Clear-enable Registers

Copyright ©

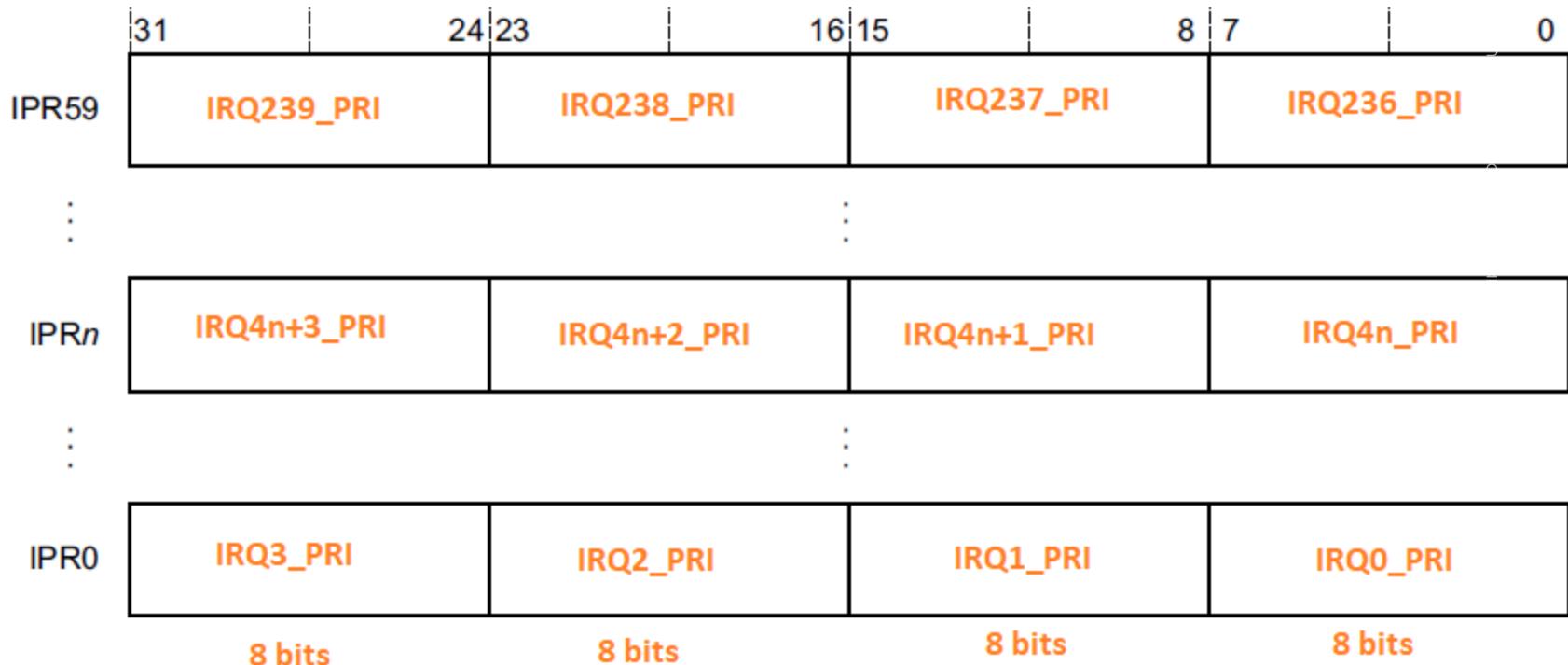
NVIC\_ICERO

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IRQ31															IRQ16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRQ15						IRQ9		.				IRQ2	IRQ1	IRQ0	

0 has no effect  
1 disable interrupt

# Interrupt Priority Registers

NVIC\_IPR0-NVIC\_IPR59



# Exercise

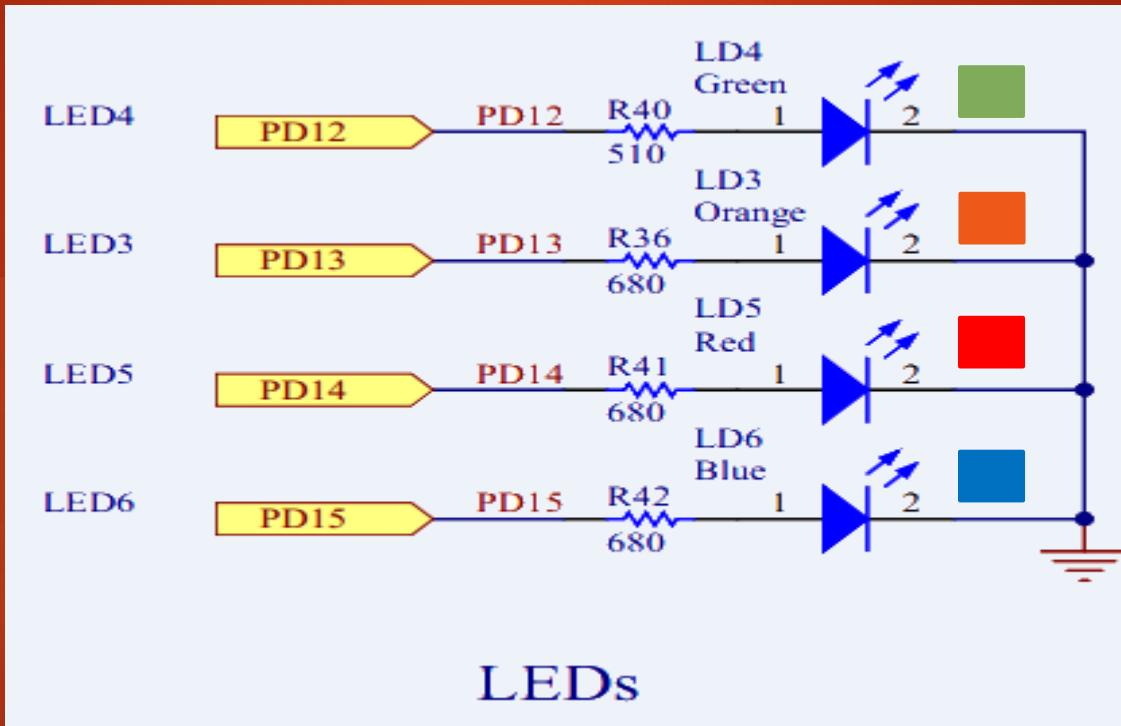
Write a program to toggle the on board LED with some delay .

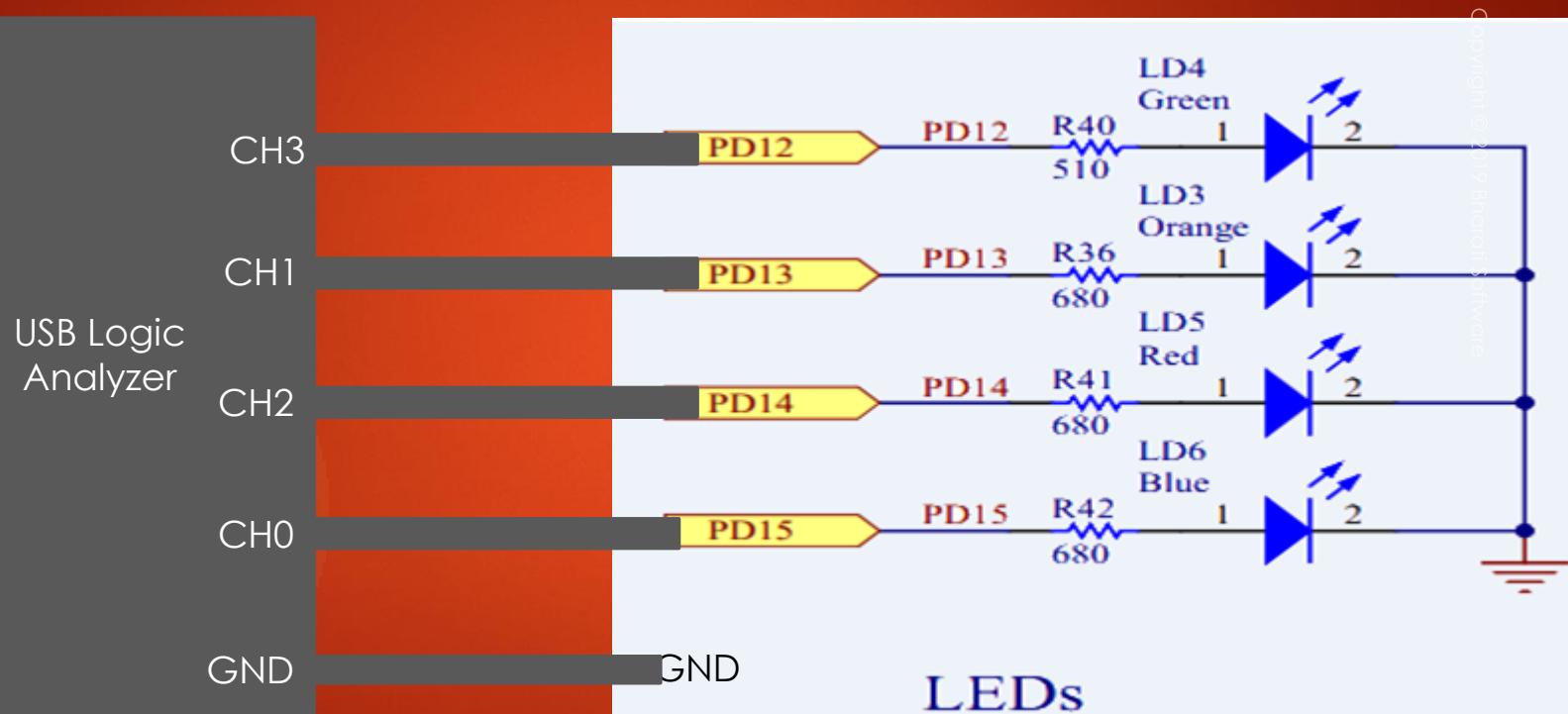
Case 1 : Use push pull configuration for the output pin

Case 2 : Use open drain configuration for the output pin

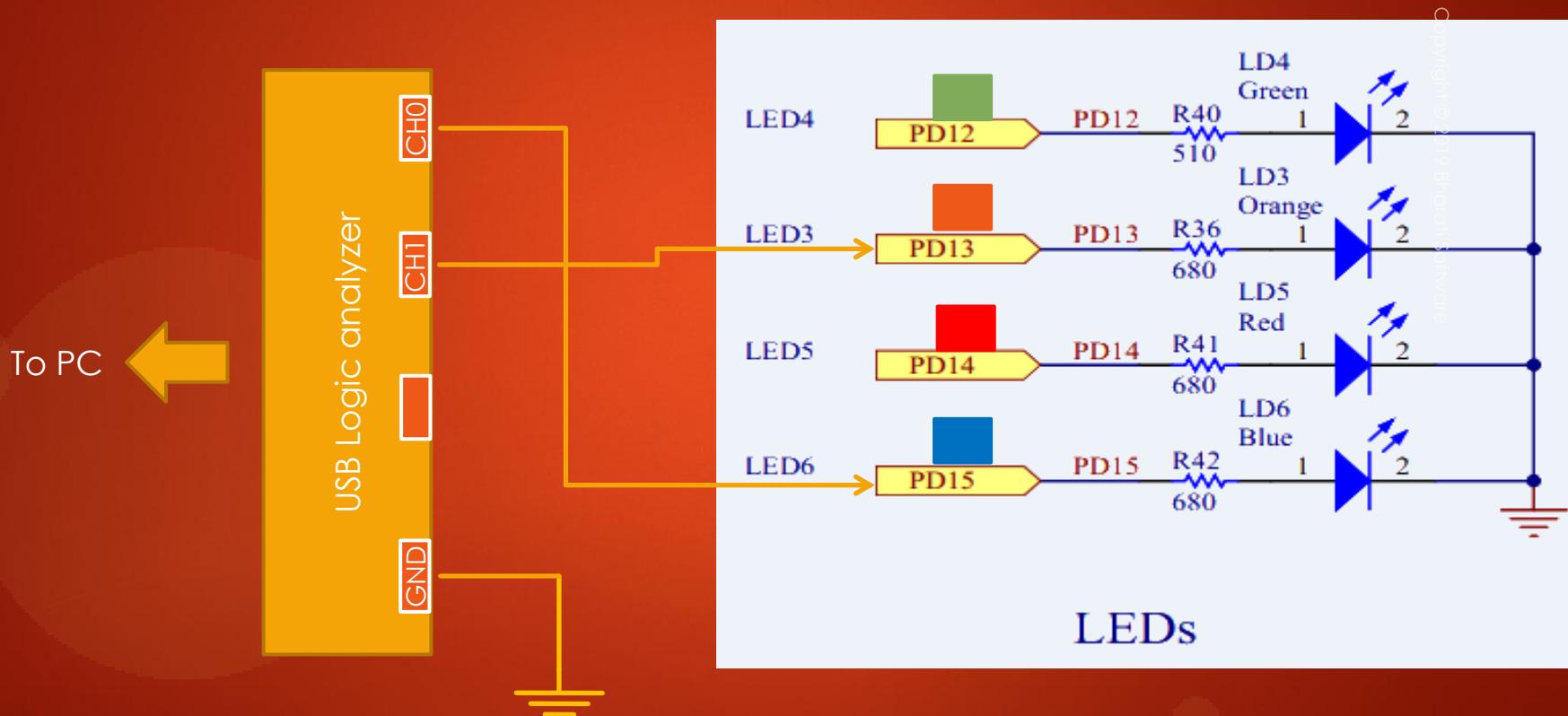
# Discovery Board LEDs

Copyright © 2019 Bharati Software



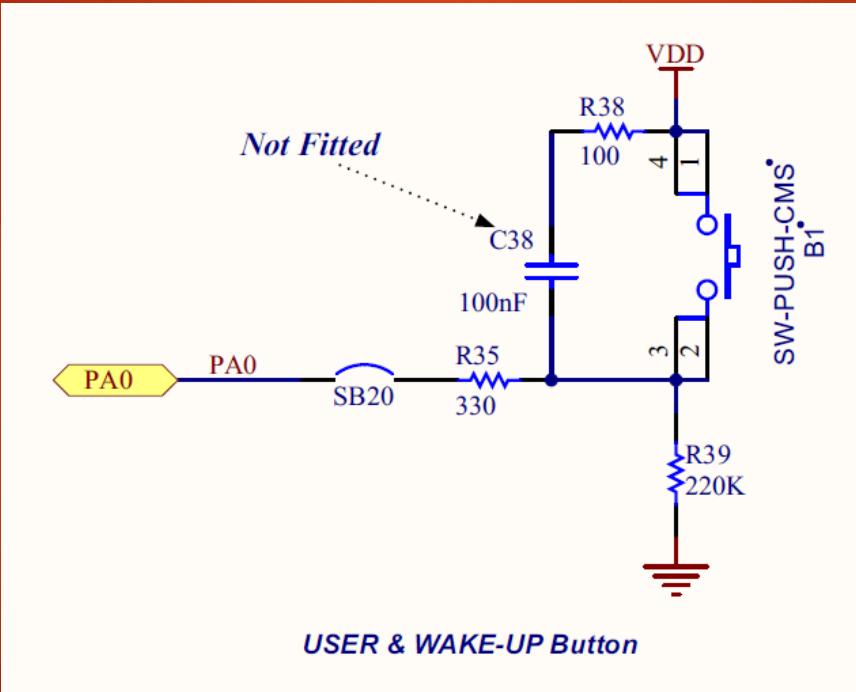


# Probing LED Toggling using USB Logic Analyzer



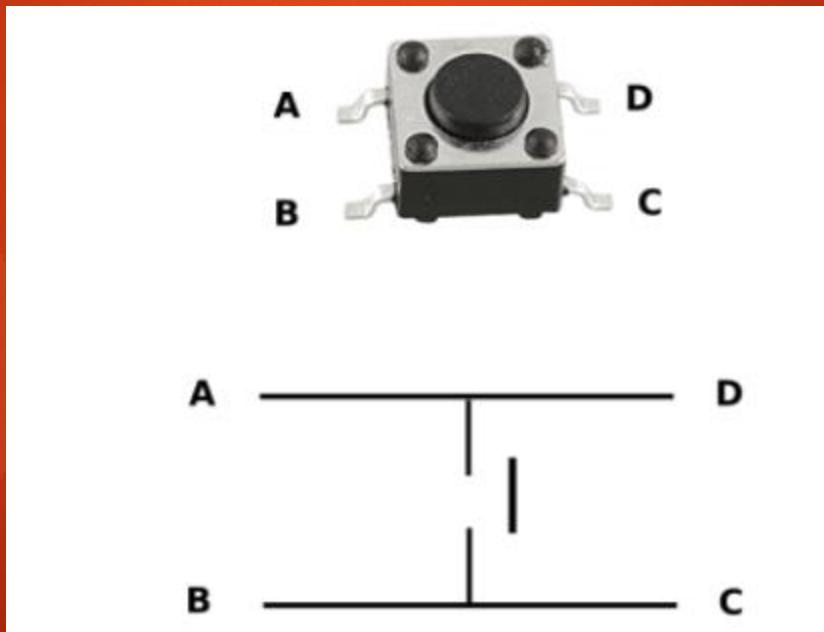
# Exercise

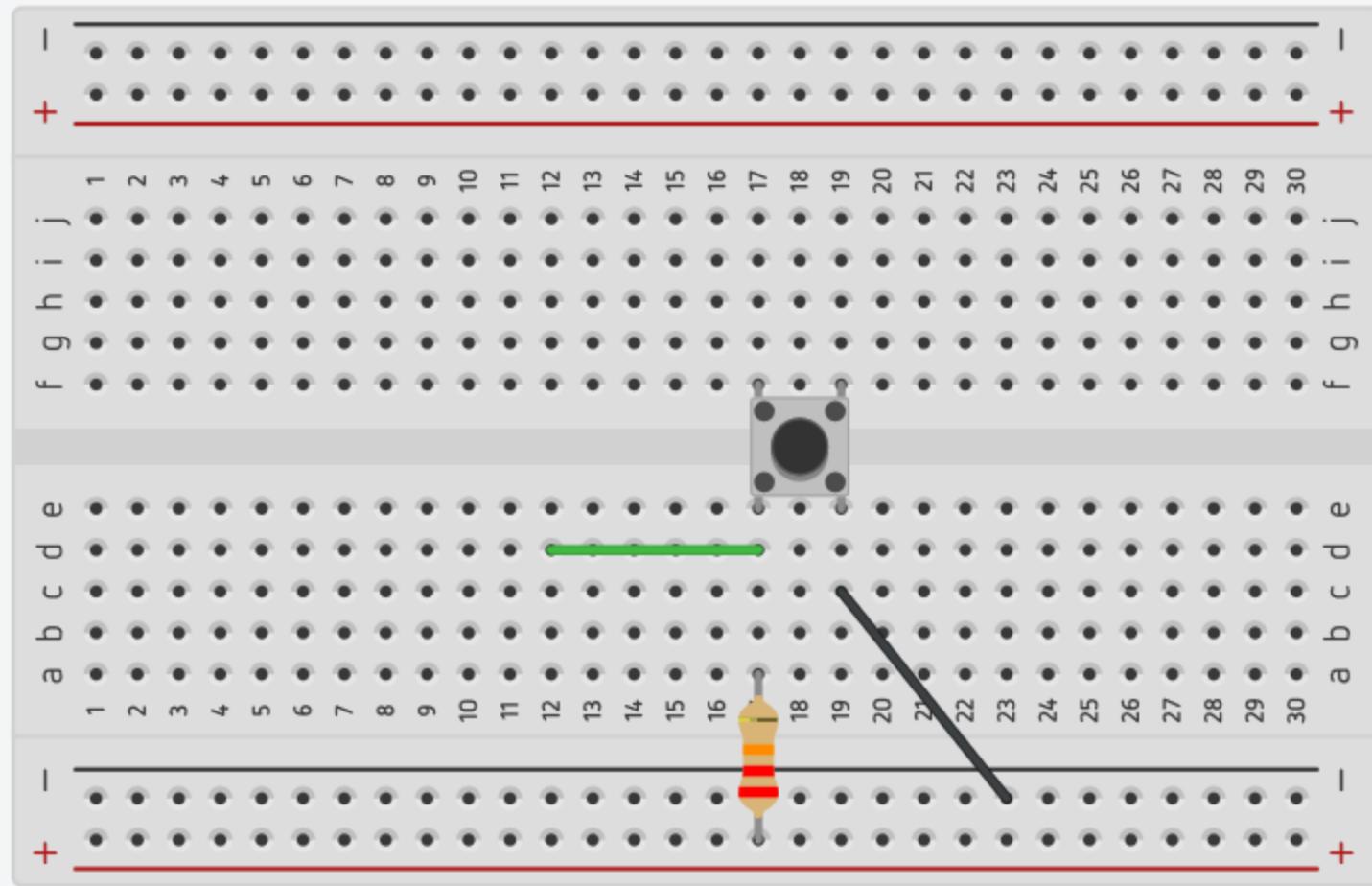
Write a program to toggle the on board LED whenever the on board button is pressed



# Exercise

Write a program to connect external button to the pin number PB12 and external LED to PA14  
Toggle the LED whenever the external button is pressed





# Exercise

Toggle a GPIO pin with no delay between  
GPIO high and GPIO low and measure the  
frequency of toggling using logic analyzer

# Exercise

Connect an external button to PD5 pin and toggle the led whenever interrupt is triggered by the button press.

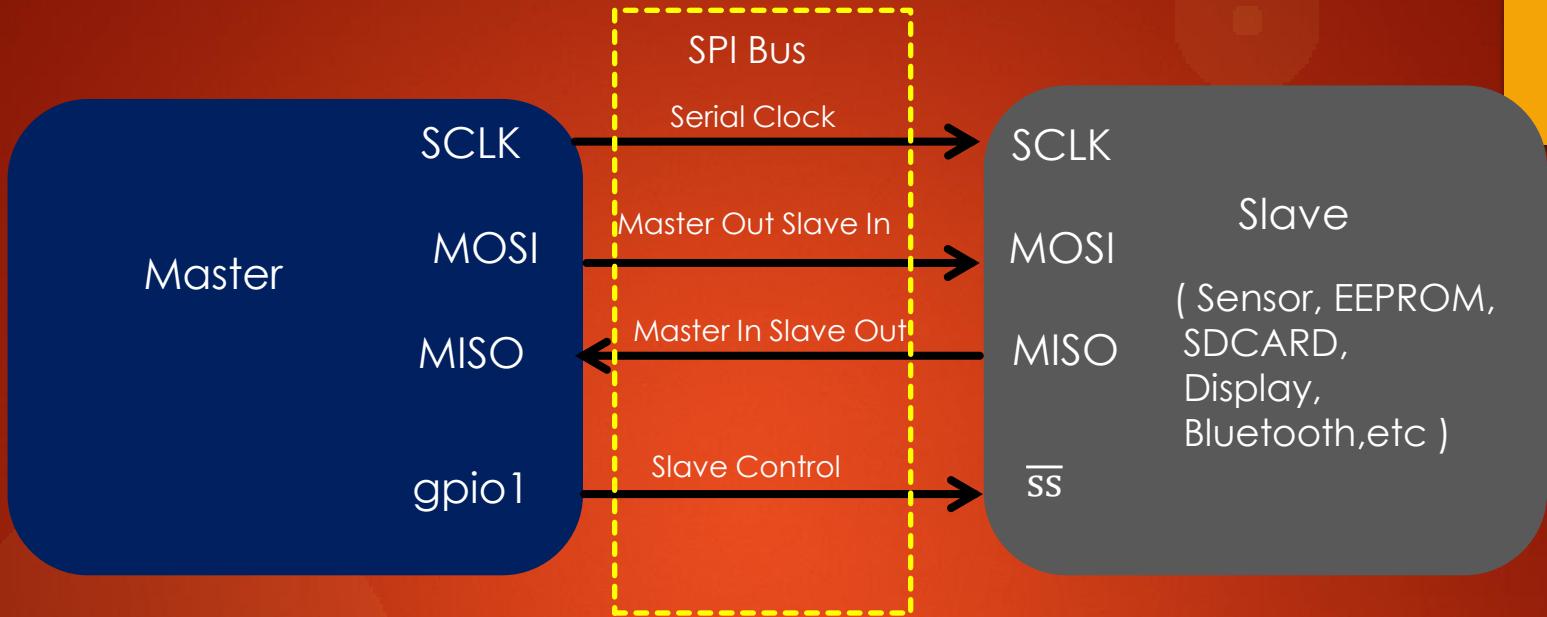
Interrupt should be triggered during falling edge of button press.



In the next lecture lets understand input mode of a pin

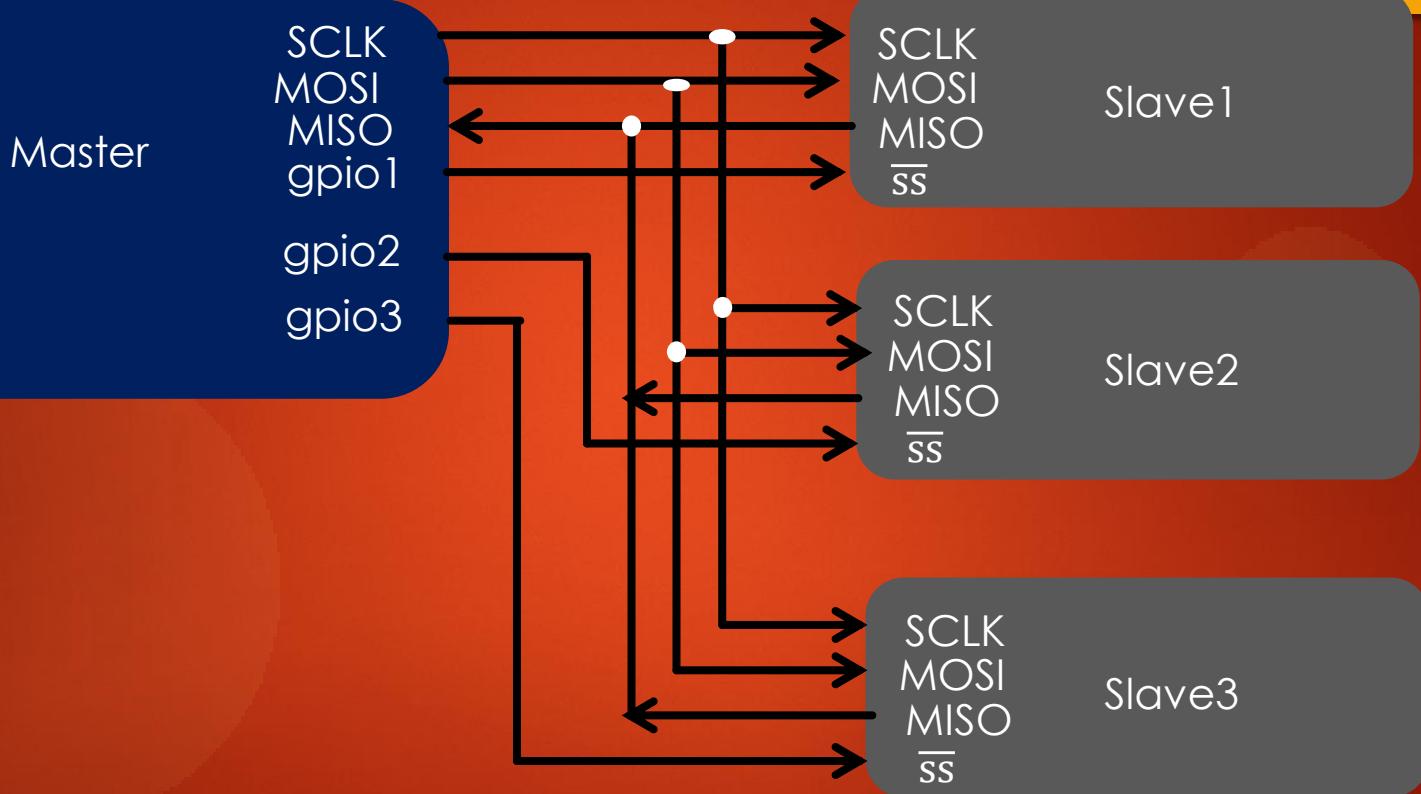
# Introduction to SPI

# Serial Peripheral Interface

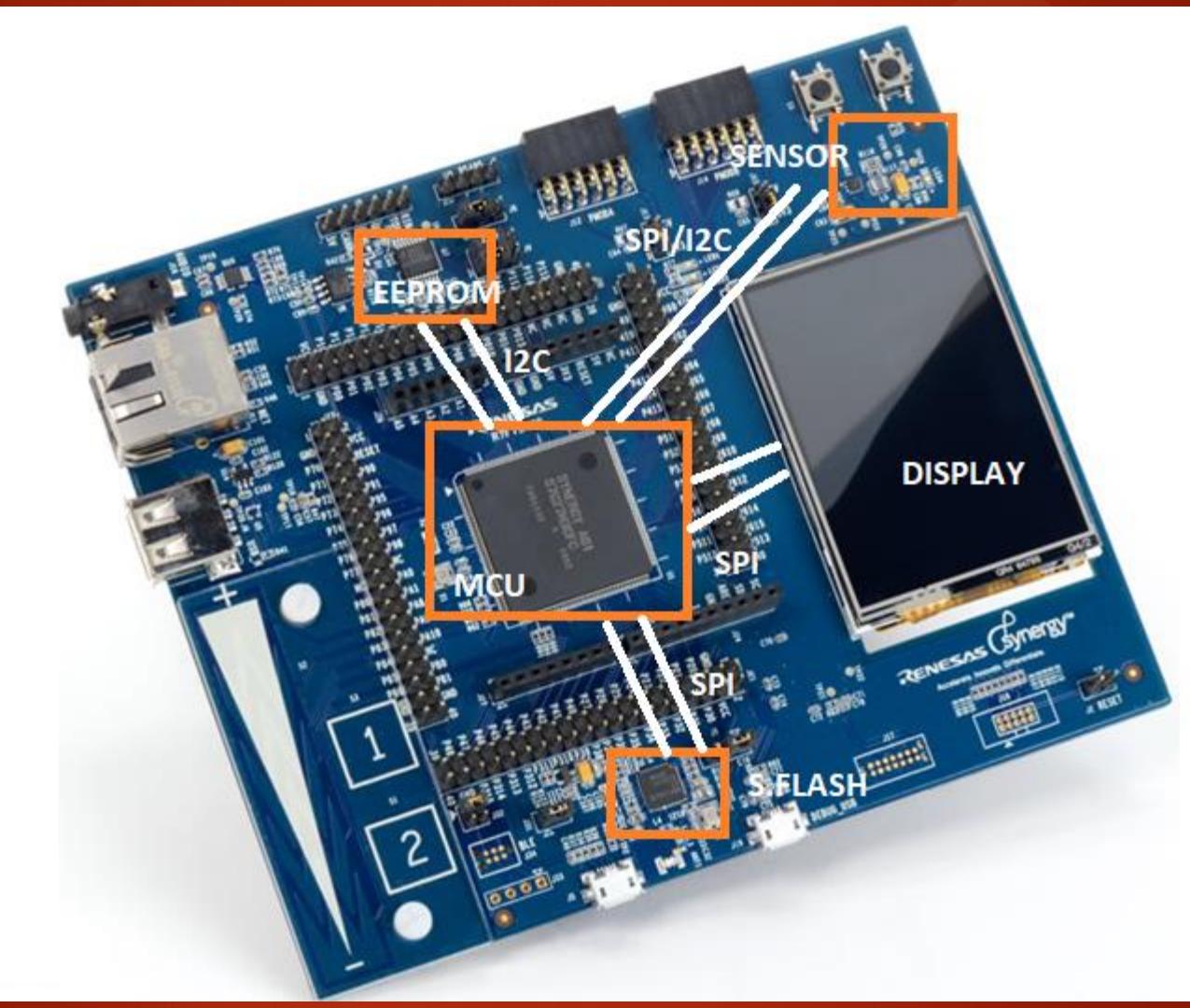


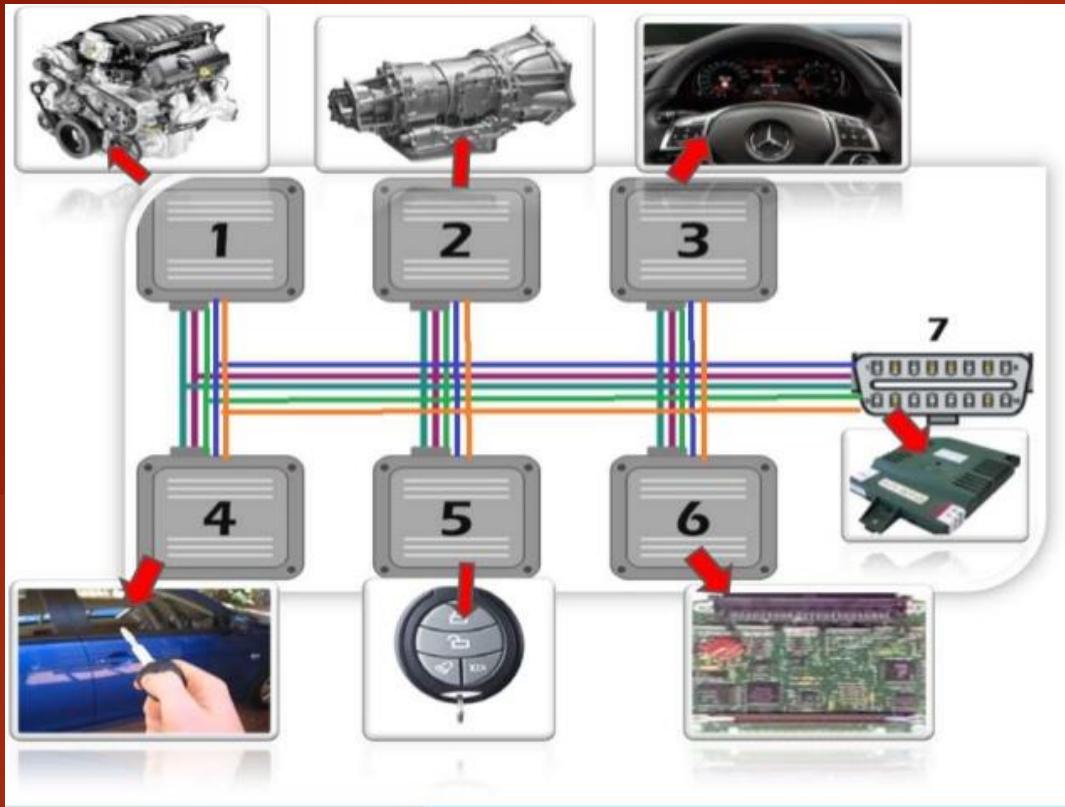


1. Four I/O pins are dedicated to SPI communication with external devices.
2. MISO: Master In / Slave Out data. In the general case, this pin is used to transmit data in slave mode and receive data in master mode
3. MOSI: Master Out / Slave In data. In the general case, this pin is used to transmit data in master mode and receive data in slave mode.
4. SCK: Serial Clock output pin for SPI master and input pin for SPI slaves.
5. NSS: Slave select pin. Depending on the SPI and NSS settings, this pin can be used to select an individual slave device for communication

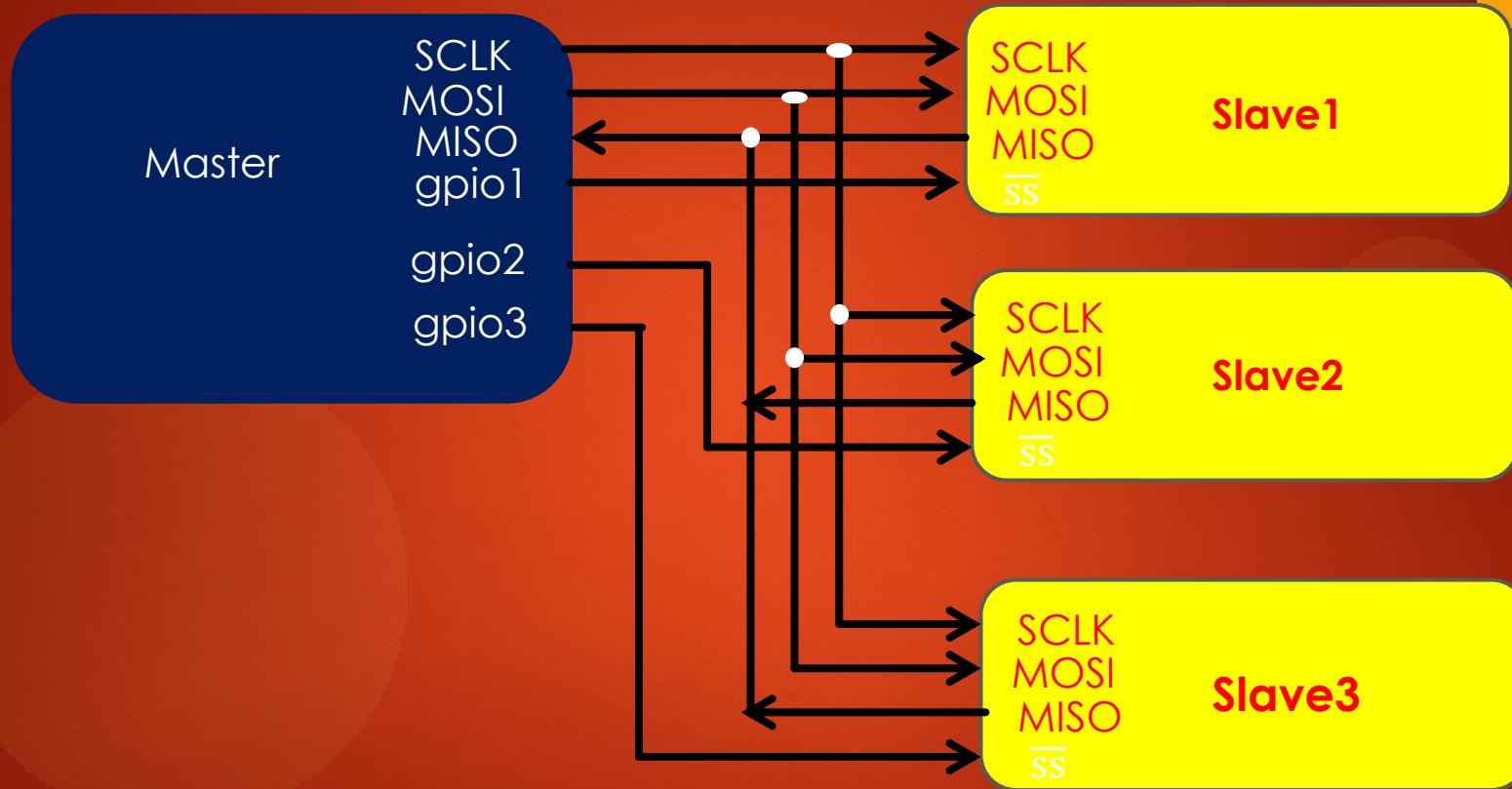


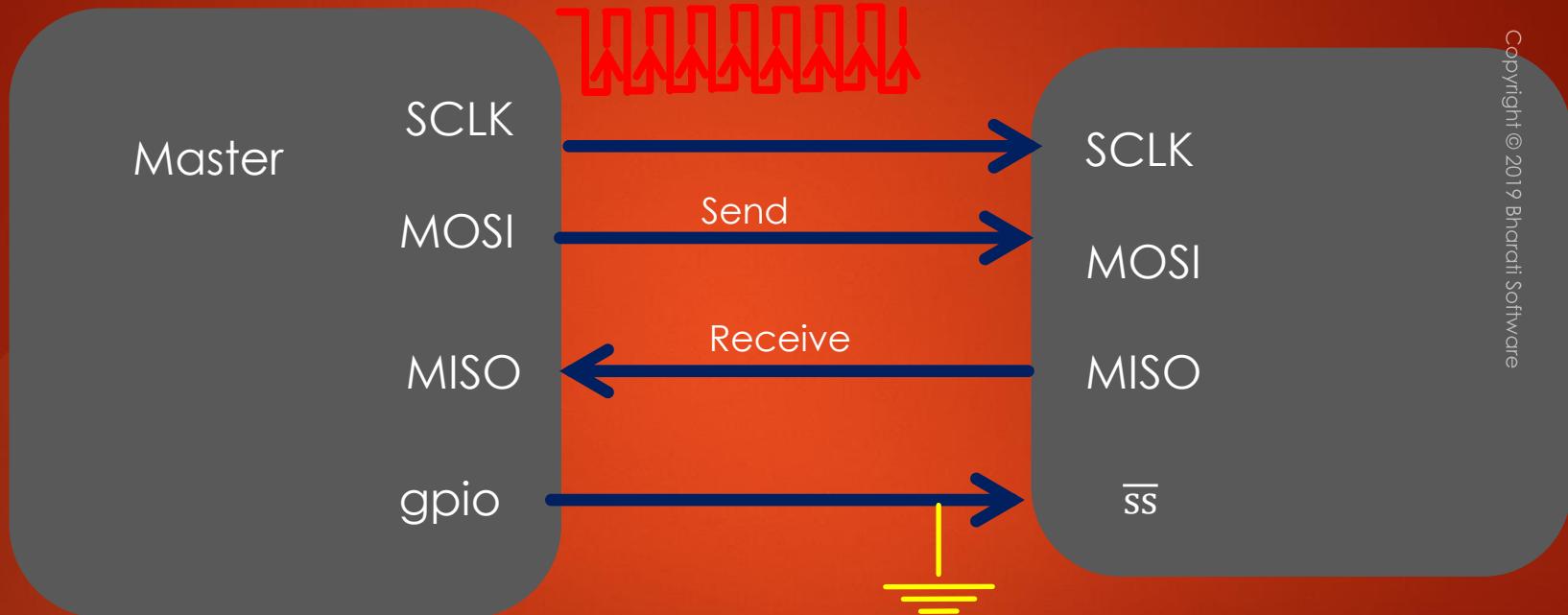
<b>Protocol</b>	<b>Type</b>	<b>Max distance(ft.)</b>	<b>Max Speed (bps)</b>	<b>Typical usage</b>
USB 3.0	dual simplex serial	9 (typical) (up to 49 with 5 hubs)	5 G	Mass storage, video
USB 2.0	half duplex serial	16 (98 ft. with 5 hubs)	1.5M, 12M, 480M	Keyboard, mouse, drive, speakers, printer, camera
Ethernet	serial	1600	10G	network communications
I2C	synchronous serial	18	3.4 M in High-speed mode.	Microcontroller communications
RS-232	asynchronous serial	50–100	20k	Modem, mouse, instrumentation
RS-485	asynchronous serial	4000	10M	Data acquisition and control systems
SPI	synchronous serial	10	fPCLK/2	





You can use CAN, ETHERNET , RS485, RS232 or combination of them , when you have to cover larger distances and want to achieve better quality of service.





# MINIMAL SPI BUS

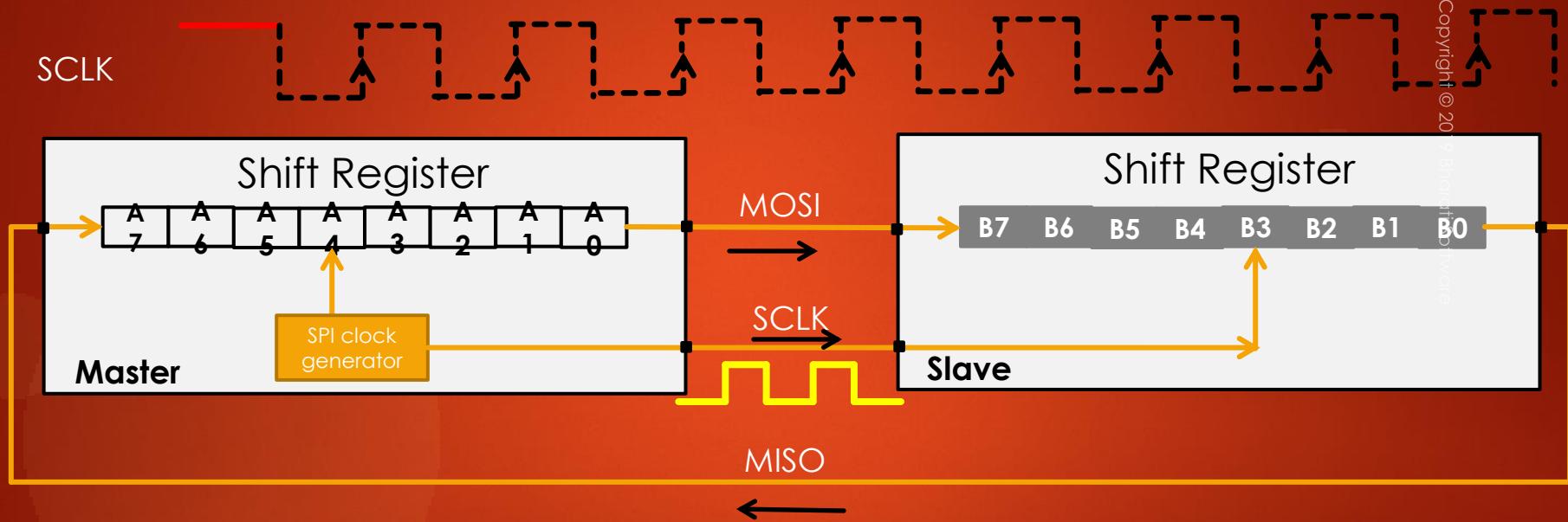
The SPI bus allows the communication between one master device and one or more slave devices. In some applications SPI bus may consists of just two wires - one for the clock signal and the other for synchronous data transfer. Other signals can be added depending on the data exchange between SPI nodes and their slave select signal management.



# SPI Hardware :Behind the scenes

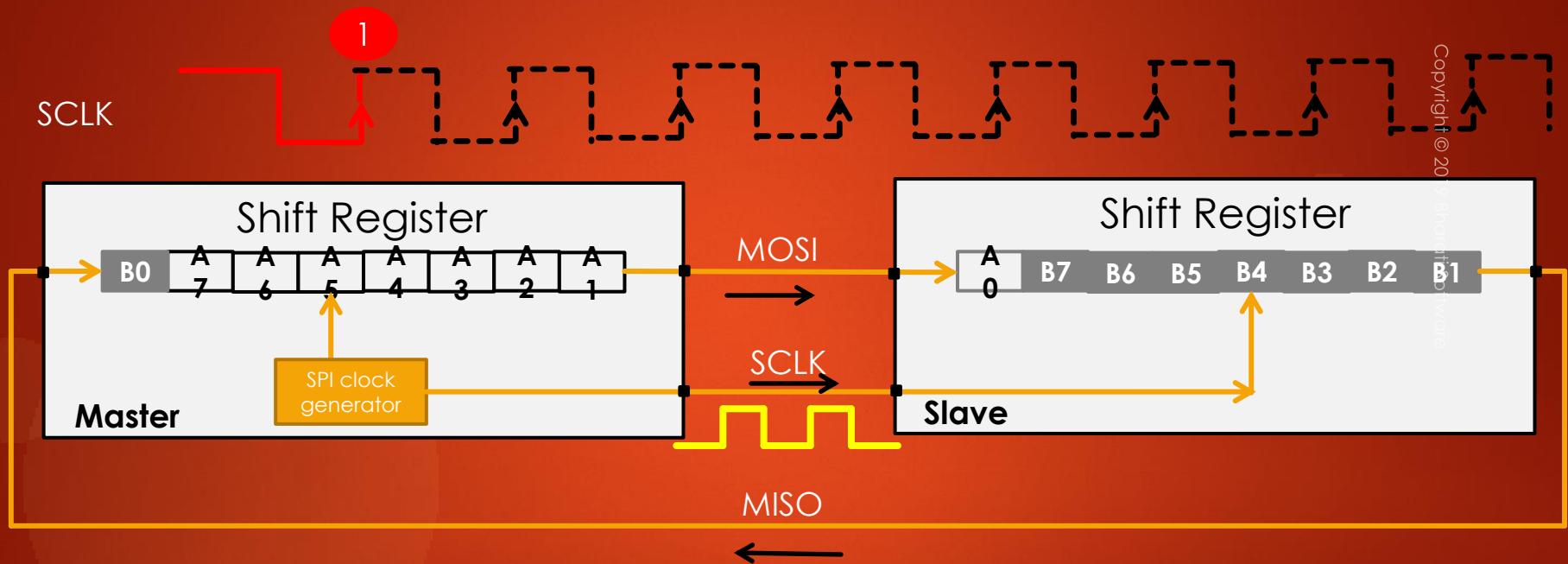
# SPI Hardware :Behind the scenes

Copyright © 2018 Silvano Moreira



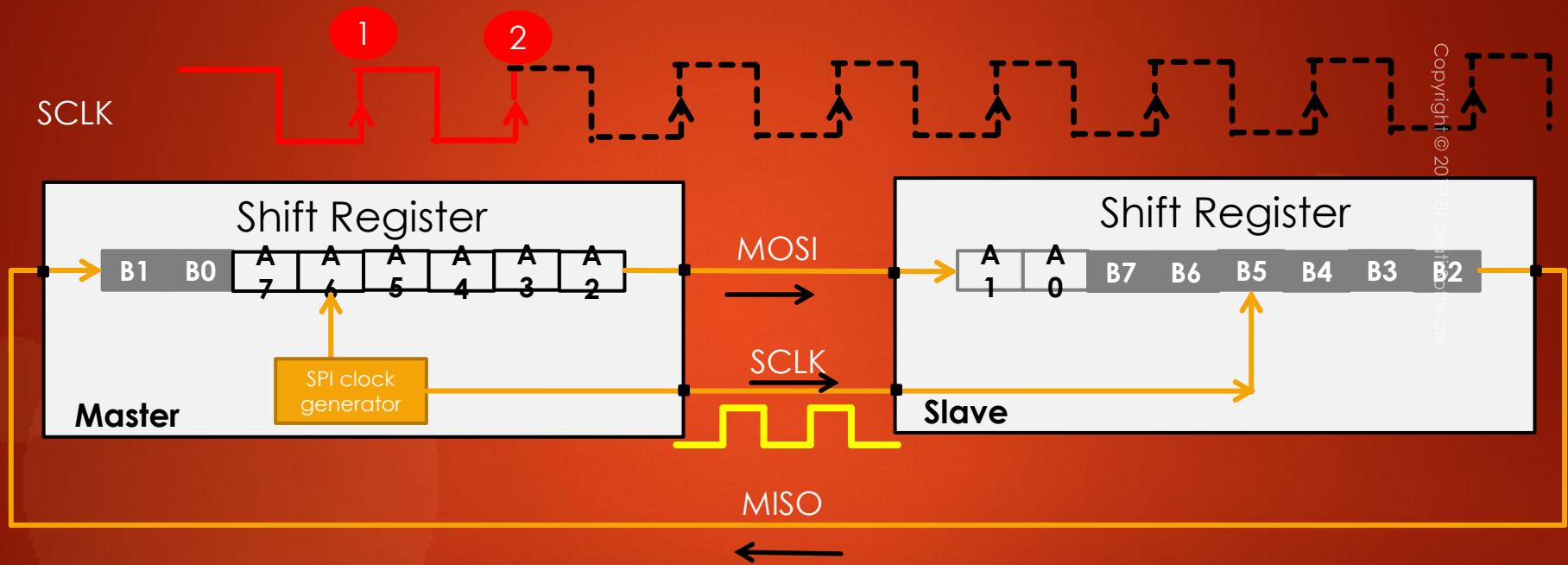
# SPI Hardware :Behind the scenes

Copyright © 2018 MikroElektronika



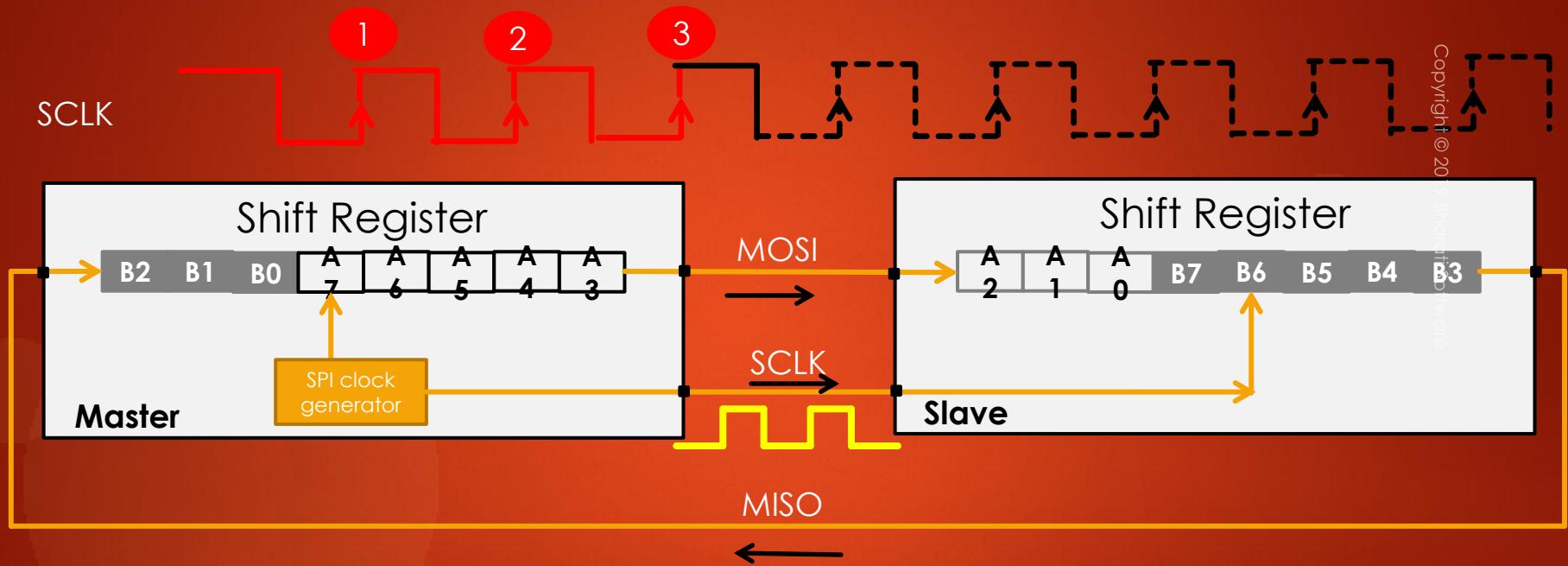
# SPI Hardware :Behind the scenes

Copyright © 2018 MikroElektronika



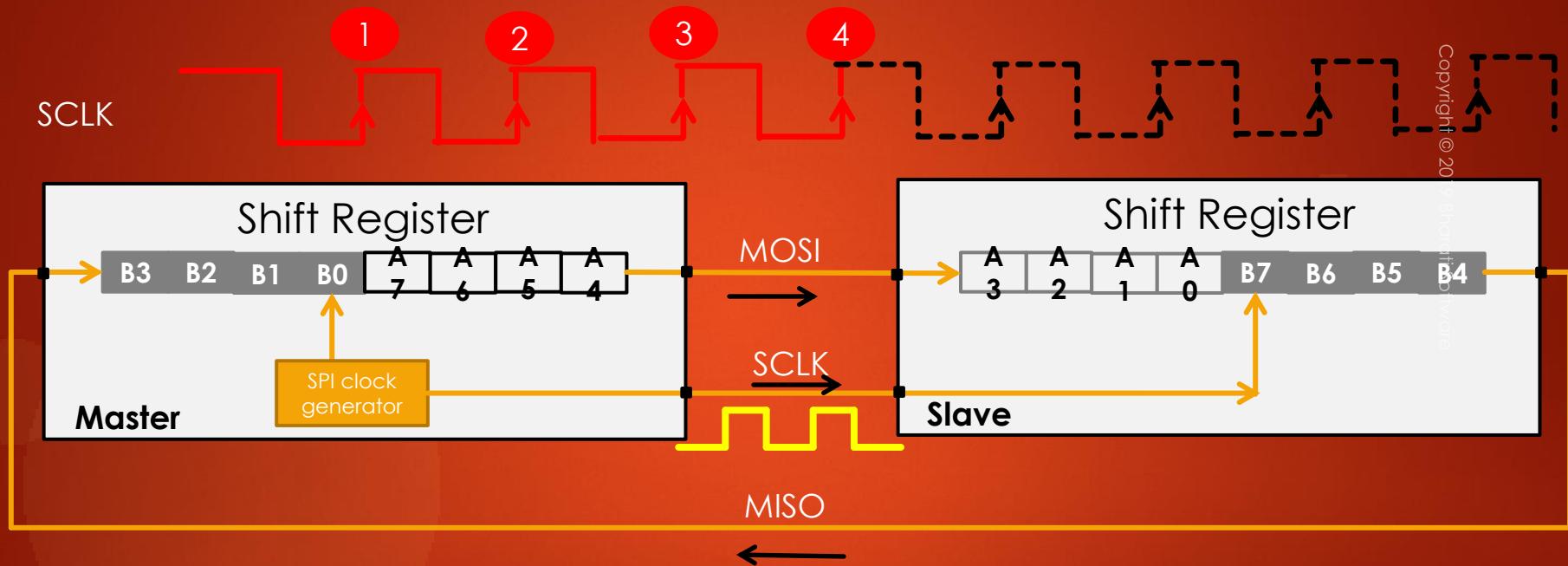
# SPI Hardware :Behind the scenes

Copyright © 2018 MikroElektronika



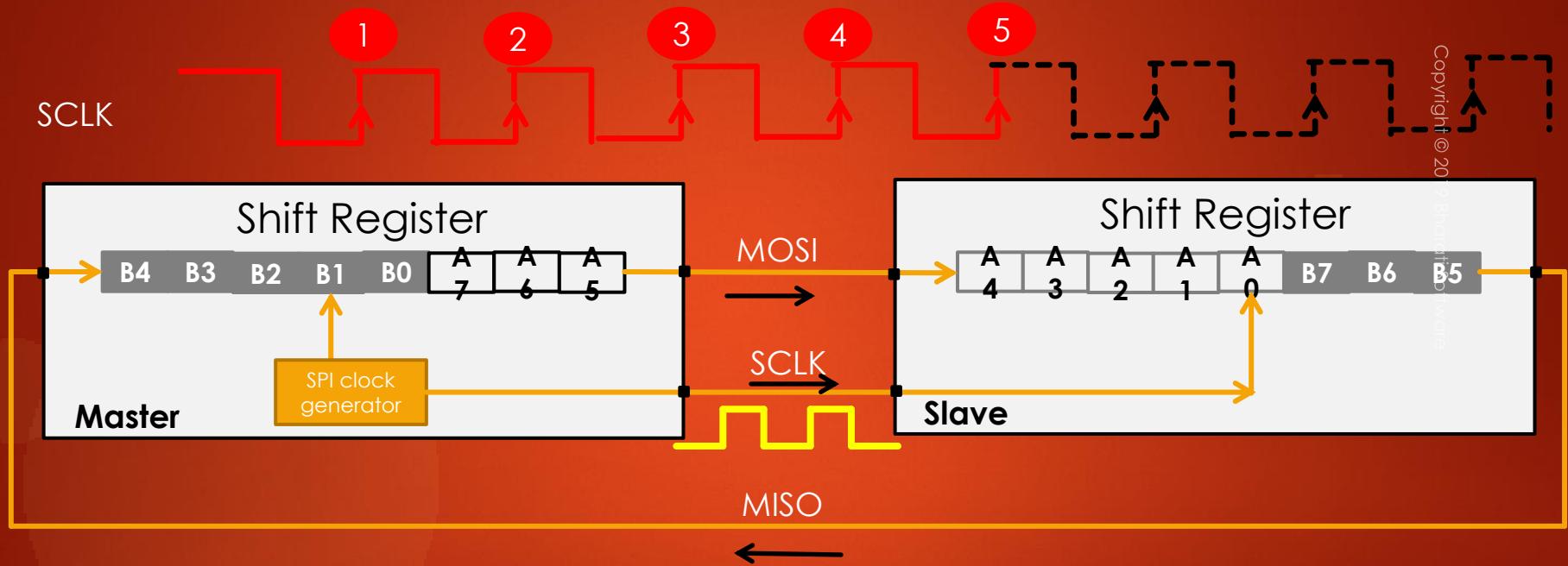
# SPI Hardware :Behind the scenes

Copyright © 2018 MikroElektronika



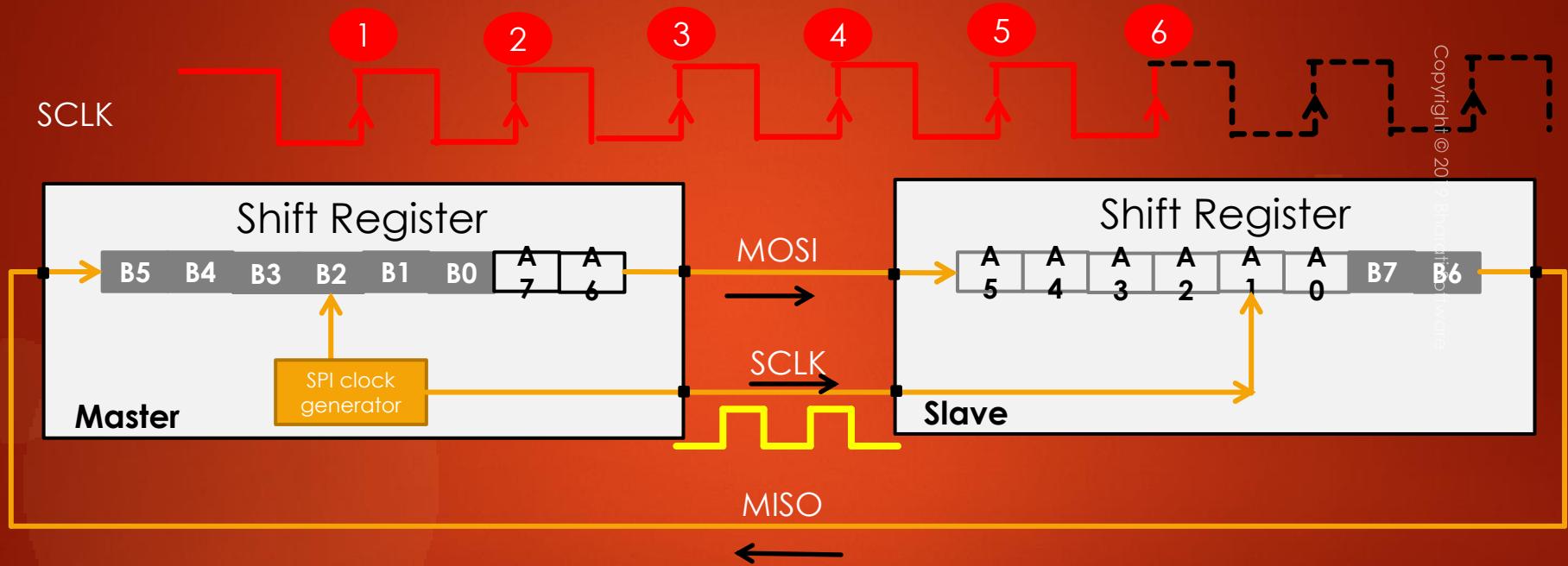
# SPI Hardware :Behind the scenes

Copyright © 2018 Edureka!

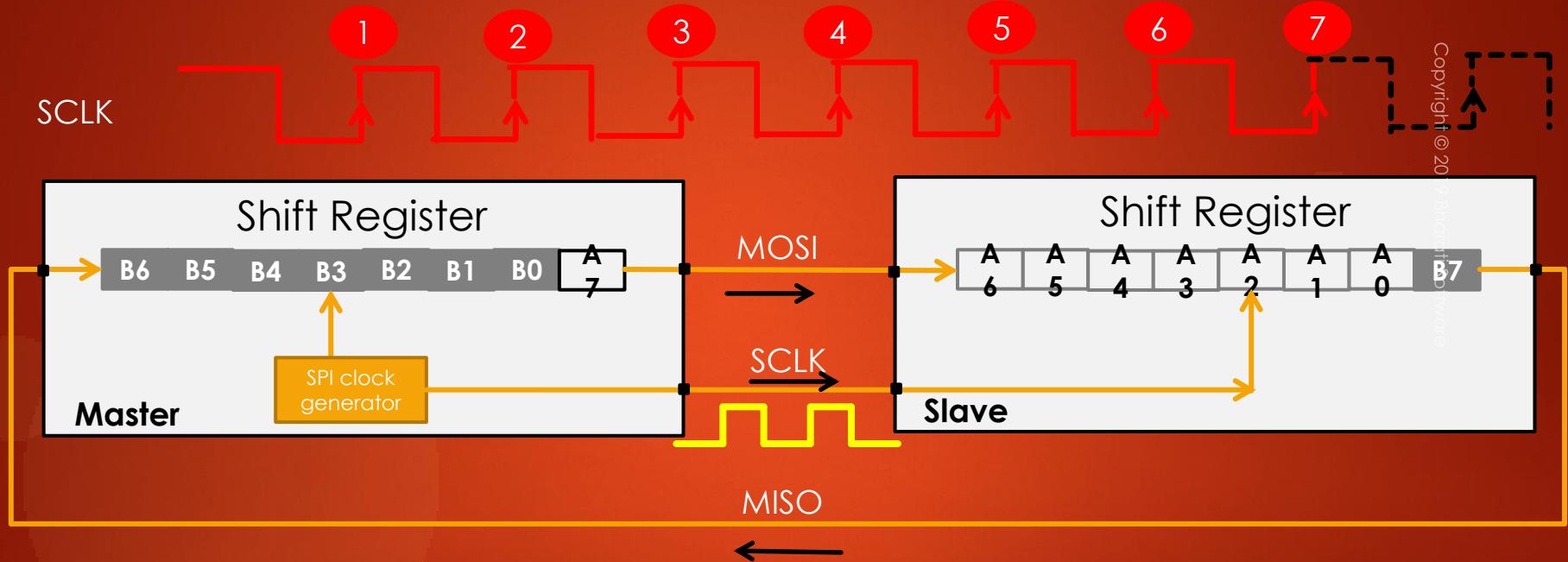


# SPI Hardware :Behind the scenes

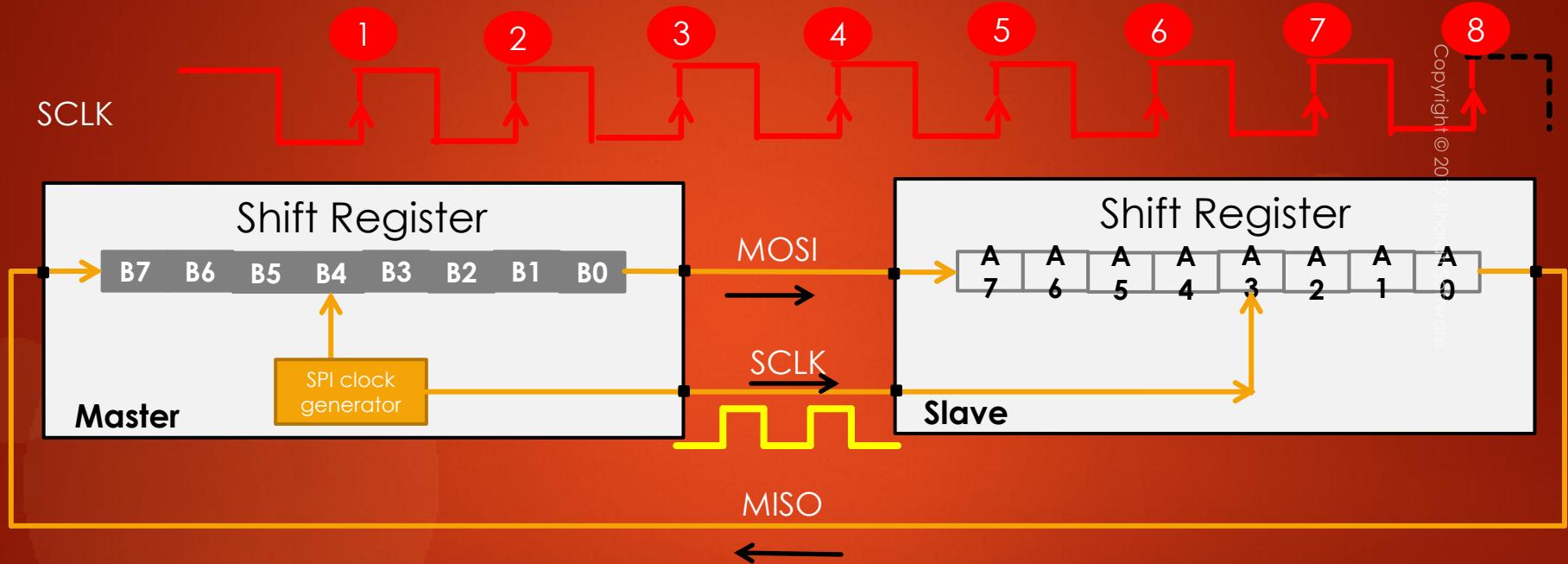
Copyright © 2018 Edureka!



# SPI Hardware :Behind the scenes



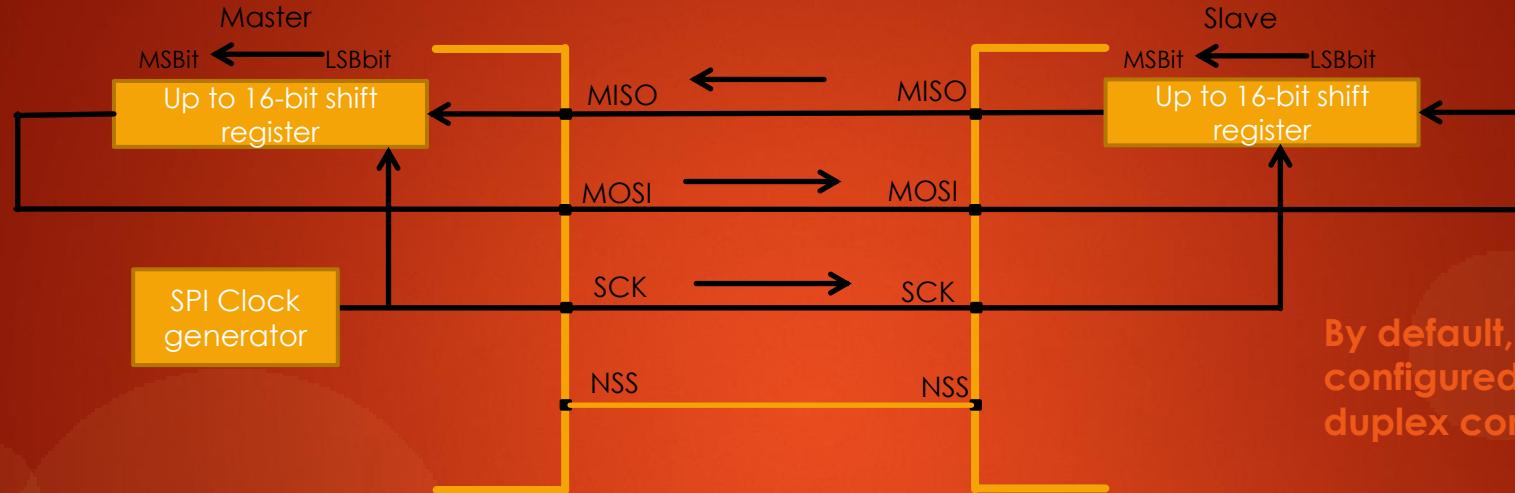
# SPI Hardware :Behind the scenes



# CUSTOMIZING SPI BUS : BUS CONFIGURATIONS

The SPI allows the MCU to communicate using different configurations, depending on the device targeted and the application requirements.

# FULL-DUPLEX COMMUNICATION

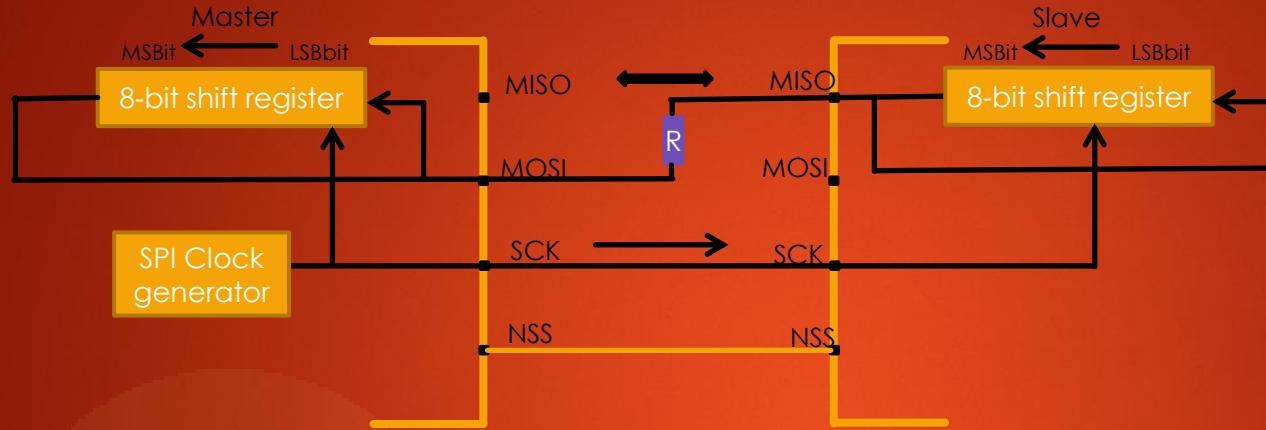


By default, the SPI is configured for full-duplex communication

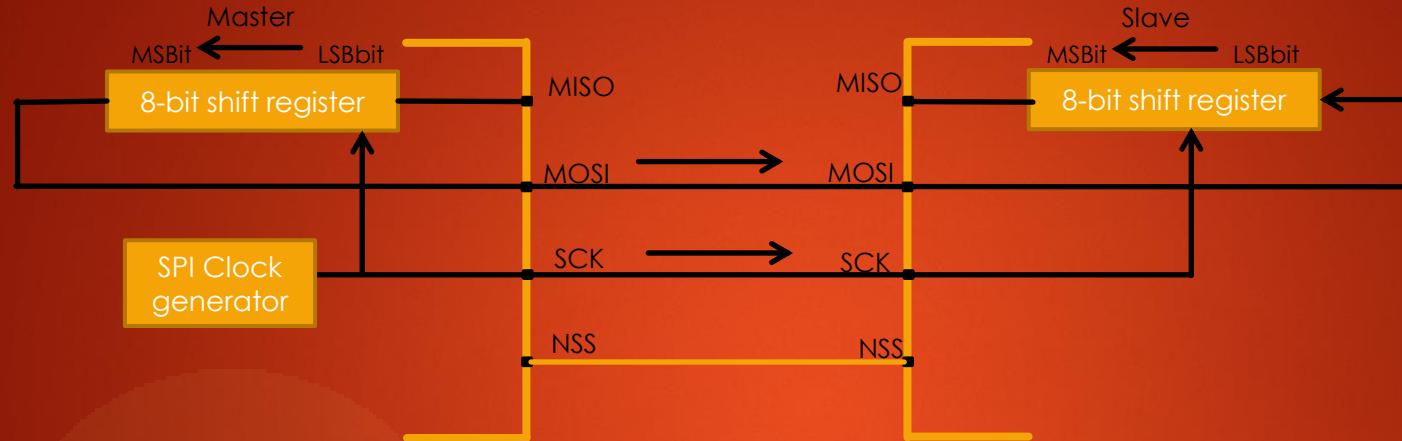
In this configuration, the shift registers of the master and slave are linked using two unidirectional lines between the MOSI and the MISO pins. During SPI communication, data is shifted synchronously on the SCK clock edges provided by the master. The master transmits the data to be sent to the slave via the MOSI line and receives data from the slave via the MISO line.

Remember that in SPI communication ,slave will not initiate data transfer unless master produces the clock.

# HALF - DUPLEX COMMUNICATION



# SIMPLEX COMMUNICATION

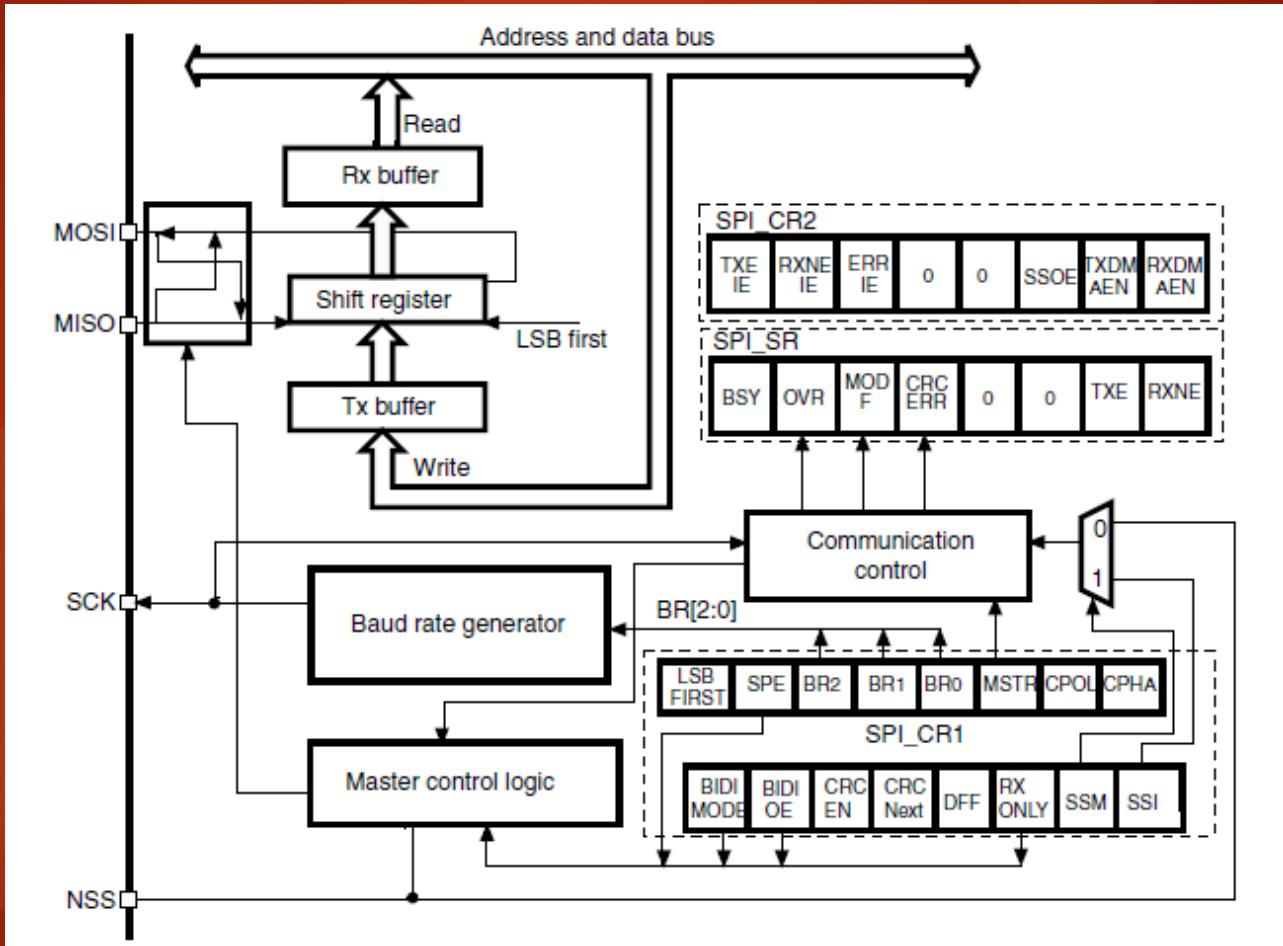


*Simplex single master, single slave application (master in transmit-only/slave in receive-only mode)*

Transmit-only , Receive Only mode :

The configuration settings are the same as for full-duplex. The application has to ignore the information captured on the unused input pin. This pin can be used as a standard GPIO

# STM32 SPI FUNCTIONAL BLOCK DIAGRAM



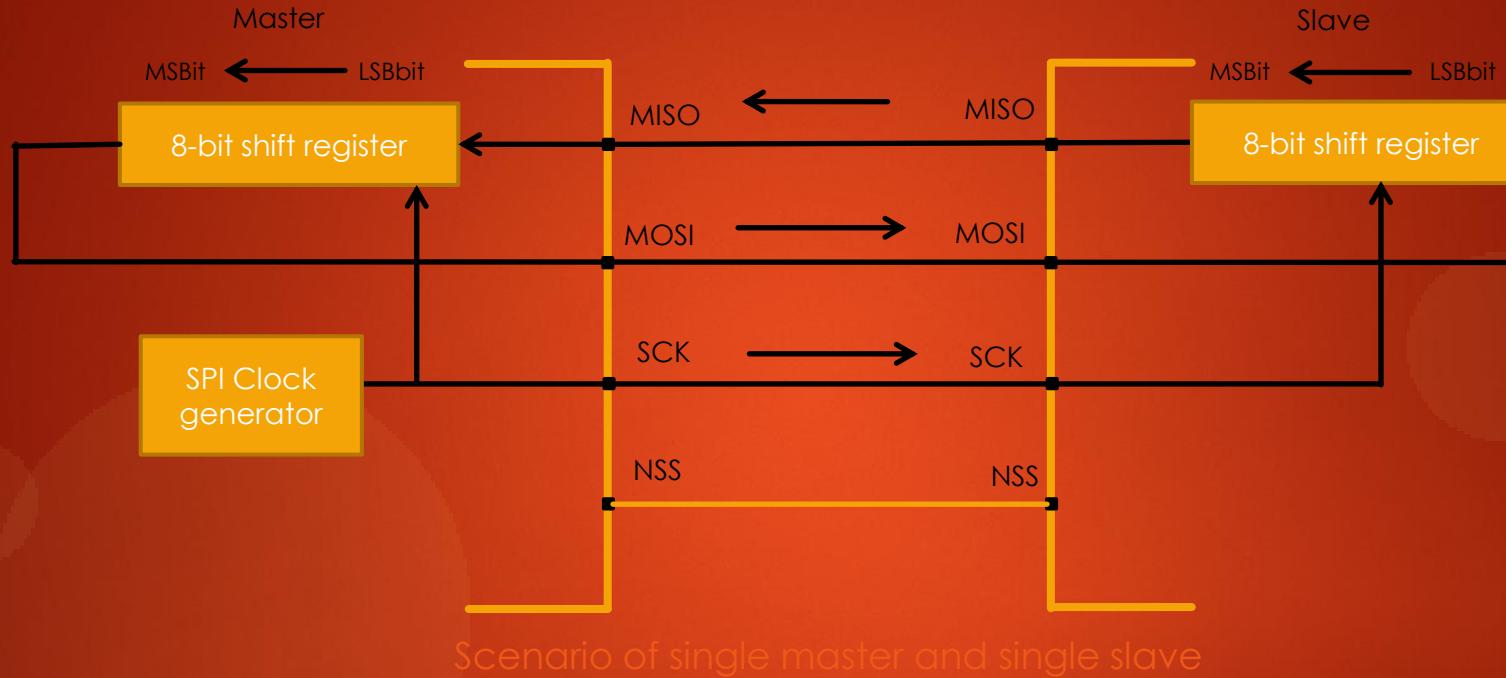
# SLAVE SELECT (NSS) PIN MANAGEMENT

## **When a device is slave mode:**

In slave mode, the NSS works as a standard “chip select” input and lets the slave communicate with the master.

## **When a device is master:**

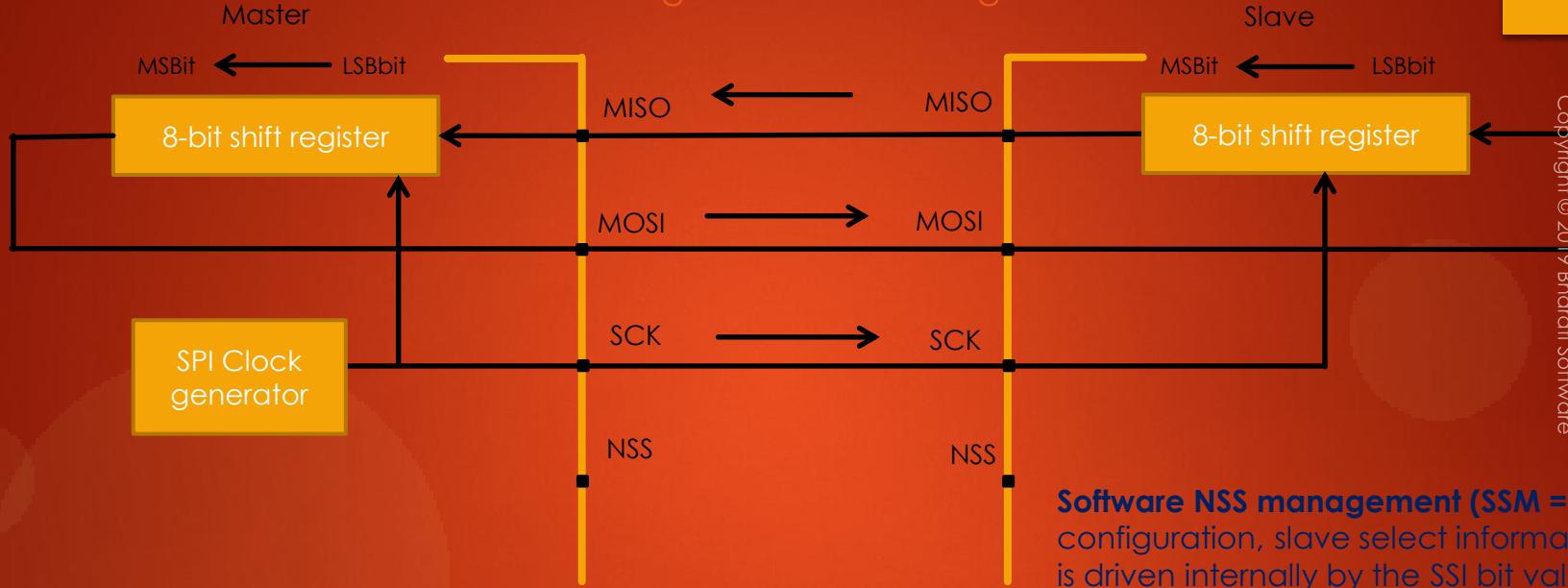
In master mode, NSS can be used either as output or input. As an input it can prevent multi-master bus collision, and as an output it can drive a slave select signal of a single slave.



# 2 Types of slave managements

Hardware slave management  
Software slave management

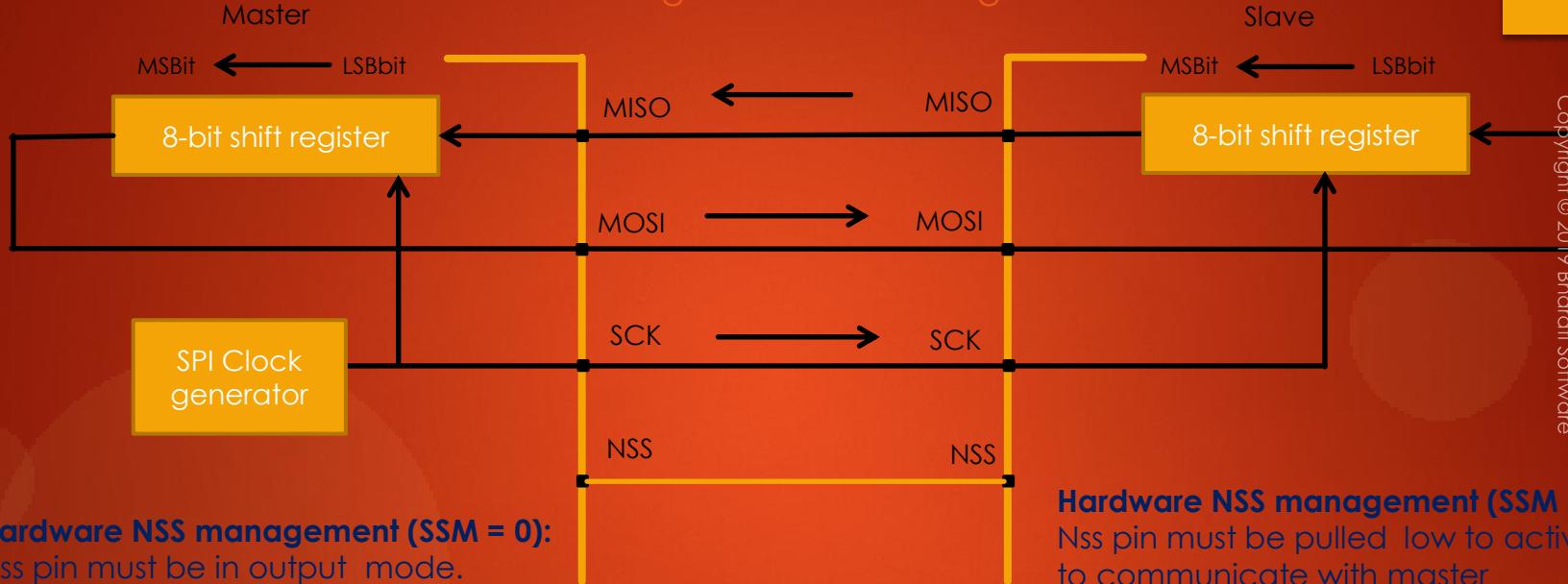
## Scenario of single master and single slave



Hardware or software slave select management can be set using the SSM bit in the SPIx\_CR1 register:

**Software NSS management (SSM = 1):** in this configuration, slave select information is driven internally by the SSI bit value in register SPIx\_CR1. The external NSS pin is free for other application uses.

## Scenario of single master and single slave



### Hardware NSS management ( $SSM = 0$ ):

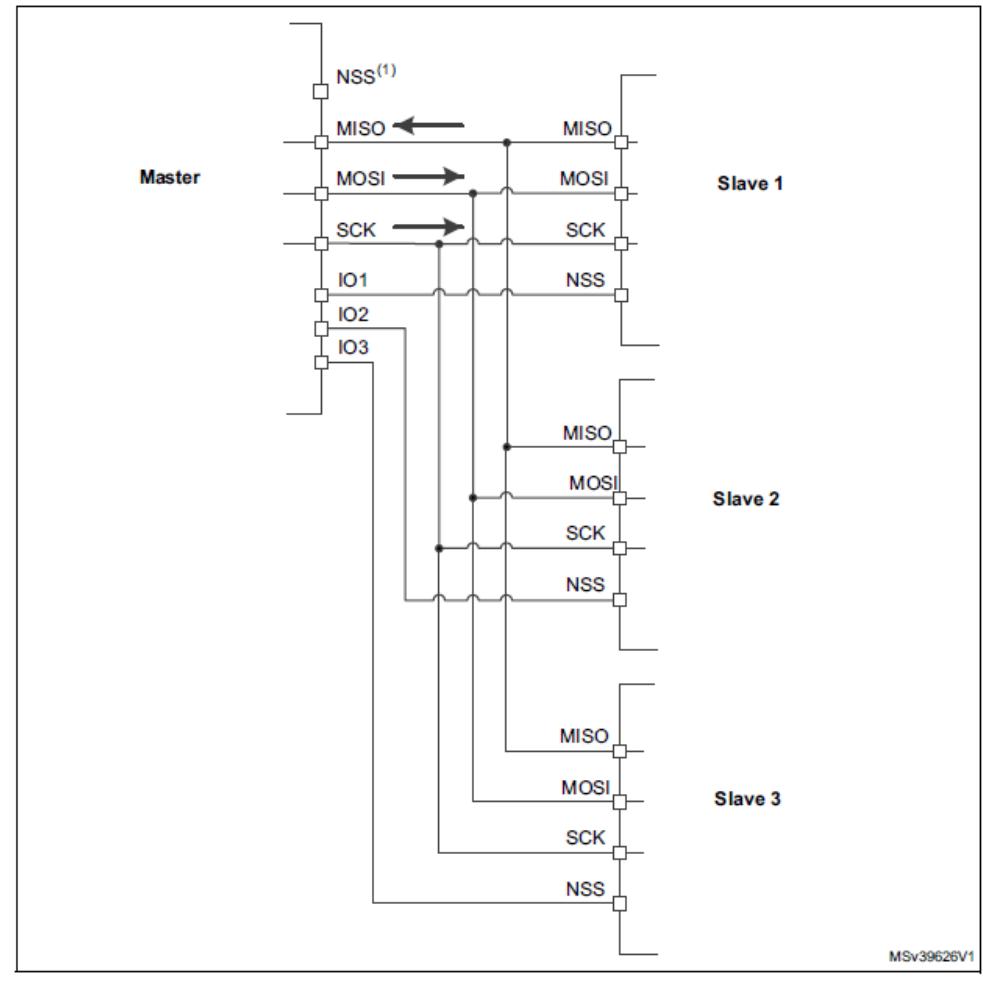
Nss pin must be in output mode.

The NSS pin is managed by the hardware

**Hardware NSS management ( $SSM = 0$ ):**  
Nss pin must be pulled low to active slave to communicate with master

Hardware or software slave select management can be set using the SSM bit in the SPIx\_CR1 register:

## Master and three independent slaves

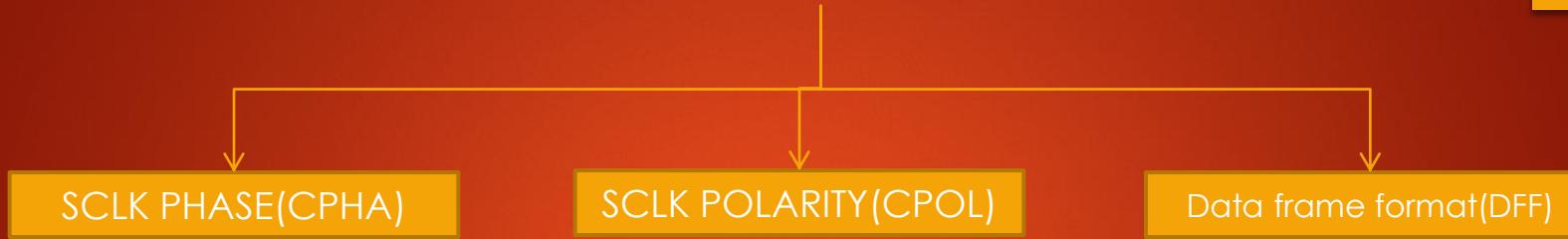


Copyright © 2012 Bharati Software

In this application you cannot use software slave management . You have to use hardware slave management

# SPI Communication Format

# SPI Communication Format

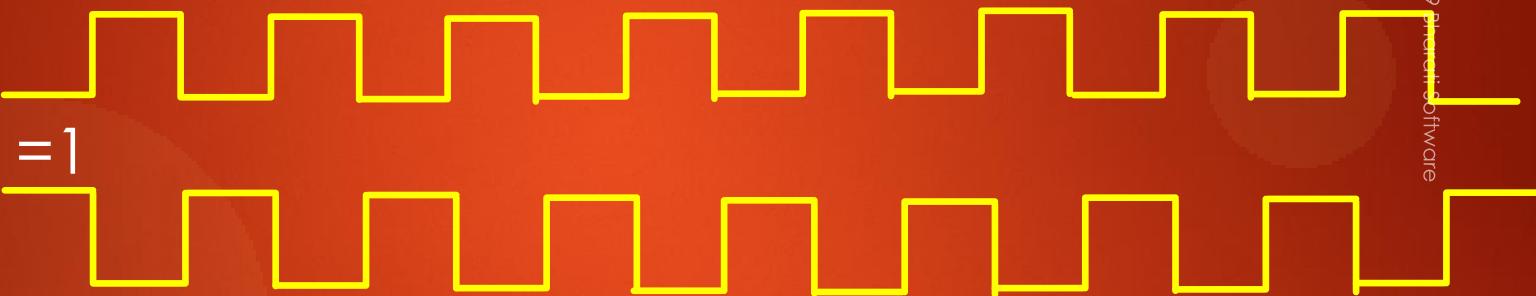


- During SPI communication, receive and transmit operations are performed simultaneously.
- The serial clock (SCK) synchronizes the shifting and sampling of the information on the data lines
- The communication format depends on the clock phase, the clock polarity and the data frame format. To be able to communicate together, the master and slaves devices must follow the same communication format.

# CPOL(CLOCK POLARITY)

- ▶ The CPOL (clock polarity) bit controls the idle state value of the clock when no data is being transferred
- ▶ If CPOL is reset, the SCLK pin has a low-level idle state. If CPOL is set, the SCLK pin has a high-level idle state.

CPOL = 0



CPOL = 1

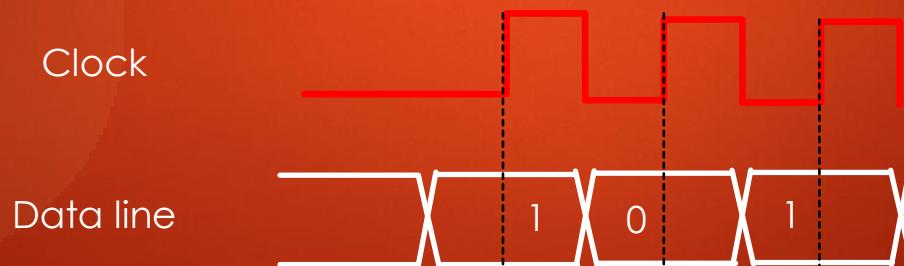
# CPHA(CLOCK PHASE)

- ▶ CPHA controls at which clock edge of the SCLK( 1<sup>st</sup> or 2<sup>nd</sup> ) the data should be sampled by the slave.
- ▶ The combination of CPOL (clock polarity) and CPHA (clock phase) bits selects the data capture clock edge.

Data toggling means, data transition to the next bit.

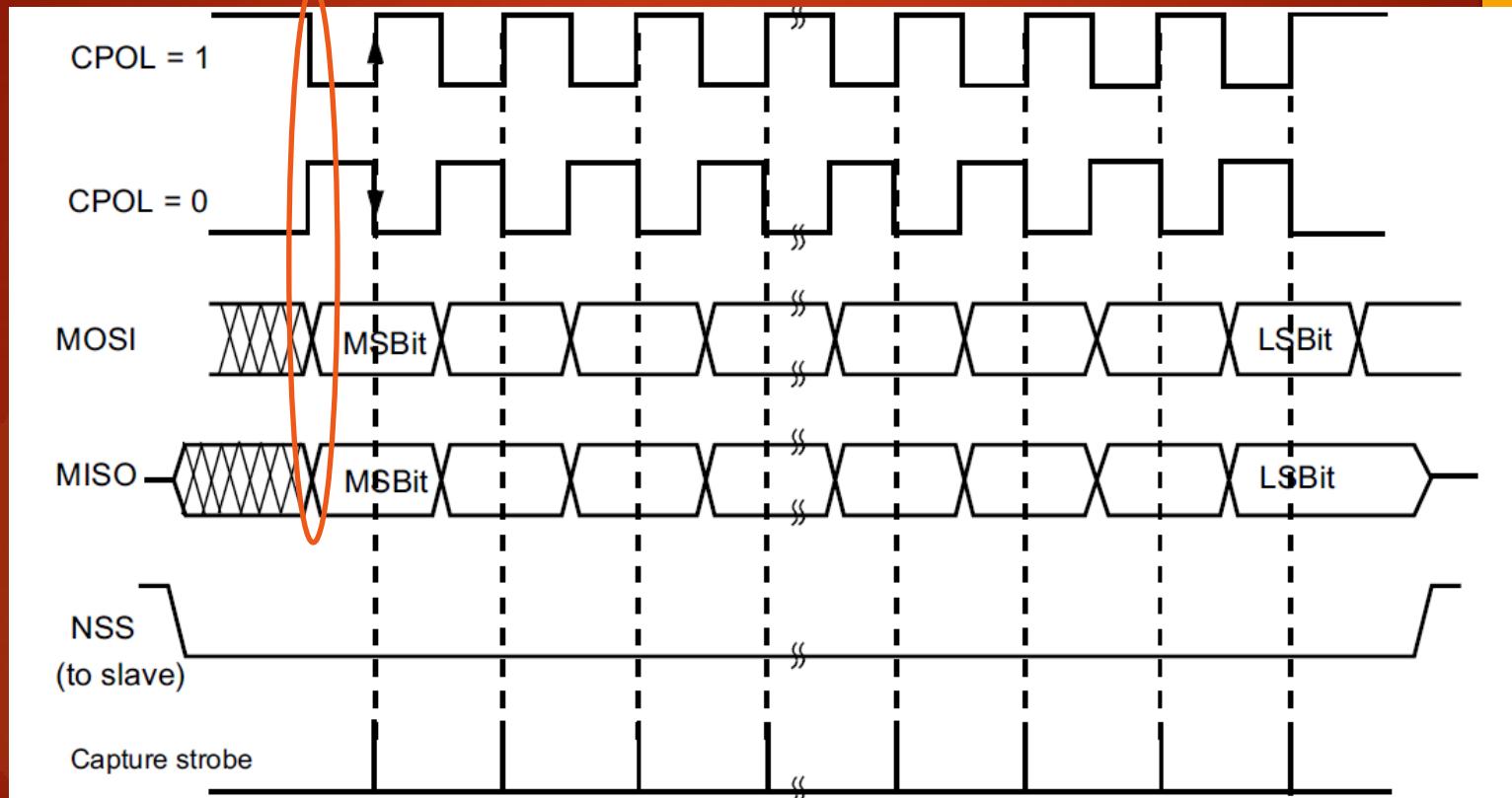


Data sampling means, sampling the data line to capture the data.



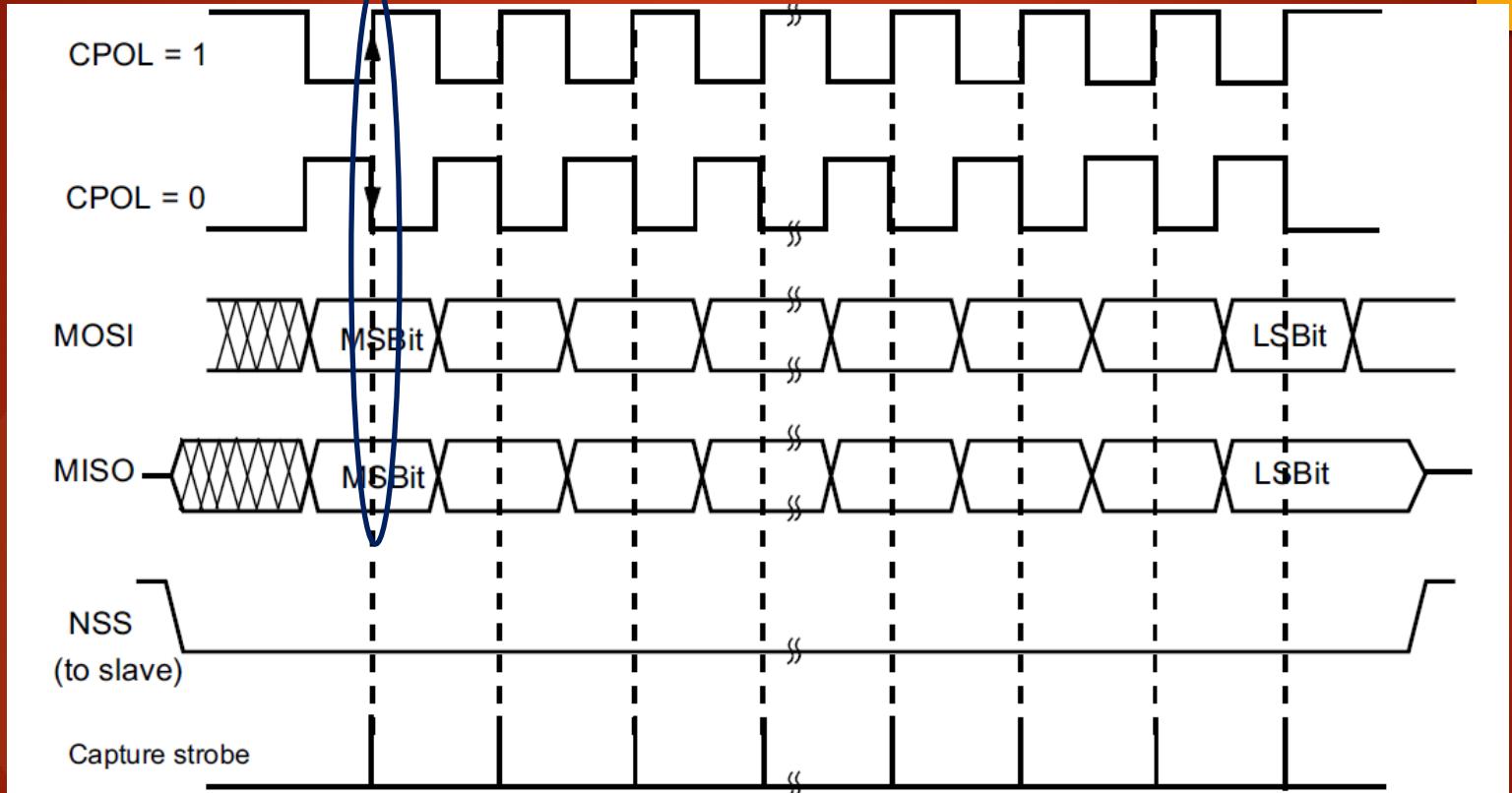
CPHA=1

1 Data will appear on the lines during first edge of the SCLK



CPHA=1

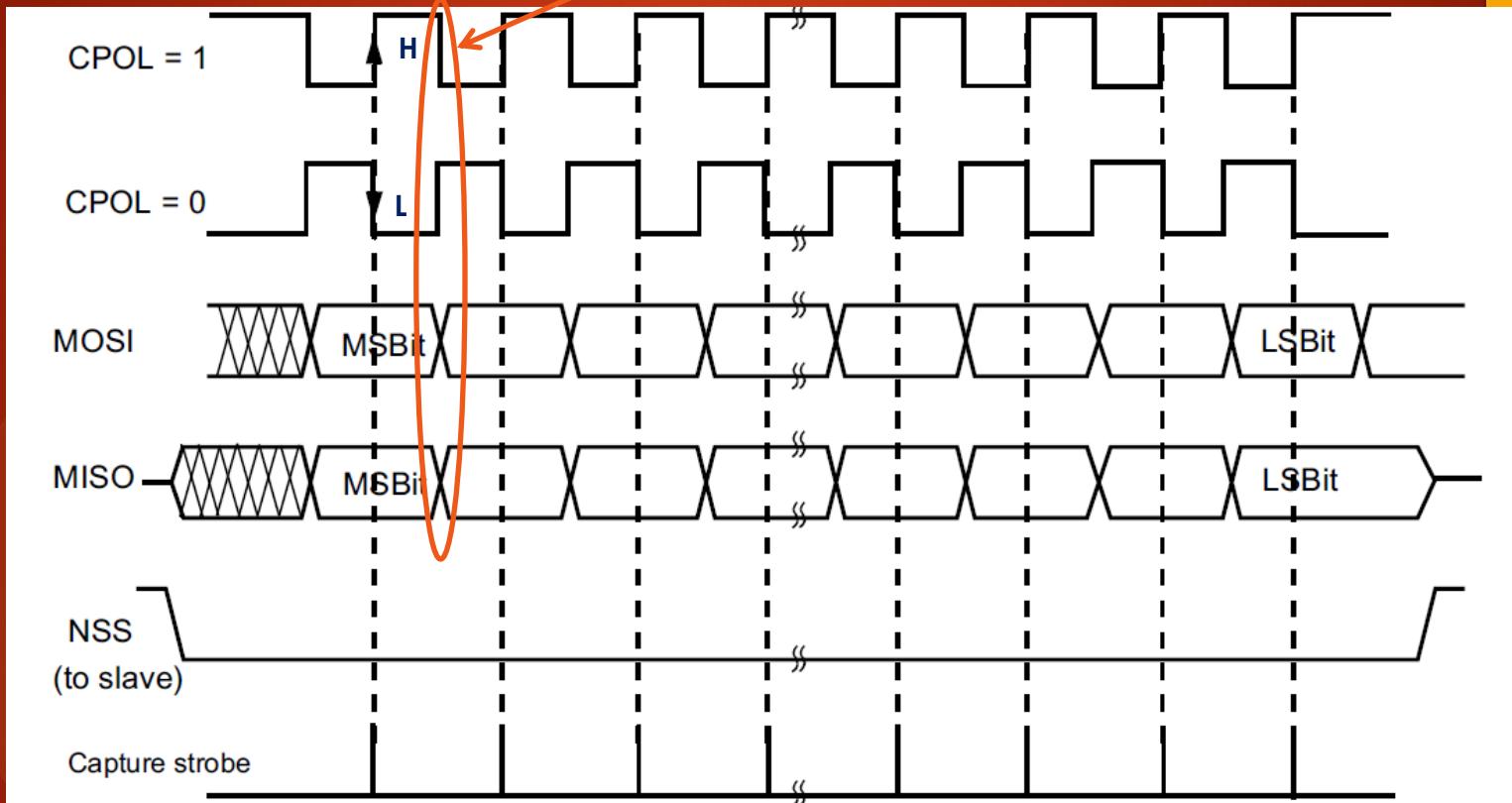
2 Slave captures the data here ( 2<sup>nd</sup> edge of the SCLK)



CPHA=1

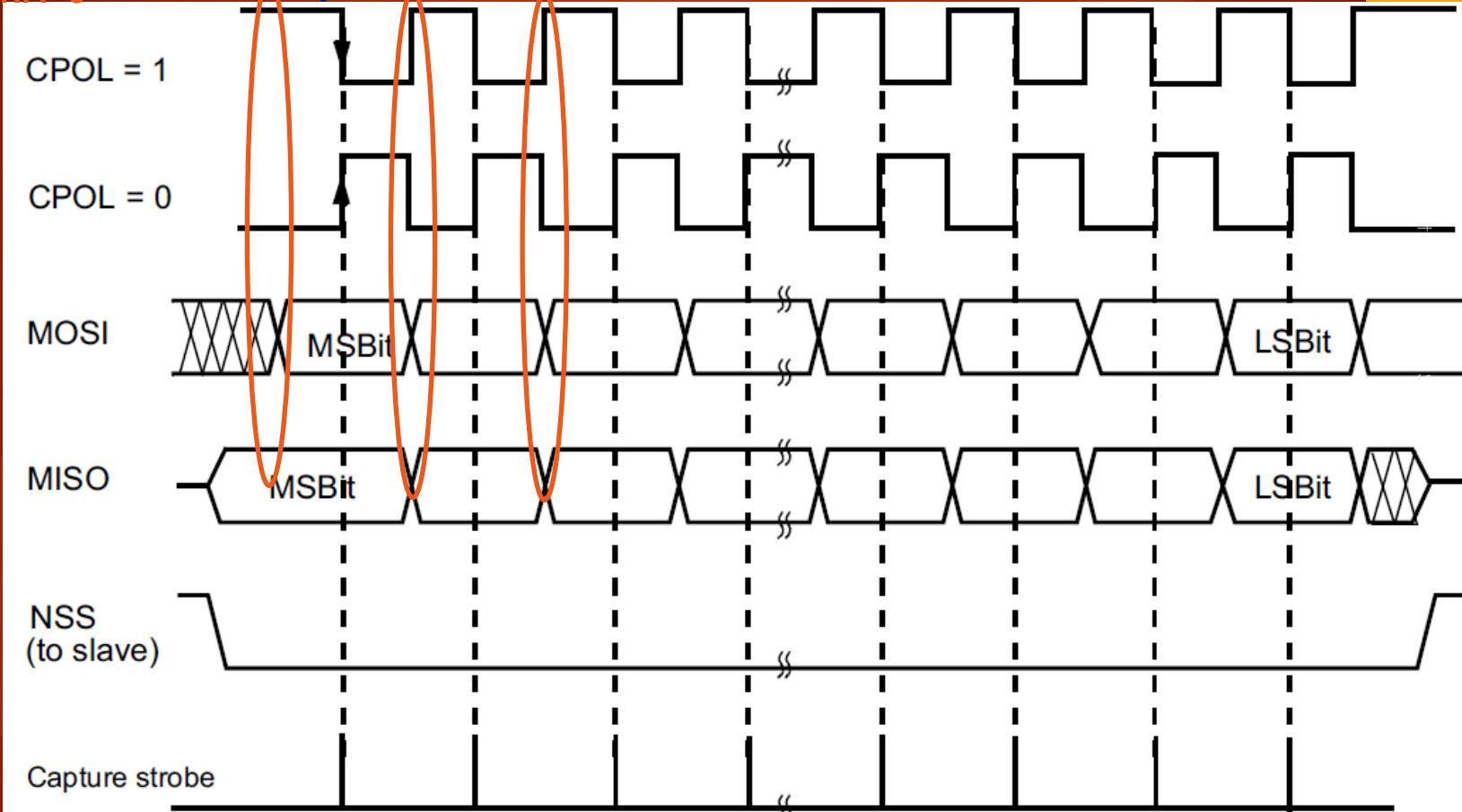
1

Data will appear on the lines during first edge of the SCLK



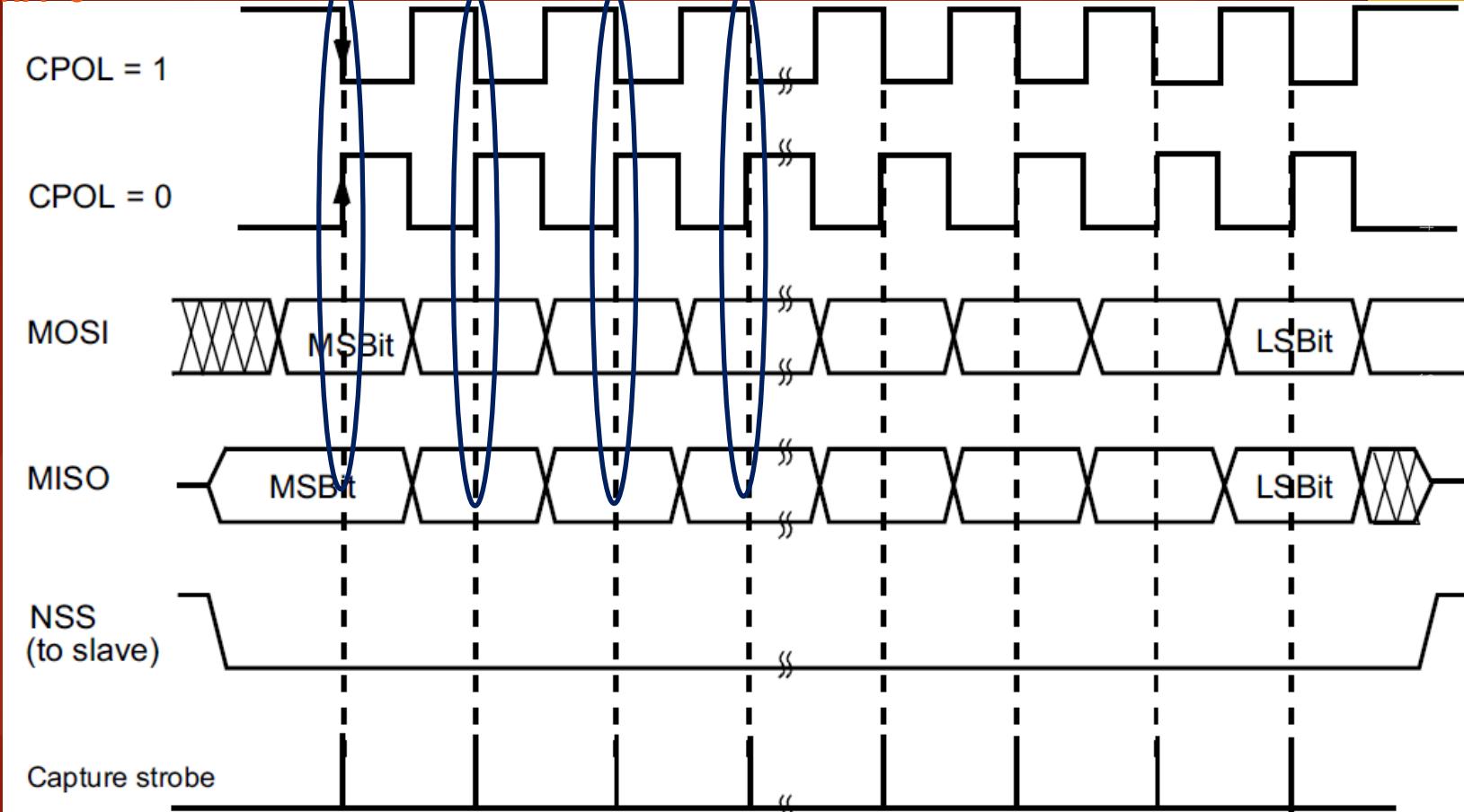
CPHA=0

1 Data will appear on the lines during 2<sup>nd</sup> edge of the clock



CPHA=0

2 Slave captures the data here ( 1<sup>st</sup> edge of the SCLK)



# Different SPI Modes

<b>Mode</b>	<b>CPOL</b>	<b>CPHA</b>
0	0	0
1	0	1
2	1	0
3	1	1

If **CPhase=1**

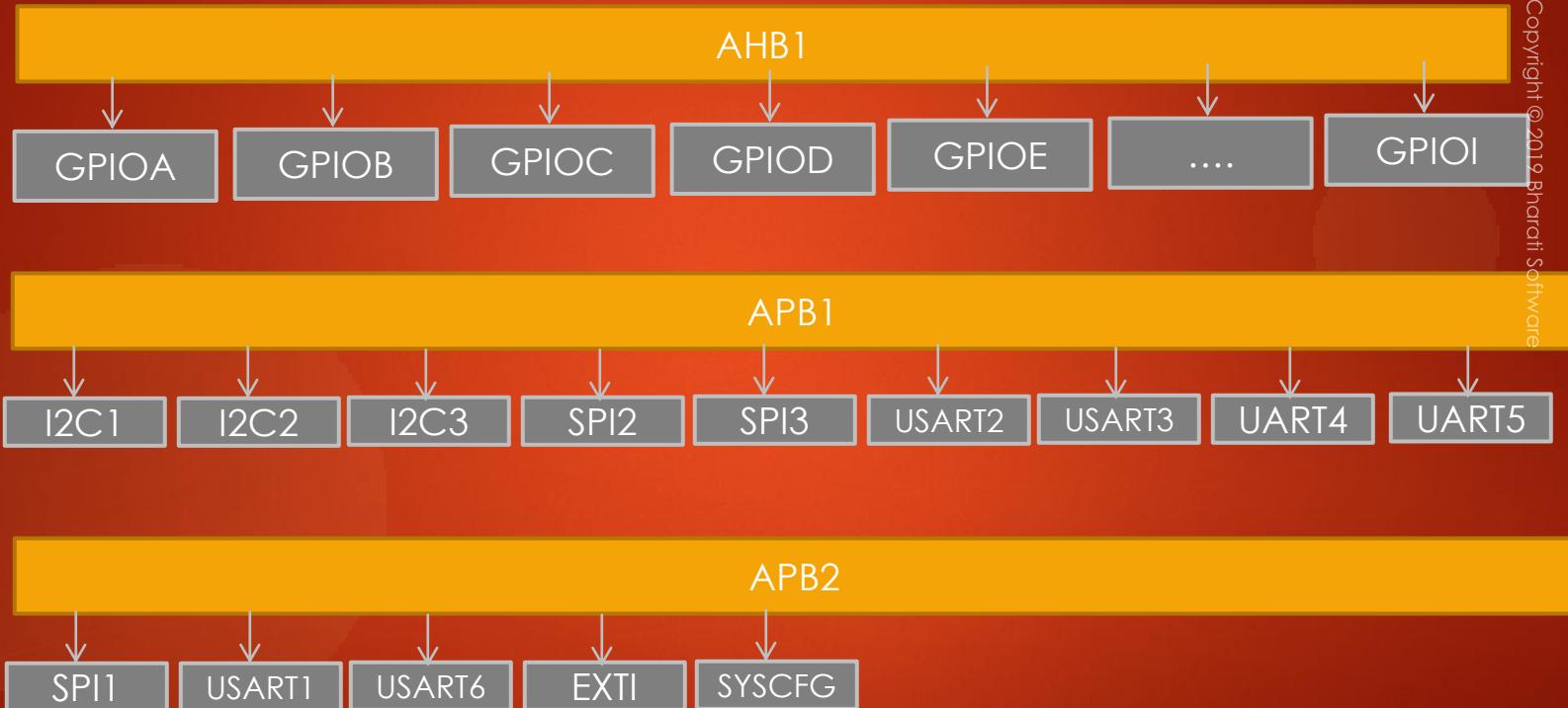
Data will be sampled on the *trailing* edge of the clock.

If **CPhase=0**

Data will be sampled on the *leading* edge of the clock.

# SPI peripherals of your MCU

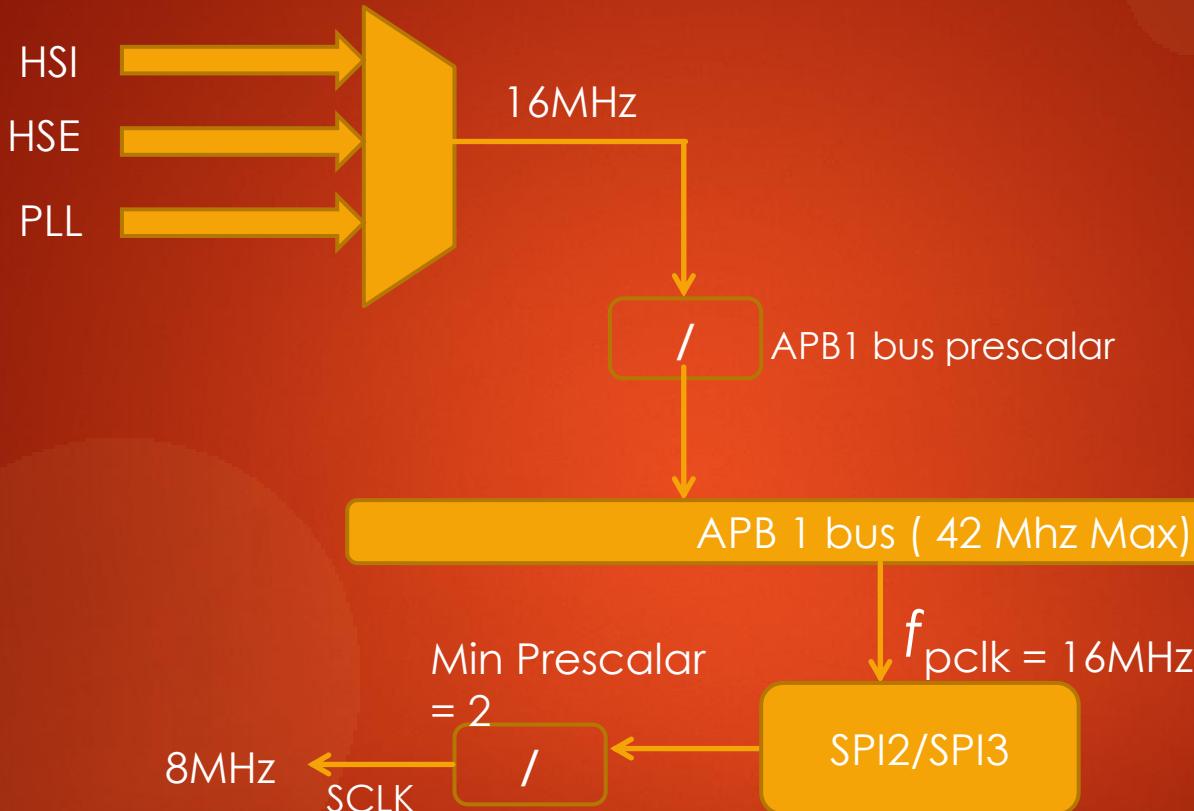
Copyright © 2012 Bharati Software

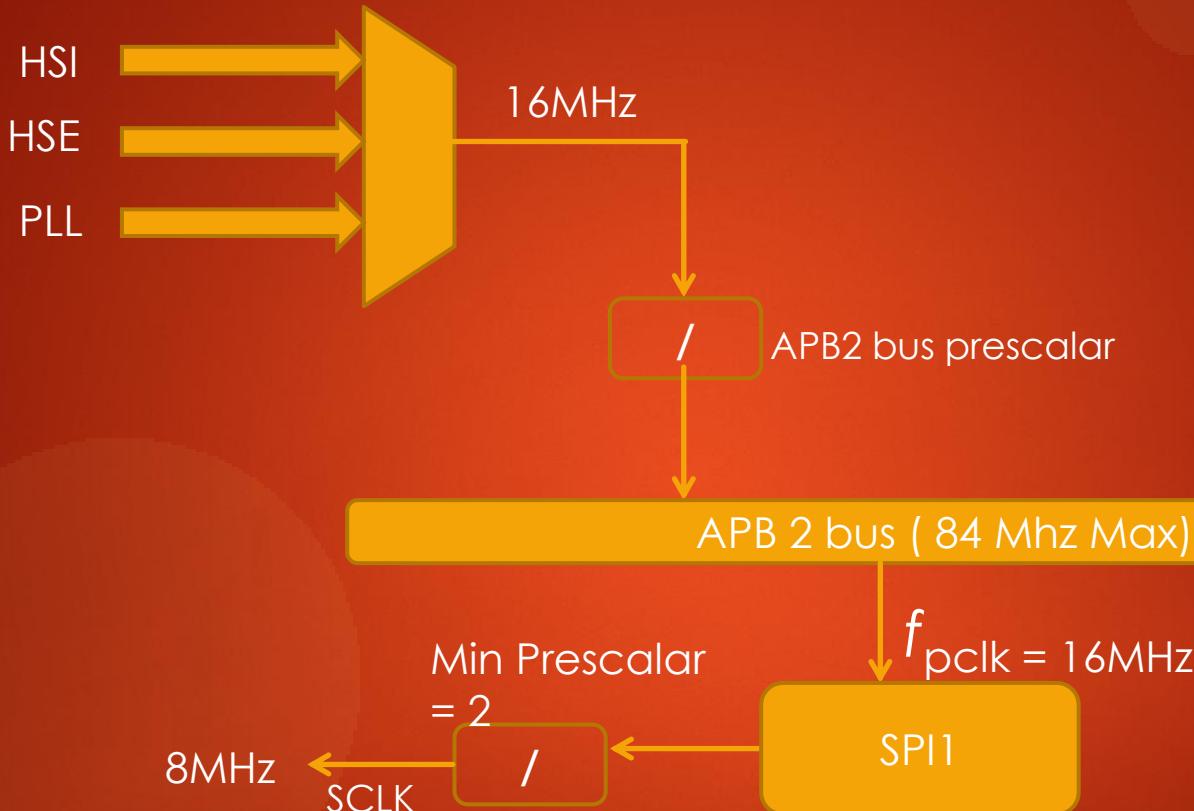


# SPI serial clock (SCLK)

What is the maximum SCLK speed of SPIx peripheral which can be achieved on a given microcontroller

First you have to know the speed of the APBx bus on which the SPI peripheral is connected





So, if we use the internal RC oscillator of 16Mhz as our system clock then SPI1/SPI2/SPI3 peripherals can able to produce the serial clock of maximum 8MHz.

Remember that in SPI communication ,slave will not initiate data transfer unless master produces the clock.

# SPI Driver Development

# Driver API requirements and user configurable items

# Sample Applications

Copyright © 2019 Bharatii Software

## Driver Layer

gpio\_driver.c , .h

i2c\_driver.c , .h

(Device header)

Stm3f407xx.h

spi\_driver.c , .h

uart\_driver.c , .h

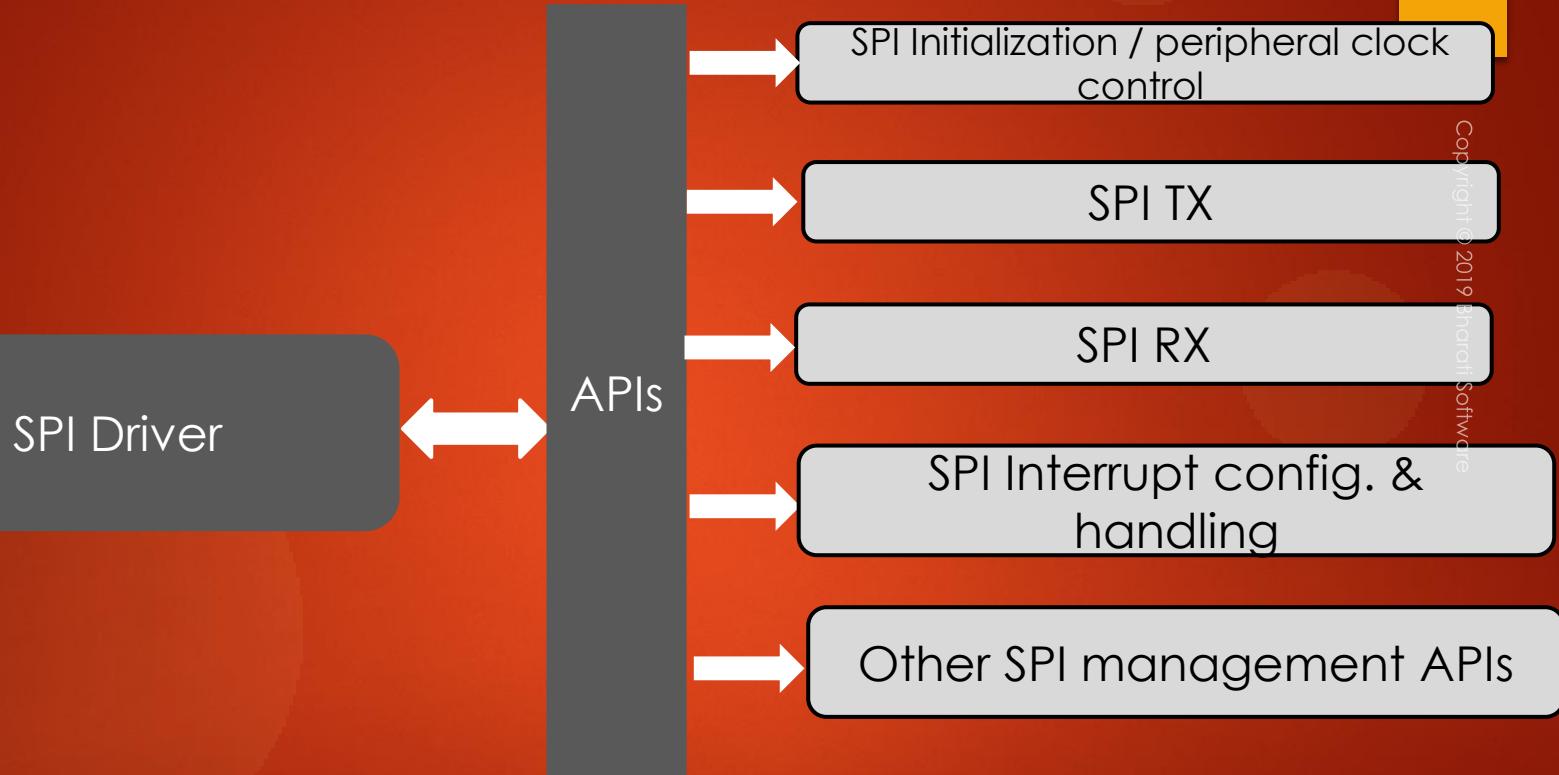
GPIO

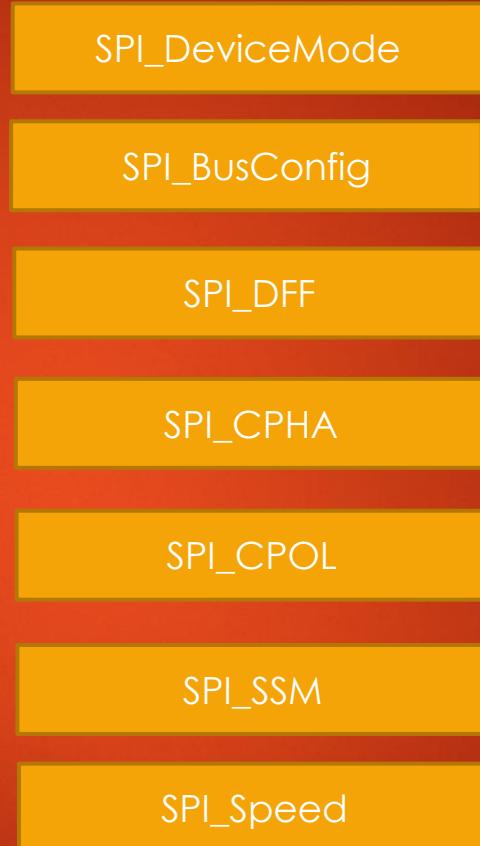
SPI

I2C

UART

**STM3F407x MCU**





Configurable items  
For user application

# SPI handle structure and configuration structure

```
/*
 * Configuration structure for SPIx peripheral
 */
typedef struct
{
    uint8_t SPI_DeviceMode;
    uint8_t SPI_BusConfig;
    uint8_t SPI_SclkSpeed;
    uint8_t SPI_DFF;
    uint8_t SPI_CPOL;
    uint8_t SPI_CPHA;
    uint8_t SPI_SSM;
}SPI_Config_t;
```

## SPI Configuration Structure

```
/*
 *Handle structure for SPIx peripheral
 */
typedef struct
{
    SPI_RegDef_t      *pSPIx;    /*!< This holds the base address of SPIx(x:0,1,2) peripheral >/
    SPI_Config_t     SPIConfig;
}SPI_Handle_t;
```

## SPI Handle Structure

# Exercise :

1. Complete SPI register definition structure and other macros ( *peripheral base addresses, Device definition , clock en , clock di , etc*) in MCU specific header file
2. Complete SPI Configuration structure and SPI handle structure in SPI header file

# Writing API prototypes

# Send data



# Receive data



# Exercise :

1. Test the SPI\_SendData API to send the string “**Hello world**” and use the below configurations
  1. SPI-2 Master mode
  2. SCLK = max possible
  3. DFF = 0 and DFF = 1

# Exercise :

## SPI Master(STM) and SPI Slave(Arduino) communication .

When the button on the master is pressed , master should send string of data to the Arduino slave connected. The data received by the Arduino will be displayed on the Arduino serial port.

- 1 .Use SPI Full duplex mode
2. ST board will be in SPI master mode and Arduino will be configured for SPI slave mode
3. Use DFF = 0
4. Use Hardware slave management (SSM = 0)
5. SCLK speed = 2MHz , fclk = 16MHz

In this exercise master is not going to receive anything from the slave. So you may not configure the MISO pin

Note.

Slave does not know how many bytes of data master is going to send. So master first sends the number bytes info which slave is going to receive next.

# Things you need

- 1 .Arduino board
- 2. ST board
- 3. Some jumper wires
- 4. Bread board



# STEP-1

Connect Arduino and ST board SPI pins as shown

# STEP-2

Power your Arduino board and download SPI  
Slave sketch to Arduino

Sketch name : 001SPISlaveRxString.ino

# Find out the GPIO pins over which SPI2 can communicate

# Exercise :

## SPI Master(STM) and SPI Slave(Arduino) command & response based communication .

When the button on the master is pressed, master sends a command to the slave and slave responds as per the command implementation .

- 1 .Use SPI Full duplex mode
2. ST board will be in SPI master mode and Arduino will be configured for SPI slave mode
3. Use DFF = 0
4. Use Hardware slave management (SSM = 0)
5. SCLK speed = 2MHz , fclk = 16MHz

# STEP-1

Connect Arduino and ST board SPI pins as shown

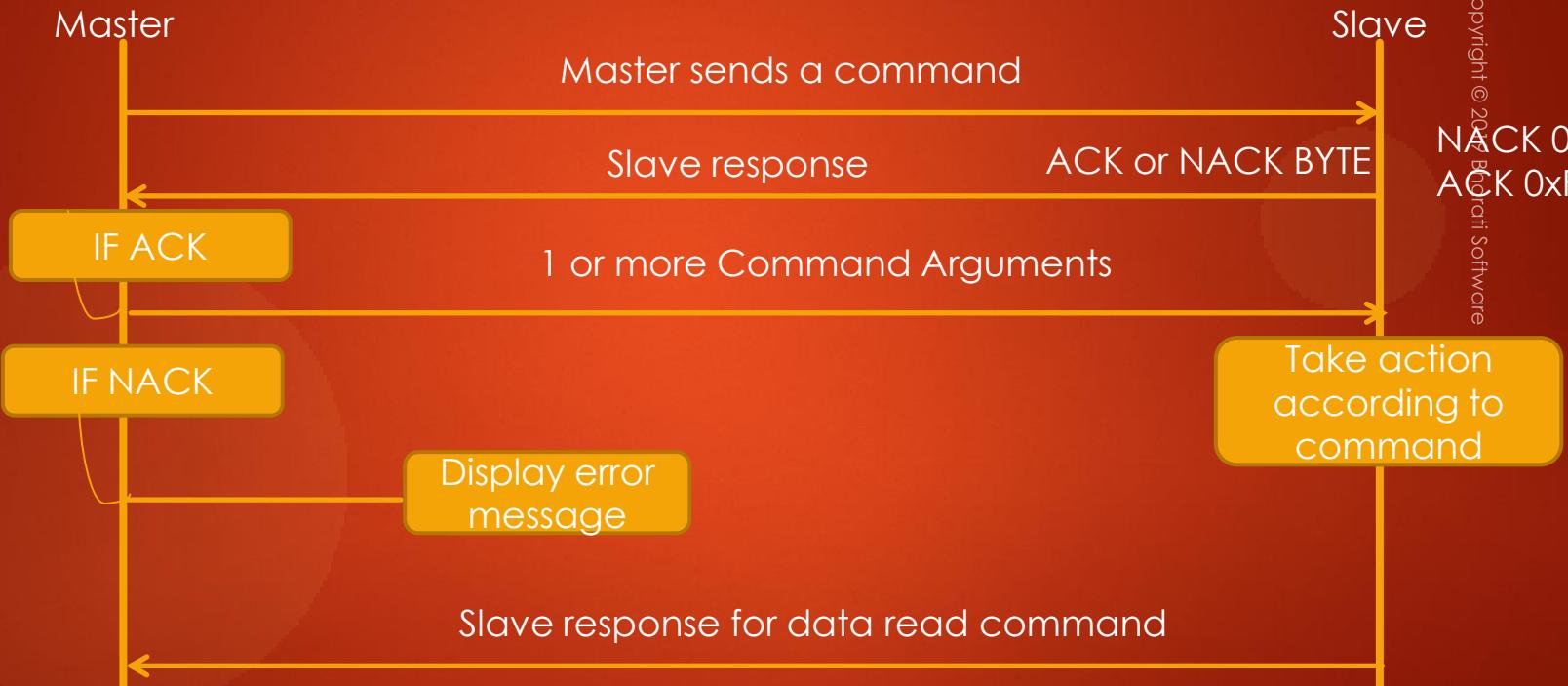
# STEP-2

Power your Arduino board and download SPI Slave sketch to Arduino

Sketch name : 002SPISlaveCmdHandling.ino

# Master & Slave communication

NACK 0xA5  
ACK 0xF5



# Command formats

< command\_code(1) >

<arg1>

<arg2>

This command is used to turn on/off led connected to specified arduino pin number

1)CMD\_LED\_CTRL

<pin no(1)>      ON/OFF

2)CMD\_SENOSR\_READ

<analog pin number(1) >

3) CMD\_LED\_READ

<pin no(1) >

4) CMD\_PRINT

<len(2)> <message(len) >

5) CMD\_ID\_READ

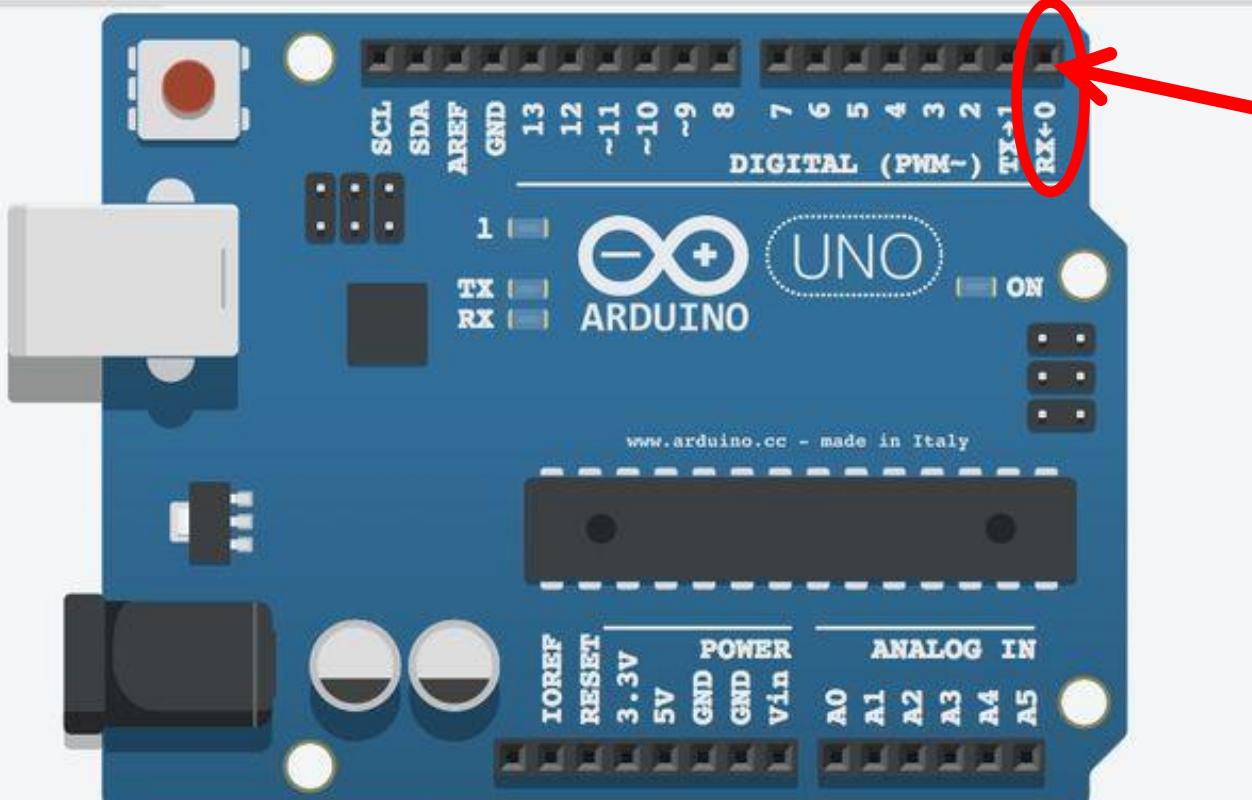
**CMD\_LED\_CTRL      <pin no>      <value>**

<pin no > digital pin number of the arduino board ( 0 to 9) ( 1 byte)

<value> 1 = ON , 0 = OFF (1 byte)

Slave Action : control the digital pin ON or OFF

Slave returns : Nothing



Remove the  
connection made  
here

Download the code

Then, connect it  
back

## **CMD\_SENOSR\_READ <analog pin number >**

<analog pin number > Analog pin number of the Arduino board ( A0 to A5)  
(1 byte)

Slave Action : Slave should read the analog value of the supplied pin

Slave returns : 1 byte of analog read value

**CMD\_LED\_READ      <pin no >**

<pin no > digital pin number of the arduino board ( 0 to 9)

Slave Action : Read the status of the supplied pin number

Slave returns : 1 byte of led status . 1 = ON , 0 = OFF

**CMD\_PRINT**

**<len>**

**<message >**

**<len>** 1 byte of length information of the message to follow

**<message>** message of ‘len’ bytes

Slave Action : Receive the message and display via serial port

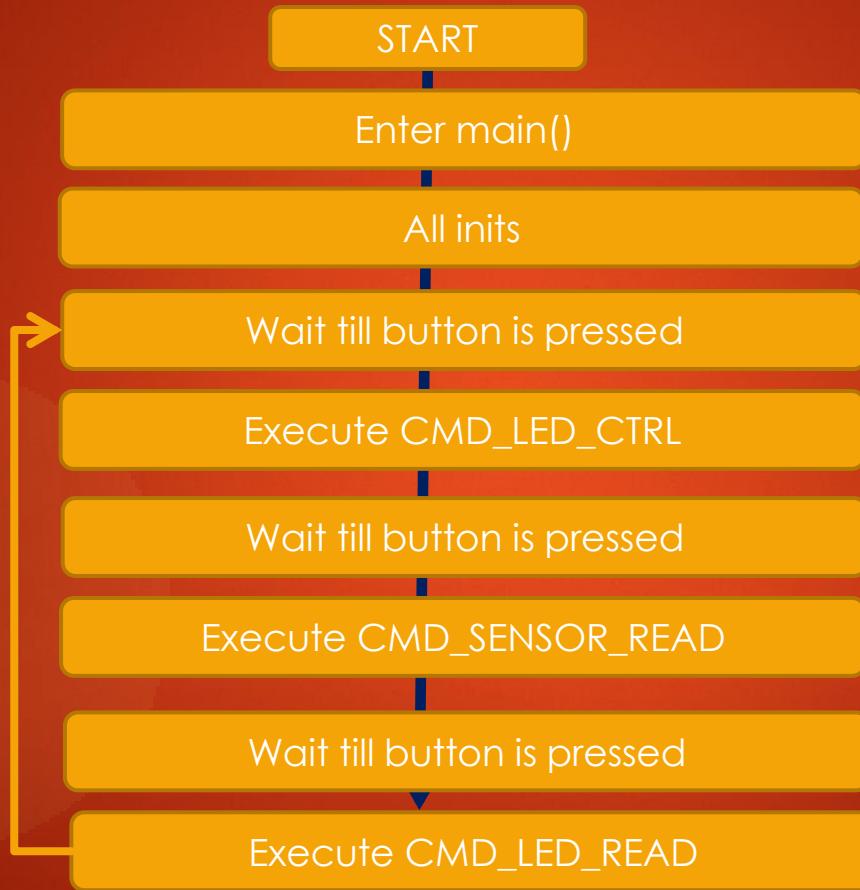
Slave returns : Nothing

## **CMD\_ID\_READ**

No arguments for this command

Slave returns : 10 bytes of board id string

# Application flow chart

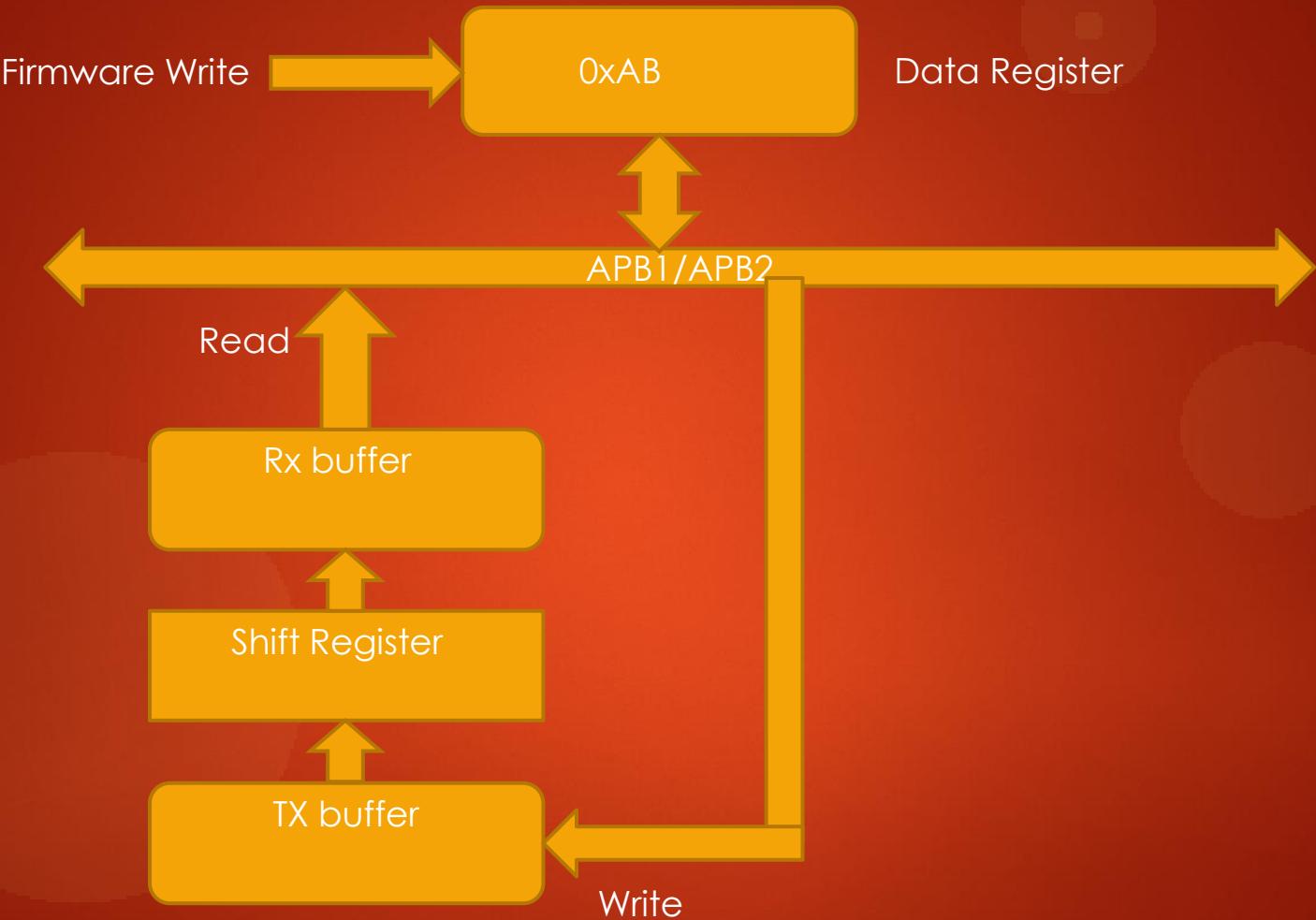


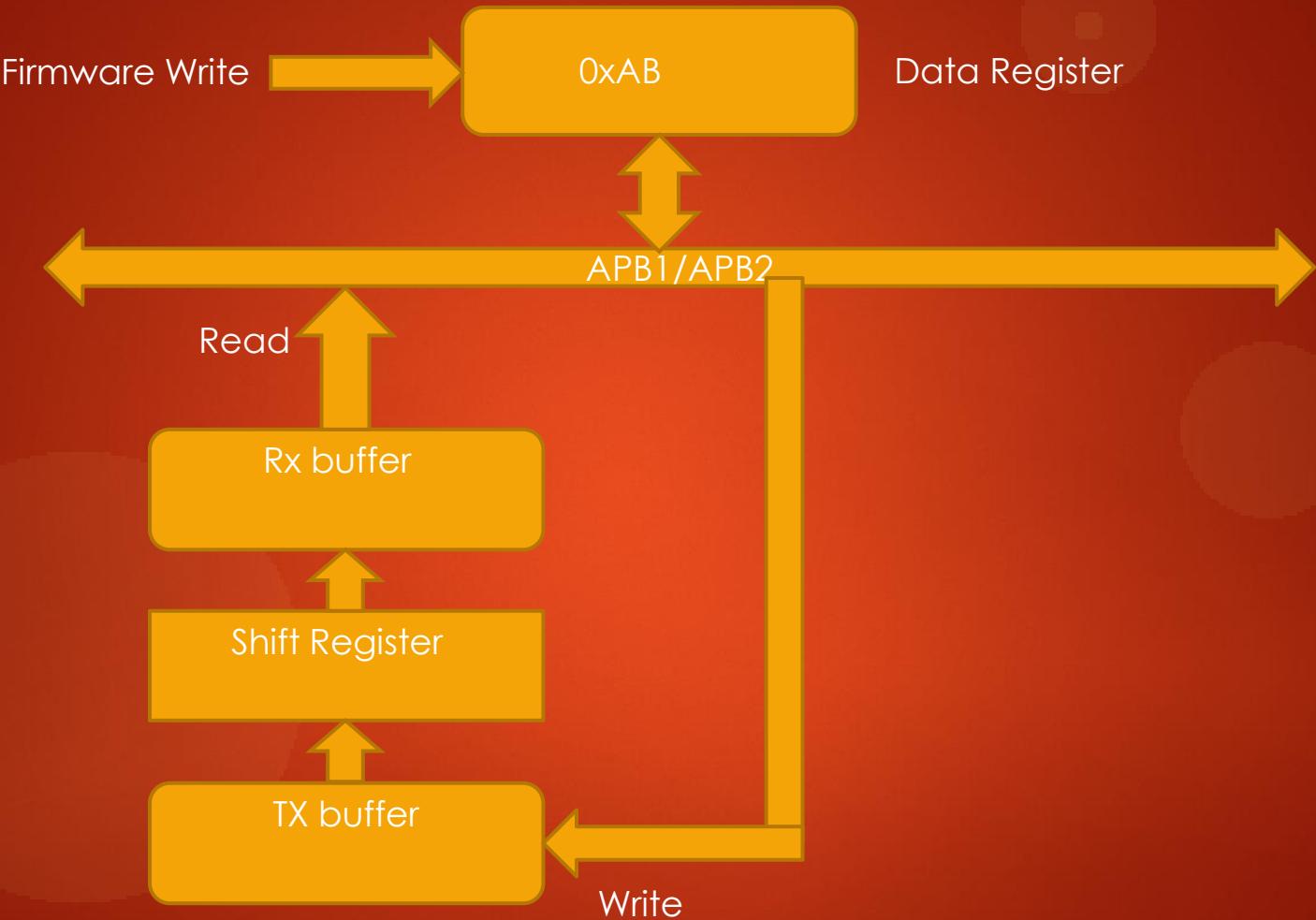


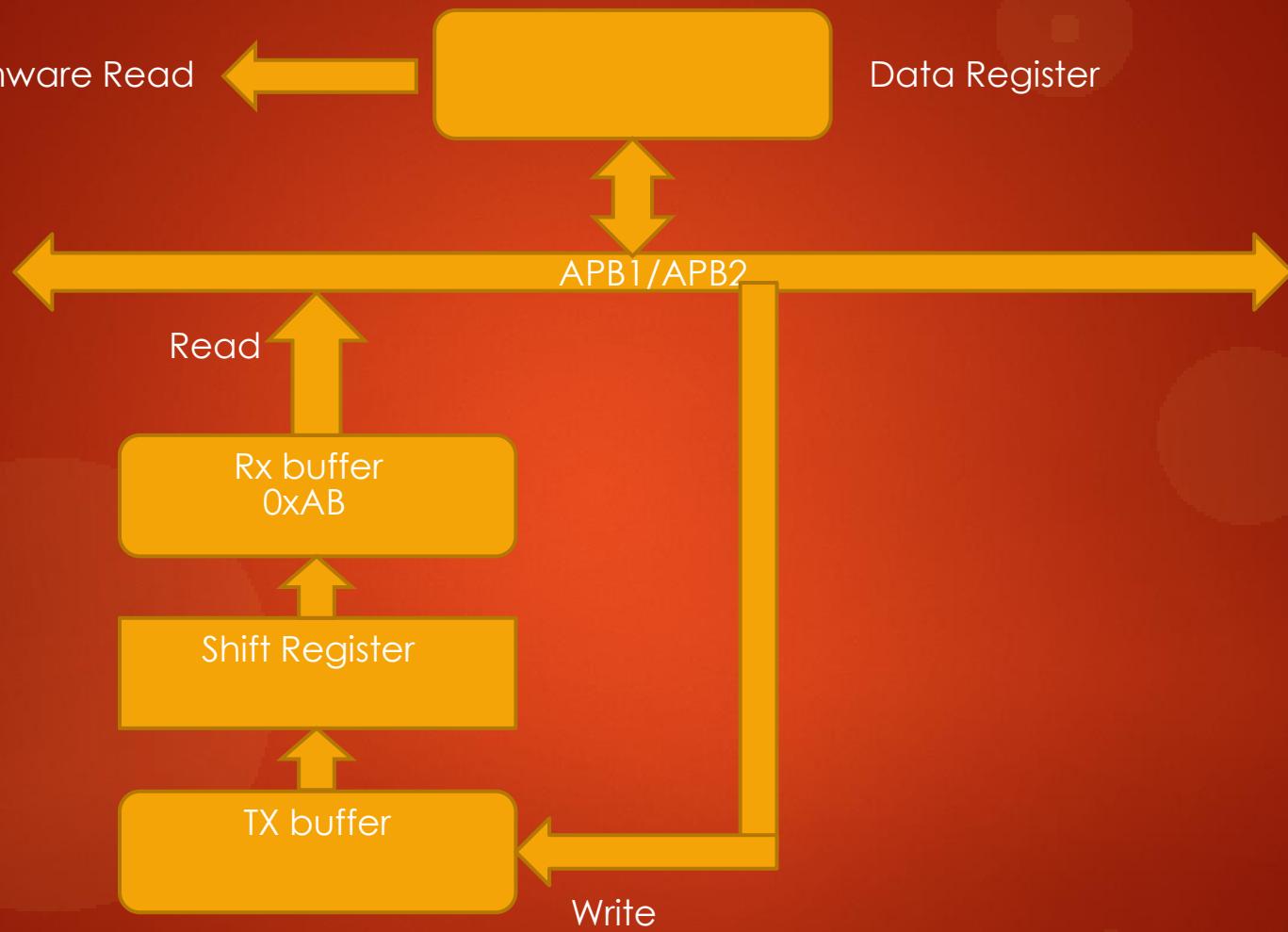
Copyright © 2019 Bharati Software



Copyright © 2019 Bharati Software







# SPI interrupts

**During SPI communication , interrupts can be generated by the following events:**

- Transmit Tx buffer ready to be loaded
- Data received in Rx buffer
- Master mode fault ( in single master case you must avoid this error happening )
- Overrun error

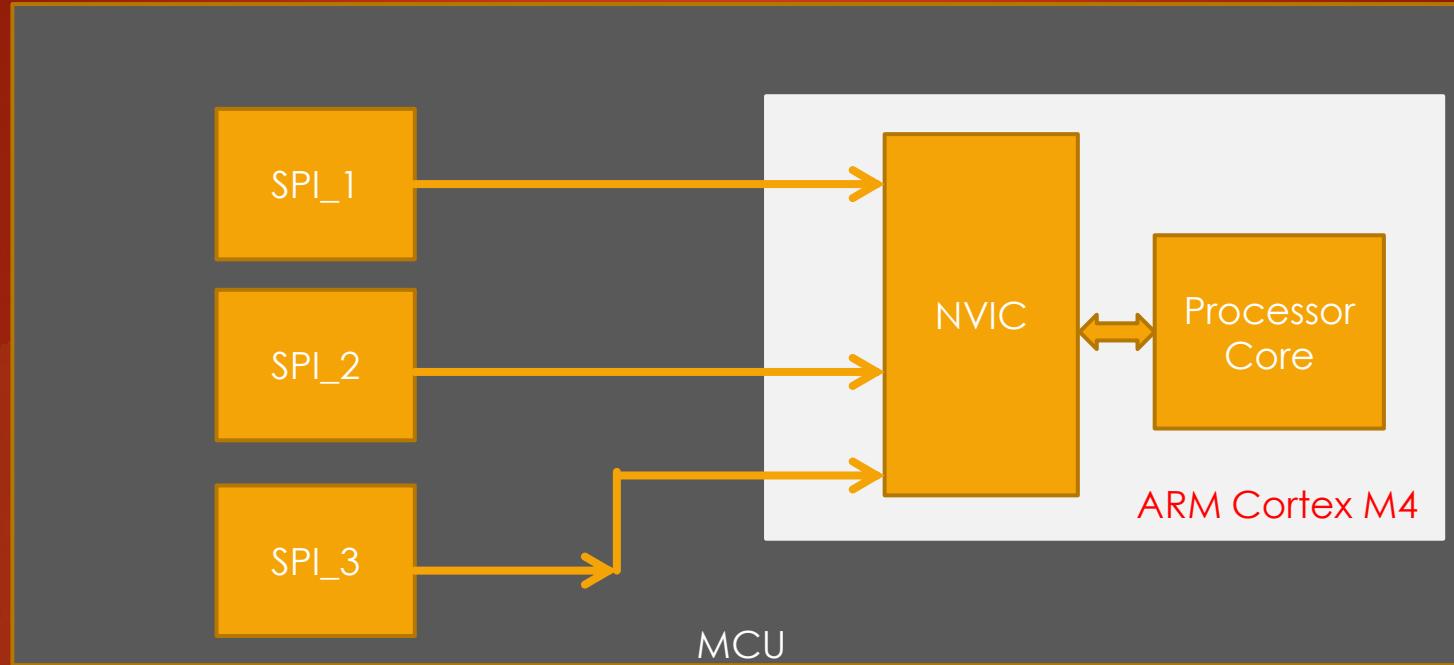
Interrupts can be enabled and disabled separately.

# SPI interrupts

**SPI interrupt requests**

Interrupt event	Event flag	Enable Control bit
Transmit Tx buffer ready to be loaded	TXE	TXEIE
Data received in Rx buffer	RXNE	RXNEIE
Master Mode fault event	MODF	
Overrun error	OVR	
CRC error	CRCERR	ERRIE
TI frame format error	FRE	

# SPI Interrupting the Processor



# Exercise-1

- ▶ Complete SPI IRQ number definition macros in MCU specific header file

# Exercise-2

- ▶ Complete the SPI IRQ Configuration APIs Implementation (reuse code from GPIO driver)

# SPI send data API (Interrupt mode)

# API to send data with interrupt mode

```
uint8_t SPI_SendDataWithIT(SPI_HandleTypeDef *pSPIHandle, uint8_t *pTxBuffer, uint8_t Len)
{
    //1 . Save the Tx buffer address and Len information in some global variables
    //2. Mark the SPI state as busy in transmission so that
    //    no other code can take over same SPI peripheral until transmission is over
    //3. Enable the TXEIE control bit to get interrupt whenever TXE flag is set in SR
    //4. Data Transmission will be handled by the ISR code ( will implement later)
}
```

# API to send data with interrupt mode

So first we have to create some place holder variables to save application's Tx address, len and SPI state.

# Handle Structure modification

```
/*
 *Handle structure for SPIx peripheral
 */
typedef struct
{
    SPI_RegDef_t      *pSPIx;    /*!< This holds the base address of SPIx(x:0,1,2) peripheral
    SPI_Config_t      SPIConfig;
    uint8_t            *pTxBuffer; /* !< To store the app. Tx buffer address > */
    uint8_t            *pRxBuffer; /* !< To store the app. Rx buffer address > */
    uint32_t           TxLen;     /* !< To store Tx len > */
    uint32_t           RxLen;     /* !< To store Rx len > */
    uint8_t            TxState;   /* !< To store Tx state > */
    uint8_t            RxState;   /* !< To store Rx state > */
}SPI_Handle_t;
```

# Possible SPI Application States

```
#define SPI_READY  
#define SPI_BUSY_IN_RX  
#define SPI_BUSY_IN_TX
```

0  
1  
2

# SPI receive data API (Interrupt mode)

# SPI receive data with interrupt

```
uint8_t SPI_ReceiveDataWithIT(SPI_HandleTypeDef *pSPIHandle, uint8_t *pRxBuffer, uint8_t Len)
{
    //1 . Save the Rx buffer address and Len information

    //2. Mark the SPI state as busy in reception so that
    //    no other code can take over same SPI peripheral until Reception is over

    //3. Enable the RXNEIE control bit to get interrupt whenever RXNE flag is set in SR

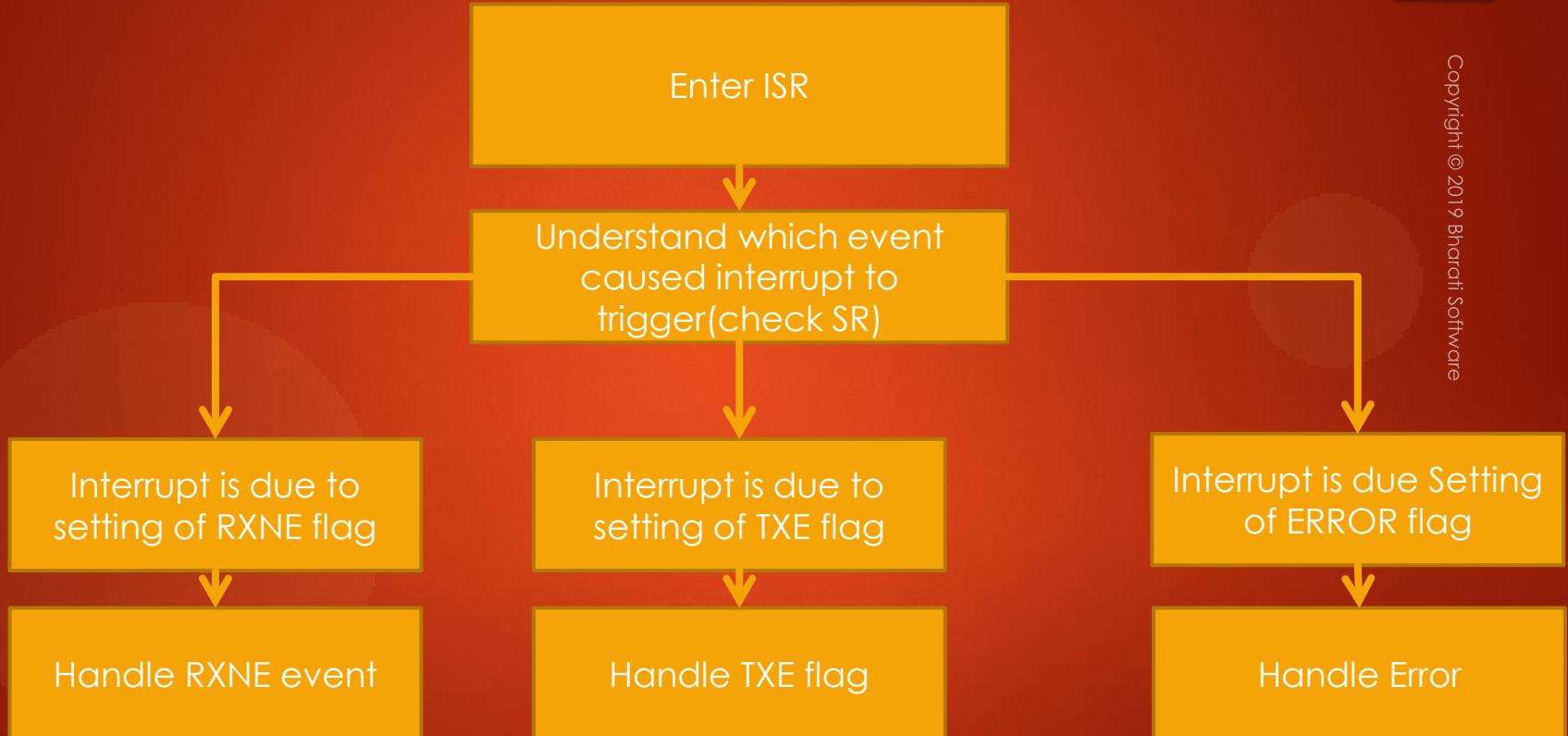
    //4. Data reception will be handled by the ISR code ( will implement later)

}
```

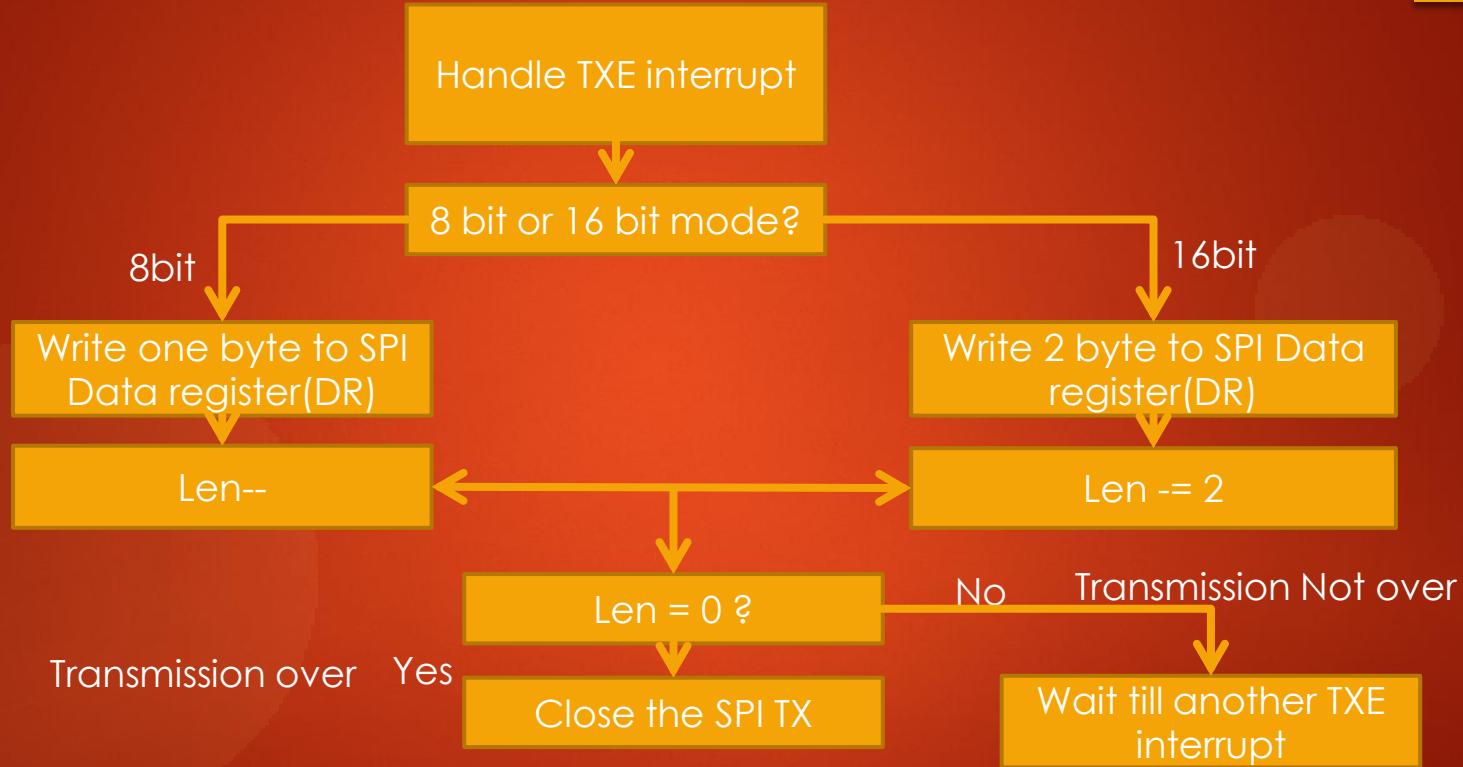
# SPI ISR Handling Implementation

# Handling SPI interrupt in the Code

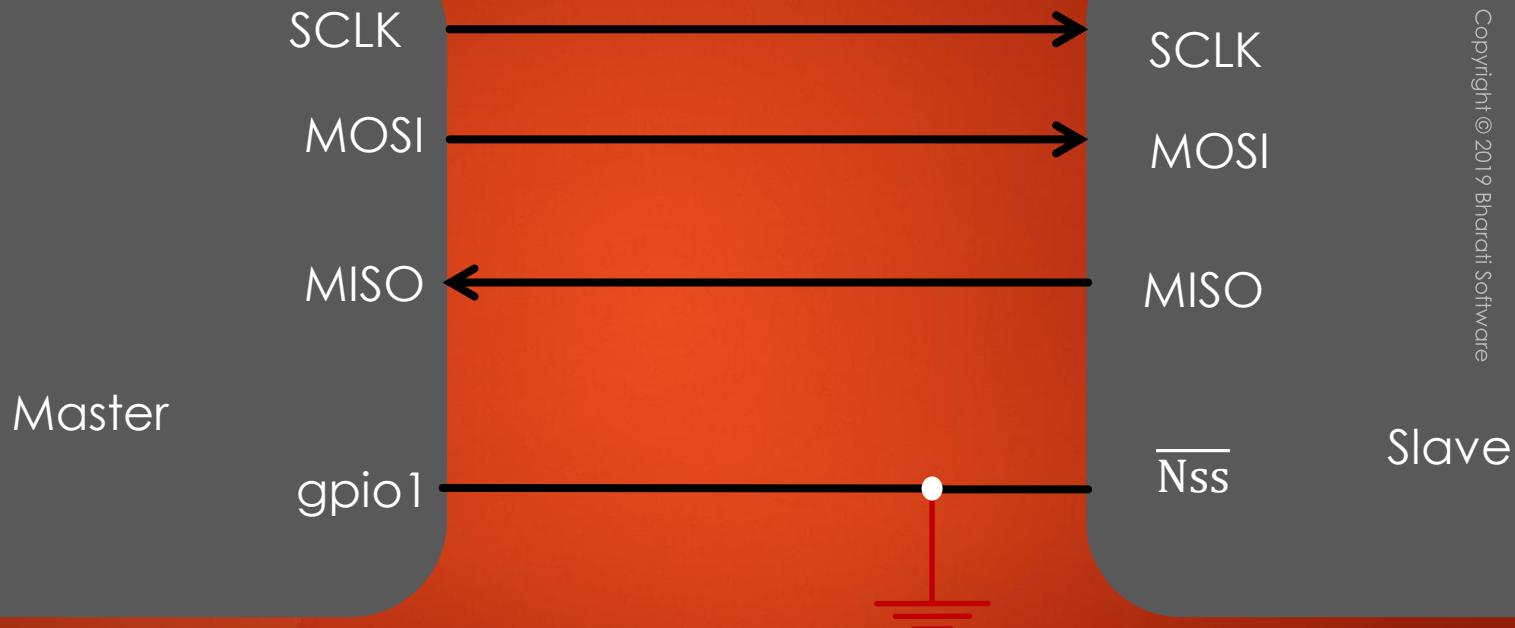
Copyright © 2019 Bharati Software



# Handling TXE interrupt



# Configuring NSS



In STM32F4xx based microcontroller, the NSS pin can be handled by 2 ways

- Software Slave Management
- Hardware Slave Management

# Software Slave Management

NSS Pin state is handled by '**SSI**' bit in the CR1 register

If **SSI** bit = 1 , then **NSS** goes **HIGH**



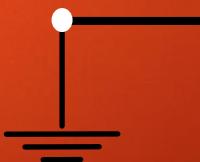
# Software Slave Management

NSS state is handled by '**SSI**' bit in the CR1 register

If **SSI** bit = 1 , then **NSS** goes **HIGH**

If **SSI** bit = 0, then **NSS** goes **LOW**

SSI=0



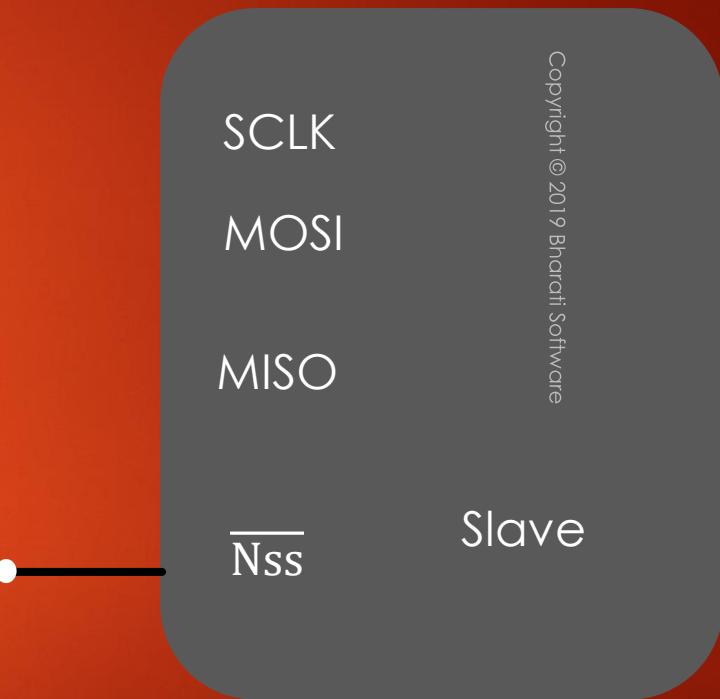
# Software Slave Management

NSS state is handled by '**SSI**' bit in the CR1 register

If **SSI** bit = 1 , then **NSS** goes **HIGH**

If **SSI** bit = 0, then **NSS** goes **LOW**

**SSI** bit is the handle for the software to control the **NSS** pin !



So, What is the advantage of using  
Software Slave Management(SSM) ?

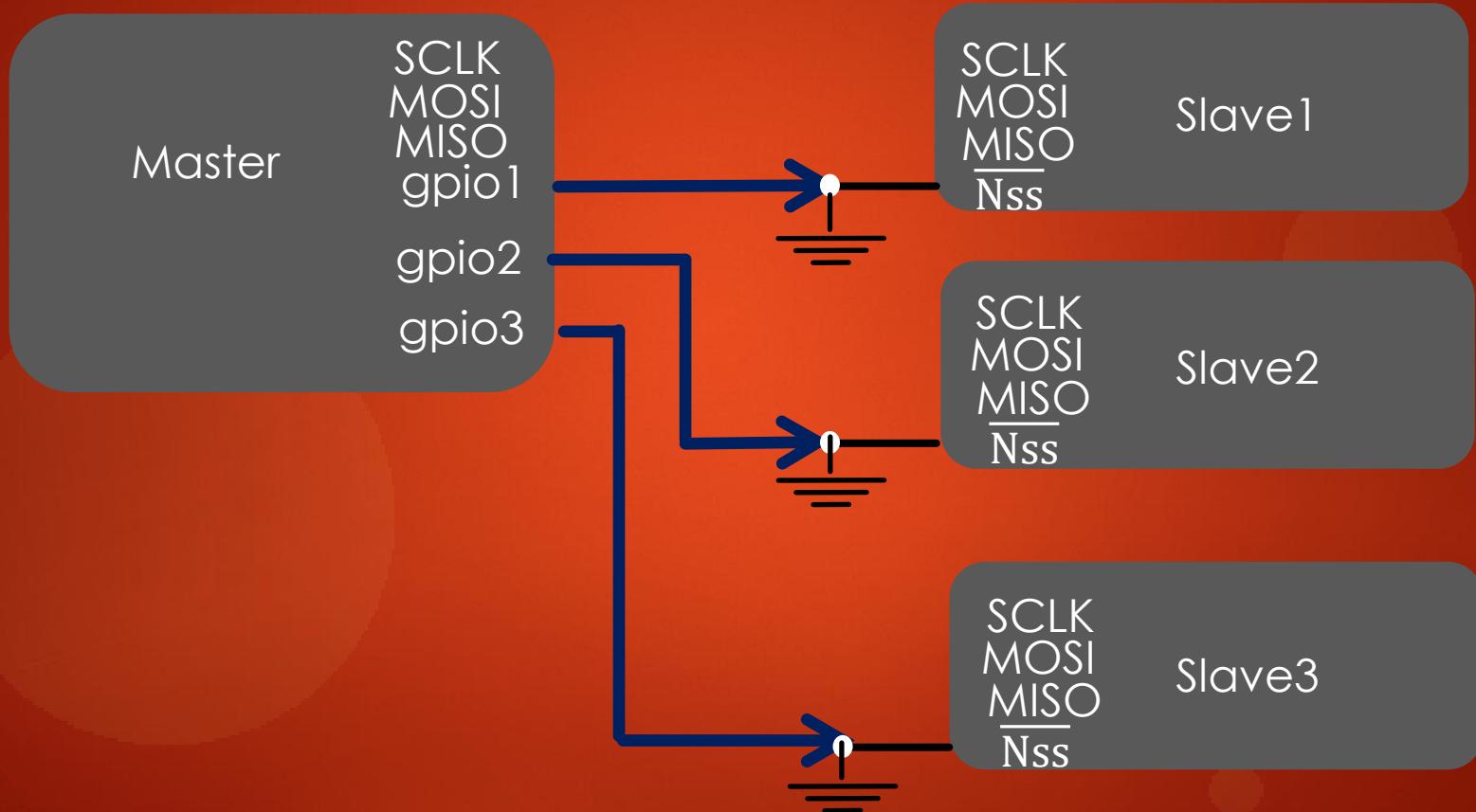
## Saves PIN



Master need not use its another pin to drive the NSS pin low, that saves one pin for master.

NSS pin is handled by SSI bit of Control register by software

# What is Hardware slave Management?



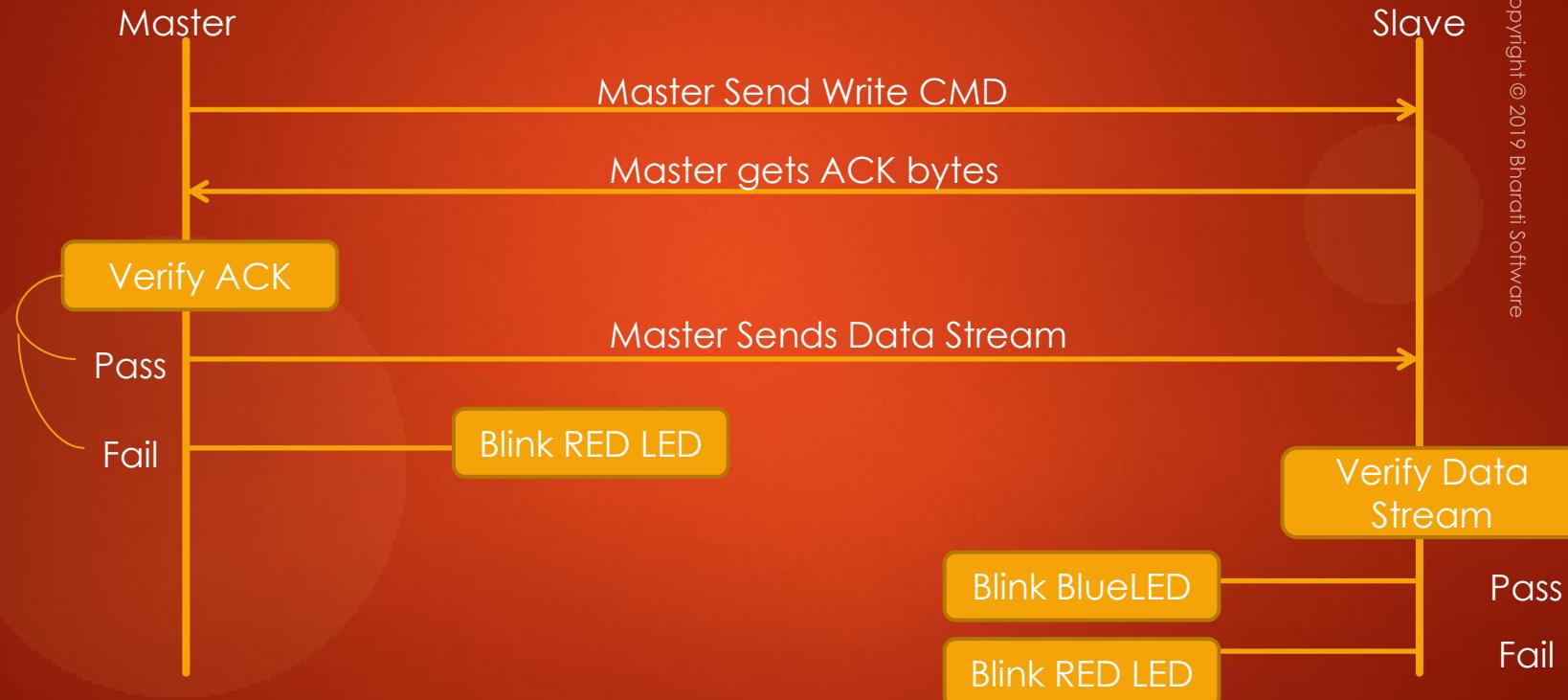
## Implementing Master rx API

If master wants **read** something form the slave, it has to **produce clock**, and to produce clock Master has to place some data in to the **tx buffer** by writing some **dummy** bytes.

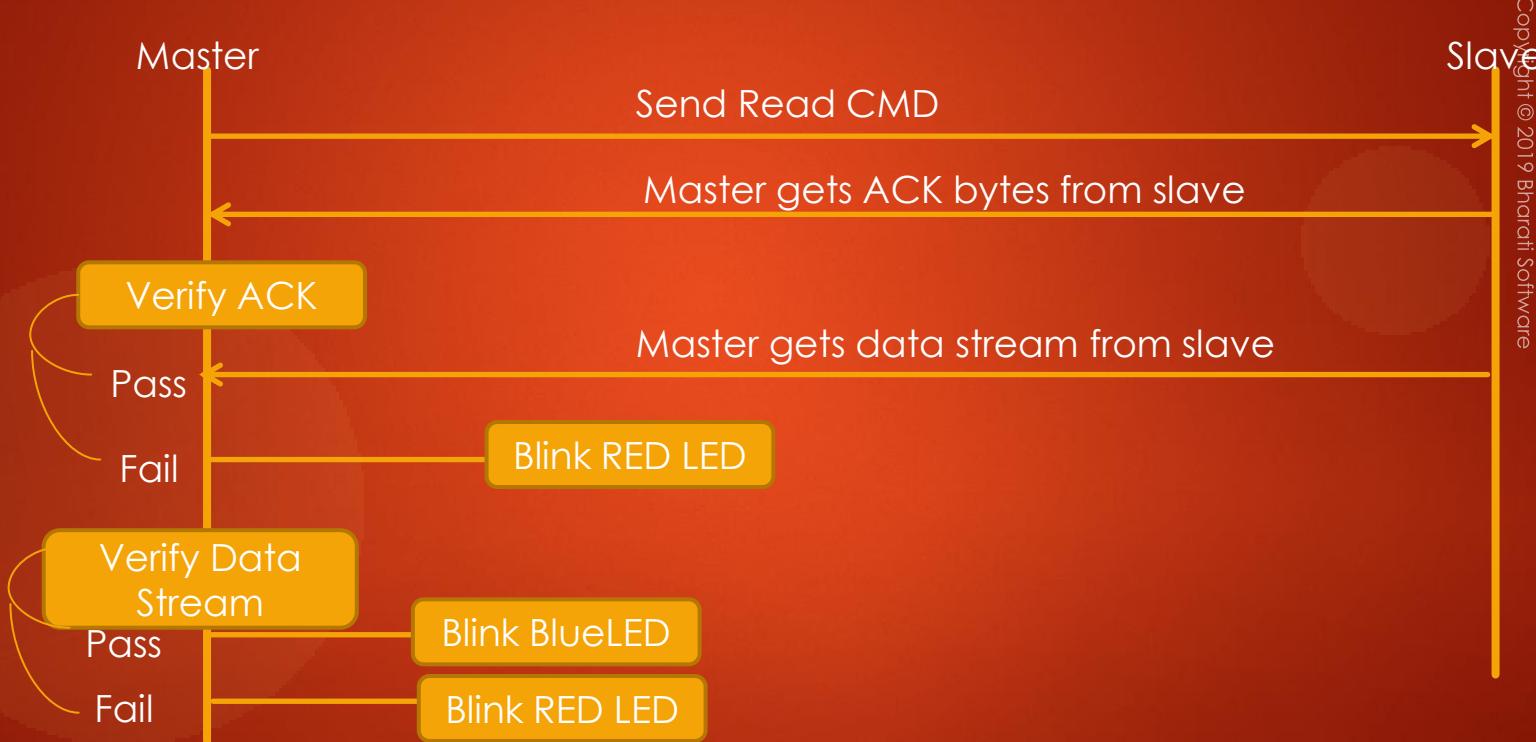
Slave can not do Tx, until master produces clock

Whenever you are in SPI slave mode and want to send some data, then slave must be ready with the data before master produces the clock.

# Master Sending Write CMD and Data Stream to Slave



# Master Sending Read CMD and Receiving Data Stream From Slave



# Master Write CMD Execution

- ▶ Master sends out the Write CMD
- ▶ Slave ACK back for the command
- ▶ If ACK is valid, then Master sends out the Data stream of known length

# Master Read CMD Execution

- ▶ Master sends out the Read CMD
- ▶ Slave ACK back for the command
- ▶ If ACK is valid , then Master gets the data stream from the slave of known length.

We selected spi2 device,  
The clock will run at 500KHz  
Data format is 8bit MSbfirst  
Spi mode is 0, because CPOL and CPHASE are 0  
The peripheral acts as master device.

# SPI Master Initialization

Copyright © 2019 Bharati Software

- ▶ We selected SPI2 device.
- ▶ SPI serial line clock will run at 500KHz
- ▶ Data format is 8bit msb first
- ▶ SPI mode is 1, because CPO=0 and CPHASE=1
- ▶ The peripheral acts as master device

SPI peripheral clock is 16Mhz, because we are running MCU at 16Mhz Internal RC Oscillator

# Common Debugging Steps

Copyright © 2019 Bharati Software

- ✓ Master mode bit must be enabled in the configuration register if you are making peripheral to work in MASTER mode.
- ✓ SPI peripheral enable bit must be enabled
- ✓ SPI peripheral clock must be enabled



# Common Problems in SPI and Debugging Tips

Copyright © 2019 Bharati Software



# Case 1

Master Can not able to produce clock and data



# Case 1: Master cannot able to produce clock and data

Reason-1:

Non-Proper Configuration of I/O lines for Alternate functionality

Debug Tip:

Recheck the GPIO Configuration registers to see what values they have



# Case 1: Master cannot able to produce clock and data

Reason-2:  
Configuration Overriding

Debug Tip:  
Dump out all the required register contents just before you begin the transmission



# Case 2

Master is sending data, but slave is not receiving data !



## Case 2: Master is sending data but slave is not receiving data !

Reason-1:

Not pulling down the slave select pin to ground before sending data to the slave

Copyright © 2019 Bharatij Software

Debug Tip:

Probe through the logic analyzer to confirm slave select line is really low during data communication



## Case 2: Master is sending data but slave is not receiving data !

Reason-2 :  
Non-Proper Configuration of I/O lines for Alternate functionality

Debug Tip:  
Probe the alternate function register



## Case 2: Master is sending data but slave is not receiving data !

Reason-3 :

Non enabling the peripheral IRQ number in the NVIC

Copyright © 2019 Bharati Software

Debug Tip:

Probe the NVIC Interrupt Mask register to see whether the bit position corresponding to the IRQ number is set or not



# Case 3

SPI interrupts are not getting triggered

Copyright © 2019 Bharati Software



# Case 3: SPI interrupts are not getting triggered

Reason-1 :

Not enabling the TXE or RXNE interrupt in the SPI configuration register

Debug Tip:

Check the SPI configuration register to see TXEIE and RXNEIE bits are really set to enable the interrupt !



# Case 3: SPI interrupts are not getting triggered

Copyright © 2019 Bharati Software

Reason-2:

Non enabling the peripheral IRQ number in the NVIC

Debug Tip:

Probe the NVIC Interrupt Mask Register to see whether the bit position corresponding to the IRQ number is set or not



# Case 4

Master is producing right data but slave is receiving the different data



## Case 4: Master is producing right data but slave is receiving the different data

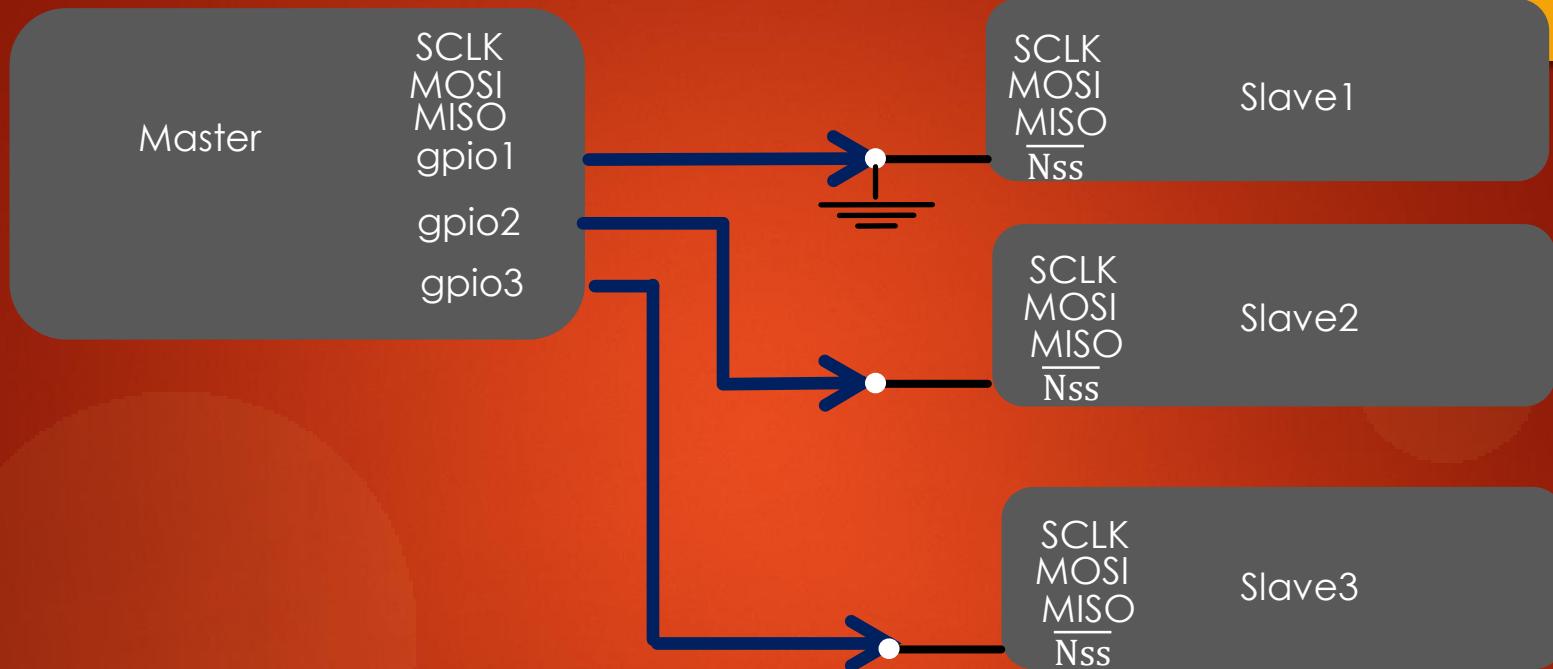
Reason-1 :

Using Long Wires in high frequency communication

Debug Tip:

use shorter wires or reduce the SPI serial frequency to 500KHz to check things work well





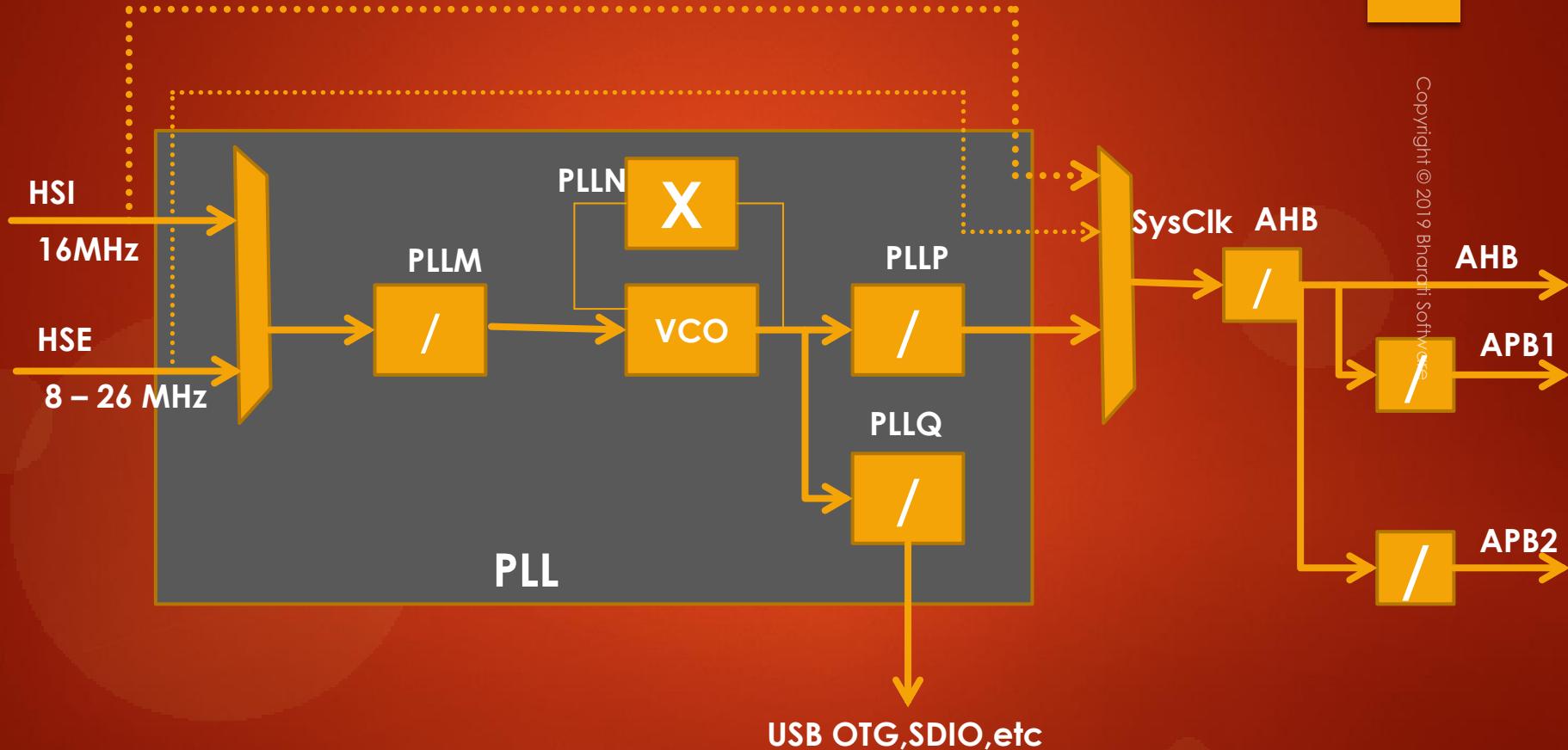
# Cross Check These Settings

- ▶ Master mode bit must be enabled in the configuration register if you are configuring the peripheral as master
- ▶ SPI peripheral enable bit must be enabled .
- ▶ SPI peripheral clock must be enabled. By default clocks to almost all peripheral in the microcontroller will be disabled to save power.

# Clock Sources in the MCU

- High Speed Internal (HSI) oscillator
- High Speed External (HSE) oscillator
- Phase Locked Loop (PLL)
- Low Speed Internal (LSI) clock
- Low Speed External (LSE) clock

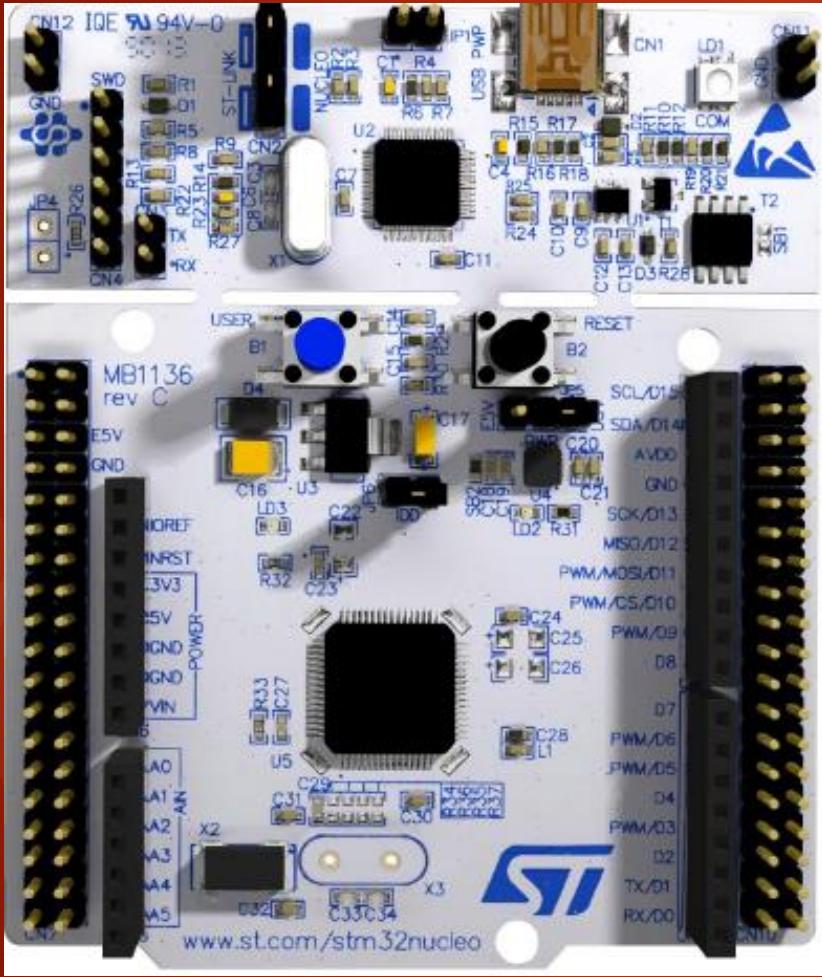
# Clock Sources in the MCU



The HSI RC oscillator which is inside the MCU has the advantage of providing a clock source at low cost . Because there is no external components required to use this clock. It also has a faster start-up time than the external crystal oscillator however, the frequency is less accurate than an external crystal oscillator.

# STM32F4xx Discovery board

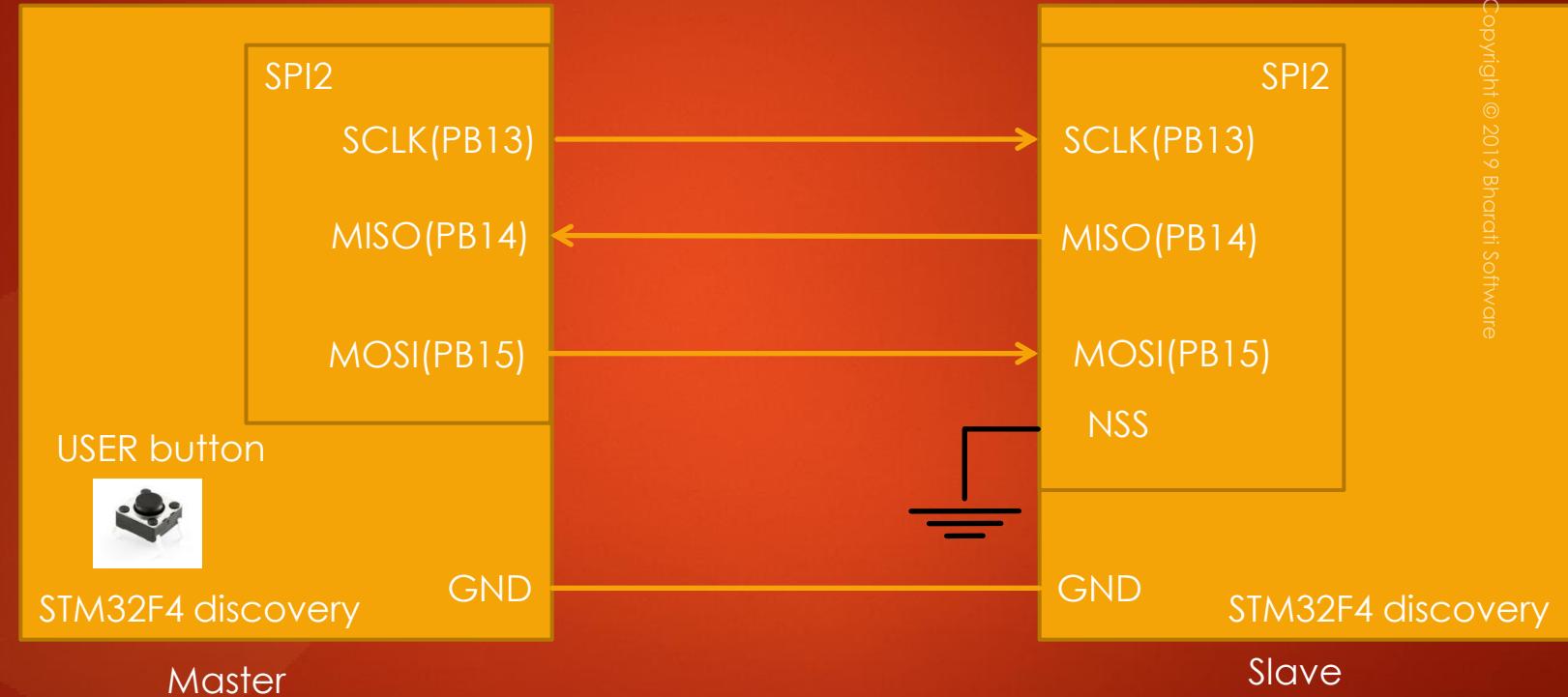




# STM32F411RE Nucleo-64

Copyright © 2019 Bharati Software

# Connection Diagram

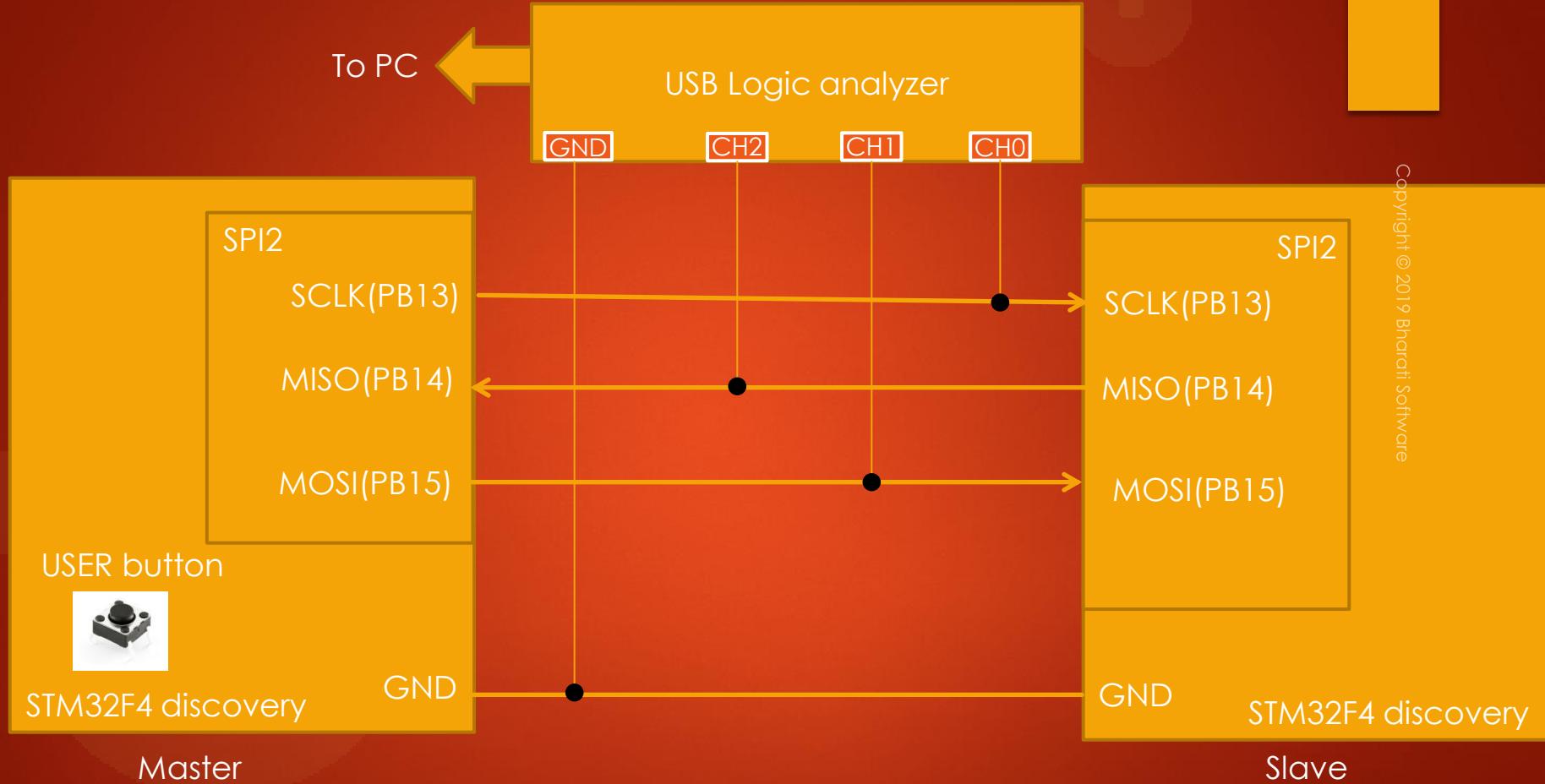


# How to enable the High Speed External Crystal Oscillator and use it as System Clock ?

Let's measure the frequency of  
different clock sources like  
HSI,HSE,PLL by using USB logic  
Analyzer !

# Connection between USB Logic Analyzer and Discovery board

# Understanding Connection Diagram



PART-2

# Mastering Microcontroller with Embedded Driver Development

FastBit Embedded Brain Academy  
Check all online courses at  
[www.fastbitlab.com](http://www.fastbitlab.com)

# About FastBit EBA

FastBit Embedded Brain Academy is an online training wing of Bharati Software.

We leverage the power of internet to bring online courses at your fingertip in the domain of embedded systems and programming, microcontrollers, real-time operating systems, firmware development, Embedded Linux.

All our online video courses are hosted in Udemy E-learning platform which enables you to exercise 30 days no questions asked money back guarantee.

For more information please visit : [www.fastbitlab.com](http://www.fastbitlab.com)

Email : [contact@fastbitlab.com](mailto:contact@fastbitlab.com)

# Introduction to I2C

# Inter-Integrated Circuit(I2C) Protocol

Copyright © 2019 Bharati Software

Pronounced as

“I squared C “ or “ I two C “

# What is I2C ?

It is just a protocol to achieve serial data communication between integrated circuits(ICs) which are very close to each other. (but more serious protocol than SPI because companies have come forward to design a specification )

I2C protocol details ( how data should sent, how data should received, how hand shaking should happen between sender and receiver, error handling, ) are complex than SPI. ( In other words SPI is simple protocol compared to I2C)

# Difference between SPI and I2C

# How different than SPI ?

## Specification



Copyright © 2019 Bharat Software

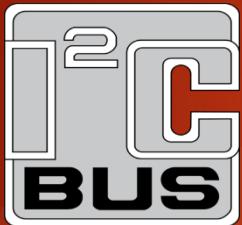
I2c is based on dedicated specification. The spec you  
can download from here

<https://www.nxp.com/docs/en/user-guide/UM10204.pdf>

For SPI there is no dedicated spec but TI and Motorola have their  
own spec

# How different than SPI ?

## Multi-Master Capability



Copyright © 2011 STMicroelectronics

I2C protocol is multi-master capable , whereas SPI has no guidelines to achieve this, but depends on MCU designers . STM SPI peripherals can be used in multi master configurations but arbitration should be handled by software code.

# How different than SPI ?

Copyright © 2011 Texas Instruments Incorporated

ACK

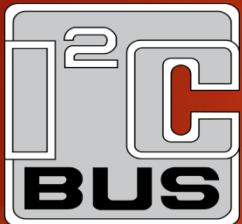


I2C hardware automatically ACKs every byte received.  
SPI does not support any automatic ACKing.

# How different than SPI ?

Copyright © 2011, PPT by S. R. Srinivasan

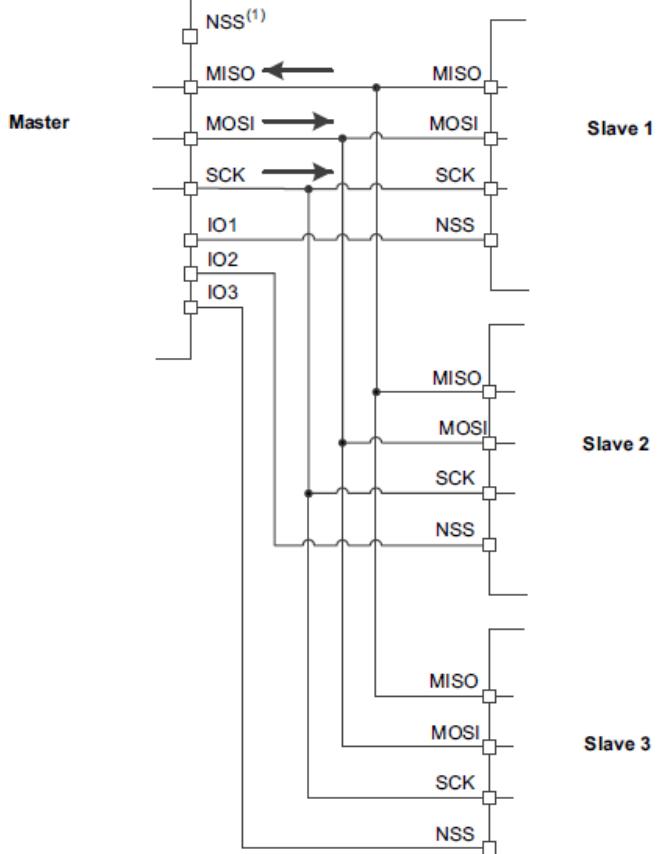
## Pins



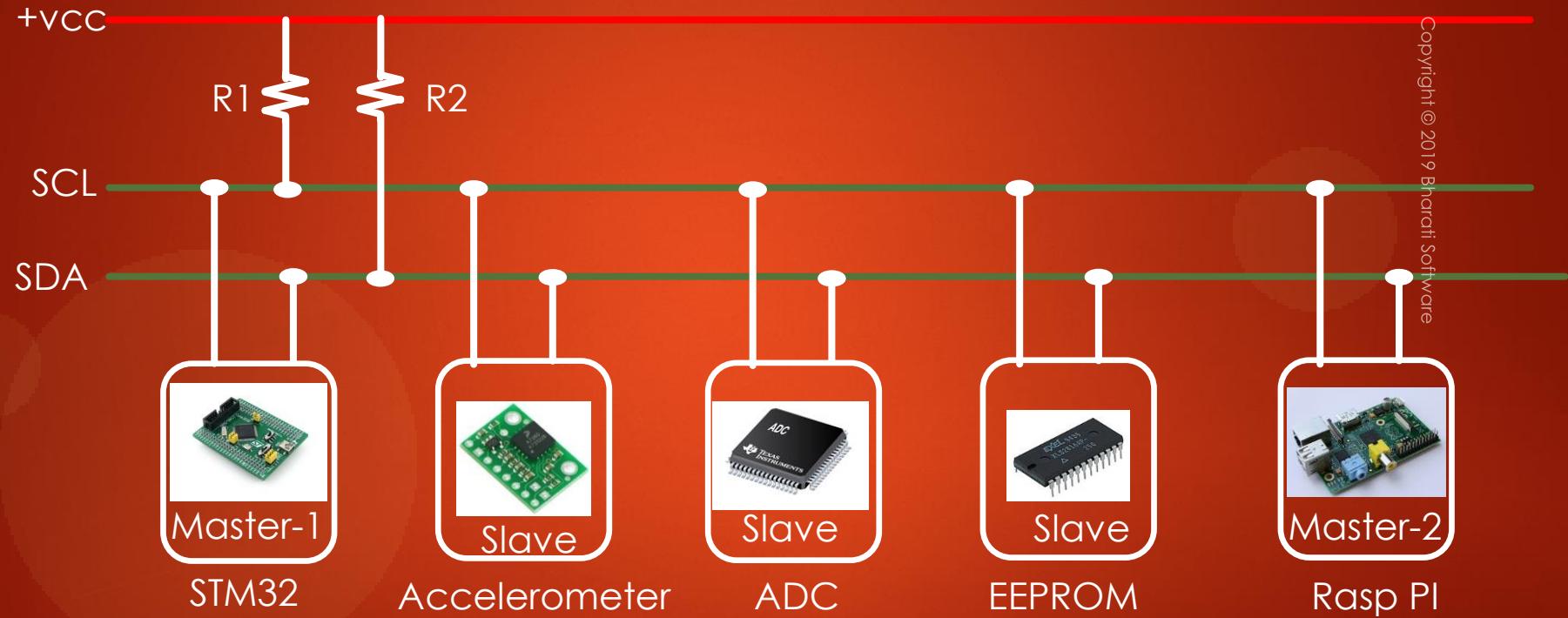
I2C needs just 2 pins for the communication  
whereas SPI may need 4 pins and even more than that  
if multiple slaves are involved

SPI Consumes more pins when more slaves are involved

### Master and three independent slaves

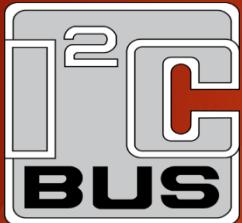


In I2C, you just need 2 pins to connect all the slaves and masters



# How different than SPI ?

## Addressing

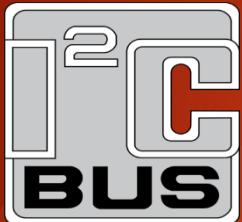


Copyright © 2011 Texas Instruments Incorporated

I2C master talks to slaves based on slave addresses, whereas in SPI dedicated pin is used to select the slave.

# How different than SPI ?

## Communication



Copyright © 2011 Texas Instruments Incorporated

I2C is half duplex, where is SPI is full duplex

# How different than SPI ?

## Speed



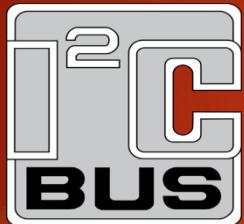
Copyright © 2011 STMicroelectronics

For I2C the max speed is 4MHz in ultra speed plus .  
For some STM microcontrollers the max speed is just 400KHz.

For SPI max speed is its  $F_{pclk}/2$  . That means if  
the peripheral clock is 20MHz, then speed can be 10MHz

# How different than SPI ?

Slave's control over serial clock



Copyright © 2011 by Texas Instruments Inc.

In I2C slave can make master wait by holding the clock down if it's busy , thanks to clock stretching feature of I2C.  
But in SPI, slave has no control over clock, programmers may use their own tricks to overcome this situation .

# Data rate

Data rate ( number of bits transferred from sender to receiver in 1 sec ) is very much lesser in I2C compared to SPI.

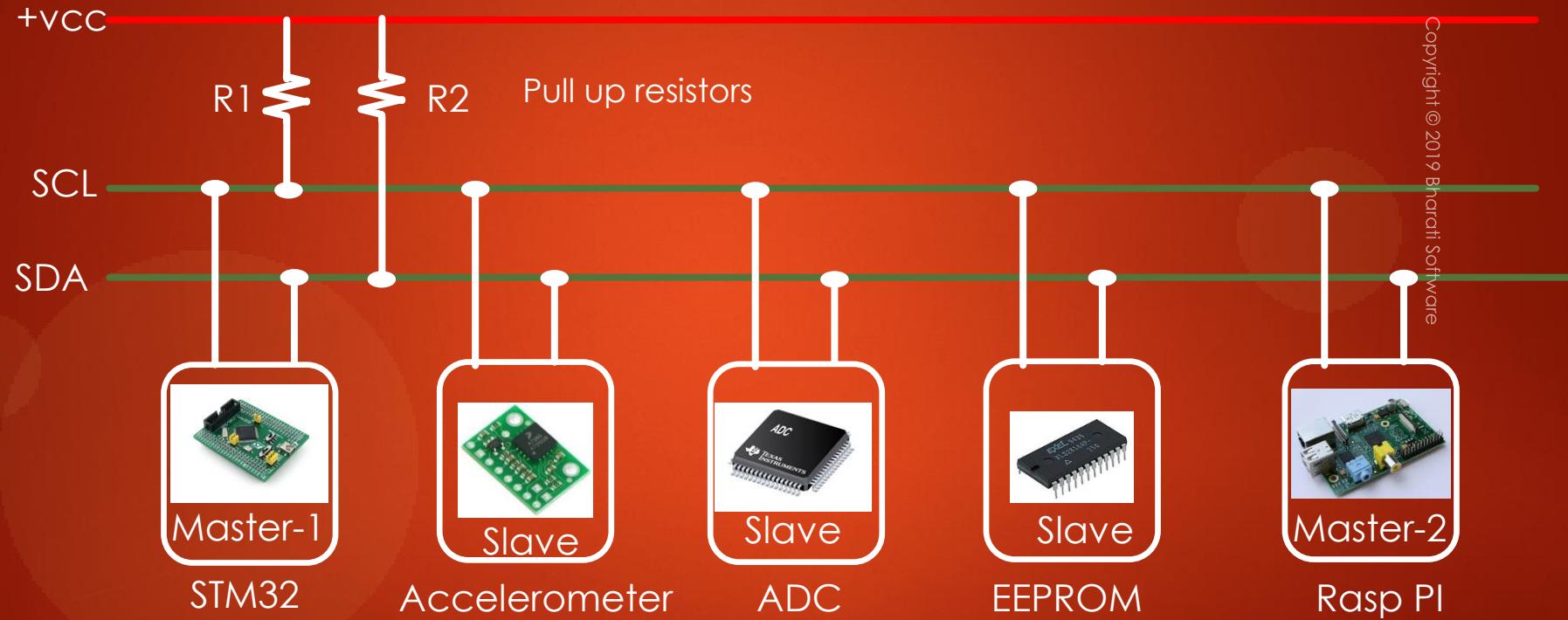
For example in STM32F4X if you have peripheral clock of 40MHz, then in I2C you can achieve data rate of 400Kbps but in SPI it is 20Mbps.

So in the above scenario SPI is 50 times faster than I2C.

## Definition of I<sup>2</sup>C-bus terminology

Term	Description
Transmitter	the device which sends data to the bus
Receiver	the device which receives data from the bus
Master	the device which initiates a transfer, generates clock signals and terminates a transfer
Slave	the device addressed by a master
Multi-master	more than one master can attempt to control the bus at the same time without corrupting the message
Arbitration	procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the winning message is not corrupted
Synchronization	procedure to synchronize the clock signals of two or more devices

In I2C, you just need 2 pins to connect all the slaves and masters

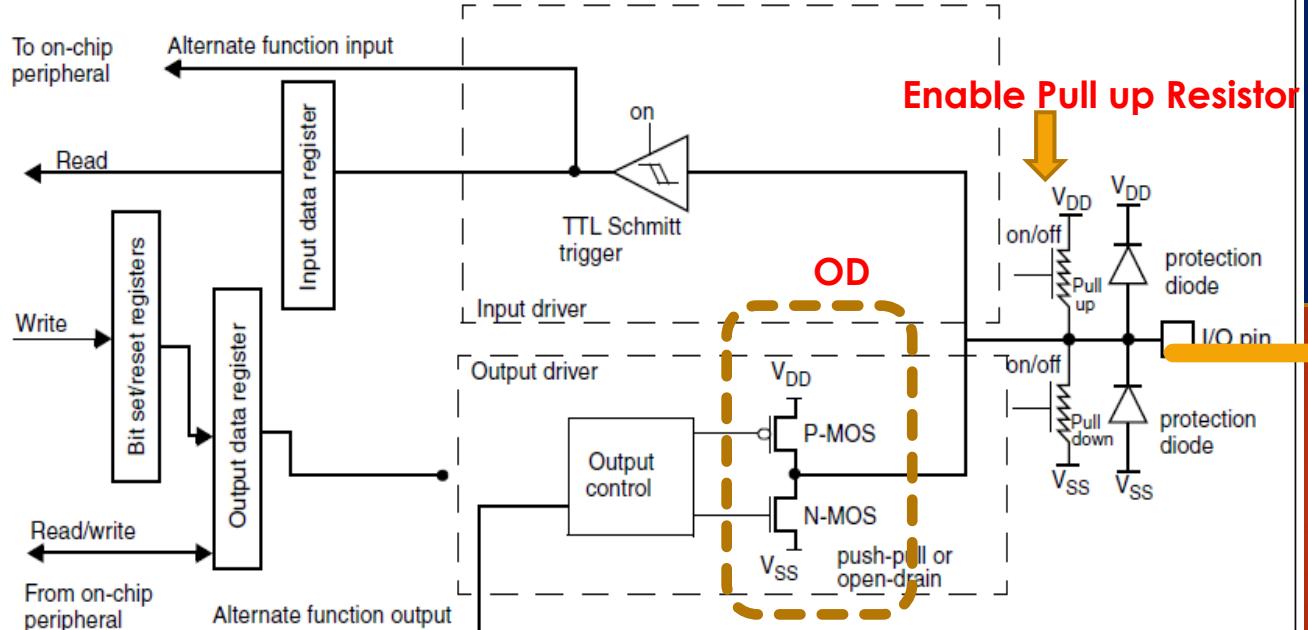


# SDA and SCL signals

- ✓ Both SDA and SCL are bidirectional lines connected to a positive supply voltage via pull-up resistors. When the bus is free, both lines are held at HIGH.
- ✓ The output stages of devices connected to the bus must have an open-drain or open-collector configuration
- ✓ The bus capacitance limits the number of interfaces connected to the bus.

# I2C: Pin Configuration

## Alternate function configuration

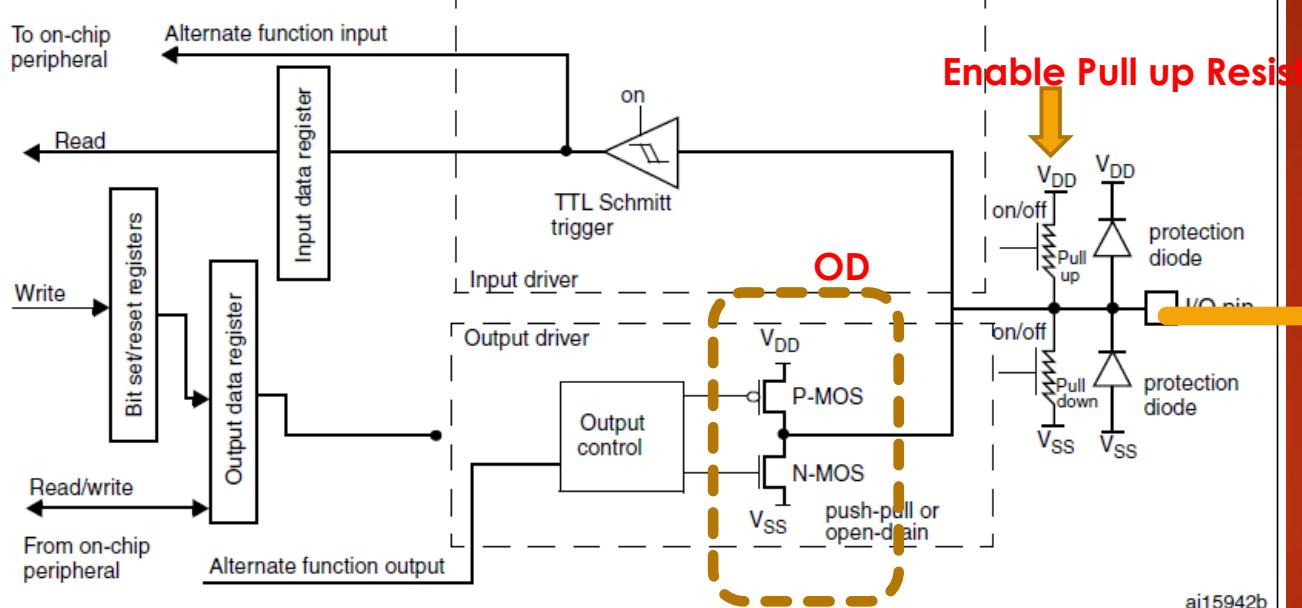


For proper functioning  
of the I2C bus pull up  
resistor value has to be  
calculated according  
to the I2C formula (will  
discuss later)

Copyright 2003, Burr-Brown Software

SCL

## Alternate function configuration



For proper functioning of the I2C bus, pull up resistor value has to be calculated according to the I2C formula (will discuss later)

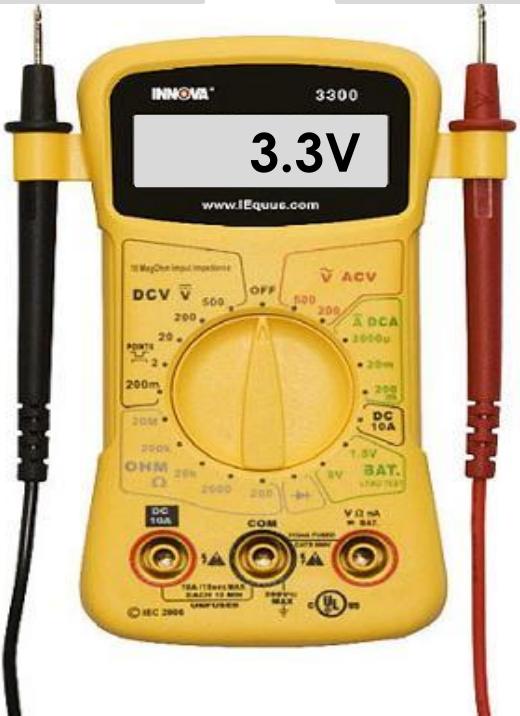
Copyright © 2003 by Microchip Technology Inc.

SDA

Trouble shooting tip : When the bus is idle, both SDA and SCL are pulled to  $+V_{DD}$

GND

SDA & SCL



## Tip

Whenever you face problems in I<sub>2</sub>C , probe the SDA and SCL line after I<sub>2</sub>C initialization. It must be held at HIGH (3.3 V or 1.8V depending up on IO voltage levels of your board)

# I2C Modes

Mode	Data rate	Notes
Standard Mode	Up to 100 Kbits/sec	Supported by STMf2F4x
Fast Mode	Up to 400Kbits/sec	Supported by STMf2F4x
Fast Mode +	Up to 1Mbits/sec	Supported by some STMf2F4x MCUs (refer RM)
High Speed mode	Up to 3.4 Mbits/sec	Not supported by F4x

# Fast Mode/Standard Mode

# Standard Mode

- ▶ In standard mode communication data transfer rate can reach up to maximum of 100kbits/sec.
- ▶ Standard mode was the very first mode introduced when first i2c spec was released.
- ▶ Standard-mode devices, however, are not upward compatible; They cant communicate with devices of Fast mode or above.

# Fast Mode

- ▶ Fast mode is a mode in i2c protocol, where devices can receive and transmit data up to 400 kbits/s.
- ▶ Fast-mode devices are downward-compatible and can communicate with Standard-mode devices in a 0 to 100 kbit/s I2C-bus system
- ▶ Standard-mode devices, however, are not upward compatible; they should not be incorporated in a Fast-mode I2C-bus system as they cannot follow the higher transfer rate and unpredictable states would occur.
- ▶ To achieve data transfer rate up to 400kbits/sec, you must put the i2c device in to fast mode

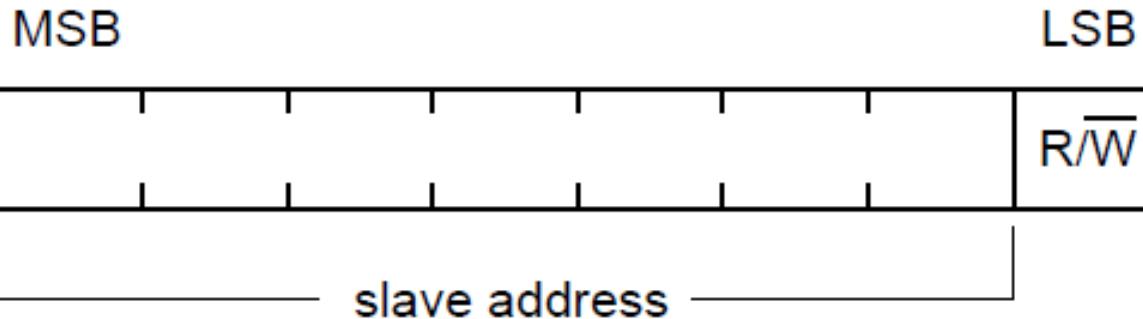
# Basics of I2C Protocol

R/nW = '1' indicates  
request for data (READ)

R/nW = '0' indicates a  
transmission (WRITE),



- ✓ Every byte put on the SDA line must be eight bits long.
- ✓ Each byte must be followed by an Acknowledge Bit
- ✓ Data is transferred with the Most Significant Bit (MSB) first



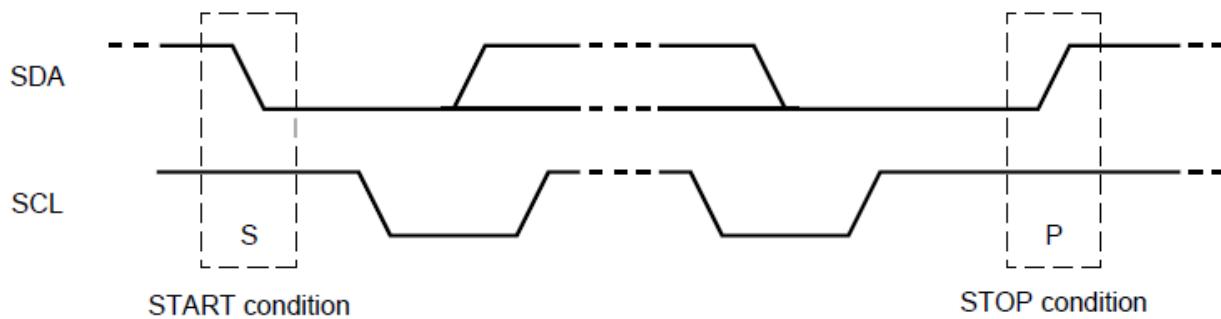
*mbc608*

**The first byte after the START procedure**

# START and STOP conditions

All transactions begin with a START (S) and are terminated by a STOP (P)

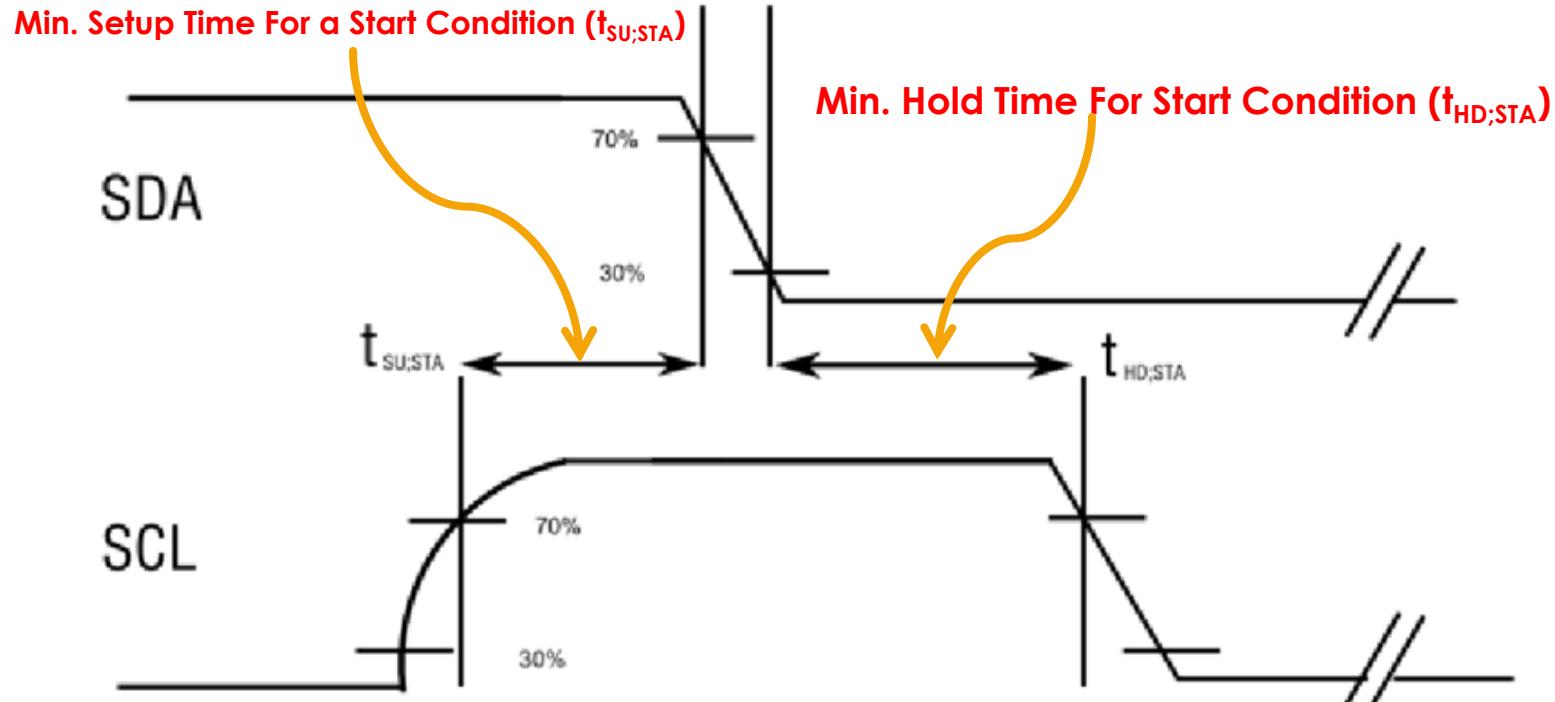
A HIGH to LOW transition on the SDA line while SCL is HIGH defines a START condition.  
A LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition.



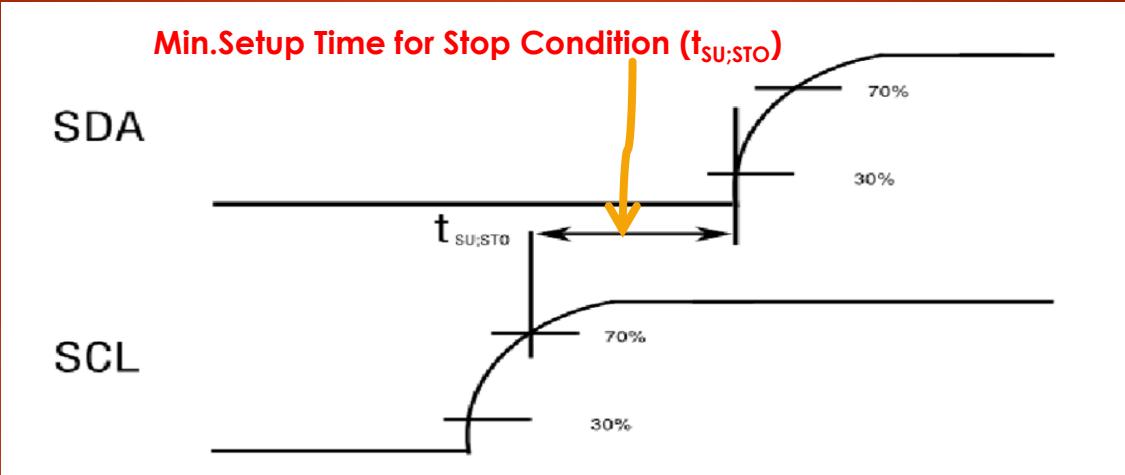
mba608

### START and STOP conditions

# I2C Protocol: Start condition timings



# I2C Protocol: stop condition timings



Setup Time for Stop Condition ( $t_{SU;STO}$ ) is measured as the time between 70% amplitude of the rising edge of SCL and 30% amplitude of a rising SDA signal during a stop condition.

# Points to Remember

- ▶ START and STOP conditions are always generated by the master. The bus is considered to be busy after the START condition.
- ▶ The bus is considered to be free again a certain time after the STOP condition.
- ▶ When the bus is free another master(if present) can get the chance to claim the bus.
- ▶ The bus stays busy if a repeated START (Sr) is generated instead of a STOP condition.
- ▶ Most of the MCU's I2C peripherals support both master and slave mode. You need not to configure the mode because when the peripheral generates the start condition it automatically becomes the master and when it generates the stop condition it goes back to slave mode.

# I2C Protocol: Address Phase

# I2C Protocol: Address Phase



# I2C Protocol: ACK/NACK

# I2C Protocol: ACK



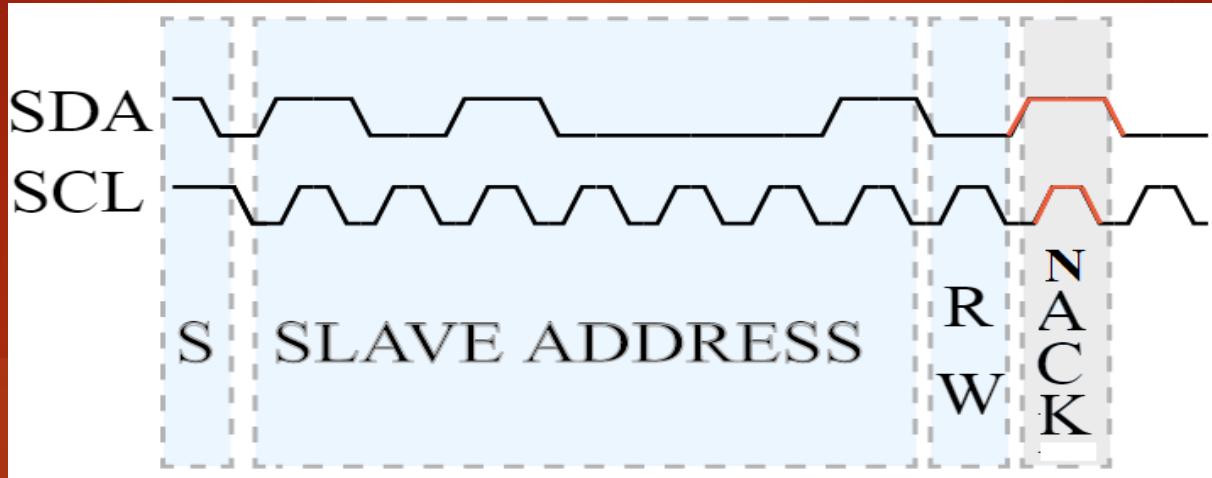
**The Acknowledge signal is defined as follows:**

The transmitter releases the SDA line during the acknowledge clock pulse so the receiver can pull the SDA line LOW and it remains stable LOW during the HIGH period of this clock pulse

Copyright © 2018 Baranit Software

- ✓ The acknowledge takes place after every byte.
- ✓ The acknowledge 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, including the acknowledge ninth clock pulse

# I2C Protocol: NACK

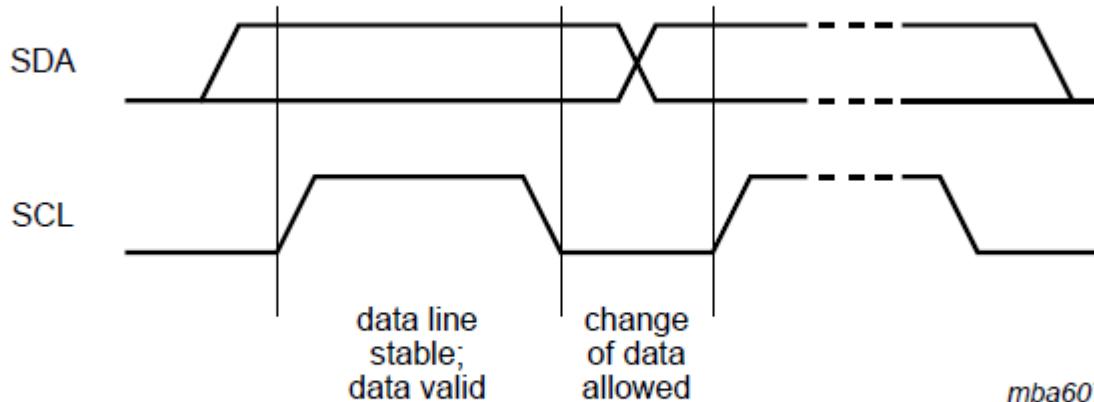


- ✓ When SDA remains HIGH during this ninth clock pulse, this is defined as the Not Acknowledge signal.
- ✓ The master can then generate either a STOP condition to abort the transfer, or a repeated START condition to start a new transfer.

# Data validity

Copyright

The data on the SDA line must be stable during the HIGH period of the clock. The HIGH Or LOW state of the data line can only change when the clock signal on the SCL line is LOW . One clock pulse is generated for each data bit transferred.

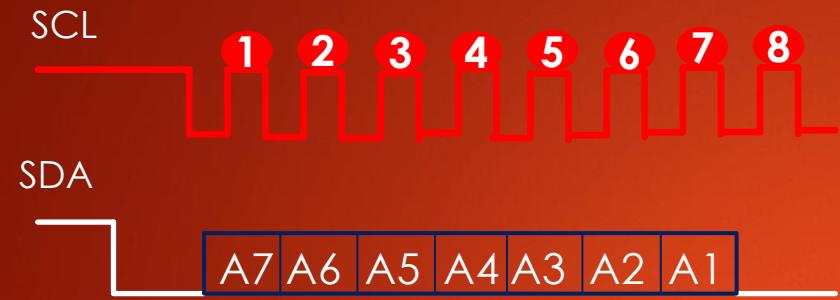


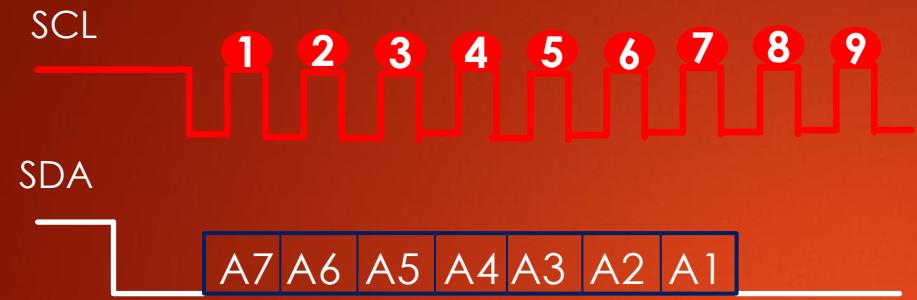
mba607

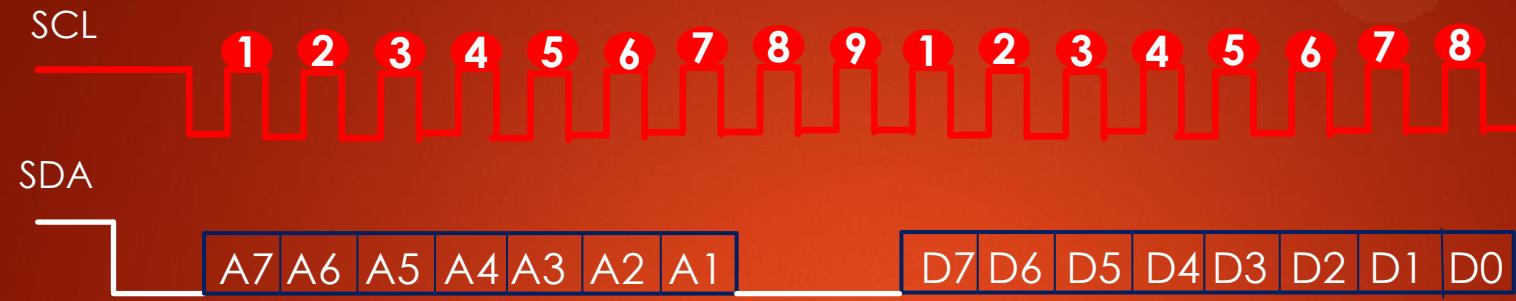
Bit transfer on the I<sup>2</sup>C-bus

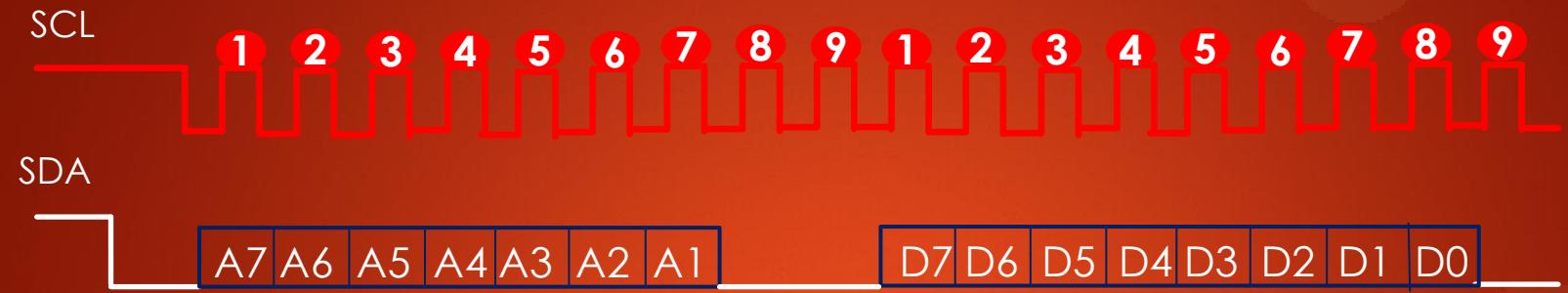
# Master Writing data to slave

Copyright © 2019 Bharati Software

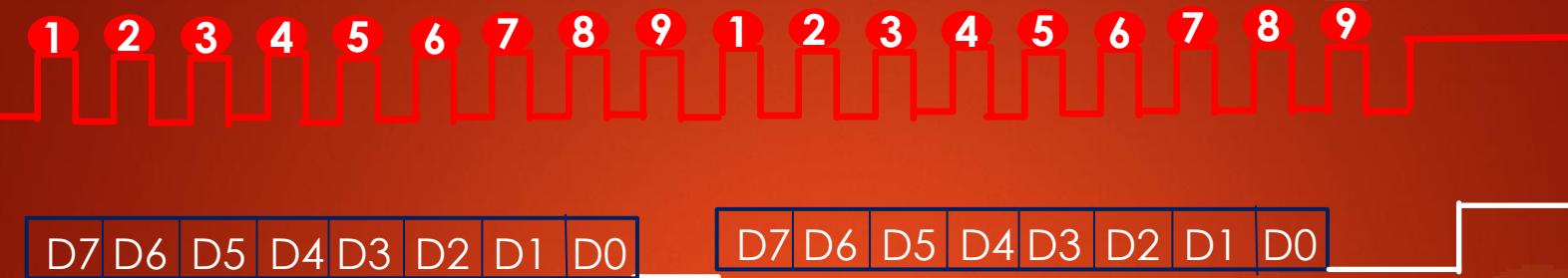








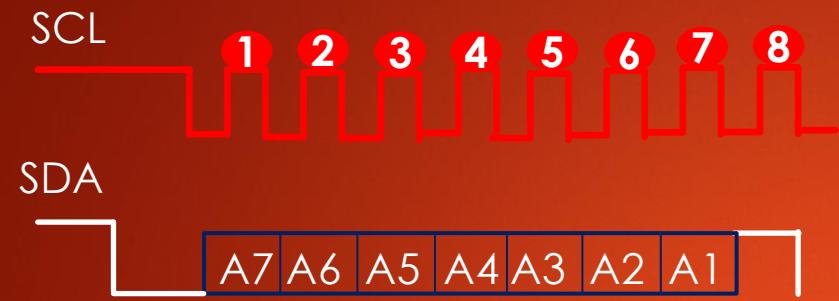


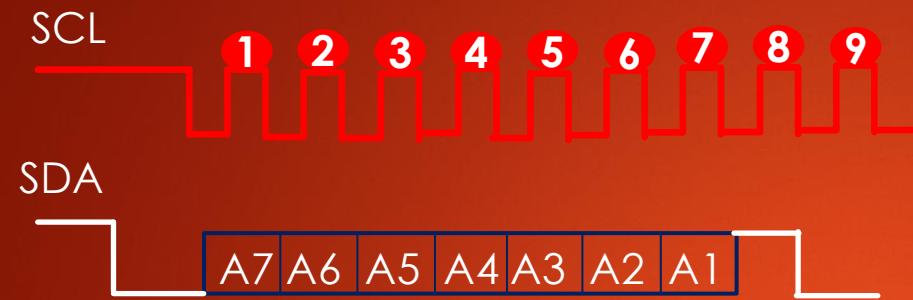


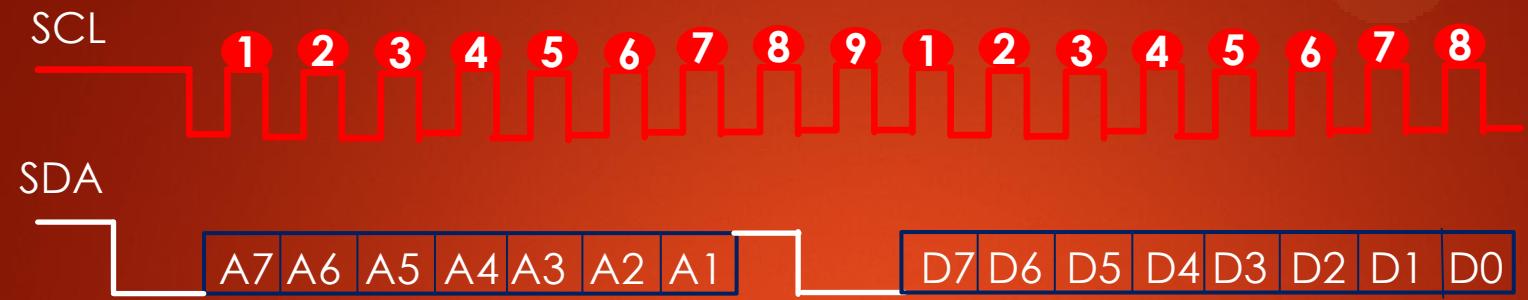
# Master Reading data from slave

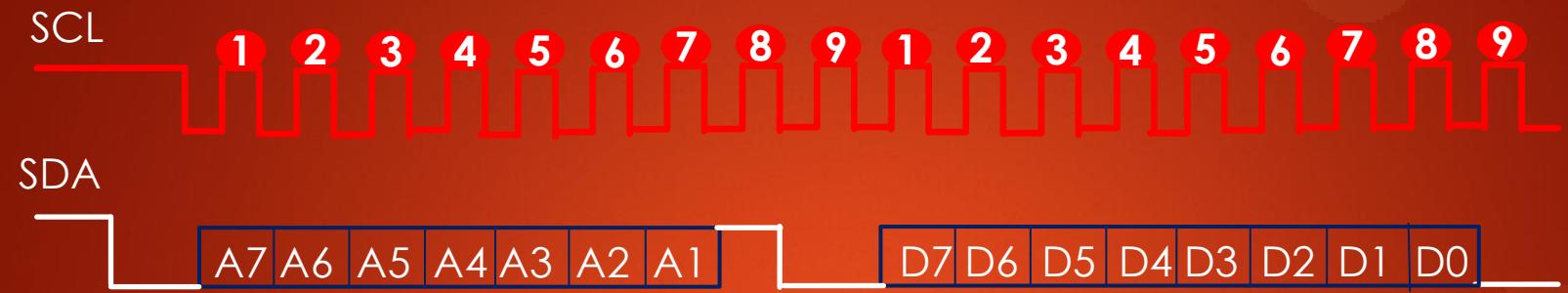
Copyright © 2019 Bharati Software

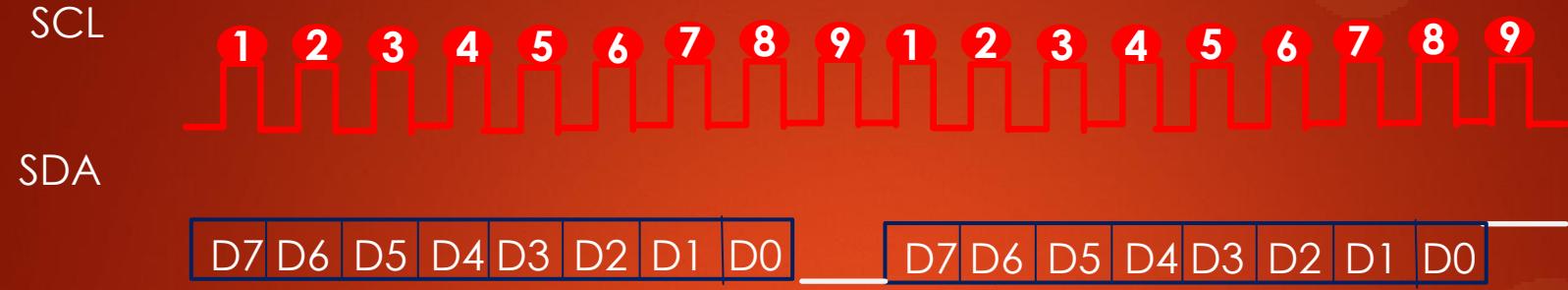












SCL

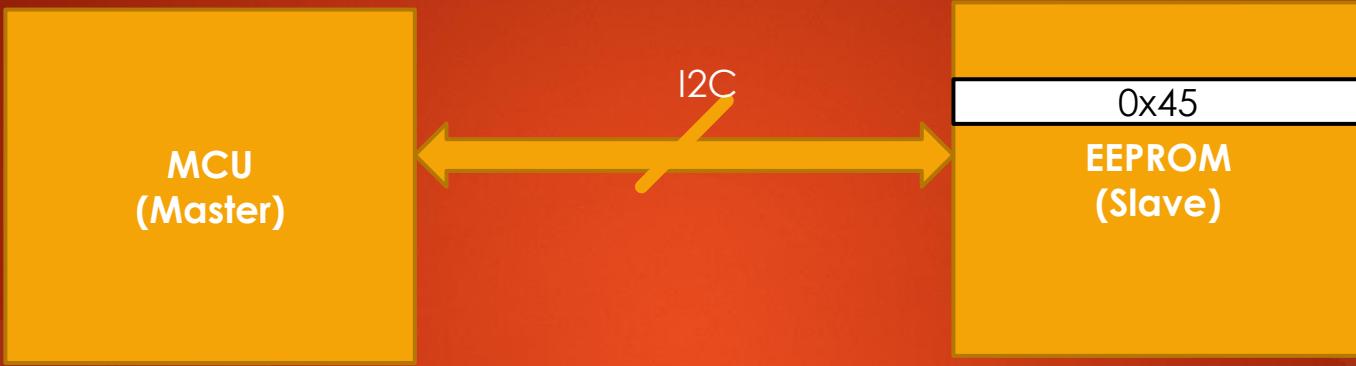


SDA



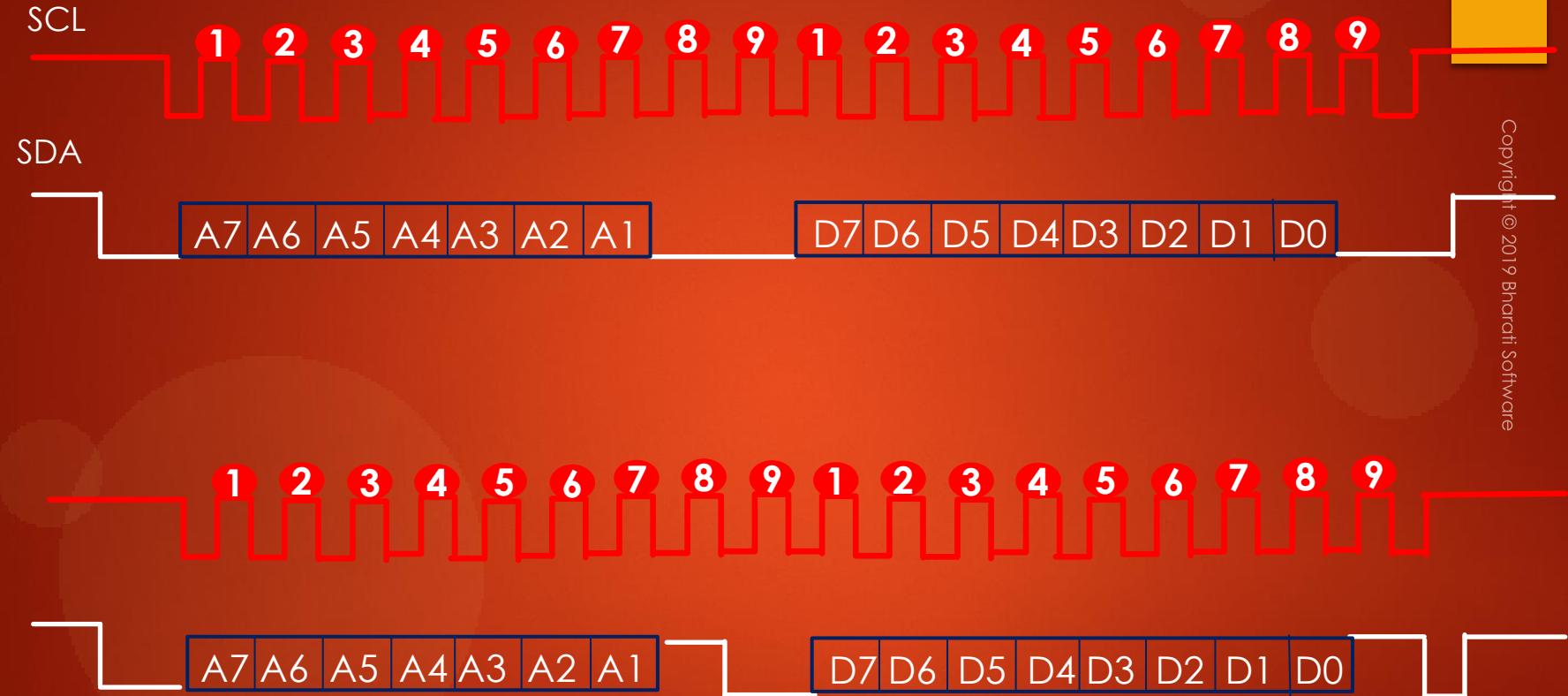
# Repeated Start( $S_r$ )

(Start again without Stop)

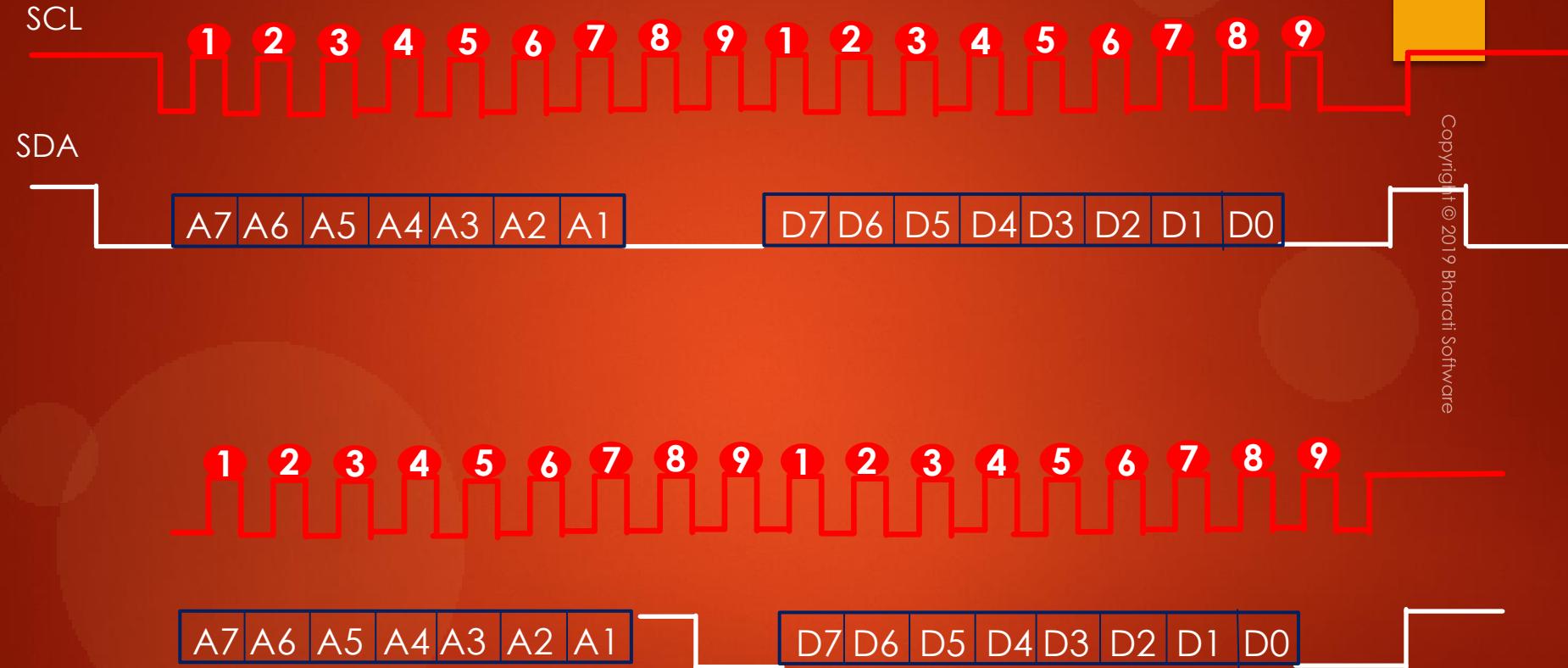


Master reading the contents of EEPROM at address 0x45

# Without repeated start

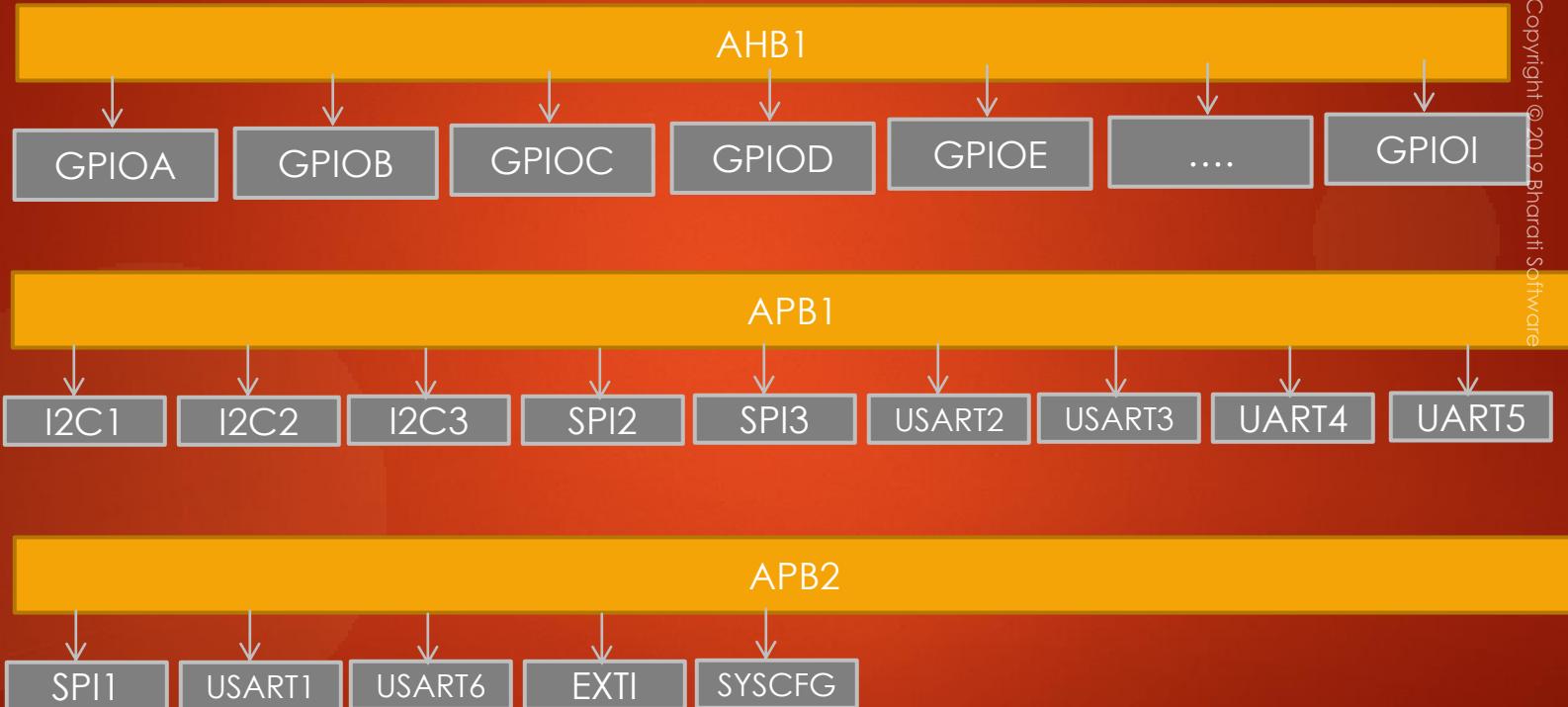


# With repeated start

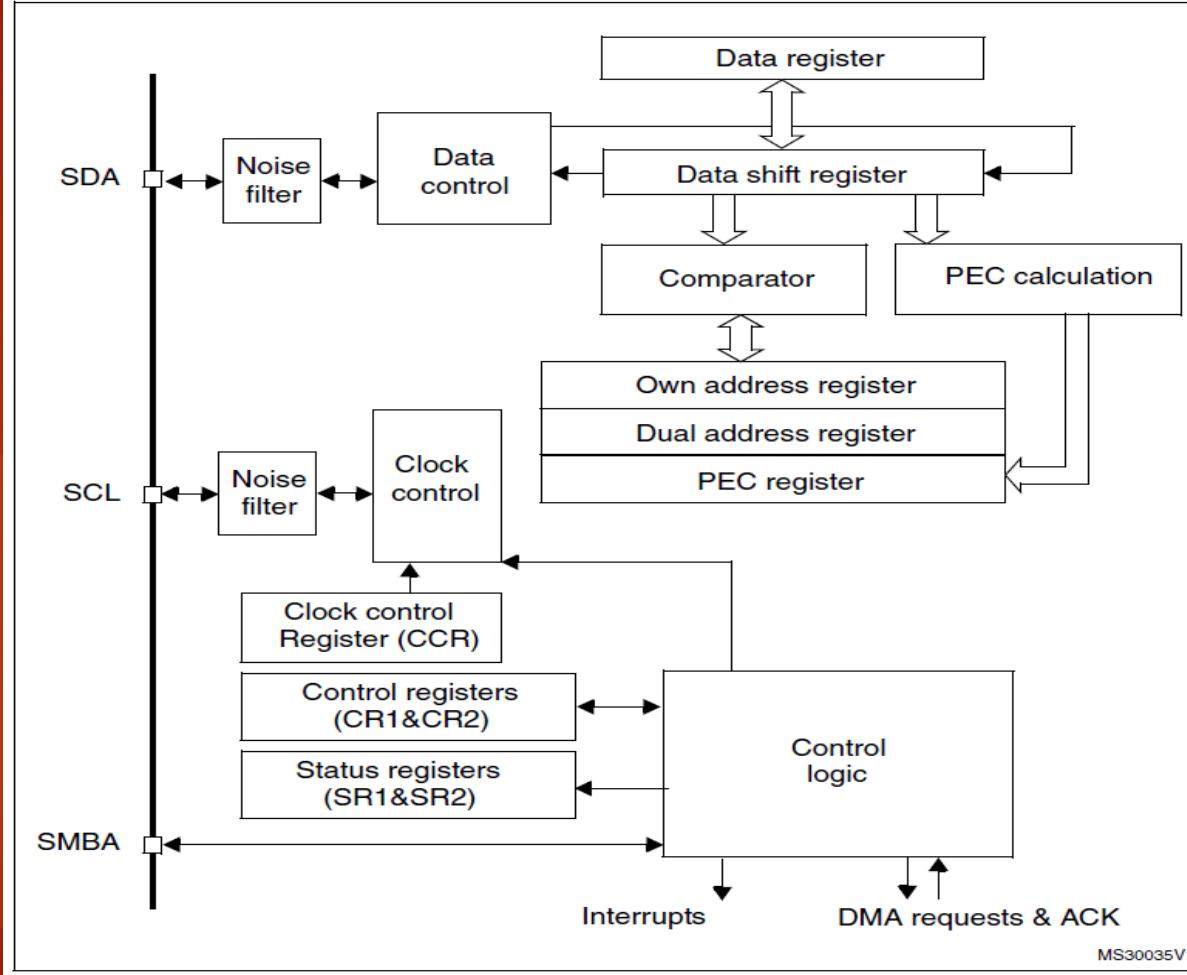


# I2C peripherals of your MCU

Copyright © 2012 Bharati Software



## I<sup>2</sup>C block diagram for STM32F40x/41x



Copyright © 2019 Bharati Software

# I2C Driver Development

# Sample Applications

Copyright © 2019 Bharatii Software

## Driver Layer

gpio\_driver.c , .h

i2c\_driver.c , .h

(Device header)

Stm3f407xx.h

spi\_driver.c , .h

uart\_driver.c , .h

GPIO

SPI

I2C

UART

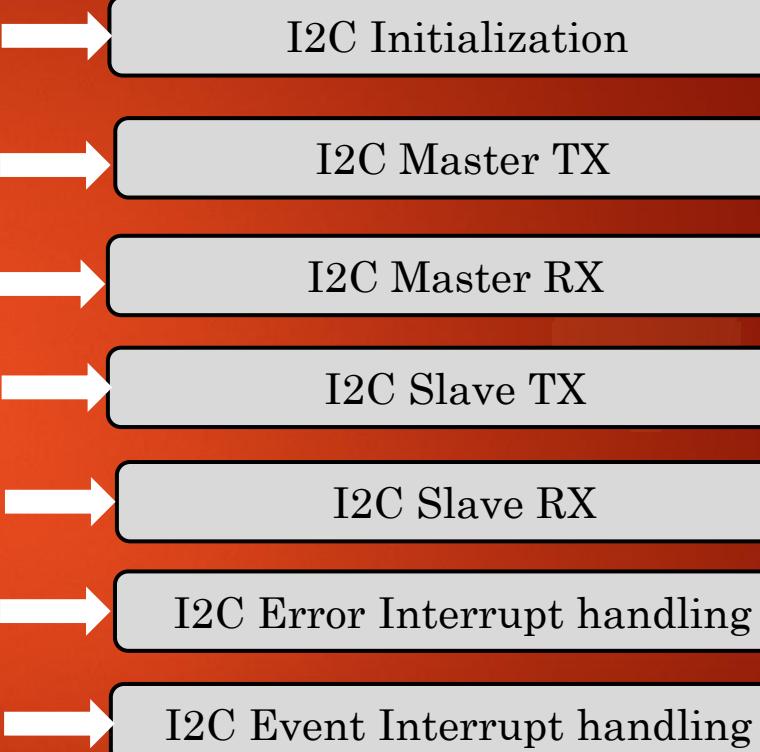
**STM3F407x MCU**

# Driver API requirements and user configurable items

I2C Driver



APIs



I2Cx  
Peripheral



I2C\_SCLSpeed

I2C\_DeviceAddress

I2C\_ACKControl

I2C\_FMDutyCycle

Configurable items  
For user application

# Exercise:

1. Create `stm32f407xx_i2c_driver.c` and `stm32f407xx_i2c_driver.h`
2. Add I2Cx related details to MCU specific header file
  - I. I2C peripheral register definition structure
  - II. I2Cx base address macros
  - III. I2Cx peripheral definition macros
  - IV. Macros to enable and disable I2Cx peripheral clock
  - V. Bit position definitions of I2C peripheral

# I2C handle structure and configuration structure

## Create Configuration and Handle structure

```
/*
 * Configuration structure for I2Cx peripheral
 */
typedef struct
{
    uint32_t I2C_SCLSpeed;
    uint8_t I2C_DeviceAddress;
    uint8_t I2C_ACKControl;
    uint16_t I2C_FMDDutyCycle;
}I2C_Config_t;

/*
 *Handle structure for I2Cx peripheral
 */
typedef struct
{
    I2C_RegDef_t *pI2Cx;
    I2C_Config_t I2C_Config;
}I2C_Handle_t;
```

- ▶ check comments

# Implementation **I2C\_Init()** API

# Steps for I2C init (Generic)

1. Configure the Mode (standard or fast )
2. Configure the speed of the serial clock (SCL)
3. Configure the device address (Applicable when device is slave)
4. Enable the Acking
5. Configure the rise time for I2C pins (will discuss later )

All the above configuration must be done when the peripheral is disabled in the control register

# I2C Serial clock (SCL) control settings

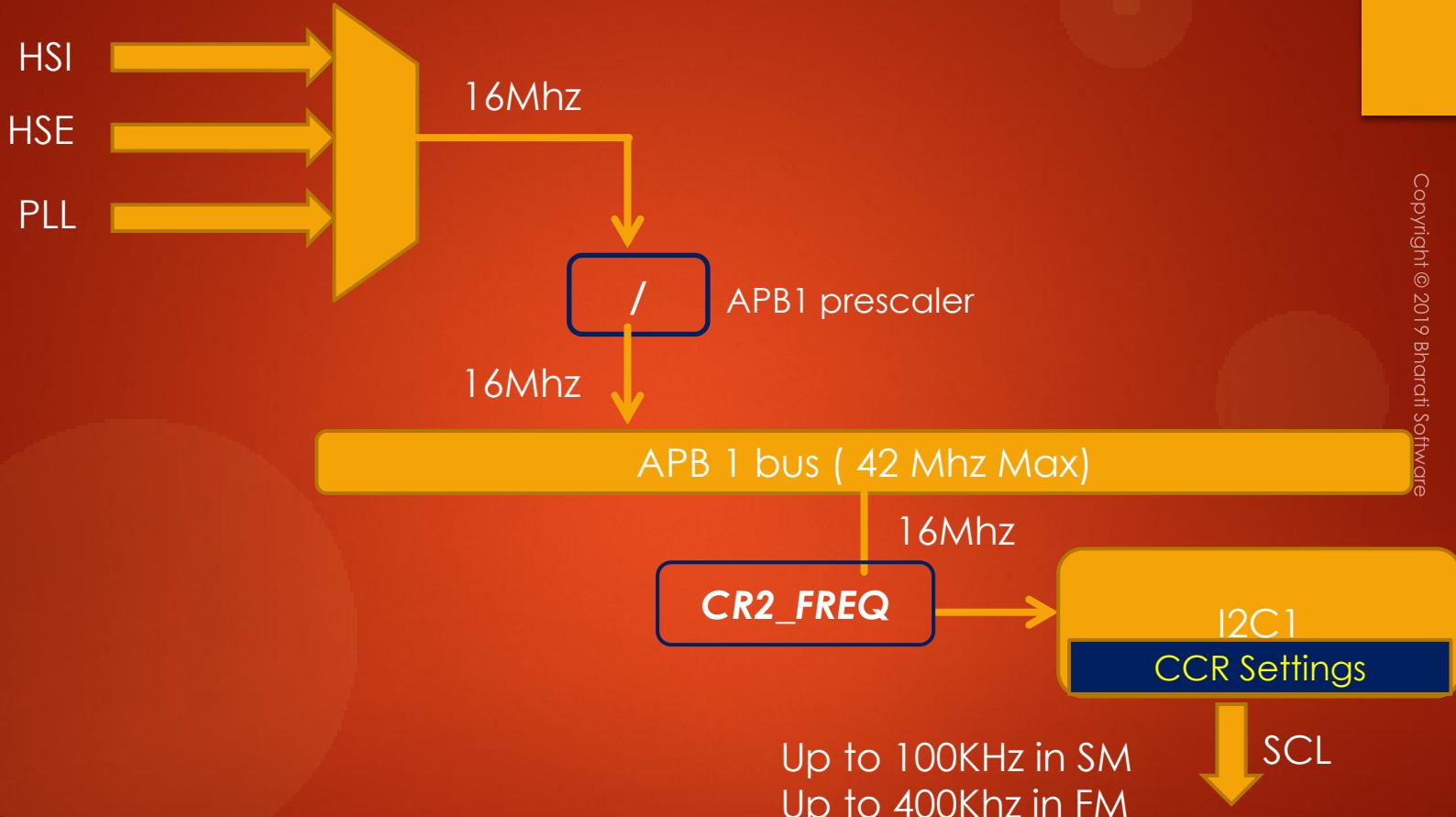
In STM32F4x I2C peripheral, CR2 and CCR registers are used to control the I2C serial clock settings and other I2C timings like setup time and hold time

Copyright ©

## I<sup>2</sup>C Control register 2 (I2C\_CR2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	LAST	DMA EN	ITBUF EN	ITEVT EN	ITERR EN	Res.	Res.					FREQ[5:0]	
			RW	RW	RW	RW	RW			RW	RW	RW	RW	RW	RW

## I<sup>2</sup>C Clock control register (I2C\_CCR)



# Example :

In SM Mode , generate a 100 kHz SCL frequency  
APB1 Clock (PCLK1) = 16MHz

- 1) Configure the mode in CCR register (15<sup>th</sup> bit )
- 2) Program FREQ field of CR2 with the value of PCLK1
- 3) Calculate and Program CCR value in CCR field of CCR register

$$T_{high(scl)} = CCR * T_{PCLK1}$$
$$T_{low(scl)} = CCR * T_{PCLK1}$$

# Example :

In FM Mode , generate a 200 kHz SCL frequency

APB1 Clock (PCLK1) = 16MHz

- 1) Configure the mode in CCR register (15<sup>th</sup> bit )
- 2) Select the duty cycle of Fast mode SCL in CCR register (14<sup>th</sup> bit)
- 3) Program FREQ field of CR2 with the value of PCLK1
- 4) Calculate and Program CCR value in CCR field of CCR register

If DUTY = 0:

$$T_{\text{high}} = CCR * \text{TPCLK1}$$

$$T_{\text{low}} = 2 * CCR * \text{TPCLK1}$$

If DUTY = 1: (to reach 400 kHz)

$$T_{\text{high}} = 9 * CCR * \text{TPCLK1}$$

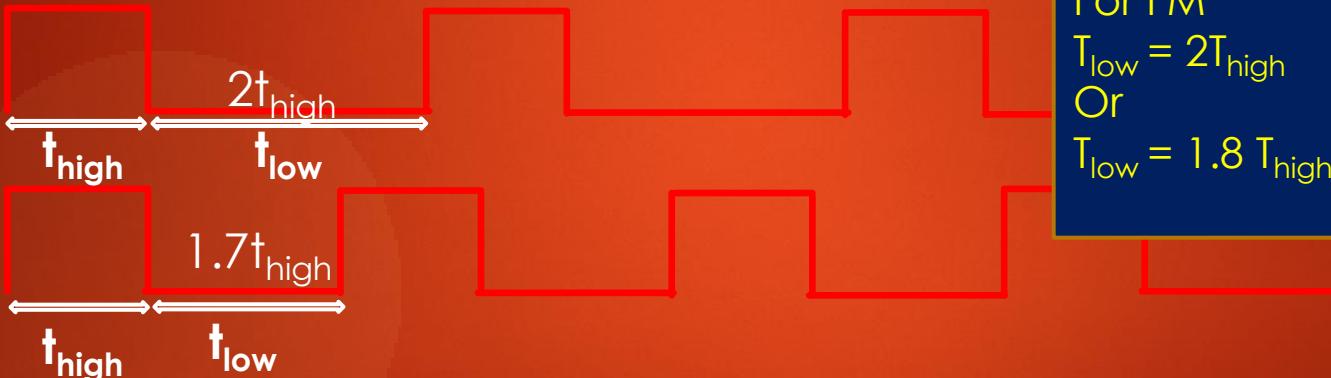
$$T_{\text{low}} = 16 * CCR * \text{TPCLK1}$$

# I2C Duty Cycle

SM



FM



In STM32f4x I2C

For SM

( $T_{low}$  may be equal to  $T_{high}$ )

For FM

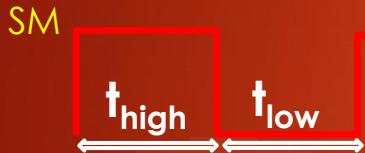
$$T_{low} = 2T_{high}$$

Or

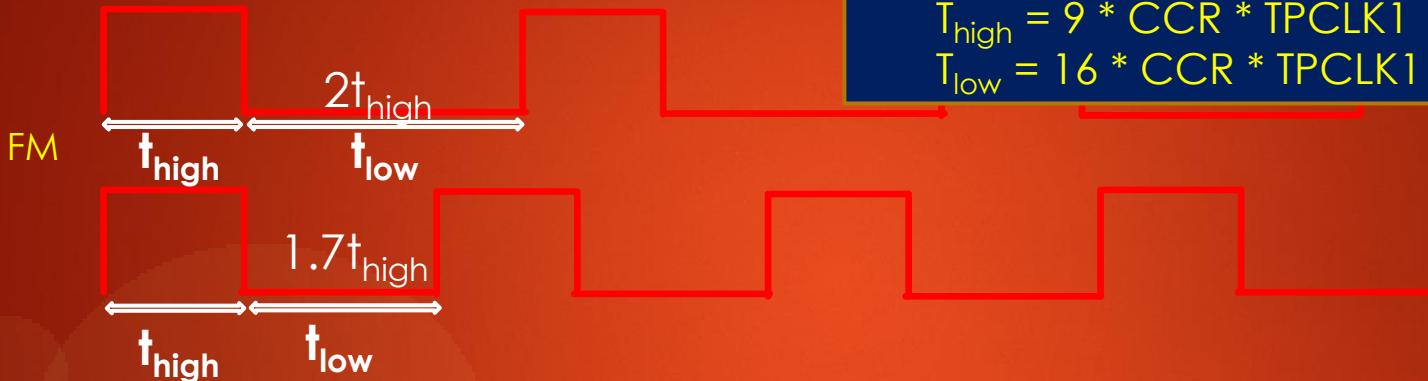
$$T_{low} = 1.8 T_{high}$$

# I2C Duty Cycle

$$T_{high(scl)} = CCR * T_{PCLK1}$$
$$T_{low(scl)} = CCR * T_{PCLK1}$$



# I2C Duty Cycle



If DUTY = 0:

$$T_{high} = CCR * TPCLK1$$

$$T_{low} = 2 * CCR * TPCLK1$$

If DUTY = 1: (to reach 400 kHz)

$$T_{high} = 9 * CCR * TPCLK1$$

$$T_{low} = 16 * CCR * TPCLK1$$



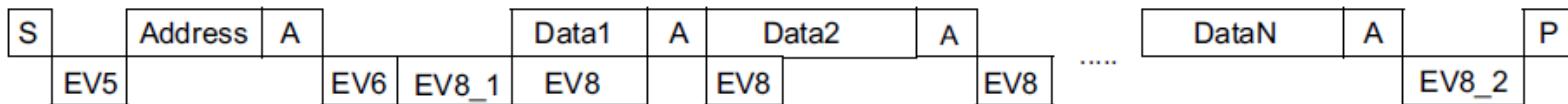
Copyright © 2019 Bharati Software

# Implementation I2C\_MasterSendData API

# Master sending data to slave

Transfer sequence diagram for master transmitter

7-bit master transmitter



Legend: S = Start, SR = Repeated start, P = stop, A = Acknowledge

EVx = Event (with interrupt if ITEVFEN = 1)

EV5: SB=1, cleared by reading SR1 register followed by writing DR register with address.

EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2.

EV8\_1: TxE=1, shift register empty, data register empty, write Data1 in DR.

EV8: TxE=1, shift register not empty, data register empty, cleared by writing DR register.

EV\_2: TxE=1, BTF=1, Program stop request, TxE and BTF are cleared by hardware by the stop condition.

1. The EV5, EV6, EV9, EV8\_1 and EV8\_2 events stretch SCL low until the end of the corresponding software sequence.
2. The EV8 event stretches SCL low if the software sequence is not complete before the end of the next byte transmission.

```
void I2C_MasterSendData(I2C_Handle_t *pI2CHandle, uint8_t *pTxBuffer, uint8_t Len, uint8_t SlaveAddr, uint8_t Sr)
{
    //1. Generate the START condition

    //2. confirm that start generation is completed by checking the SB flag in the SR1
    //   Note: Until SB is cleared SCL will be stretched (pulled to LOW)

    //3. Send the address of the slave with r/nw bit set to w(0) (total 8 bits )

    //4. Confirm that address phase is completed by checking the ADDR flag in teh SR1

    //5. clear the ADDR flag according to its software sequence
    //   Note: Until ADDR is cleared SCL will be stretched (pulled to LOW)

    //6. send the data until Len becomes 0

    //7. when Len becomes zero wait for TXE=1 and BTF=1 before generating the STOP condition
    //   Note: TXE=1 , BTF=1 , means that both SR and DR are empty and next transmission should begin
    //   when BTF=1 SCL will be stretched (pulled to LOW)

    //8. Generate STOP condition and master need not to wait for the completion of stop condition.
    //   Note: generating STOP, automatically clears the BTF
}
```

# Exercise :

## I2C Master(STM) and I2C Slave(Arduino) communication .

When button on the master is pressed , master should send data to the Arduino slave connected. The data received by the Arduino will be displayed on the Arduino serial port.

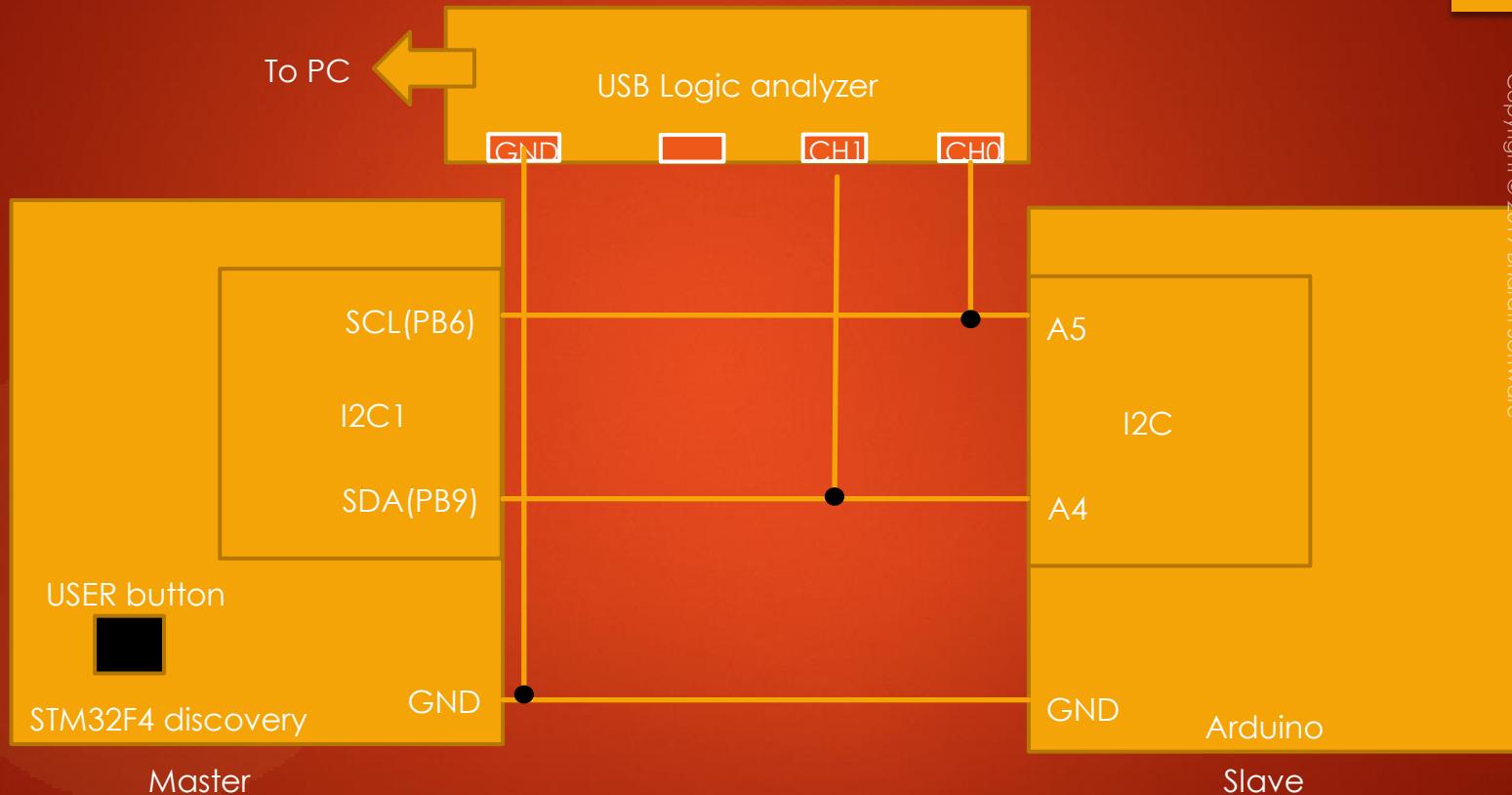
- 1 . Use I2C SCL = 100KHz(Standard mode )
2. Use internal pull resistors for SDA and SCL lines

# Things you need

- 1 .Arduino board
2. ST board
3. Some jumper wires
4. Bread board
5. 2 Pull up resistors of value  $4.4K \Omega$ ( only if your pin doesn't support internal pull up resistors )



# STEP-1 Connect Arduino and ST board SPI pins as shown



# STEP-2

Power your Arduino board and download I2C Slave sketch to Arduino

Sketch name : 001I2CSlaveRxString.ino

# Find out the GPIO pins over which I2C1 can communicate

# I2C pull up resistance , Rise time and Bus capacitance discussion

# Pull-up Resistor( $R_p$ ) Calculation

$R_p$  (min) is a function of VCC, VOL (max), and IOL:

$$R_p(\text{min}) = \frac{(V_{\text{CC}} - V_{\text{OL}}(\text{max}))}{I_{\text{OL}}}$$

$V_{\text{OL}}$  = LOW-level output voltage

$I_{\text{OL}}$  = LOW-level output current

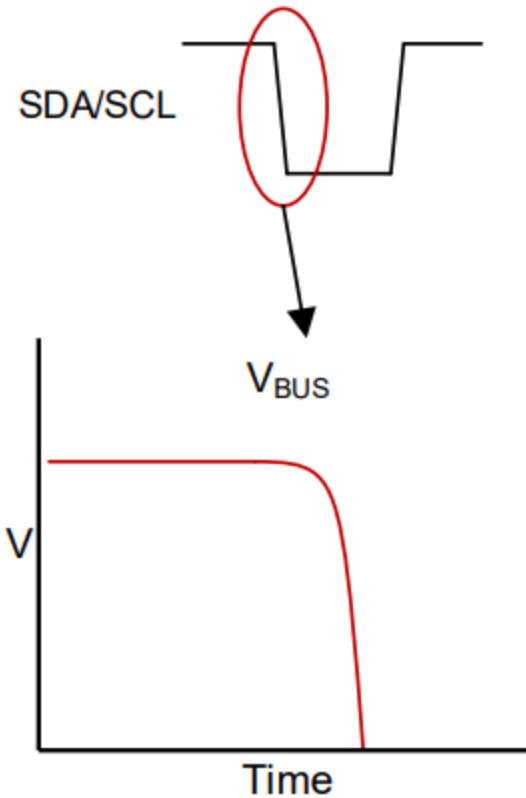
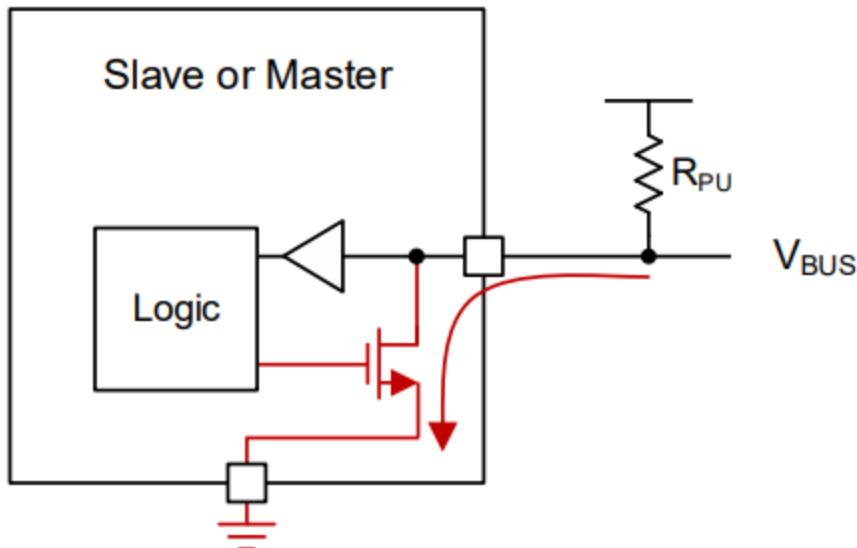
$t_r$  = rise time of both SDA and SCL signals

$C_b$  = capacitive load for each bus line

The maximum pullup resistance is a function of the maximum rise time ( $t_r$ ):

$$R_p(\text{max}) = \frac{t_r}{(0.8473 \times C_b)}$$

Image taken from :  
<http://www.ti.com/lit/an/slva704/slva704.pdf>



**Pulling the Bus Low With An Open-Drain Interface**

# Rise ( $t_r$ ) and Fall ( $t_f$ ) Times



$t_r$  is defined as the amount of time taken by the rising edge to reach 70% amplitude from 30% amplitude for either SDA and SCL

I2C spec cares about  $t_r$  value and you have to respect it while calculating the pull up resistor value.

Higher value of pull up resistors(weak pull-ups) increases  $t_r$  value. ( not acceptable if  $t_r$  crosses max. limit mentioned in the spec )

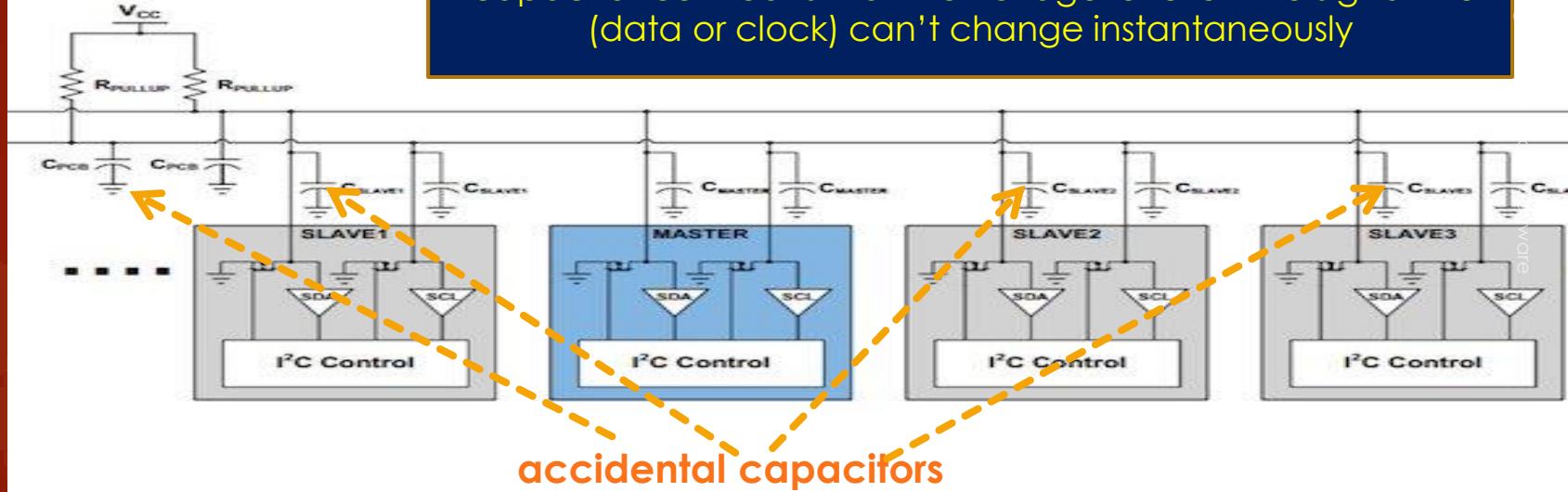
Lower value of pull up resistors ( strong pull-ups) decreases  $t_r$  value ( good) but they also lead higher current consumption (bad)

Using very high value of pull up resistors may cause issues like this where the pin may not able to cross the  $V_{IH}$  limit (Input Level High Voltage)



# I2C Bus Capacitance( $C_b$ )

capacitance means that the voltage level on the signal line (data or clock) can't change instantaneously



# I2C Bus Capacitance( $C_b$ )

- ✓ Bus capacitance is a collection of individual pin capacitance wrt gnd , capacitance between the sda and scl , parasitic capacitance, capacitance added by the devices hanging on the bus , bus length (wire) , dielectric material etc.
- ✓ Bus capacitance limits how long your i2c wiring can be and how many devices you can connect on the bus.
- ✓ For maximum allowed bus capacitance check the spec.

# Exercise

For Fast-mode I2C communication with the following parameters, calculate the pullup resistor value.  $C_b = 150 \text{ pF}$ ,  $VCC = 3.3 \text{ V}$

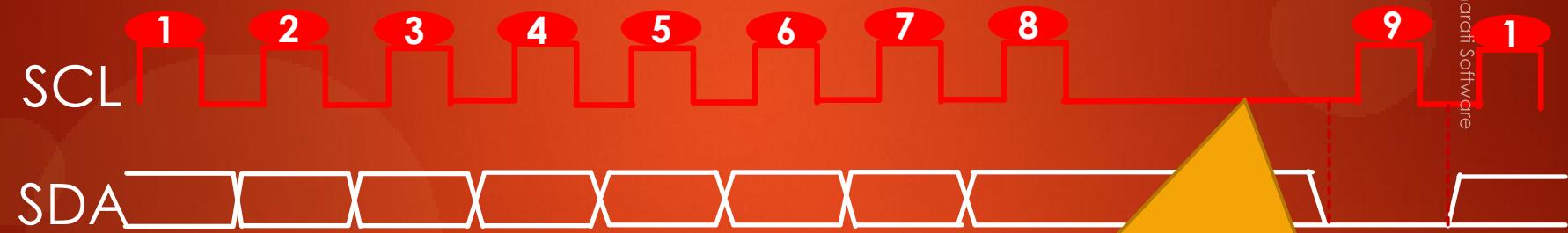
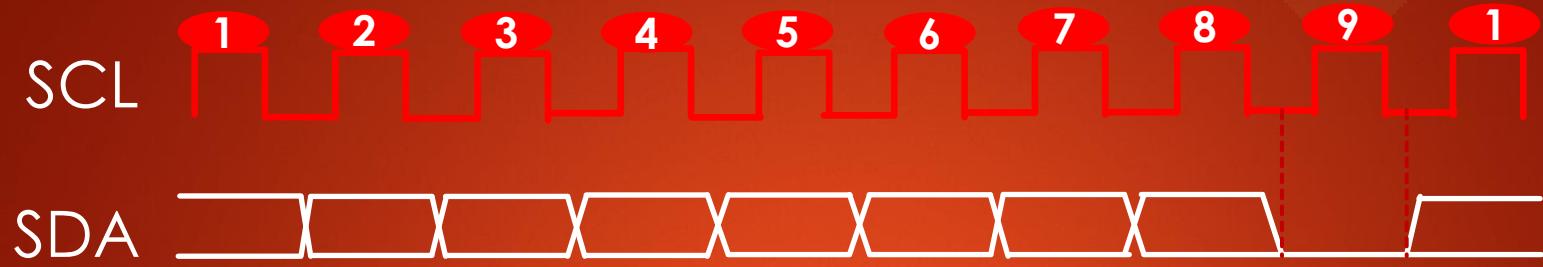
# TRISE calculation

Copyright © 2019 Bharati Software

# Clock Stretching

# Clock Stretching

- ▶ Clock stretching pauses a transaction by holding the SCL line LOW.
- ▶ The transaction cannot continue until the line is released HIGH again. Clock stretching is optional and in fact, most slave devices do not include an SCL driver so they are unable to stretch the clock.
- ▶ Slaves can hold the SCL line LOW after reception and acknowledgment of a byte to force the master into a wait state until the slave is ready for the next byte transfer in a type of handshake procedure
- ▶ If a slave cannot receive or transmit another complete byte of data until it has performed some other function, for example servicing an internal interrupt, it can hold the clock line SCL LOW to force the master into a wait state. Data transfer then continues when the slave is ready for another byte of data and releases clock line SCL.



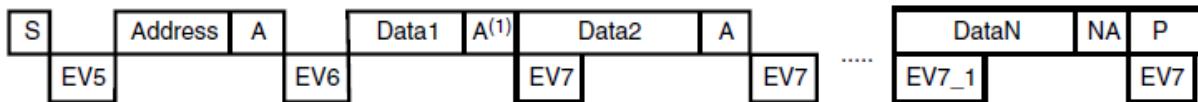
The slave is not ready for more data, so it buys time by holding the clock low.  
The master will wait for the clock line to be released before proceeding to the next frame

# Implementation I2C\_MasterReceiveData API

# Master Receiving data from slave

## Transfer sequence diagram for master receiver

7-bit master receiver



**Legend:** S= Start, S<sub>r</sub> = Repeated Start, P= Stop, A= Acknowledge, NA= Non-acknowledge, EVx= Event (with interrupt if ITEVFEN=1)

**EV5:** SB=1, cleared by reading SR1 register followed by writing DR register.

**EV6:** ADDR=1, cleared by reading SR1 register followed by reading SR2. In 10-bit master receiver mode, this sequence should be followed by writing CR2 with START = 1.

In case of the reception of 1 byte, the Acknowledge disable must be performed during EV6 event, i.e. before clearing ADDR flag.

**EV7:** RxNE=1 cleared by reading DR register.

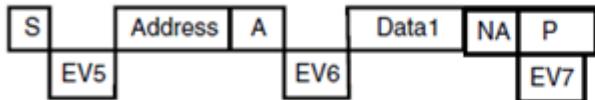
**EV7\_1:** RxNE=1 cleared by reading DR register, program ACK=0 and STOP request

1. If a single byte is received, it is NA.
2. The EV5, EV6 and EV9 events stretch SCL low until the end of the corresponding software sequence.
3. The EV7 event stretches SCL low if the software sequence is not completed before the end of the next byte reception.
4. The EV7\_1 software sequence must be completed before the ACK pulse of the current byte transfer.

# Master Receiving 1 byte from slave

## Transfer sequence diagram for master receiver

7-bit master receiver



Legend: S= Start, S<sub>r</sub>= Repeated Start, P= Stop, A= Acknowledge, NA= Non-acknowledge, EVx= Event (with interrupt if ITEVFEN=1)

EV5: SB=1, cleared by reading SR1 register followed by writing DR register.

EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2. In 10-bit master receiver mode, this sequence should be followed by writing CR2 with START = 1.

In case of the reception of 1 byte, the Acknowledge disable must be performed during EV6 event, i.e. before clearing ADDR flag.

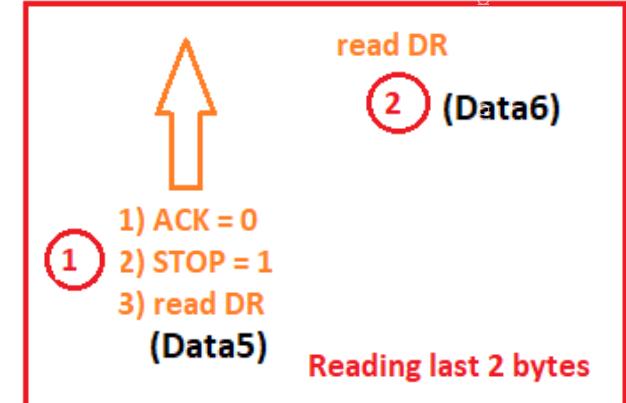
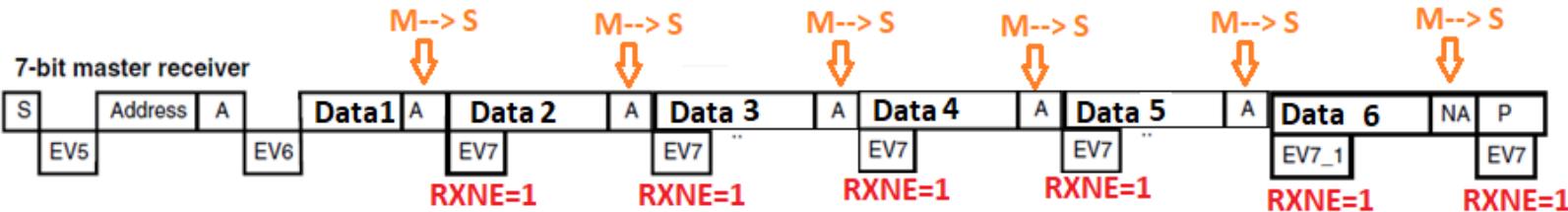
EV7: RxNE=1 cleared by reading DR register.

EV7\_1: RxNE=1 cleared by reading DR register, program ACK=0 and STOP request

1. If a single byte is received, it is NA.
2. The EV5, EV6 and EV9 events stretch SCL low until the end of the corresponding software sequence.
3. The EV7 event stretches SCL low if the software sequence is not completed before the end of the next byte reception.
4. The EV7\_1 software sequence must be completed before the ACK pulse of the current byte transfer.

# Master Receiving more than 1 byte

Transfer sequence diagram for master receiver when Len > 1



# Exercise :

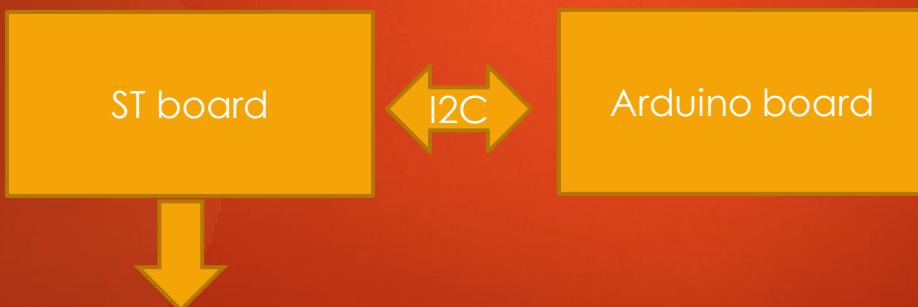
## I2C Master(STM) and I2C Slave(Arduino) communication .

When button on the master is pressed , master should read and display data from Arduino Slave connected. First master has to get the length of the data from the slave to read subsequent data from the slave.

- 1 . Use I2C SCL = 100KHz(Standard mode )
2. Use internal pull resistors for SDA and SCL lines

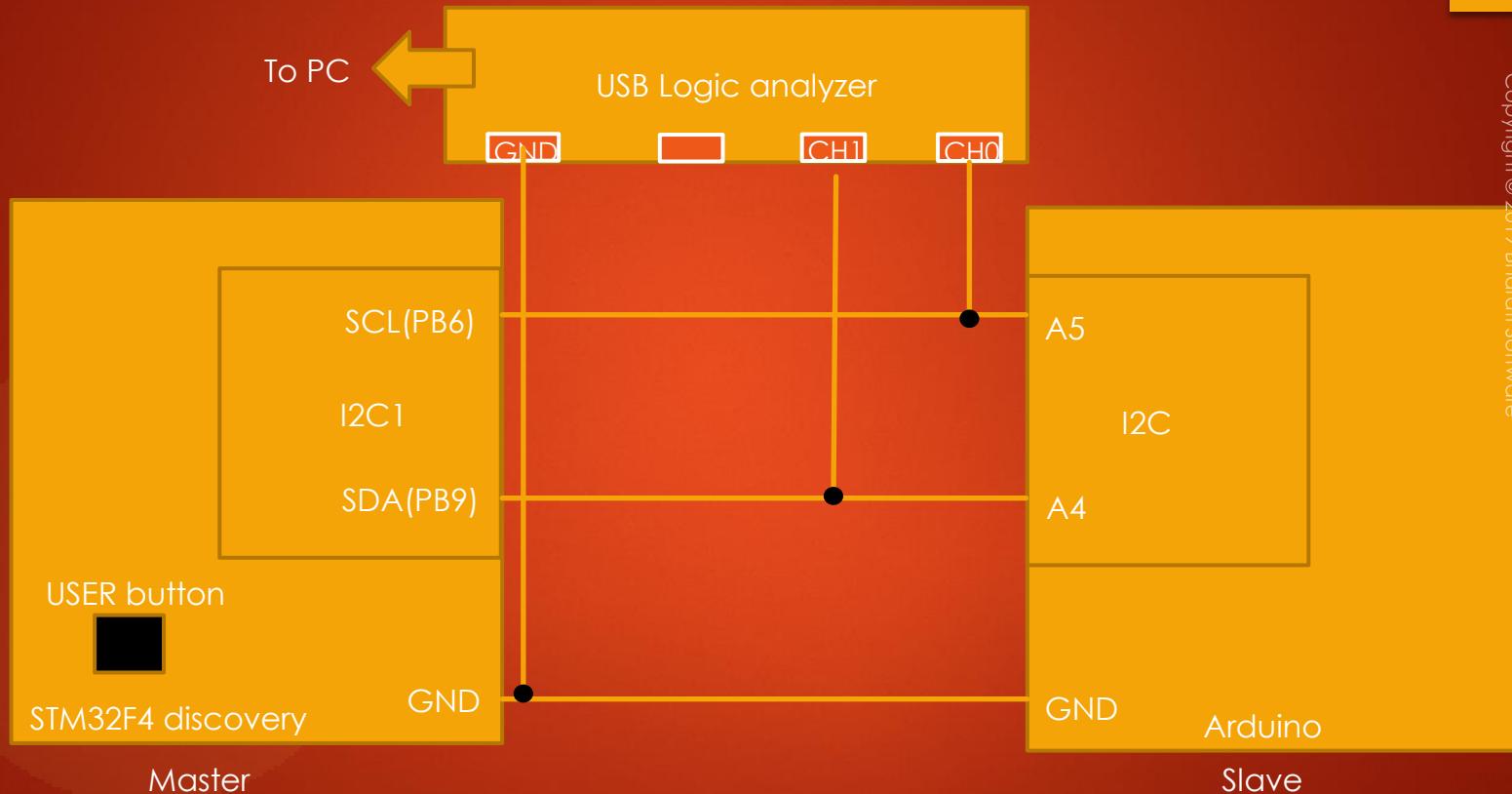
# Things you need

- 1 .Arduino board
2. ST board
3. Some jumper wires
4. Bread board
5. 2 Pull up resistors of value 4.7K  $\Omega$ ( only if your pin doesn't support internal pull up resistors )



Print received data using semihosting

# STEP-1 Connect Arduino and ST board SPI pins as shown



# STEP-2

Power your Arduino board and download I2C Slave sketch to Arduino

Sketch name : 002I2CSlaveTxString.ino

## Procedure to read the data from Arduino Slave

1

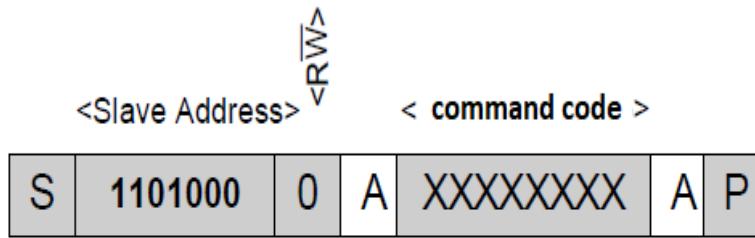
Master sends command code 0x51 to read the length(1 byte) of the data from the slave

2

Master sends command code 0x52 to read the complete data from the slave

# I2C transactions to read the 1 byte length information from slave

## Data Write— Master sending command to slave



S - Start

A - Acknowledge (ACK)

P - Stop



Master to slave



Slave to master

## Data Read— Master reading response from the slave

<Slave Address><sup>V</sup>  
<sup>RW</sup>  
^

<Data >

S	1101000	1	A	XXXXXXXX	$\bar{A}$	P
---	---------	---	---	----------	-----------	---

S - Start

A - Acknowledge (ACK)

P - Stop

$\bar{A}$  - Not Acknowledge (NACK)



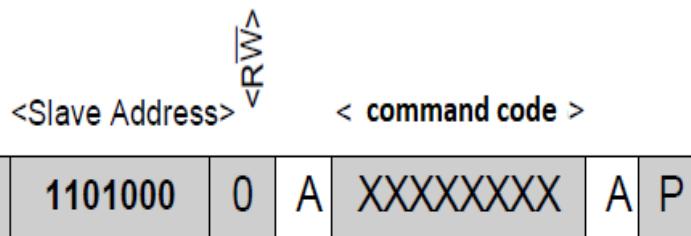
Master to slave



Slave to master

# I2C transactions to read “length” bytes of data from the slave

## Data Write—Master sending command to slave



S - Start

A - Acknowledge (ACK)

P - Stop

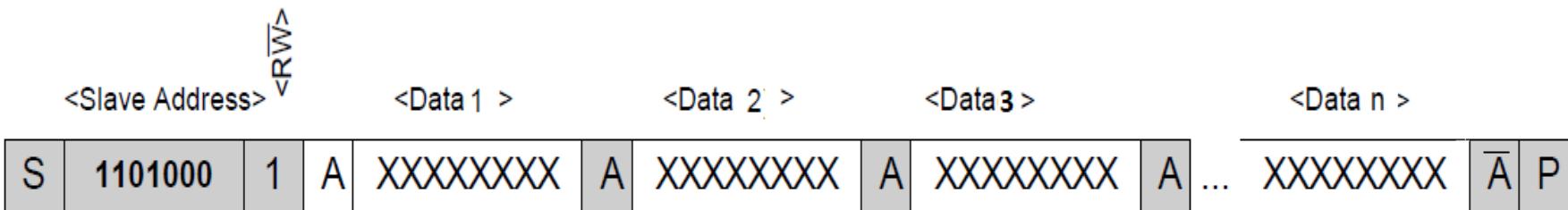


Master to slave



Slave to master

## Data Read— Master reading response from the slave



S - Start

A - Acknowledge (ACK)

P - Stop

Ā - Not Acknowledge (NACK)



Master to slave



Slave to master

START

Reset both the boards

Wait for button press on the master

Master first sends command code 0x51  
to slave (to read length)

Master reads “length” from slave  
( 1 byte)

Master reads “length” number of bytes  
from the slave

Master display the data received using  
printf (semihosting)

END

# I2C Functional block and Peripheral Clock

$f_{\text{pclk}}$  (Peripheral Clock Frequency ) must be at least 2MHz to achieve standard mode I2c frequencies that are up to 100khz

$f_{\text{pclk}}$  must be at least 4Mhz to achieve FM mode i2c frequencies. that is above 100khz but below 400Khz

$f_{\text{pclk}}$  must be a multiple of 10Mhz to reach the 400khz max i2c fm mode clock frequency

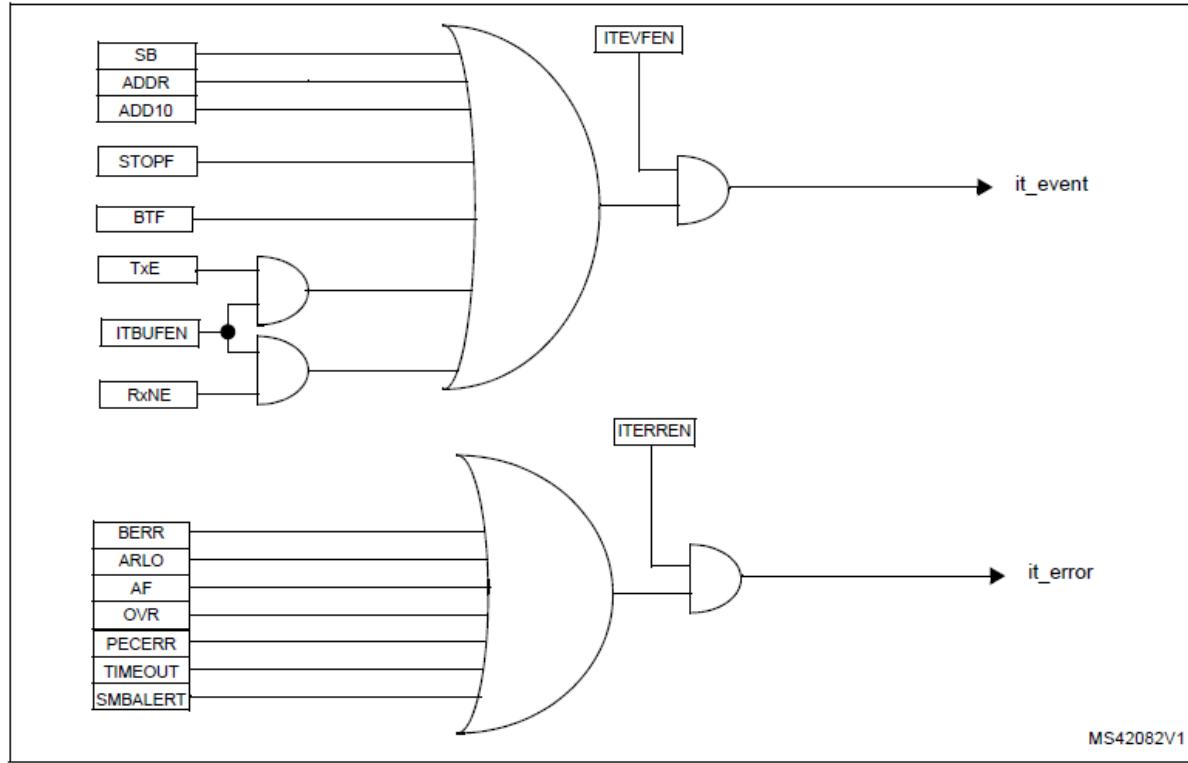
# I2C Interrupts

# I2C IRQs and Interrupt mapping

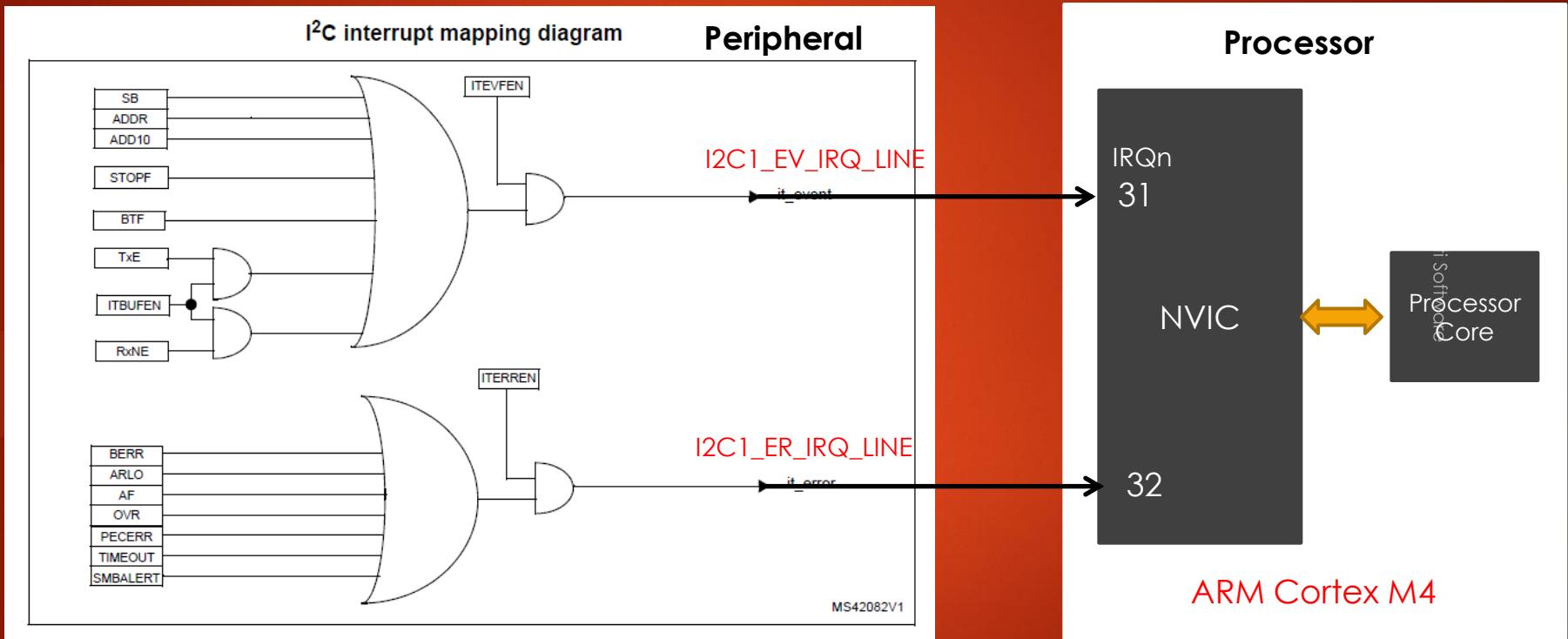
## I<sup>2</sup>C Interrupt requests

Interrupt event	Event flag	Enable control bit
Start bit sent (Master)	SB	ITEVFEN
Address sent (Master) or Address matched (Slave)	ADDR	
10-bit header sent (Master)	ADD10	
Stop received (Slave)	STOPF	
Data byte transfer finished	BTF	
Receive buffer not empty	RxNE	ITEVFEN and ITBUFEN
Transmit buffer empty	TxE	
Bus error	BERR	ITERREN
Arbitration loss (Master)	ARLO	
Acknowledge failure	AF	
Overrun/Underrun	OVR	
PEC error	PECERR	
Timeout/Tlow error	TIMEOUT	
SMBus Alert	SMBALERT	

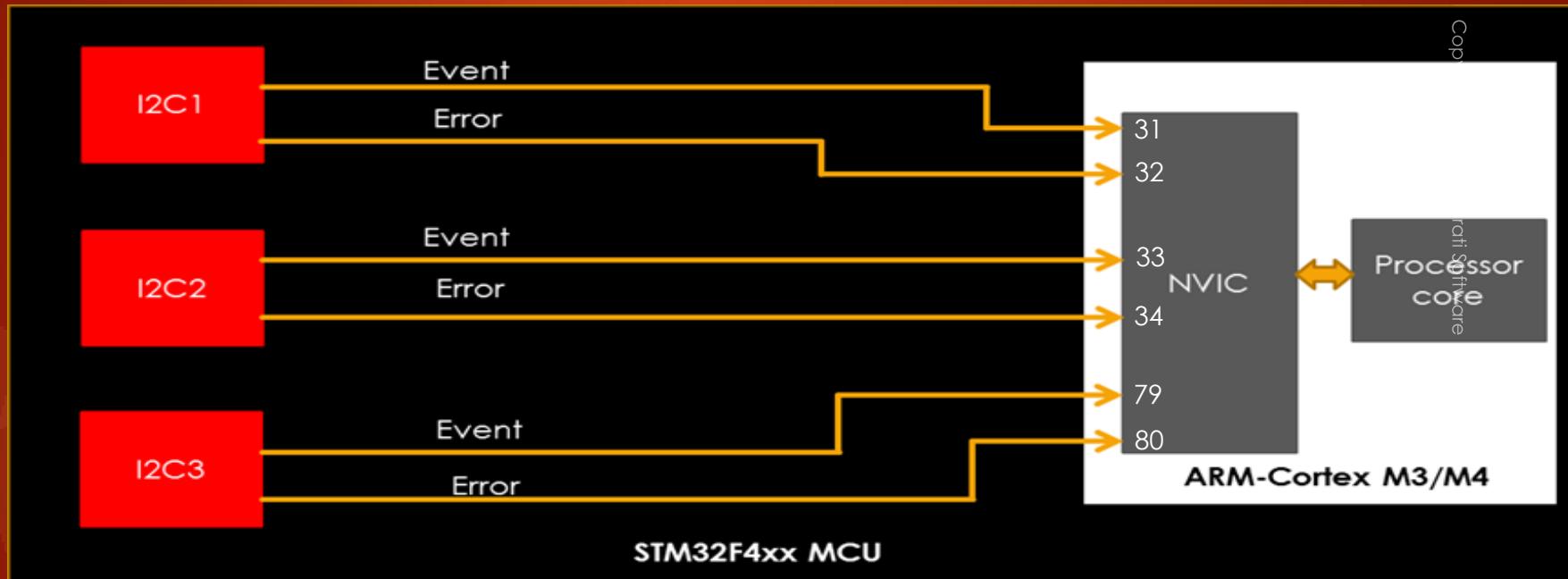
## I<sup>2</sup>C interrupt mapping diagram



# I<sup>2</sup>C1 interrupts



# I2C Interrupting the Processor



*IRQ numbers are specific to MCUs. Please check the vector table of your RM*

## Bus Error

This error happens when the interface detects an SDA rising or falling edge while SCL is high, occurring in a non-valid position during a byte transfer

## Arbitration Loss Error

This error can happen when the interface loses the arbitration of the bus to another master

# ACK Failure Error

This error happens when no ACK is returned for the byte sent

# Overrun Error

Happens during reception, when a new byte is received and the data register has not been read yet and the New received byte is lost.

# Under-run Error

Happens when In transmission when a new byte should be sent and the data register has not been written yet and the same byte is sent twice

# PEC Error

Happens when there is CRC mismatch, if you have enabled the CRC feature

# Time-Out Error

Happens when master or slave stretches the clock , by holding it low more than recommended amount of time.

# BTF flag in TX and preventing underrun

During Txing of a data byte, if TXE=1, then that means data register is empty.

And if the firmware has not written any byte to data register before shift register becomes empty(previous byte transmission), then *BTF flag will be set and clock will be stretched to prevent the under run.*

# BTF flag in RX and preventing overrun

If RXNE =1, Then it means new data is waiting in the data register, and if the firmware has not read the data byte yet before Shift register is filled with another new data, then also the BTF flag will be set and clock will be stretched to prevent the overrun.

# I2C sending and receiving data in interrupt mode

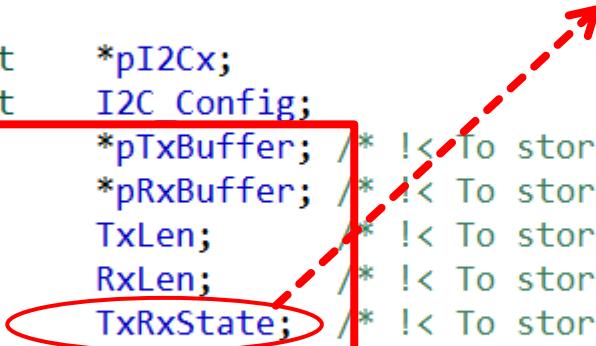
# I2C\_MasterSendDataIT API

# I2C\_MasterReceiveDataIT API

# Modifying handle structure to store place holder variables

```
/*
 *Handle structure for I2Cx peripheral
 */
typedef struct
{
    I2C_RegDef_t      *pI2Cx;
    I2C_Config_t      I2C_Config;
    uint8_t            *pTxBuffer; /* !< To store the app. Tx buffer address > */
    uint8_t            *pRxBuffer; /* !< To store the app. Rx buffer address > */
    uint32_t           TxLen;     /* !< To store Tx len > */
    uint32_t           RxLen;     /* !< To store Rx len > */
    uint8_t            TxRxState; /* !< To store Communication state > */
    uint8_t            DevAddr;   /* !< To store slave/device address > */
    uint32_t           RxSize;    /* !< To store Rx size > */
    uint8_t            Sr;        /* !< To store repeated start value > */
}I2C_Handle_t;
```

```
/*
 * I2C application states
 */
#define I2C_READY          0
#define I2C_BUSY_IN_RX     1
#define I2C_BUSY_IN_TX     2
```



# Implementing **I2C\_MasterSendDataIT** API

# Implementing **I2C\_MasterReceiveDataIT API**

# Adding I2C IRQ number macros

# Implementing I2C\_IRQInterruptConfig(); I2C\_IRQPriorityConfig();

# I2C ISR handling

ISR1

Interrupt handling for  
interrupts generated by I2C  
events

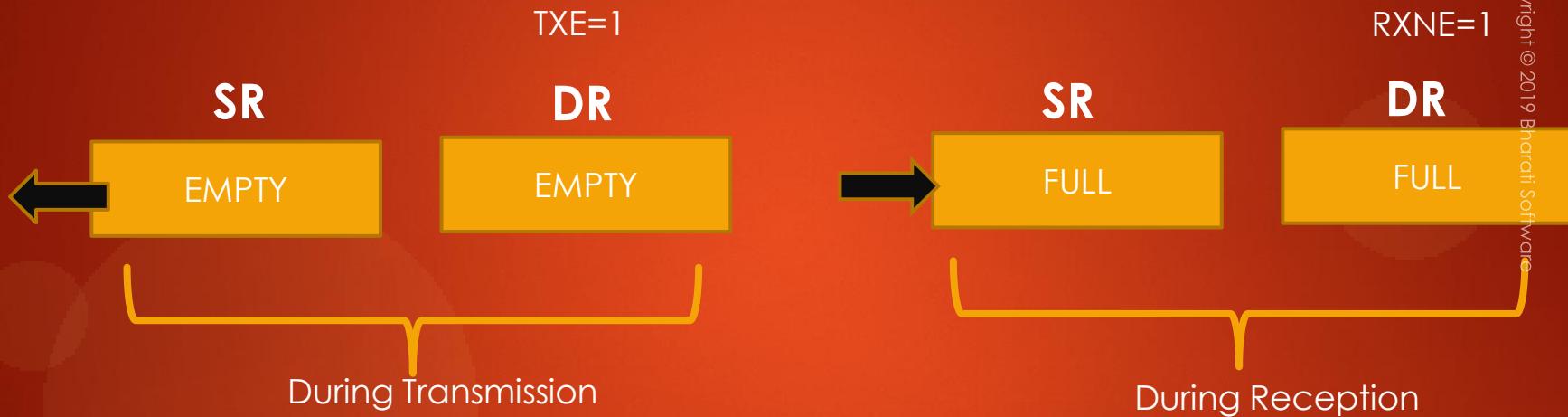
I2C\_EV\_IRQHandlering

ISR2

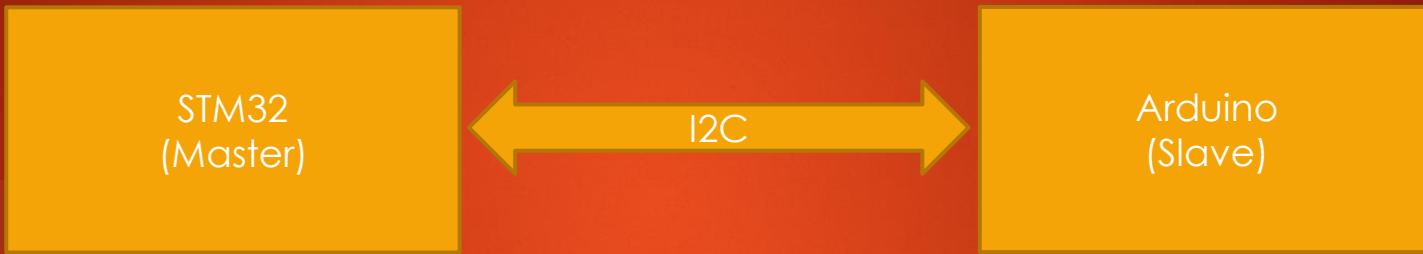
Interrupt handling for  
interrupts generated by I2C  
errors

I2C\_ER\_IRQHandlering

# When BTF flag is set



# I2C Slave Programming





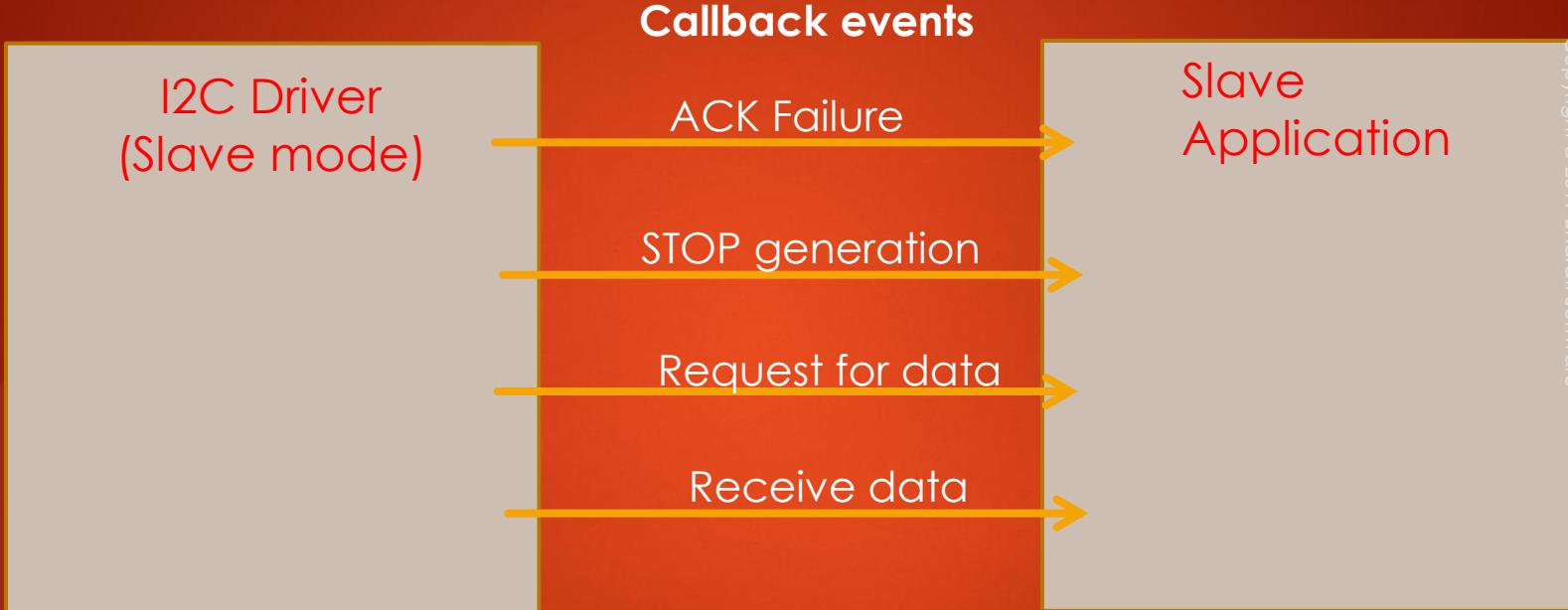


START  
STOP  
Read  
Write

I2C Slave

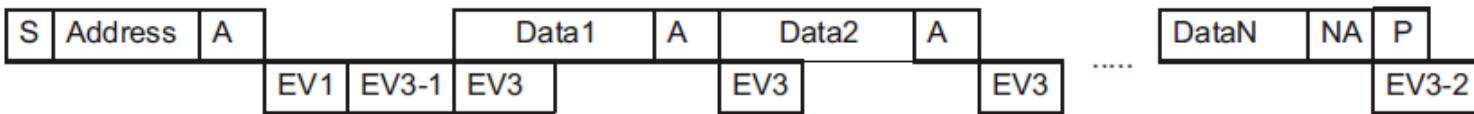
Request for data

Receive data



## Transfer sequence diagram for slave transmitter

### 7-bit slave transmitter



**Legend:** S= Start,  $S_r$ = Repeated Start, P= Stop, A= Acknowledge, NA= Non-acknowledge, EVx= Event (with interrupt if ITEVFEN=1)

EV1: ADDR=1, cleared by reading SR1 followed by reading SR2

EV3-1: TxE=1, shift register empty, data register empty, write Data1 in DR.

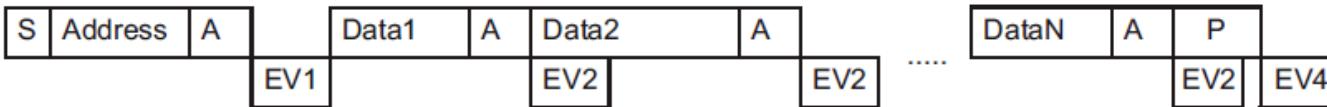
EV3: TxE=1, shift register not empty, data register empty, cleared by writing DR

EV3-2: AF=1; AF is cleared by writing '0' in AF bit of SR1 register.

1. The EV1 and EV3\_1 events stretch SCL low until the end of the corresponding software sequence.
2. The EV3 event stretches SCL low if the software sequence is not completed before the end of the next byte transmission.

## Transfer sequence diagram for slave receiver

7-bit slave receiver



**Legend:** S= Start, Sr = Repeated Start, P= Stop, A= Acknowledge,  
EVx= Event (with interrupt if ITEVFEN=1)

EV1: ADDR=1, cleared by reading SR1 followed by reading SR2

EV2: RxNE=1 cleared by reading DR register.

EV4: STOPF=1, cleared by reading SR1 register followed by writing to the CR1 register

1. The EV1 event stretches SCL low until the end of the corresponding software sequence.
2. The EV2 event stretches SCL low if the software sequence is not completed before the end of the next byte reception.
3. After checking the SR1 register content, the user should perform the complete clearing sequence for each flag found set.

Thus, for ADDR and STOPF flags, the following sequence is required inside the I2C interrupt routine:

READ SR1

if (ADDR == 1) {READ SR1; READ SR2}

if (STOPF == 1) {READ SR1; WRITE CR1}

The purpose is to make sure that both ADDR and STOPF flags are cleared if both are found set.

# Exercise :

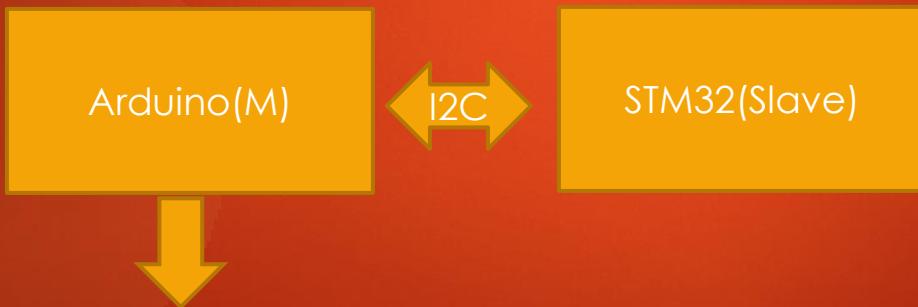
## I2C Master(Arduino) and I2C Slave(STM32) communication .

Master should read and display data from STM32 Slave connected. First master has to get the length of the data from the slave to read subsequent data from the slave.

- 1 . Use I2C SCL = 100KHz(Standard mode )
2. Use internal pull resistors for SDA and SCL lines

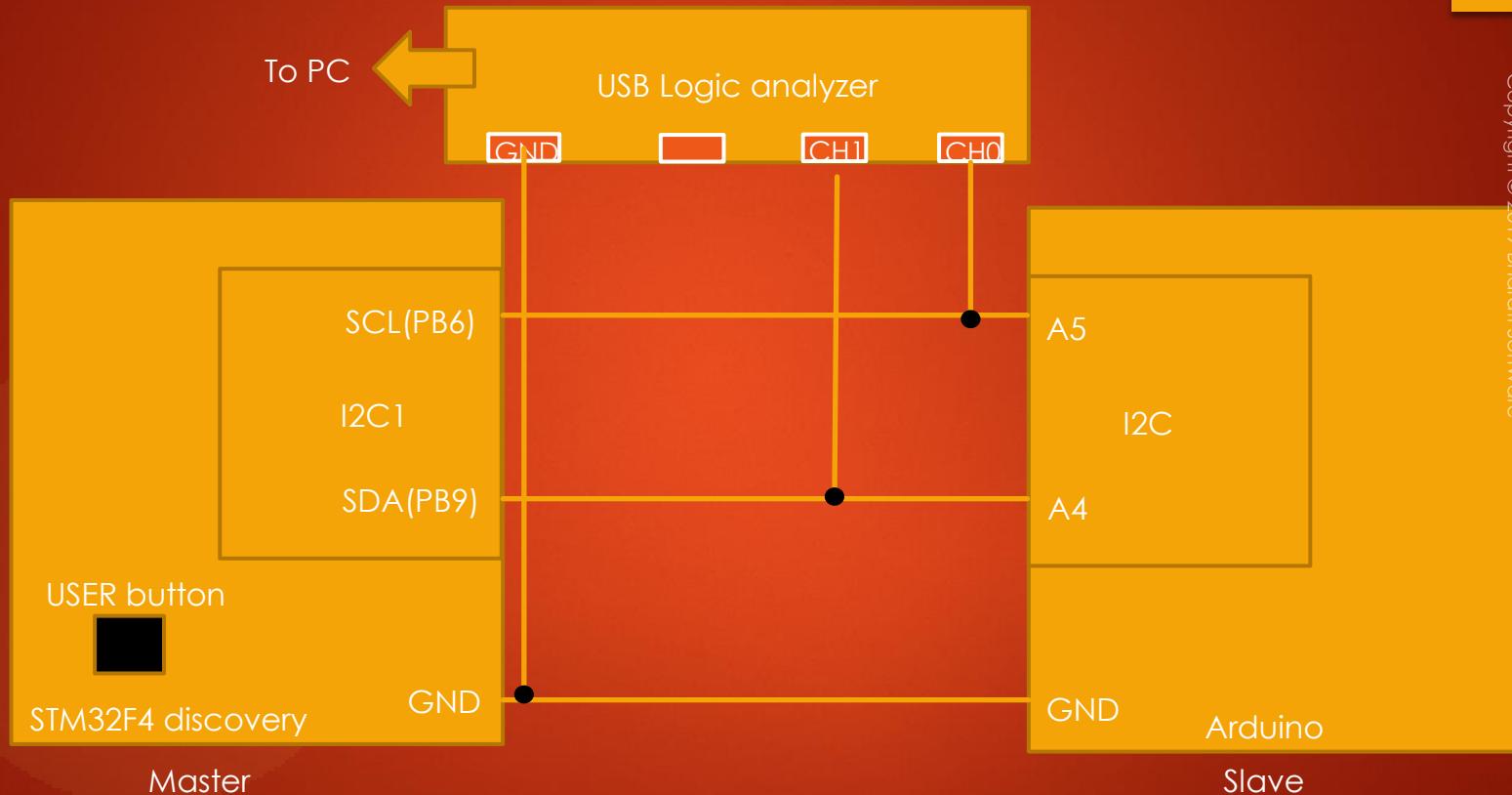
# Things you need

- 1 .Arduino board
2. ST board
3. Some jumper wires
4. Bread board
5. 2 Pull up resistors of value 4.7K  $\Omega$ ( only if your pin doesn't support internal pull up resistors )



Print received data on to Arduino serial port

# STEP-1 Connect Arduino and ST board SPI pins as shown



# STEP-2

Power your Arduino board and download I2C Slave sketch to Arduino

Sketch name: 003I2CMasterRxString.ino

## Procedure to read the data from STM32 Slave

1

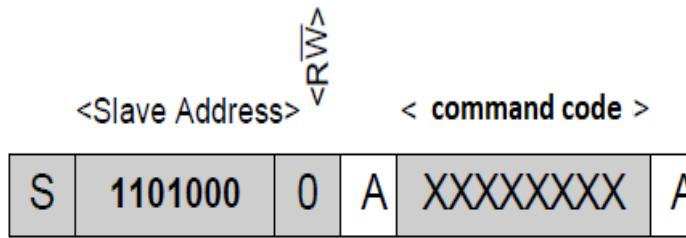
Master sends command code 0x51 to read the length(1 byte) of the data from the slave

2

Master sends command code 0x52 to read the complete data from the slave

# I2C transactions to read the 1 byte length information from slave

## Data Write— Master sending command to slave



Master : Arduino

S - Start

A - Acknowledge (ACK)

P - Stop

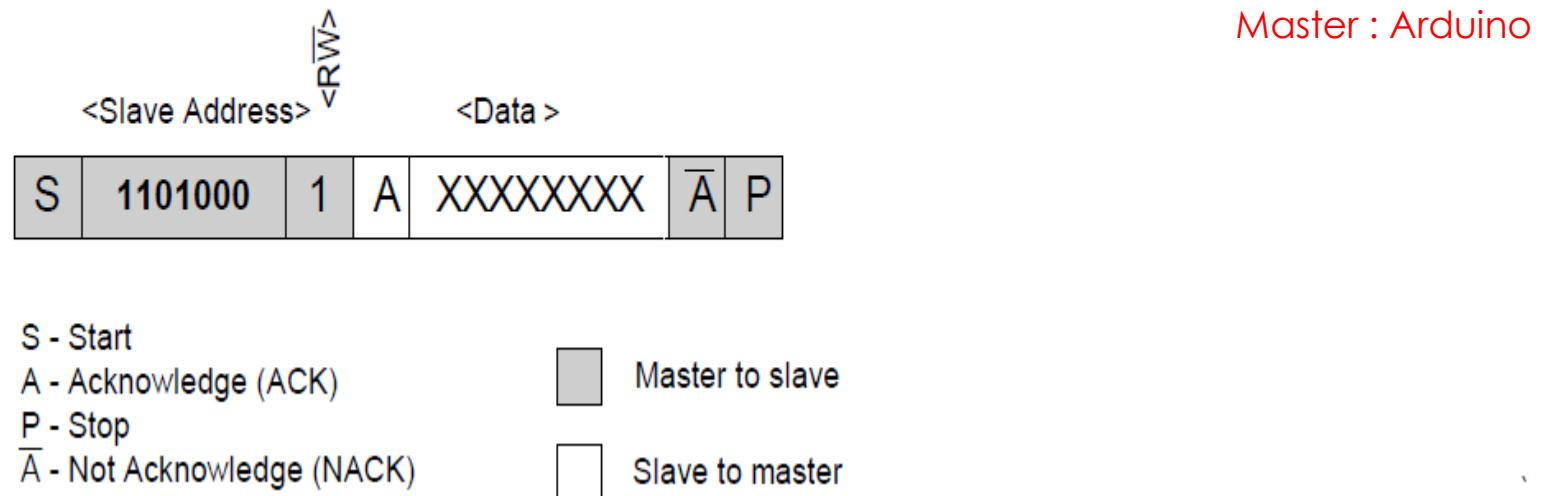


Master to slave



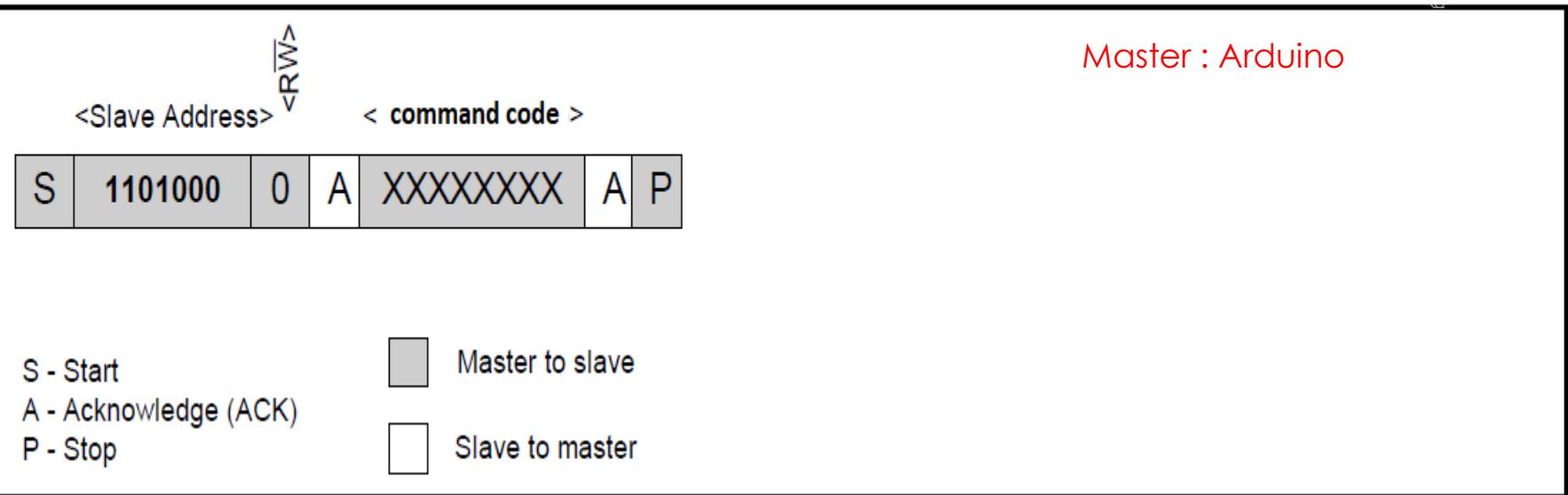
Slave to master

## Data Read— Master reading response from the slave



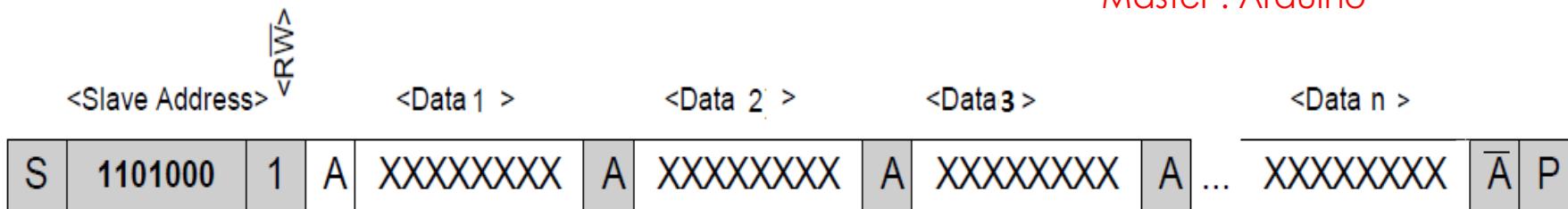
# I2C transactions to read “length” bytes of data from the slave

# Data Write—Master sending command to slave



## Data Read— Master reading response from the slave

Master : Arduino



S - Start

A - Acknowledge (ACK)

P - Stop

A - Not Acknowledge (NACK)



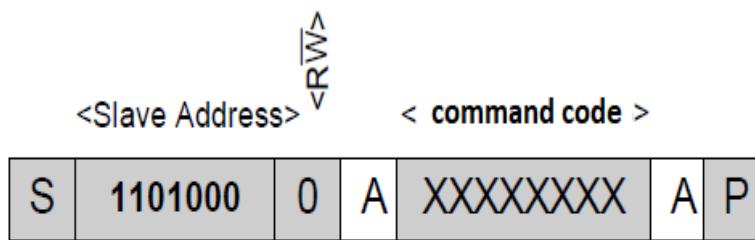
Master to slave



Slave to master

I2C transactions to read the 4 bytes of length information from the slave

## Data Write— Master sending command to slave



S - Start

A - Acknowledge (ACK)

P - Stop



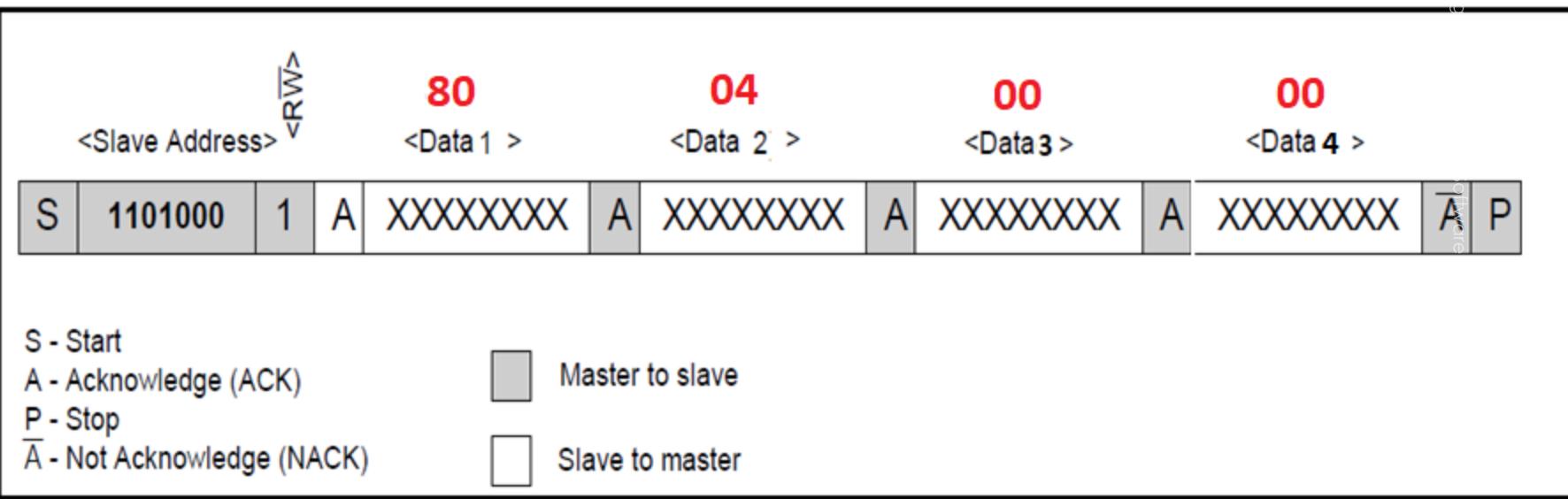
Master to slave



Slave to master

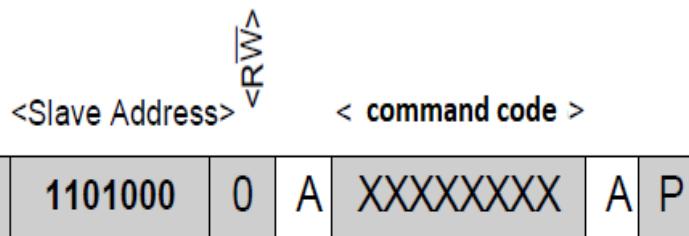
## Data Read— Master reading response from the slave

0x00000480



# I2C transactions to read “length” bytes of data from the slave

## Data Write—Master sending command to slave



S - Start

A - Acknowledge (ACK)

P - Stop



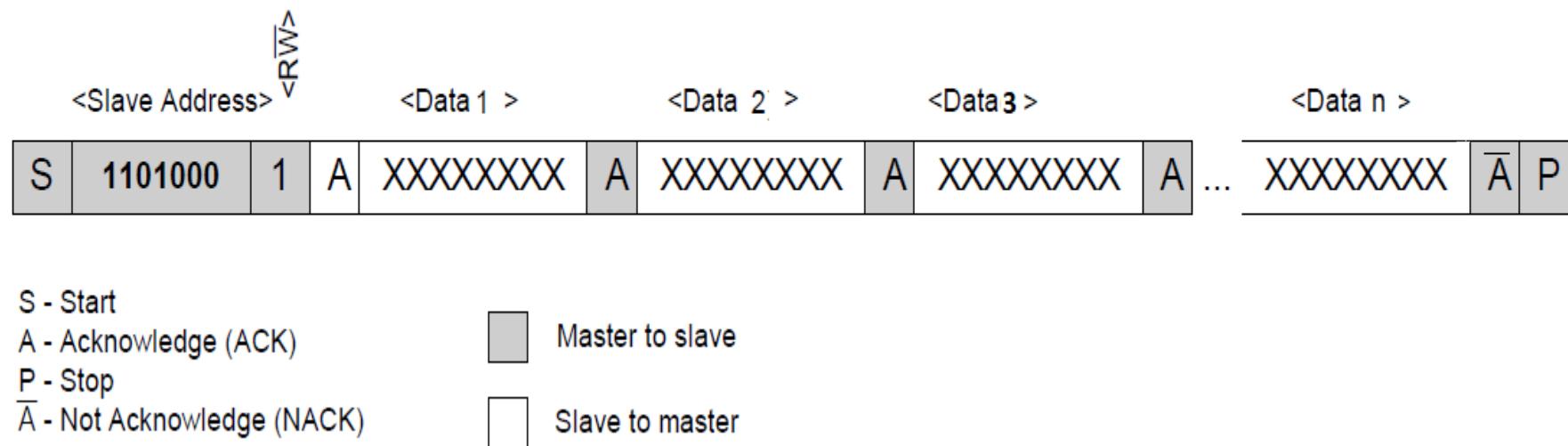
Master to slave



Slave to master

If length is  $\leq 32$  then only one “READ” Transaction

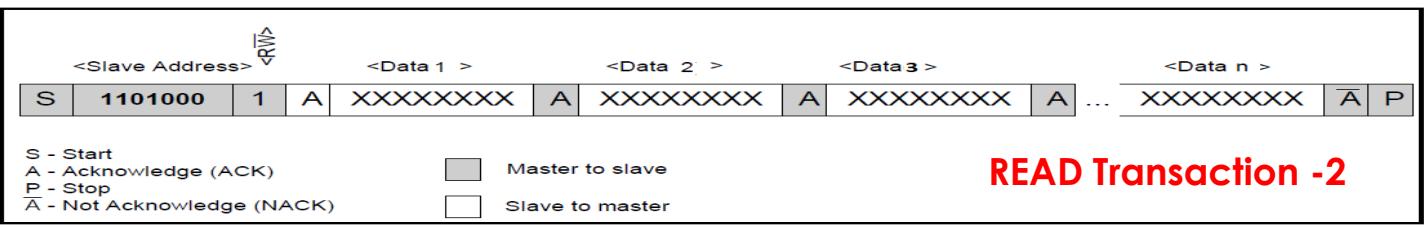
## Data Read— Master reading response from the slave



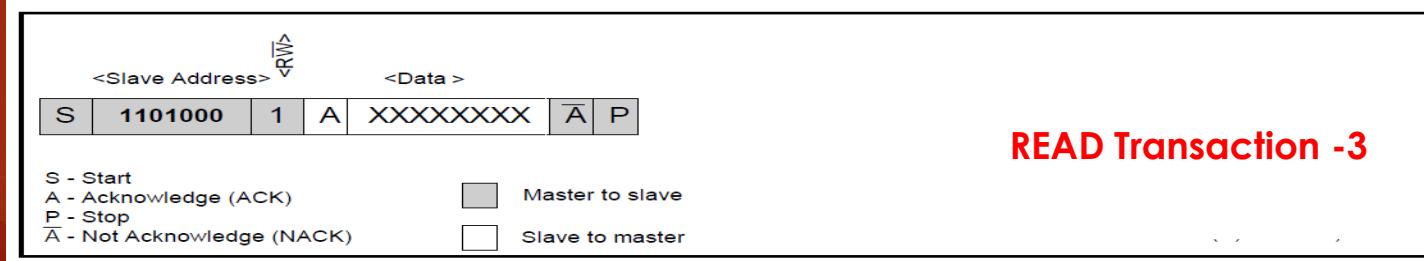
## Data Read— Master reading response from the slave



## Data Read— Master reading response from the slave



## Data Read— Master reading response from the slave



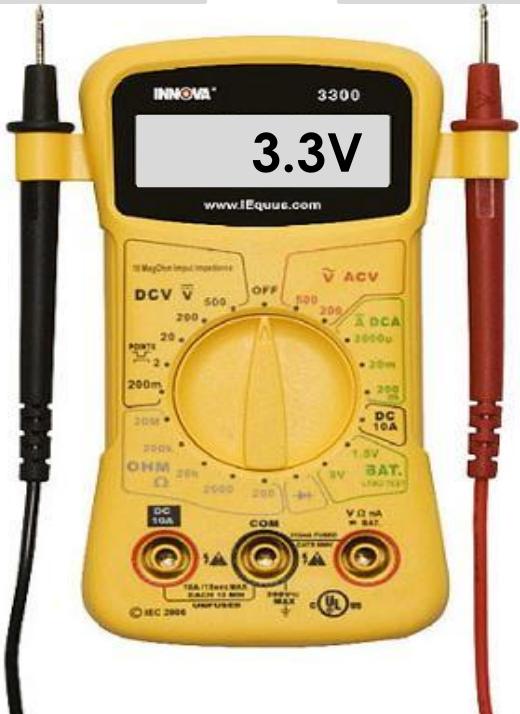
# Common Problems in I2C and Debugging Tips

Copyright © 2019 Bharati Software



GND

SDA & SCL



## Tip

Whenever you face problem in I2C , probe the SDA and SCL line after I2C initialization. It must be held at HIGH level.

# Problem-1: SDA and SCL line not held HIGH Voltage after I2C pin initialization

Copyright © 2019 Bharati Software

Reason-1:

Not activating the pullup resistors if you are using the internal pull up resistor of an I/O line

Debug Tip:

worth checking the configuration register of an I/O line to see whether the pullups are really activated or not, best way is to dump the register contents.

## Problem-2: ACK failure

Reason-1:

Generating the address phase with wrong slave address

Debug Tip:

verify the slave address appearing on the SDA line by using logic analyser.

## Problem-2: ACK failure

Reason-2:

Not enabling the ACKing feature in the I2C control register

Debug Tip:

Cross check the I2C Control register ACK enable field

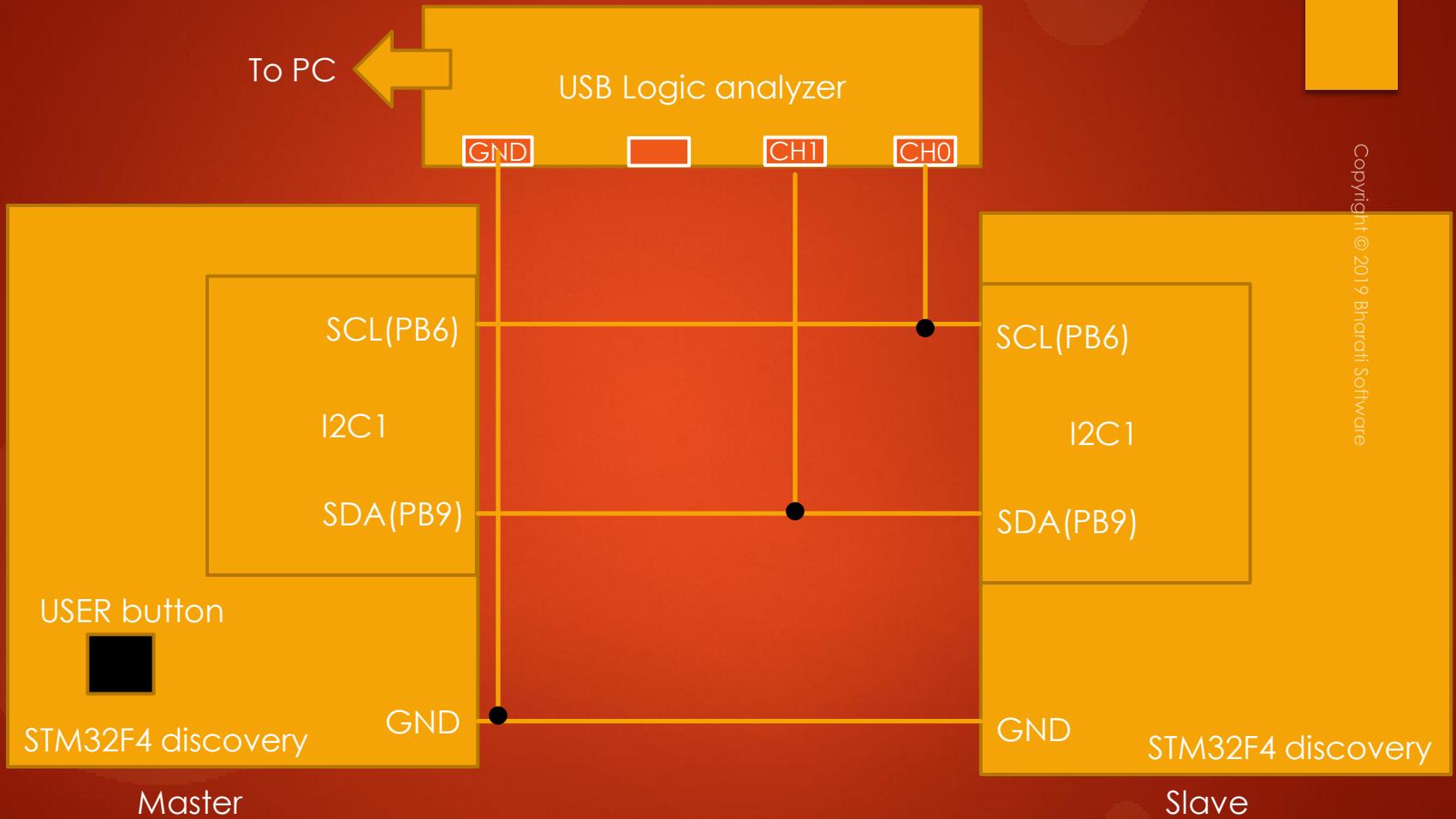
# Problem-3: Master is not producing the clock

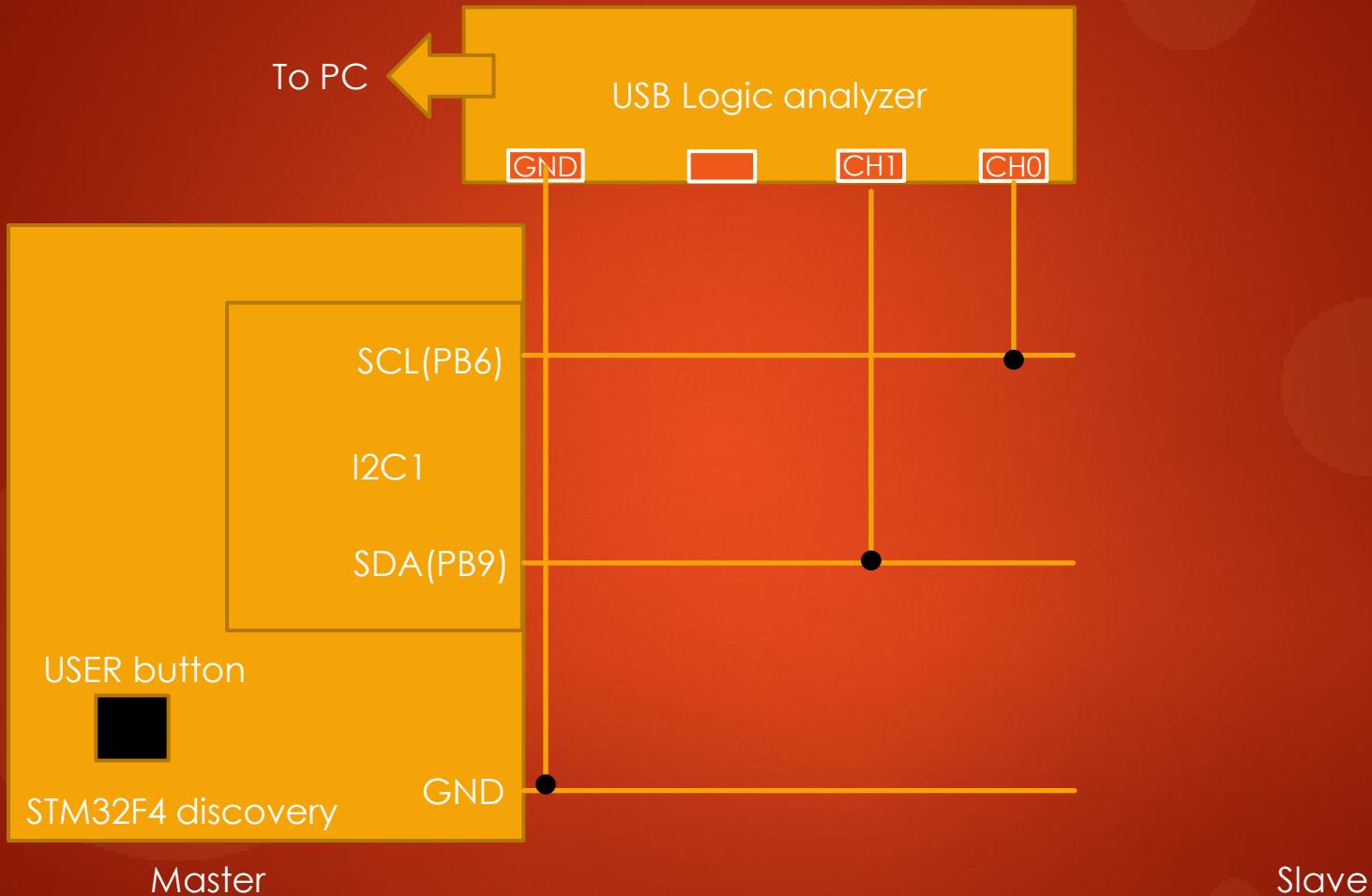
Debug Tip 1 :

First Check whether I2C peripheral clock is enabled and set to at least 2MHz to produce standard mode i2c serial clock frequency

Debug Tip 2 :

Check whether GPIOs which you used for SCL and SDA functionality are configured properly for the alternate functionality





# I2C Clock Stretching

Clock Stretching simply means that holding the clock to 0 or ground level.

The moment clock is held at low, then the whole I2C interface pauses until clock is given up to its normal operation level.

# So, What is the use of clock stretching ?

I2C devices, either Master or Slave, uses this feature ***to slow down the communication*** by stretching SCL to low, ***which prevents the clock to Rise high again*** and the i2c communication stops for a while

There are situations where an ***I2C slave is not able to co-operate with the clock speed*** given by the master and needs to slow down a little.

If slave needs time, then it takes the advantage of clock stretching , and by holding clock at low, it momentary pauses the I2C operation.

# Modifying I2C application to send/receive more than 32 bytes

# Introduction :UART vs USART

# Exploring UART Interrupt Mapping

UART  
Universal  
Asynchronous  
Receiver  
Transmitter

USART  
Universal  
Synchronous  
Asynchronous  
Receiver  
Transmitter

UART Peripheral supports only asynchronous mode

USART supports both asynchronous and synchronous modes.

You can use USART module both in synchronous mode as well as in asynchronous mode

There is no specific port for USART communication. They are commonly used in conjugation with protocols like RS-232, RS-434, USB etc.

In synchronous transmission, the clock is sent separately from the data stream and no start/stop bits are used

# USART hardware components

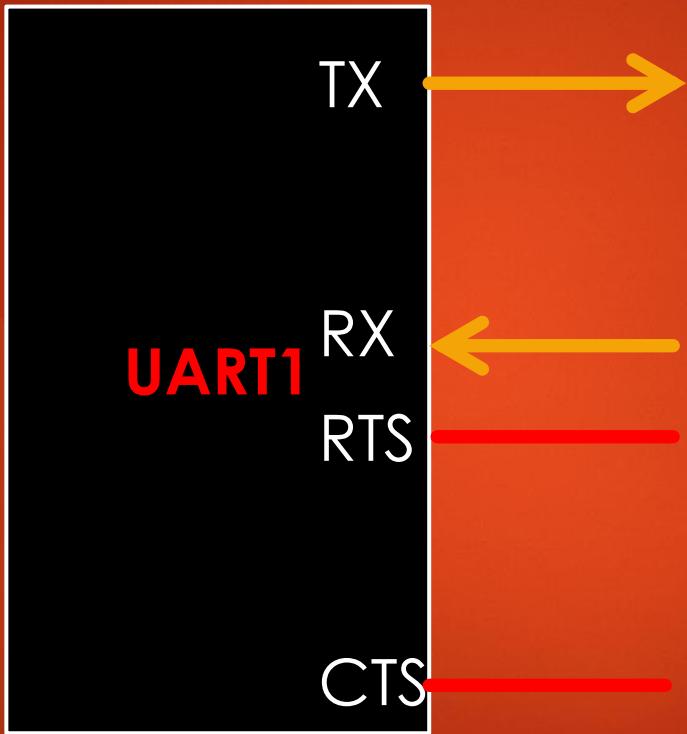
- ▶ Baud rate generator
- ▶ TX and RX shift registers
- ▶ Transmit/Receive control blocks
- ▶ Transmit/Receive buffers
- ▶ First-in, First-out (FIFO) buffer memory

USART is just a piece of hardware in your microcontroller which transmits and receives data bits either in Synchronous mode or in Asynchronous mode.

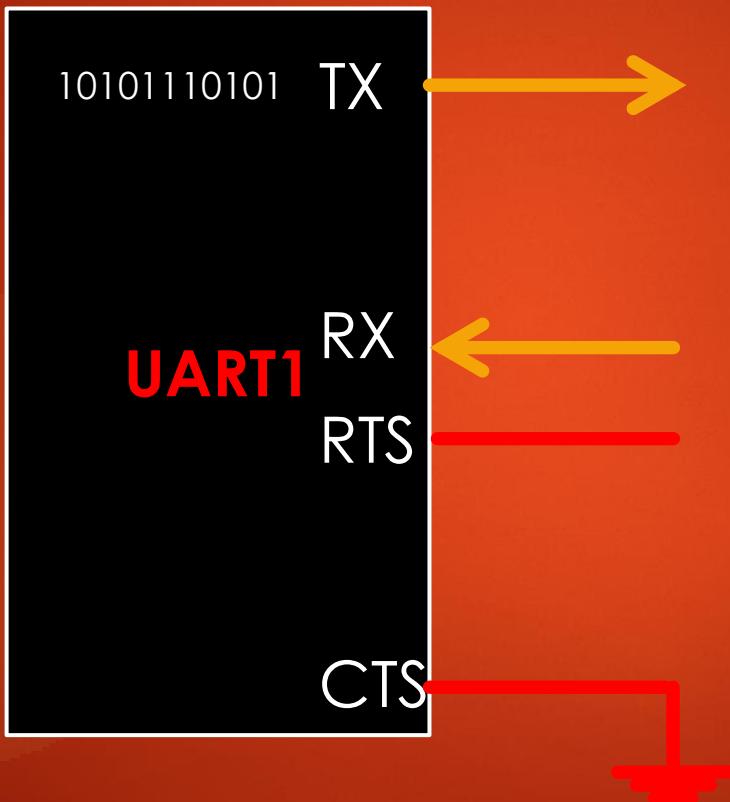
If it is Asynchronous mode, then clock will not be sent along with the data, instead we use synchronization bits like start and stop along with the useful data.

# Understanding UART pins

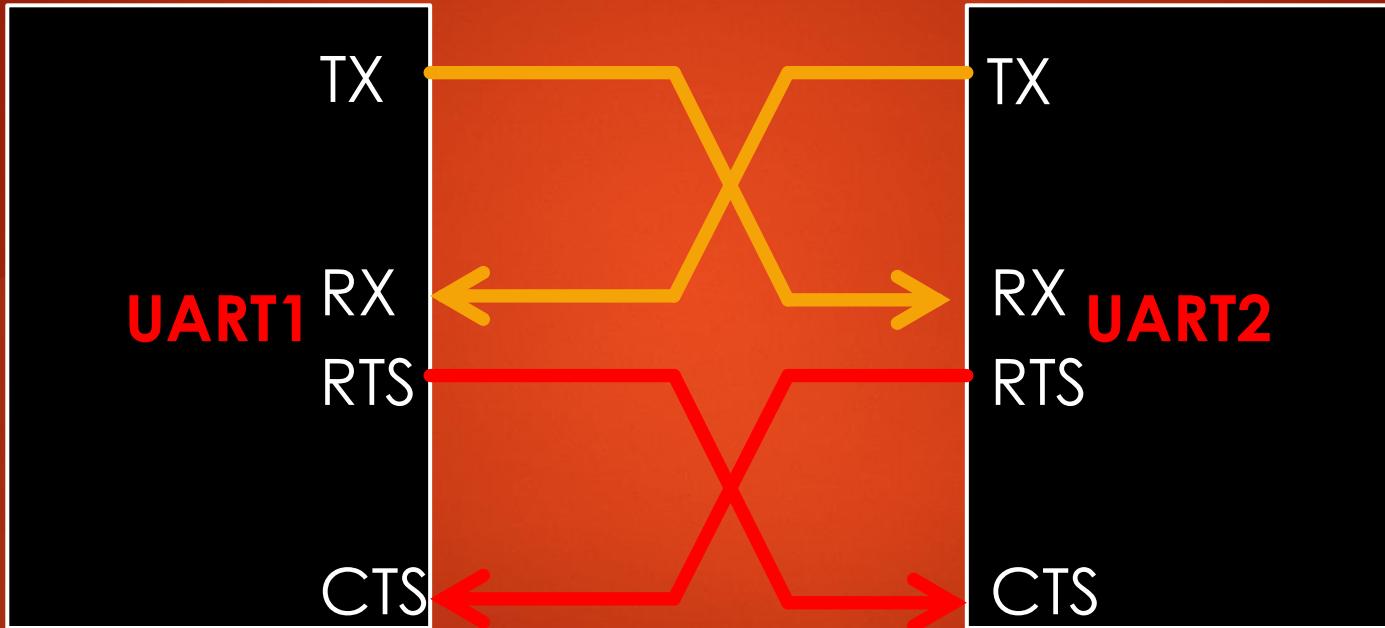
# Understanding UART pins



# Understanding UART pins



# Understanding UART pins

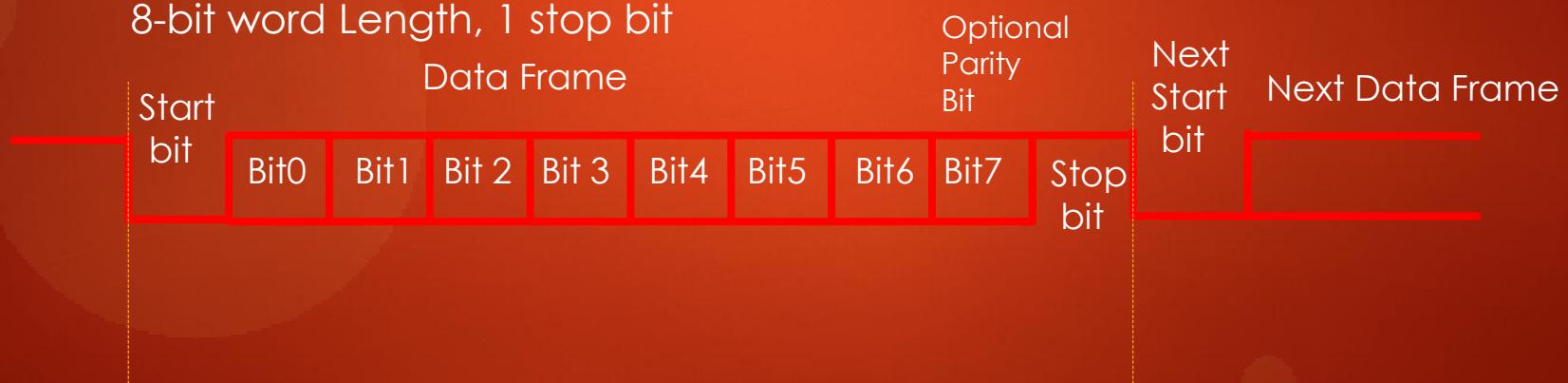


# UART frame formats

9-bit word Length, 1 stop bit

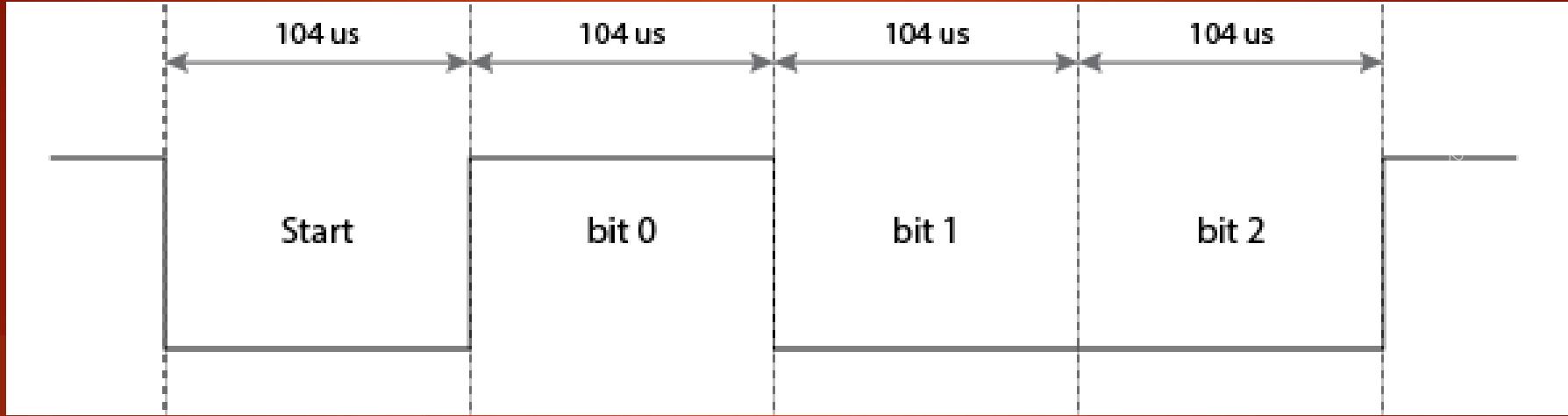


8-bit word Length, 1 stop bit

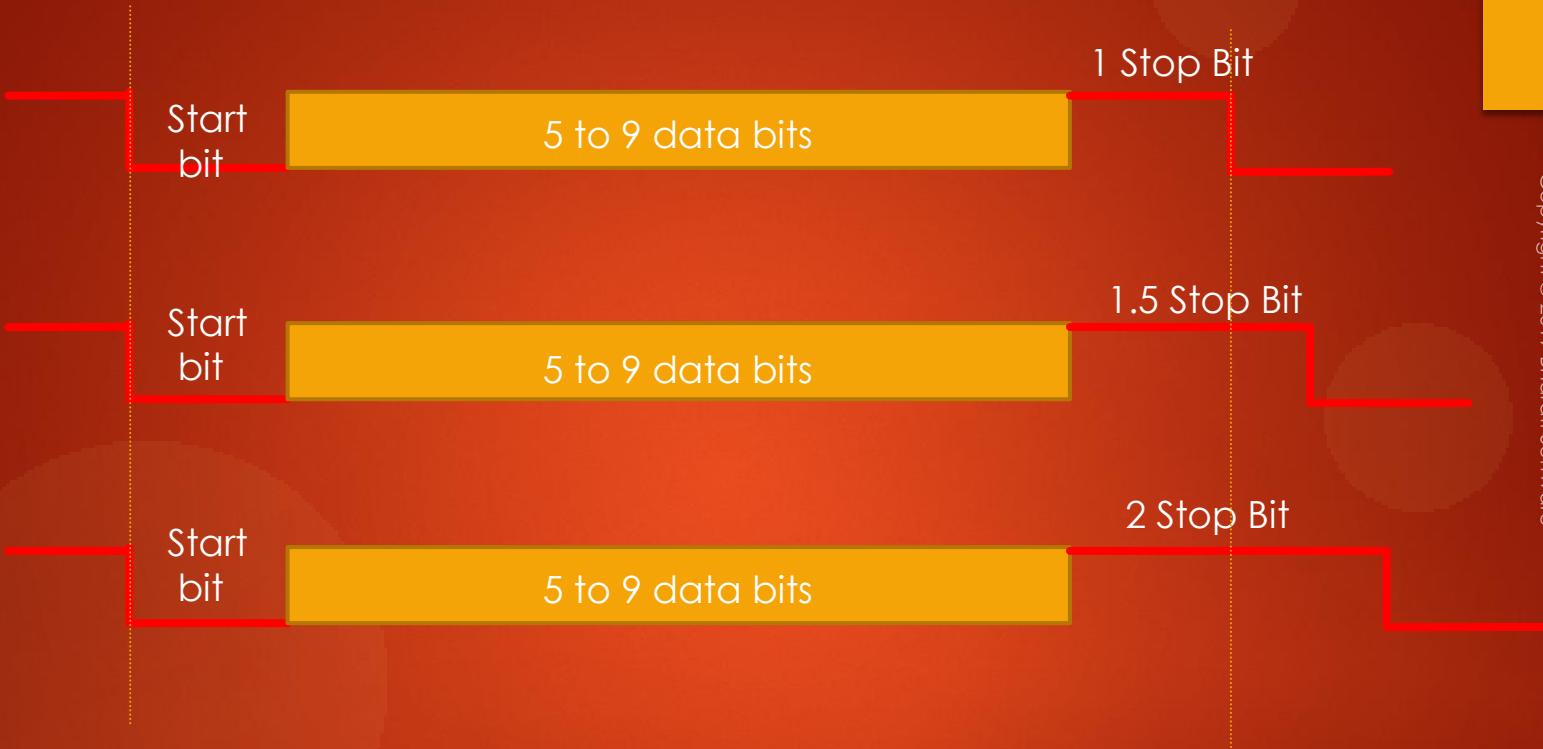


# UART Baudrate

The significance of baud rate is **how fast the** data is sent over a serial line. It's usually expressed in units of bits-per-second (bps). If you invert the baud rate, you can find out, just how long it takes to transmit a single bit. This value determines how long the transmitter holds a serial line high or low .



# UART Synchronization bits



# UART Parity

Adding Parity bit is a simplest method of error detection. Parity is simply the number of ones appearing in the binary form of a number.

55(Decimal)



0b00110111

Parity = 5

Parity

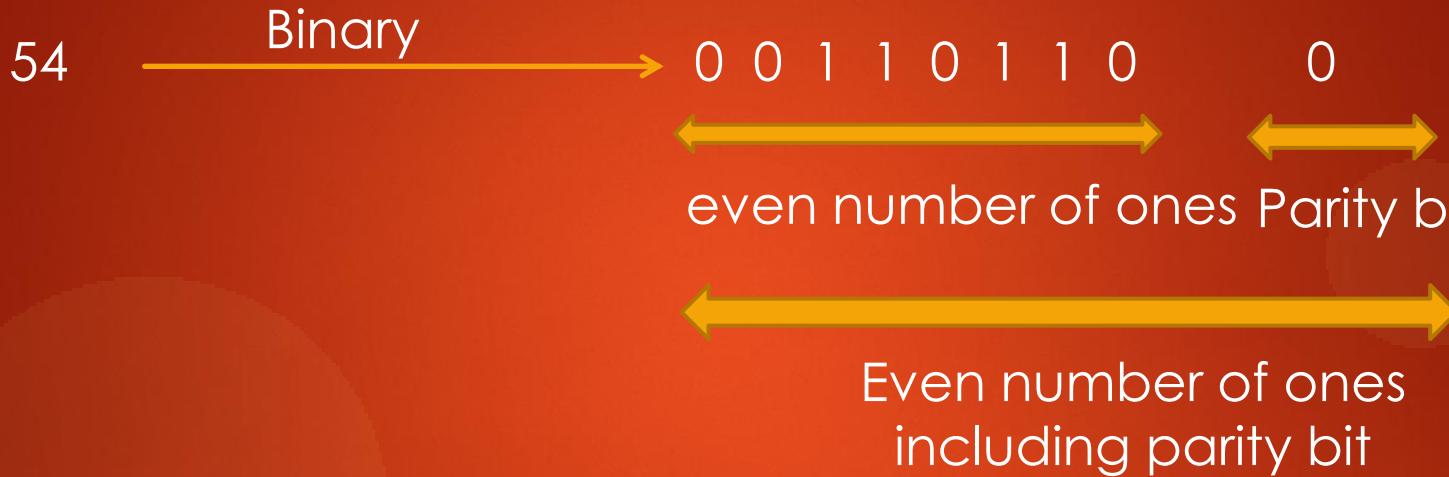
Even Parity

Odd parity

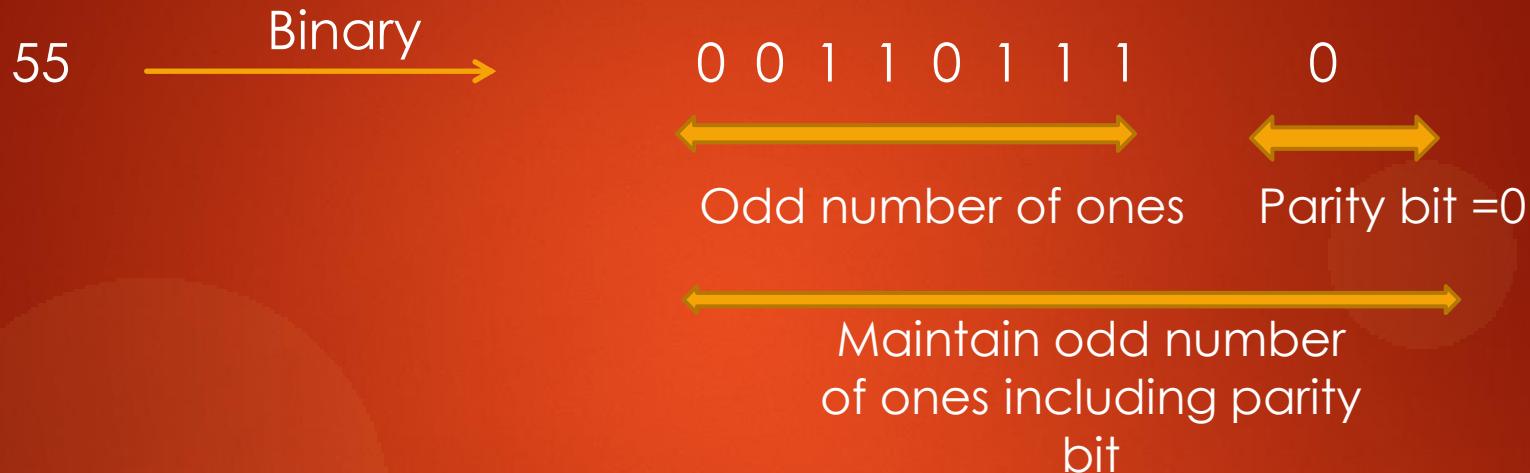
# Even parity



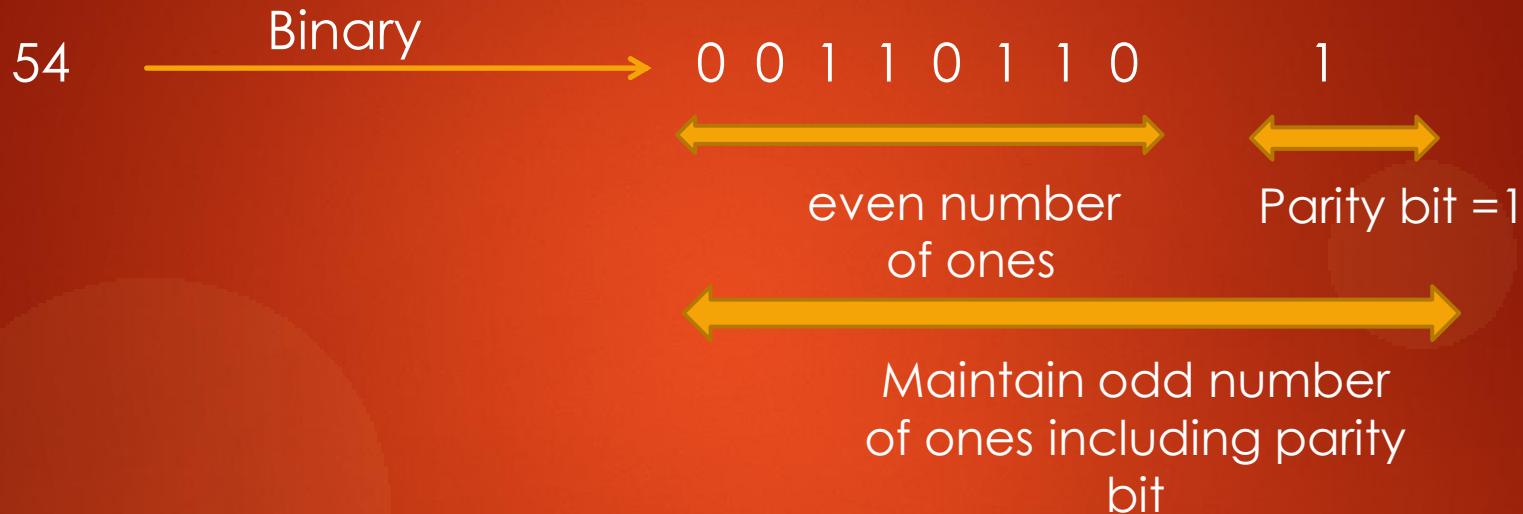
# Even parity



# Odd parity



# Odd parity



# Simply remember

Even parity results in even number of 1s, whereas odd parity results in odd number of 1s, when counted including the parity bit.

# Why use the Parity Bit?

Data gets corrupted  
during transmission

1101011

1100011

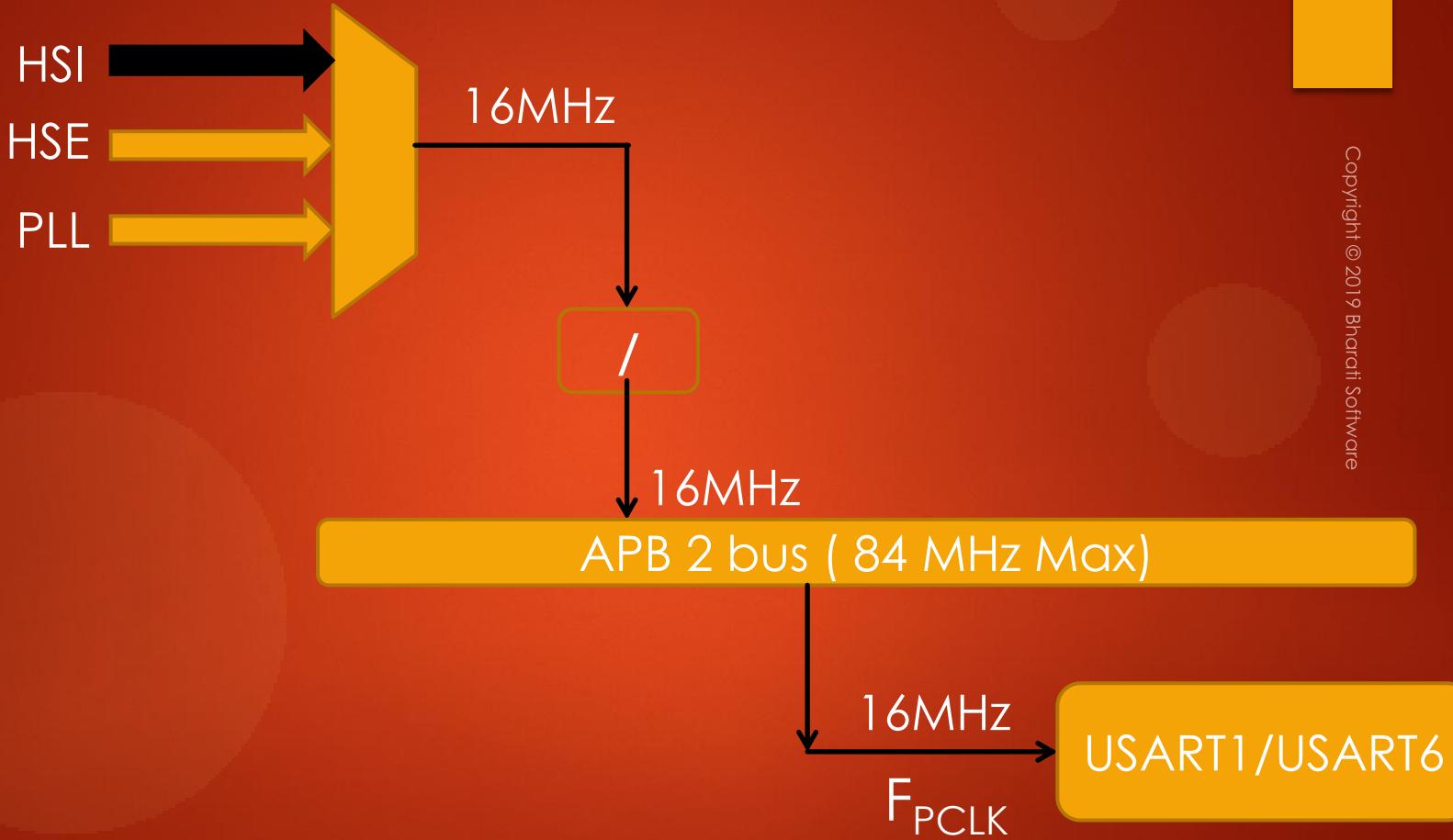


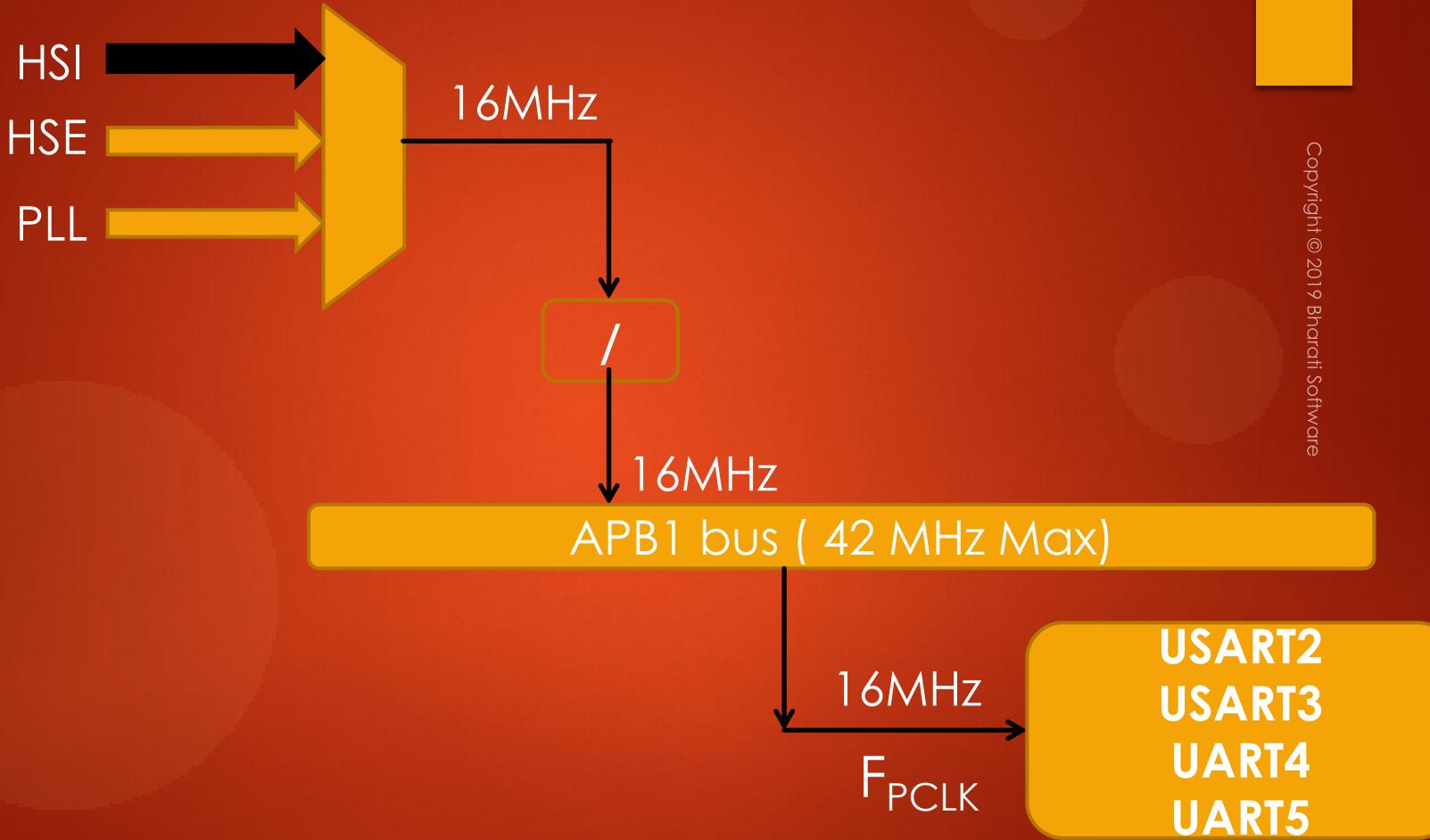
Represents EVEN parity  
. i.e even number of 1s  
including the parity bit

Here it has a odd number  
of 1s including the parity  
bit. Which implies that  
data is corrupted and has  
to be discarded.

# UART Functional block

# UART Peripheral Clock



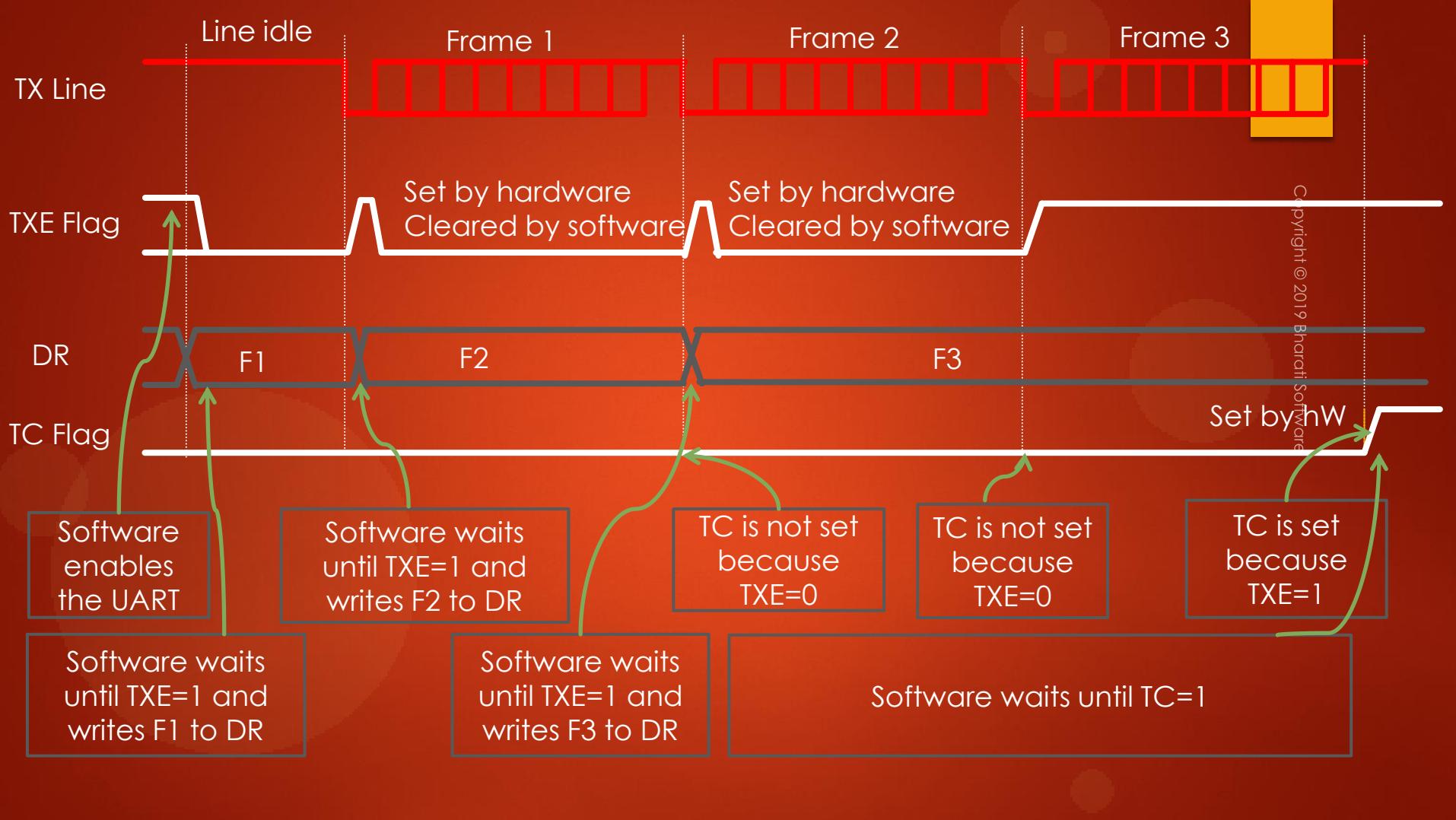


# UART Transmitter

# Steps to do Data Transmission

# Steps to do Data Transmission

- ▶ Program the M bit in USART\_CR1 to define the word length
- ▶ Program the number of stop bits in USART\_CR2 register.
- ▶ Select the desired baud rate using the USART\_BRR register
- ▶ Set the TE bit in USART\_CR1 to enable the transmit block. .
- ▶ Enable the USART by writing the UE bit in USART\_CR1
- ▶ Now if txe flag is set , then Write the data byte to send , in the USART\_DR register .
- ▶ After writing the last data into the USART\_DR register, wait until TC=1

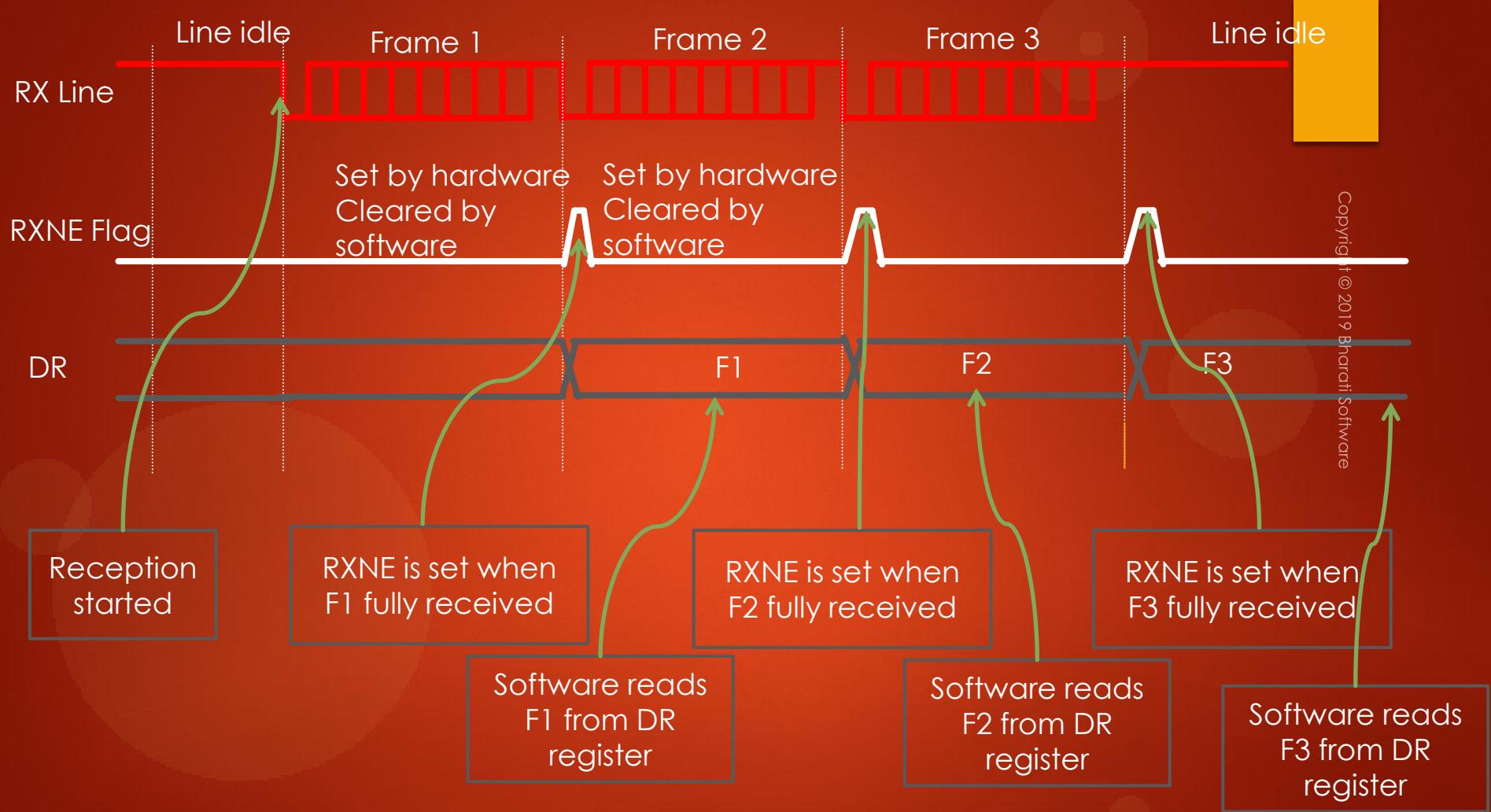


# UART Receiver

# Steps to do Data Reception

# Steps to do Data Reception

- ▶ Program the M bit in USART\_CR1 to define the word length
- ▶ Program the number of stop bits in USART\_CR2 register.
- ▶ Select the desired baud rate using the USART\_BRR register
- ▶ Enable the USART by writing the UE bit in USART\_CR1
- ▶ Set the RE bit in the USART\_CR1 register , which enables the receiver block of the usart peripheral
- ▶ When a character is received, wait till The RXNE bit is set and read the data byte from the data register
- ▶ The RXNE bit must be cleared by reading the data register , before the end of the reception of the next character to avoid an overrun error.



# UART Interrupts

# Over sampling

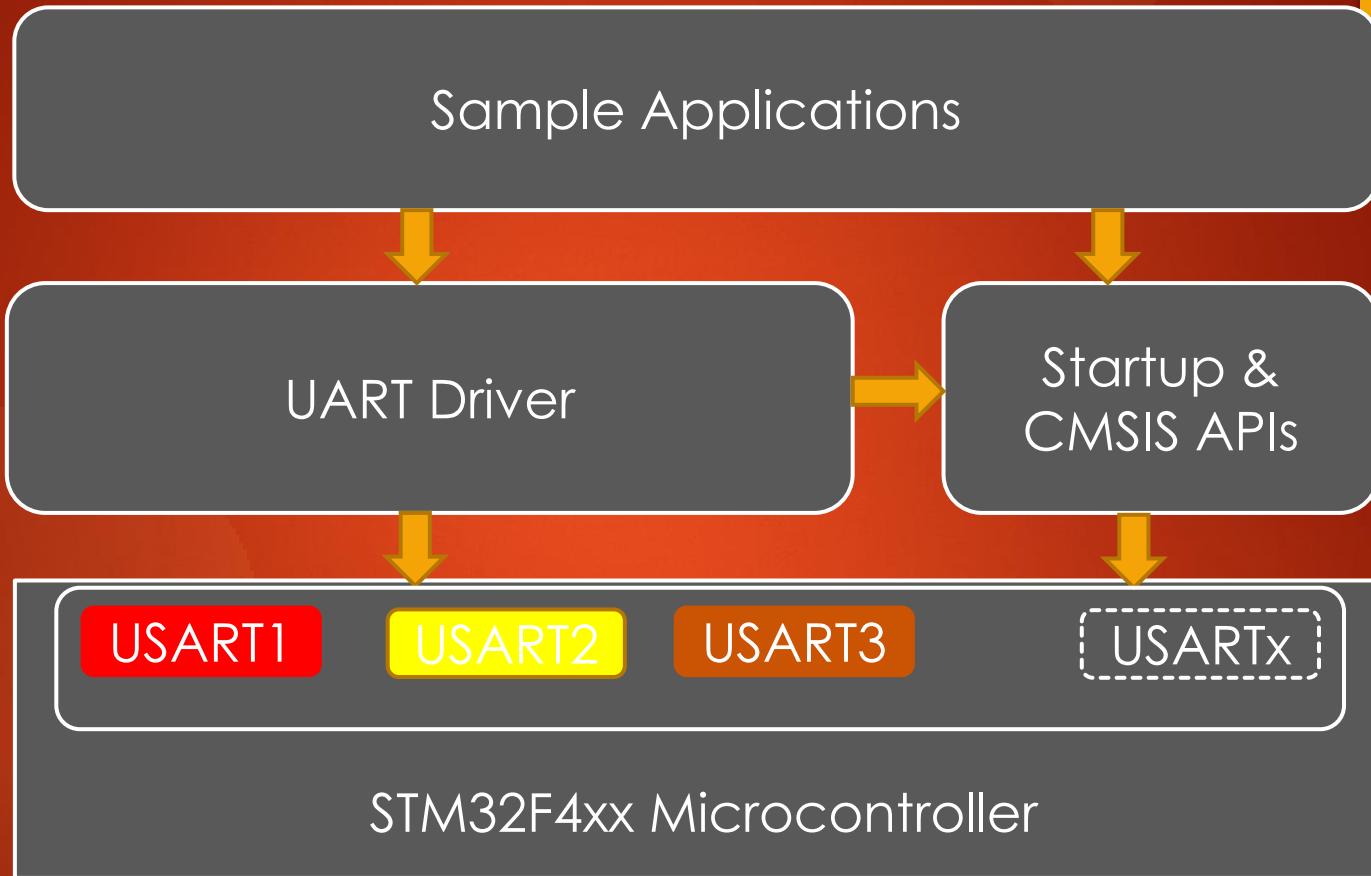
- ▶ What is oversampling
- ▶ How does it effect the baudrate

# Connecting UART Pins to PC

- ▶ Using TTL to USB Converter
- ▶ Using RS232 cable with DB9 Connector

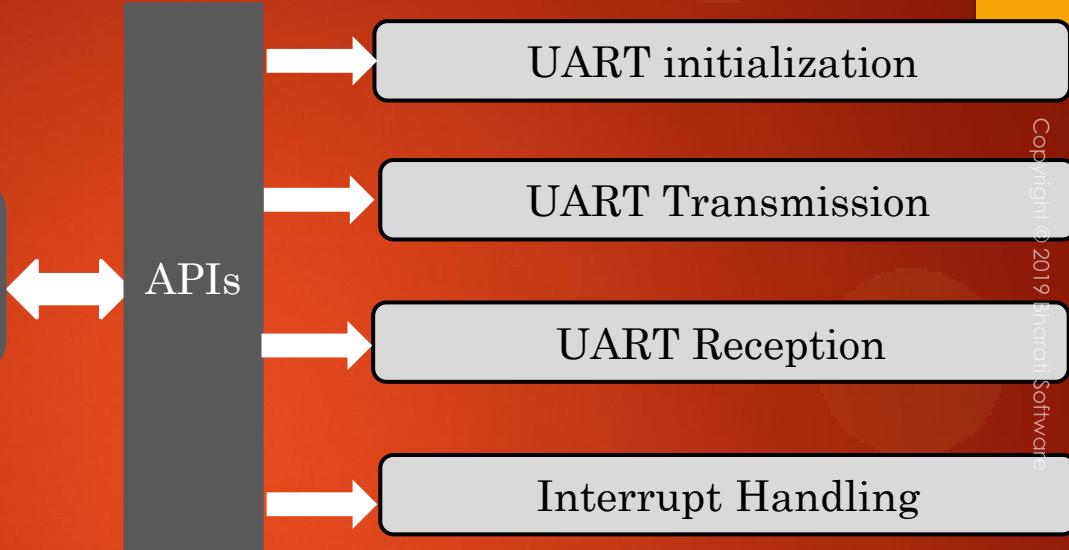
- ▶ Using TTL to USB Converter
- ▶ **Using RS232 cable with DB9 Connector**

# Overview



# Understanding the Requirements

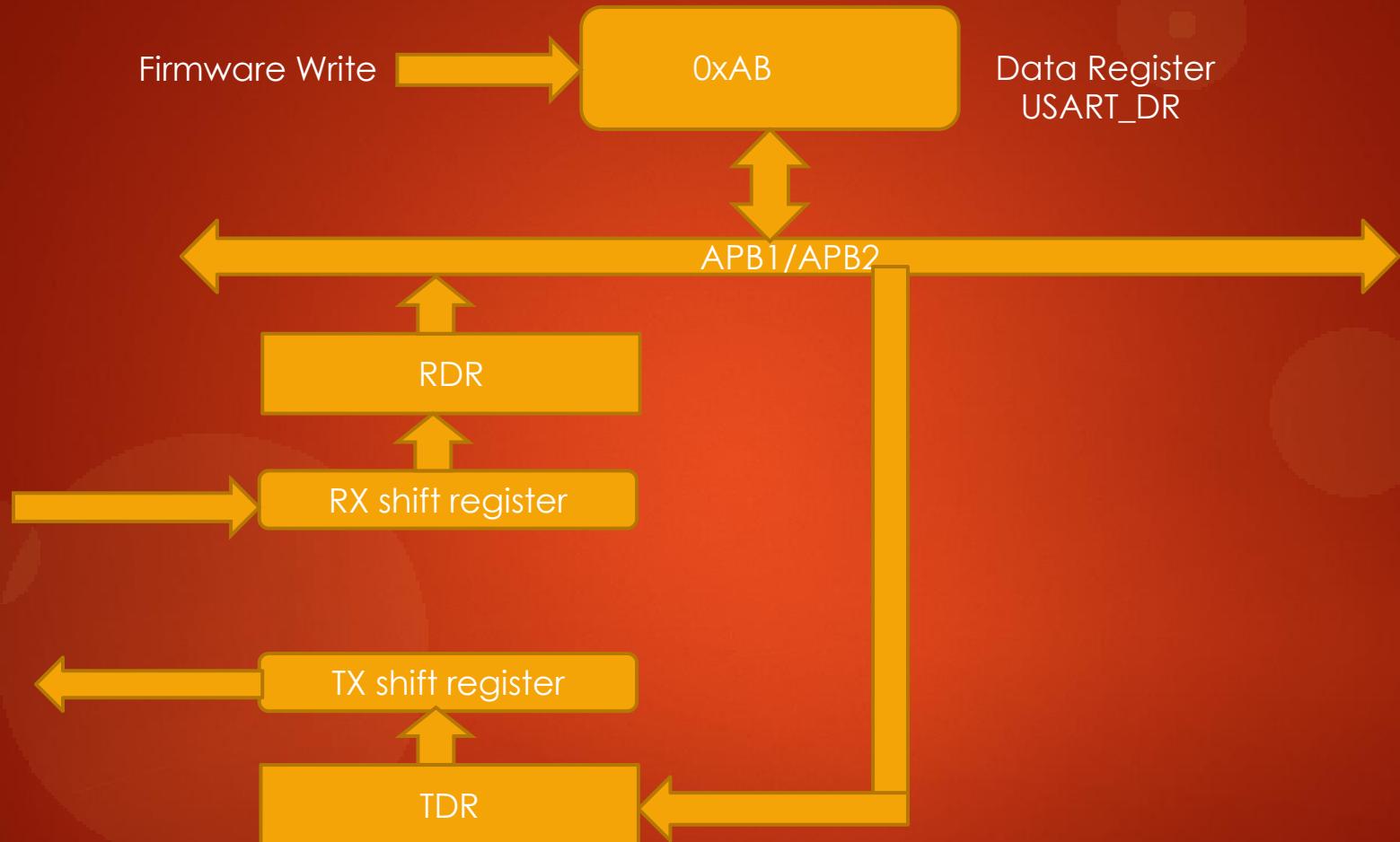
UART Driver

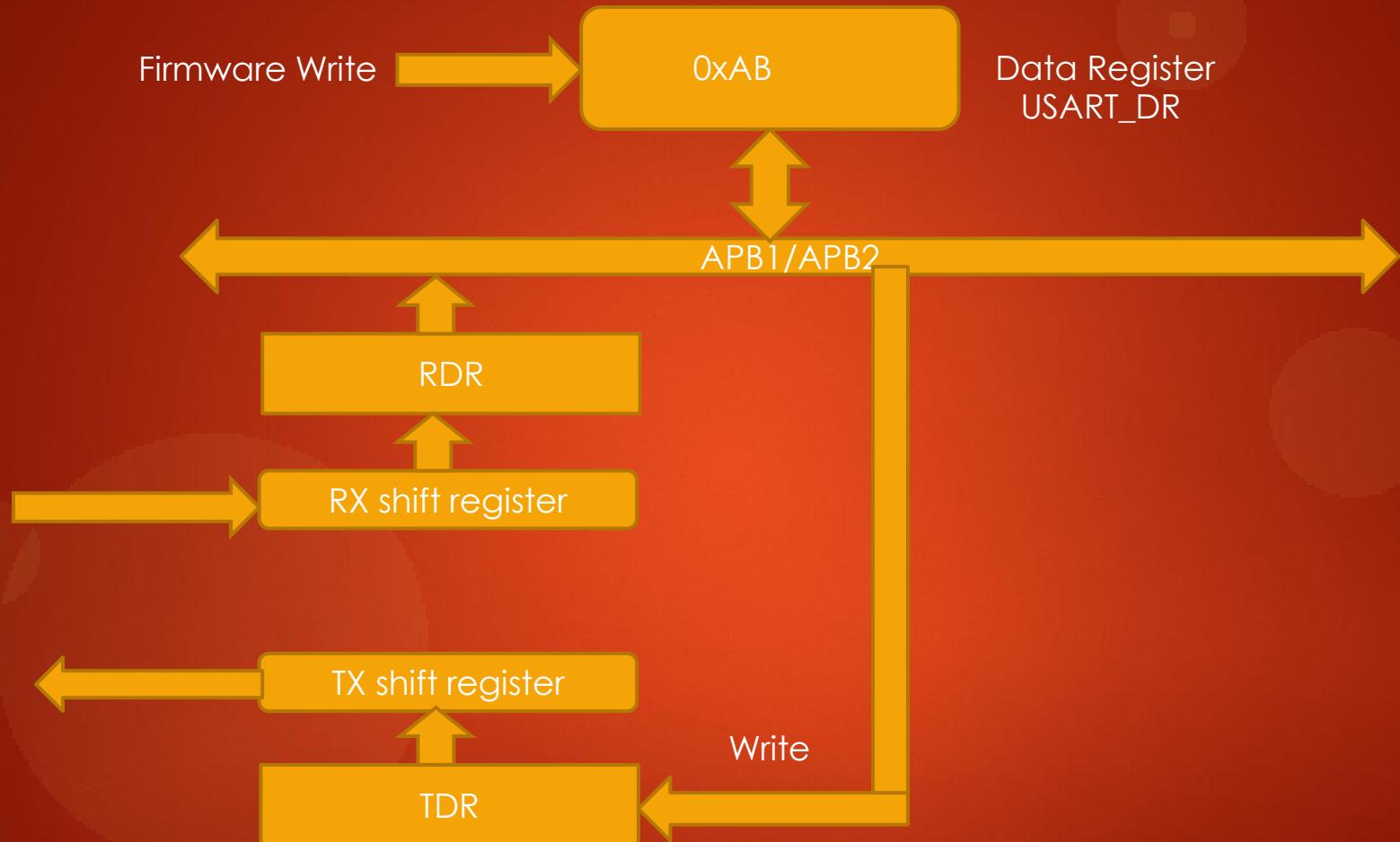


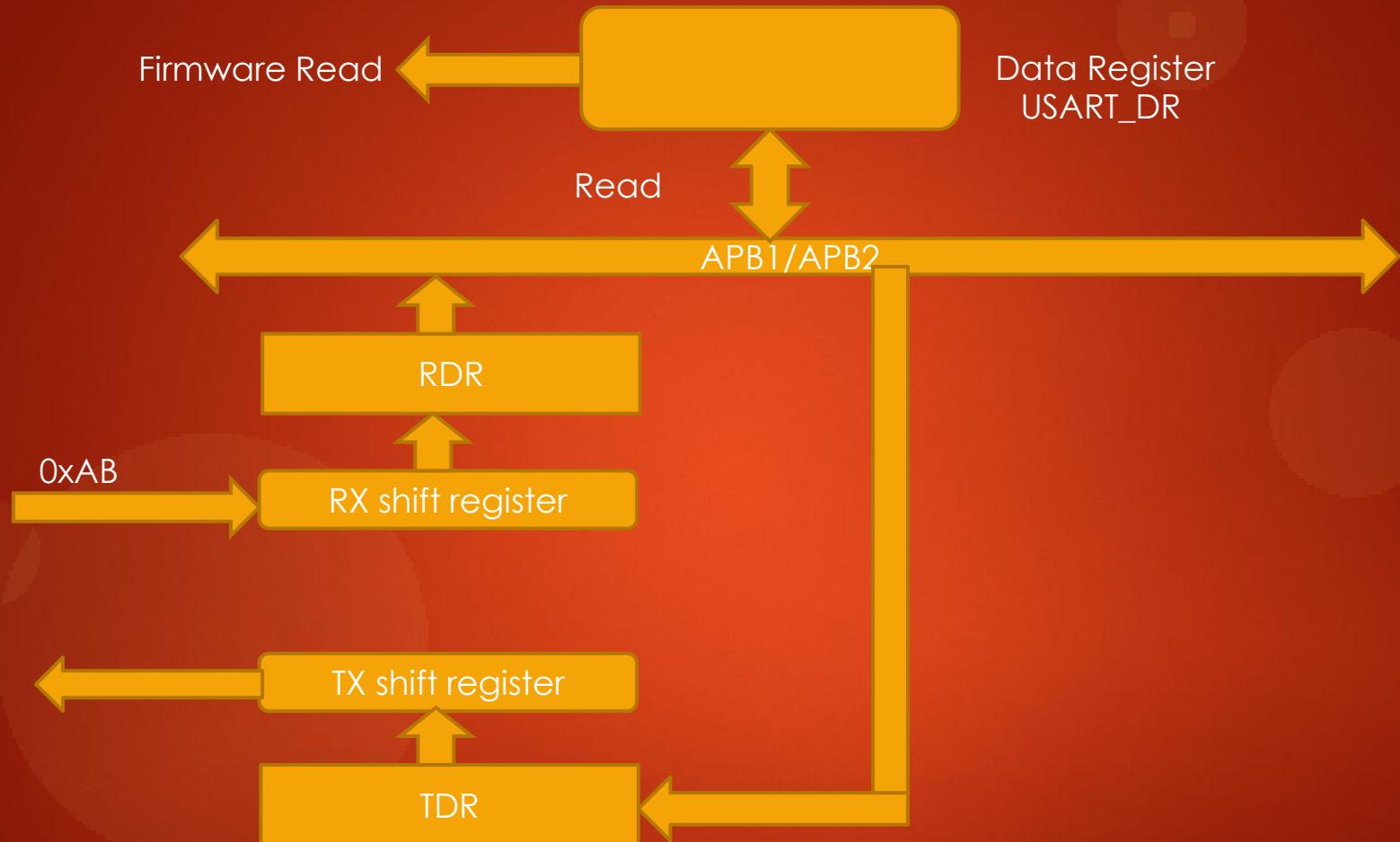
# Exploring different UART Peripherals and pins of the MCU

U(S)ARTs	Pin Pack 1		Pin pack 2		Pin pack 3		Bus
	TX	RX	TX	RX	TX	RX	
USART1	PA9	PA10	PB6	PB7			APB2
USART2	PA2	PA3	PD5	PD6			APB1
USART3	PB10	PB11	PC10	PC11	PD8	PD9	APB1
UART4	PA0	PA1	PC10	PC11			APB1
UART5	PC12	PD2					APB1
USART6	PC6	PC7	PG14	PG9			APB2

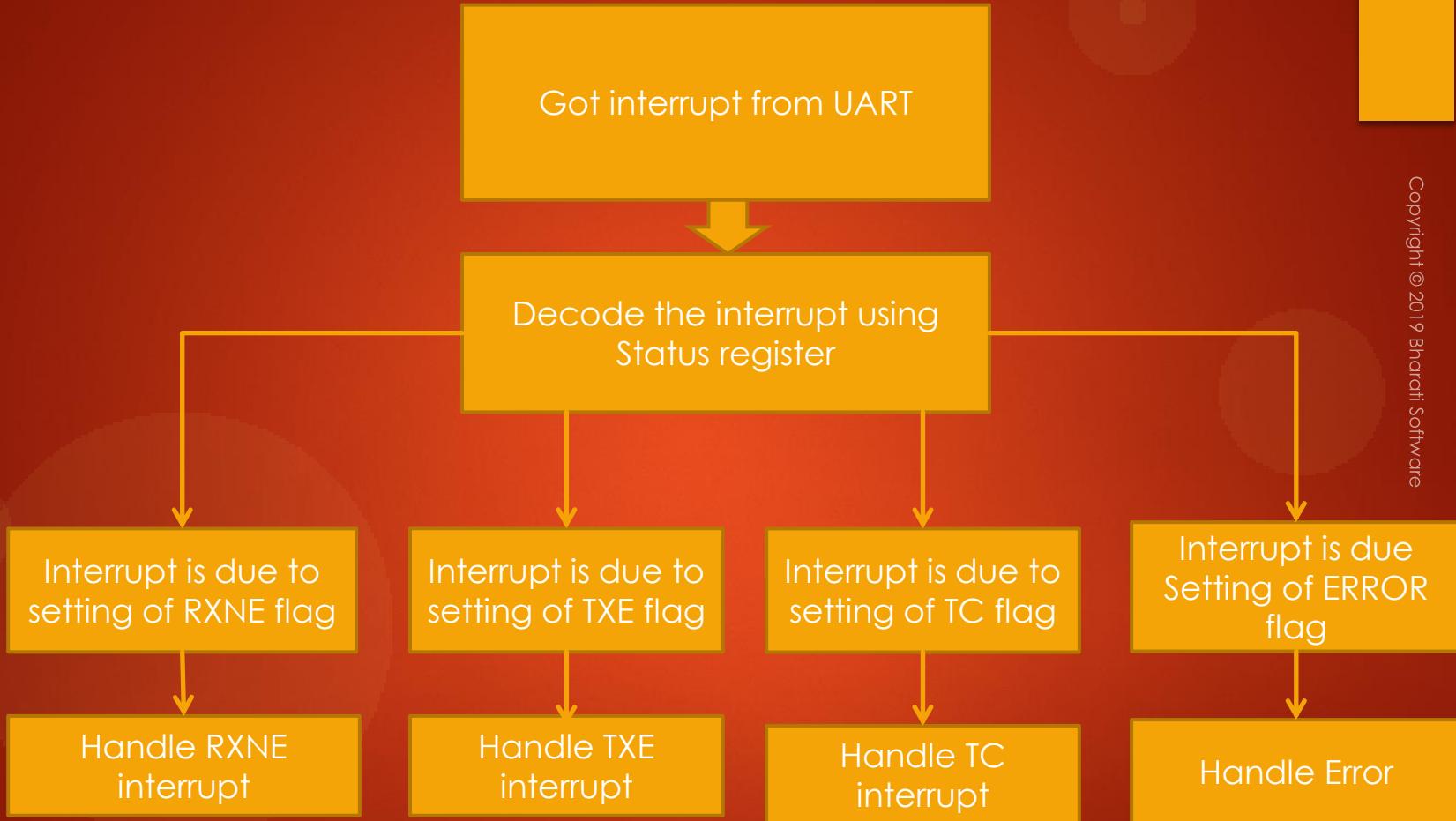
# UART Data Register

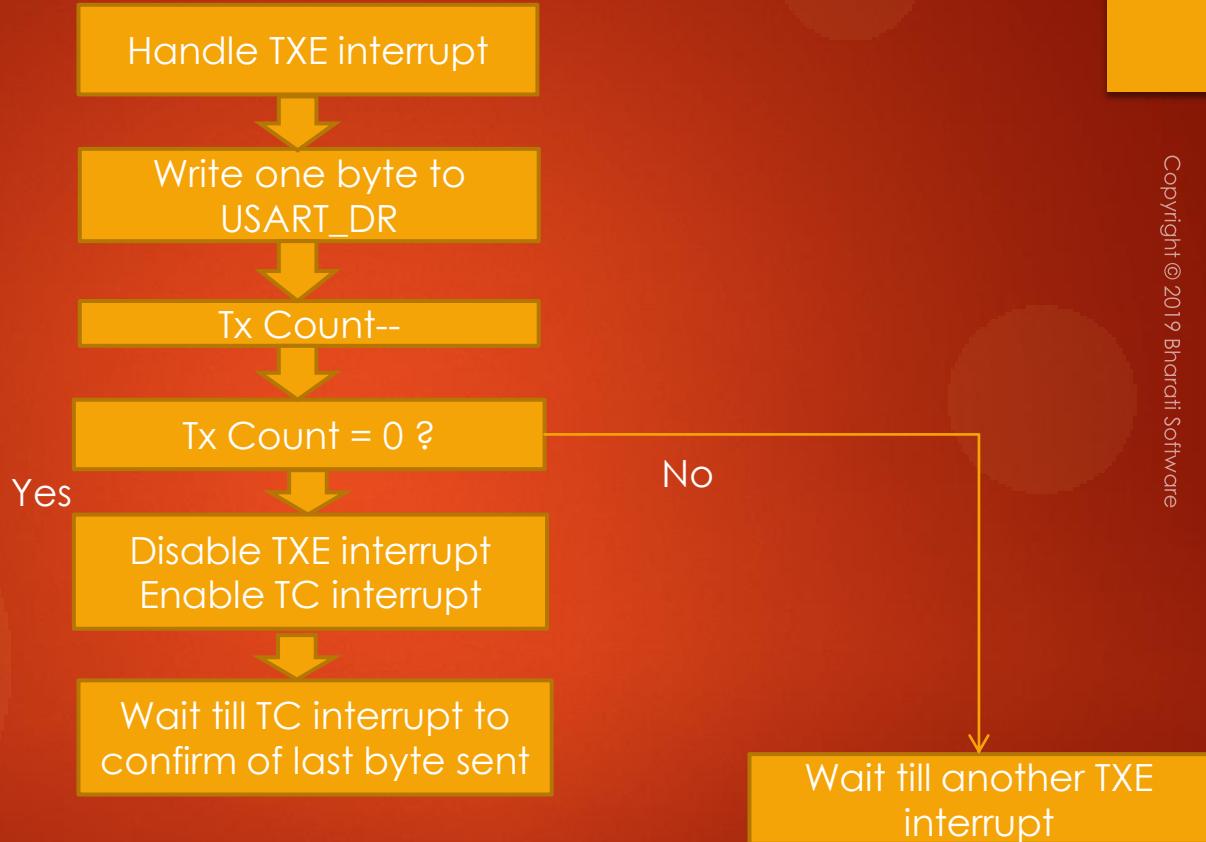




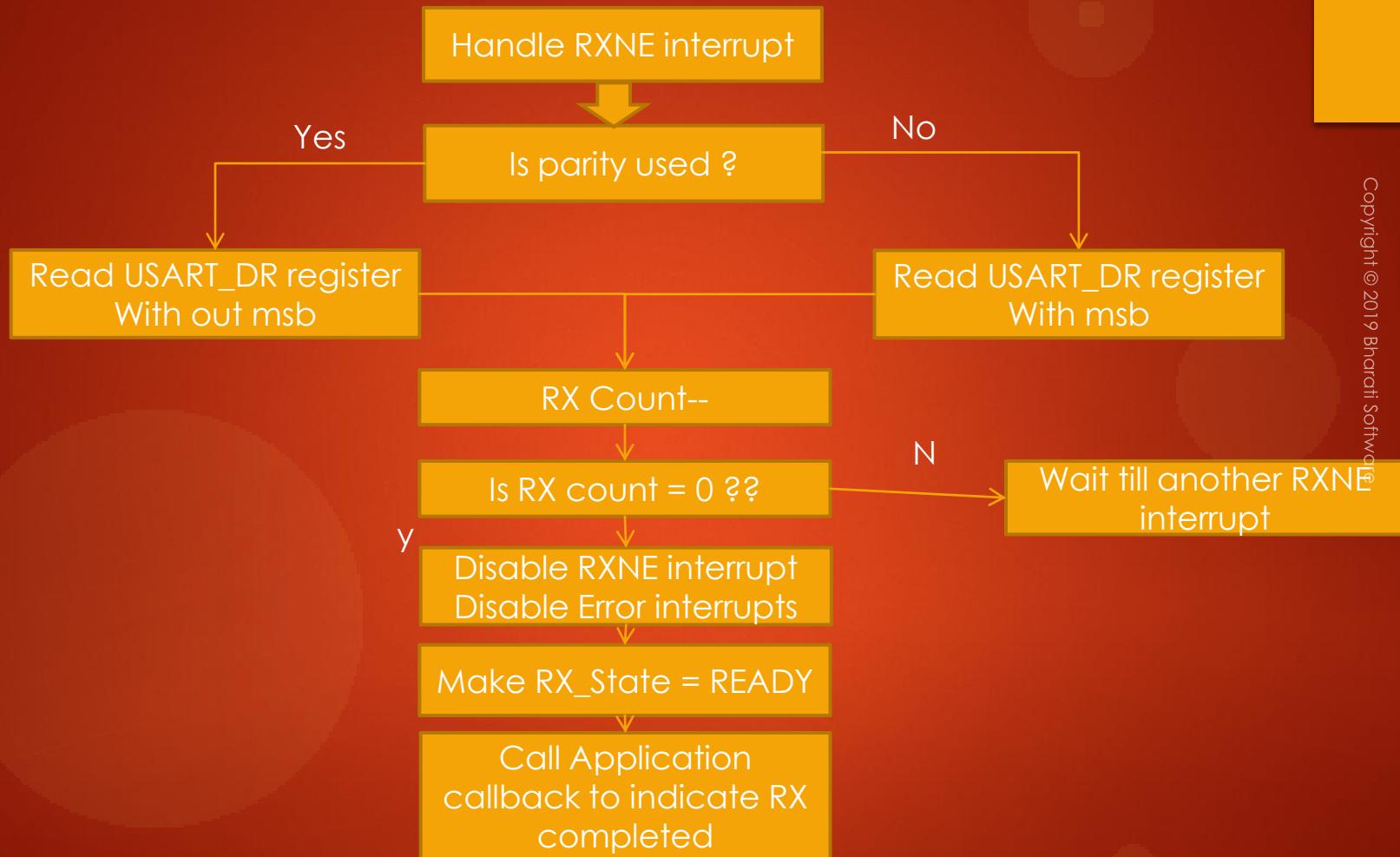


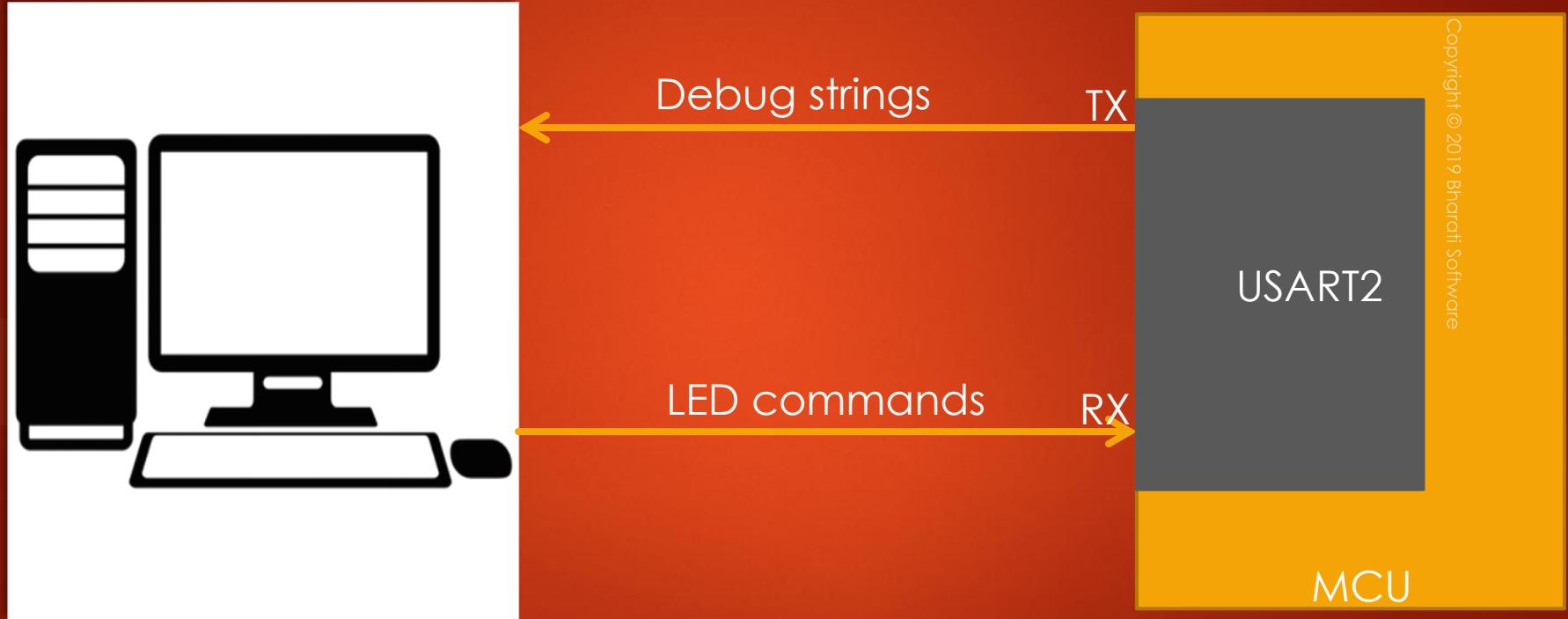
# UART Status Register









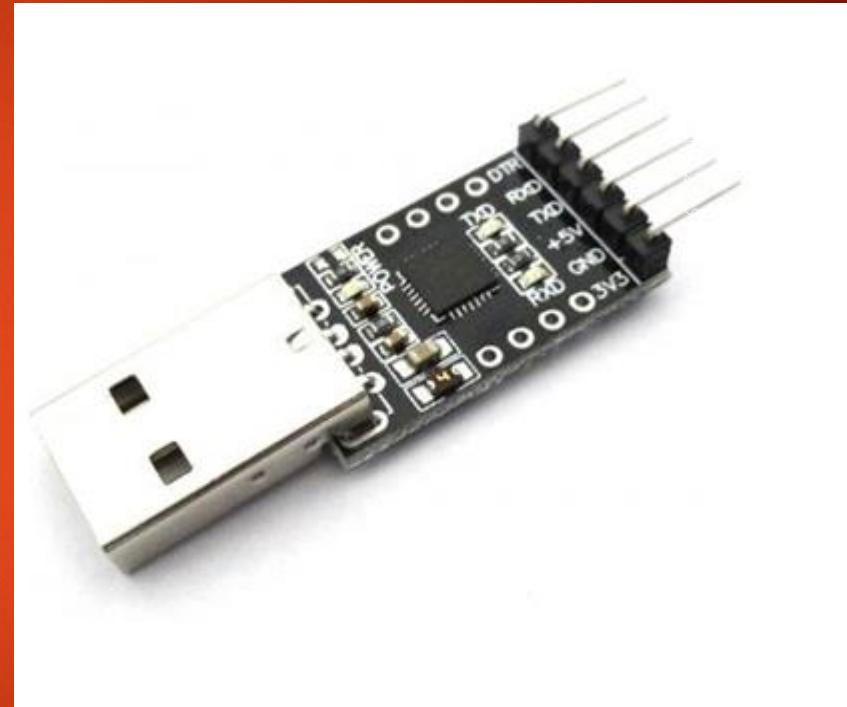


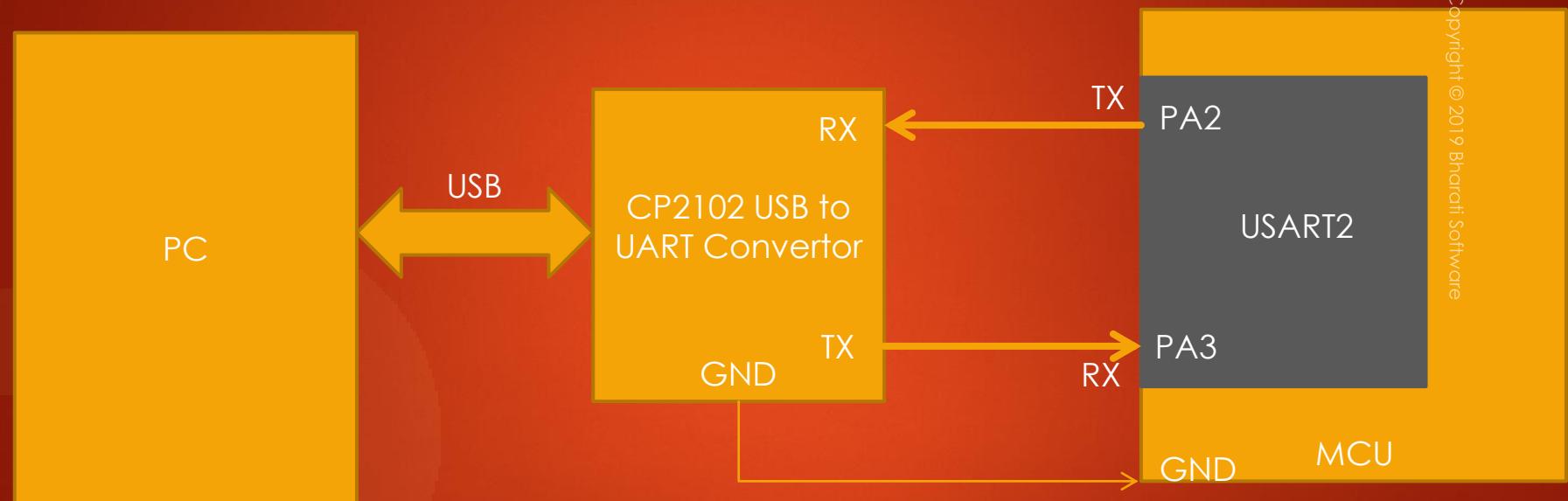
Commands	Function
<b>LEDOH</b>	Turn ON LED Orange
<b>LEDOL</b>	Turn OFF LED Orange
<b>LEDBH</b>	Turn ON LED Blue
<b>LEDBL</b>	Turn OFF LED Blue
<b>LEDRH</b>	Turn ON LED Red
<b>LEDRL</b>	Turn OFF LED Red
<b>LEDGH</b>	Turn ON LED Green
<b>LEDGL</b>	Turn OFF LED Green
<b>LEDAH</b>	Turn ON ALL LEDs
<b>LEDAL</b>	Turn OFF ALL LEDs

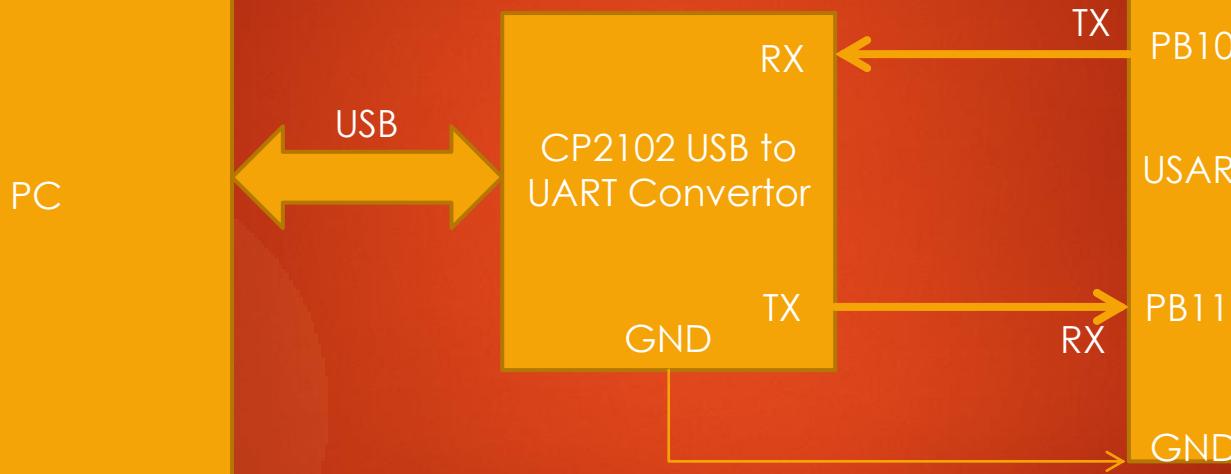
# Different USART Peripherals and Associated Pin Packs

U(S)ARTs	Pin Pack 1		Pin pack 2		Pin pack 3		Bus
	TX	RX	TX	RX	TX	RX	
USART1	PA9	PA10	PB6	PB7			APB2
USART2	PA2	PA3	PD5	PD6			APB1
USART3	PB10	PB11	PC10	PC11	PD8	PD9	APB1
UART4	PA0	PA1	PC10	PC11			APB1
UART5	PC12	PD2					APB1
USART6	PC6	PC7	PG14	PG9			APB2

# USB to TTL USB UART serial port converter module







**Uartprintf library uses USART3 to output debug messages**

# Implementing UART\_PRINTF

By using this printf function you can able to print debug messages over UART, just like standard printf

write a code for uart  
printf  
uartprintf.c

create a static  
library  
uartprintf.lib

Use uartprintf.lib in any  
application to enable  
the prints over UART

Both transmitting as well as receiving devices should operate at the same rate.

2400

4800

9600

19200

38400

57600

115200

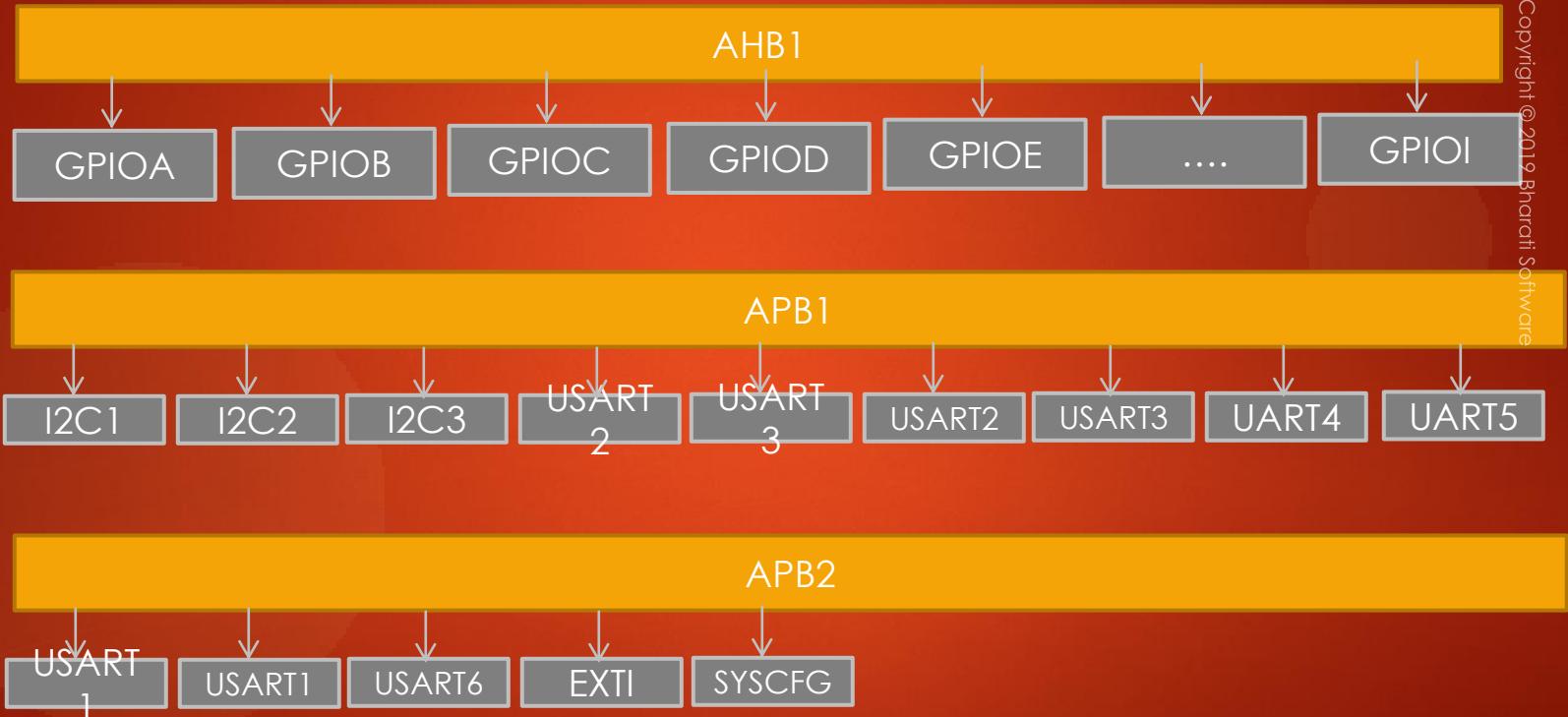
$F_{\text{pclk}} = 16\text{MHz}$

OVER8=0

Desired baudrate	USARTDIV value	Hex conversion (USART_BRR)
9600bps	104.1875	0x683
115200 bps	8.6875	0x8A
921600 bps(Max)	1.0625	0x11

# USART and UART peripherals of your MCU

Copyright © 2019 Bharati Software



# USART Driver Development

# Sample Applications

Copyright © 2019 Bharatii Software

## Driver Layer

gpio\_driver.c , .h

i2c\_driver.c , .h

(Device header)

Stm3f407xx.h

USART\_driver.c , .h

uart\_driver.c , .h

GPIO

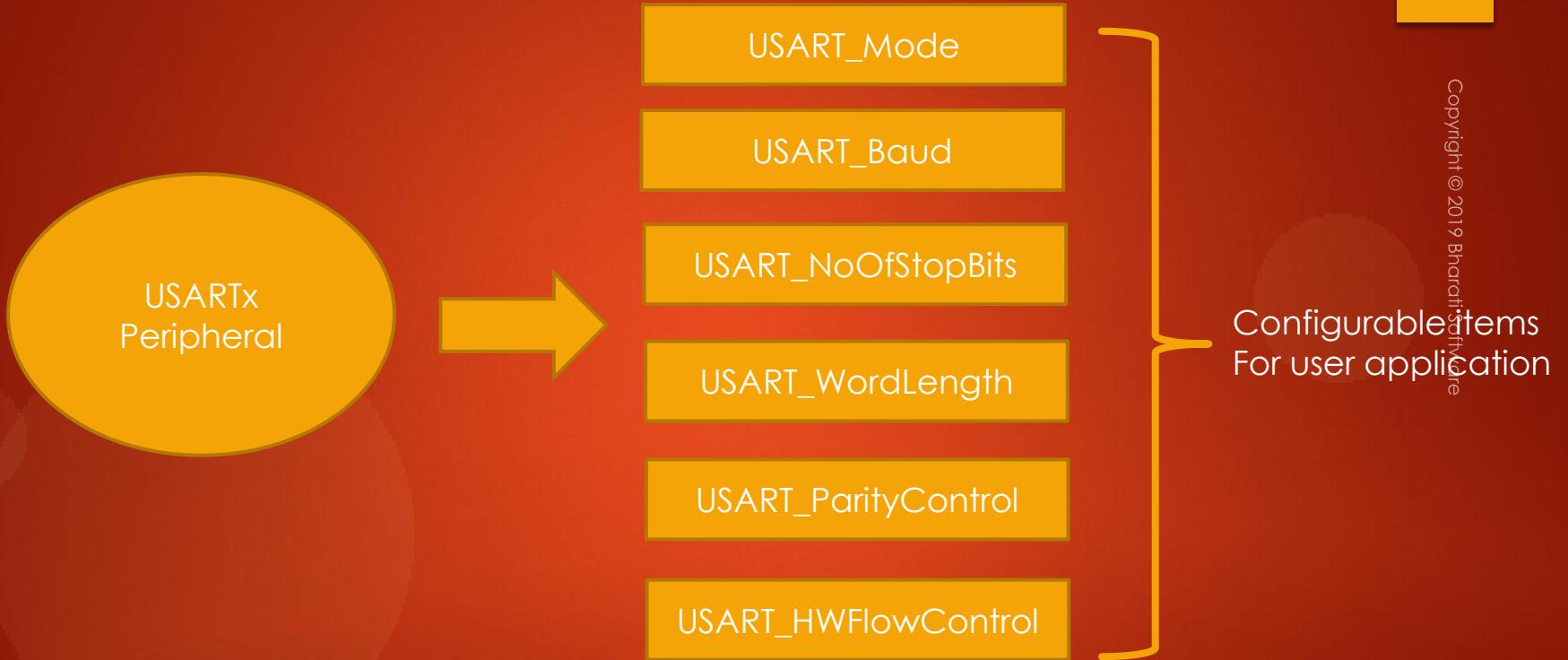
USART

I2C

UART

**STM3F407x MCU**

# Driver API requirements and user configurable items



USART Driver

APIs

USART Initialization / peripheral  
clock control

USART TX

USART RX

USART Interrupt config. &  
handling

Other USART management  
APIs

# USART handle structure and configuration structure

```
/*
 * Configuration structure for USARTx peripheral
 */
typedef struct
{
    uint8_t USART_Mode;
    uint32_t USART_Baud;
    uint8_t USART_NoOfStopBits;
    uint8_t USART_WordLength;
    uint8_t USART_ParityControl;
    uint8_t USART_HWFlowControl;
}USART_Config_t;
```

## USART Configuration Structure

```
/*
 * Handle structure for USARTx peripheral
 */
typedef struct
{
    USART_RegDef_t *pUSARTx;
    USART_Config_t   USART_Config;
}USART_Handle_t;
```

## USART Handle Structure

# Exercise :

1. Create USART driver header file and source file
2. Complete USART register definition structure and other macros (*peripheral base addresses, Device definition , clock en , clock di , etc*) in MCU specific header file
3. Also add USART register bit definition macros in MCU specific header file
4. Add USART Configuration structure and USART handle structure in USART header file

# Writing API prototypes

```
/*implement the below APIs by taking reference from I2C or SPI driver */

void USART_PeripheralControl(USART_RegDef_t *pUSARTx, uint8_t EnOrDi);

void USART_PeripheralControl(USART_RegDef_t *pUSARTx, uint8_t EnOrDi);

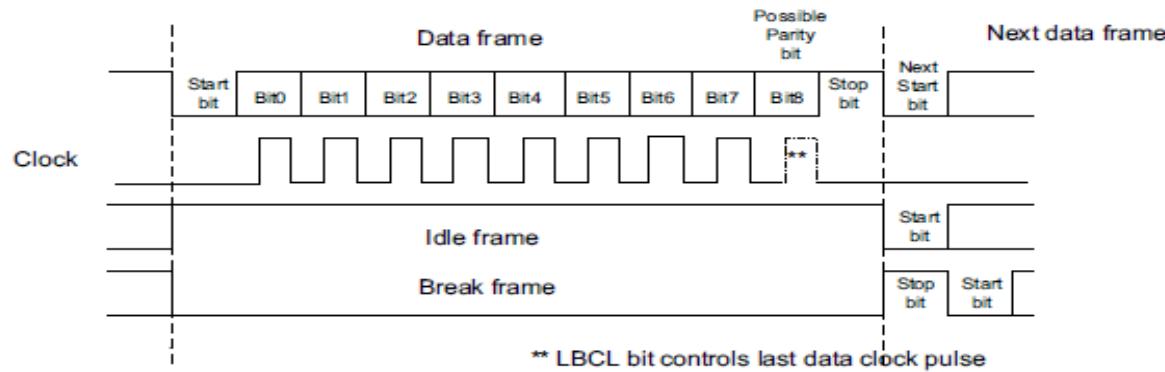
uint8_t USART_GetFlagStatus(USART_RegDef_t *pUSARTx, uint8_t StatusFlagName);
void USART_ClearFlag(USART_RegDef_t *pUSARTx, uint16_t StatusFlagName);

void USART_IRQHandlerConfig(uint8_t IRQNumber, uint8_t EnorDi);
void USART_IRQPriorityConfig(uint8_t IRQNumber, uint32_t IRQPriority);
```

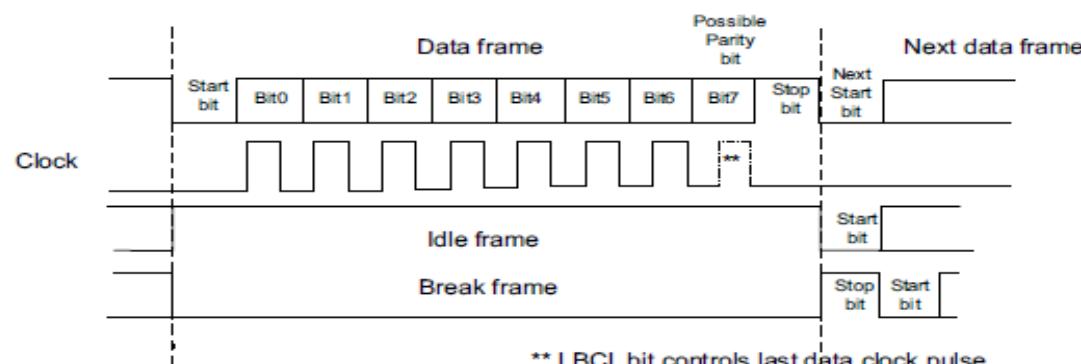
# Implementing **USART\_Init**

## Word length programming

9-bit word length (M bit is set), 1 Stop bit



8-bit word length (M bit is reset), 1 Stop bit

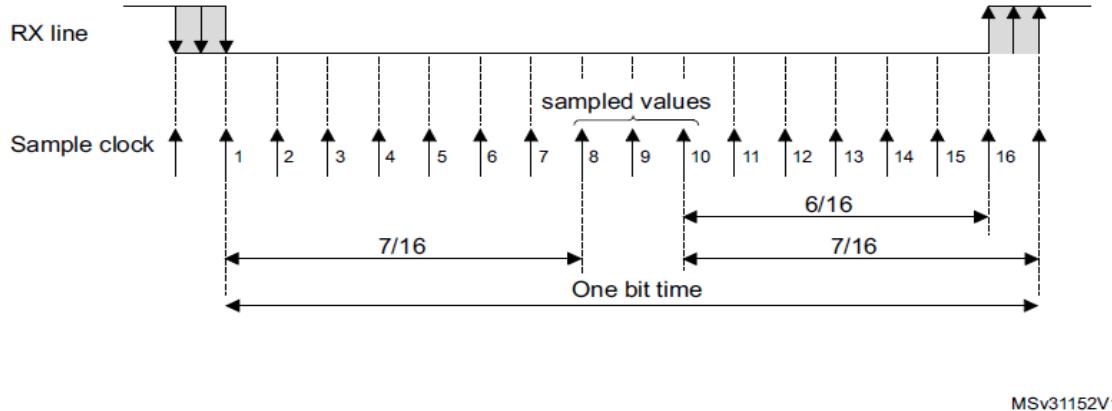


# USART: Oversampling

# USART: Oversampling

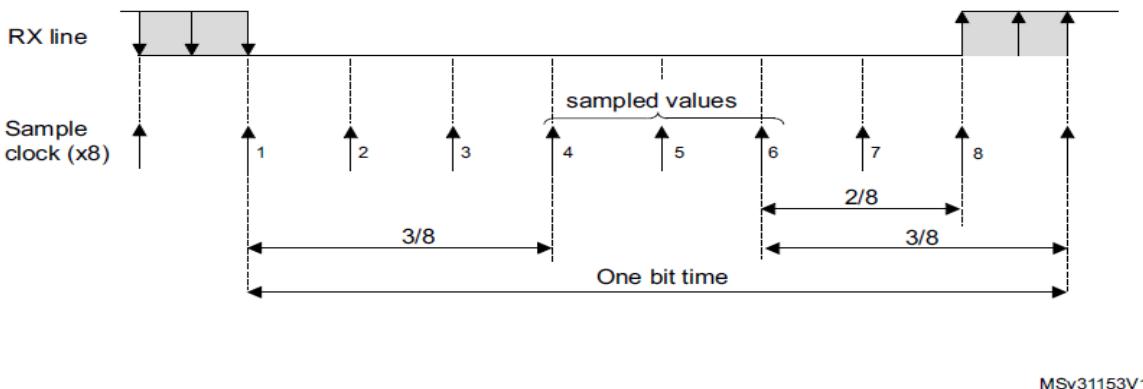
- ▶ The receiver implements different user-configurable oversampling techniques (except in synchronous mode) for data recovery by discriminating between valid incoming data and noise.
- ▶ The oversampling method can be selected by programming the OVER8 bit in the USART\_CR1 register and can be either 16 or 8 times the baud rate clock
- ▶ Configurable oversampling method by 16 or by 8 to give flexibility between speed and clock tolerance

## Data sampling when oversampling by 16



Receiver block sampling the Rx line during reception when oversampling by 16 is used

## Data sampling when oversampling by 8



Receiver block sampling the Rx line during reception when oversampling by 8 is used

# Sampled values Vs Noise

Noise detection from sampled data

Sampled value	NE status	Received bit value
000	0	0
001	1	0
010	1	0
011	1	1
100	1	0
101	1	1
110	1	1
111	0	1

# Noise Error

When noise is detected in a frame:

- ✓ The NF flag bit is set by hardware when noise is detected on a received frame
- ✓ The invalid data is transferred from the Shift register to the USART\_DR register
- ✓ The application may consider or discard the frame based on application logic

# Selecting the proper oversampling method

The receiver implements different user-configurable oversampling techniques (except in synchronous mode) for data recovery by discriminating between valid incoming data and noise.

**If you select oversampling by 8 (OVER8=1) then you can achieve max baudrate up to  $F_{PCLK}/8$ , but In this case the maximum receiver tolerance to clock deviation is reduced**

**If you select oversampling by 16 (OVER8=0) then you can achieve max baudrate up to  $F_{PCLK}/16$ , In this case the maximum receiver tolerance to clock deviation is increased.**

# UART Baudrate Calculation

$$\text{Tx/Rx baud} = \frac{f_{CK}}{8 \times \text{USARTDIV}}$$

if OVER8 = 1

*Peripheral Clock*

↓  
Divide factor to generate different baud rates

$$\text{Tx/Rx baud} = \frac{f_{CK}}{16 \times \text{USARTDIV}}$$

if OVER8 = 0

The baud rate for the receiver and transmitter (Rx and Tx) are both set to the same value as programmed in the Mantissa and Fraction values of USARTDIV.

$$\text{Tx/Rx baud} = \frac{f_{CK}}{8 \times (2 - \text{OVER8}) \times \text{USARTDIV}}$$

OVER8=1 , if Oversampling by 8 is used  
OVER8=0, if Oversampling by 16 is used

$$\text{Tx/Rx baud} = \frac{f_{CK}}{8 \times (2 - \text{OVER8}) \times \text{USARTDIV}}$$

$f_{CK}$  = 16MHz

(USART Peripheral Clock)

Tx/Rx baud = 9600 bps

(Desired Baudrate)

OVER8=0

(Oversampling by 16)

$$\begin{aligned}\text{USARTDIV} &= 16M / (8 * 2 * 9600) \\ &= 104.17\end{aligned}$$

# Programming USART Baudrate register (USART\_BRR)

$$\text{USARTDIV} = 104.1875$$

(For baudrate 9600bps with  $F_{CK} = 16\text{MHz}$  and  $\text{OVER8}=0$ )

Now this value we have to convert in to hex and then program the USART\_BRR register to achieve desired baud rate.

**USART uses a fractional baud rate generator - with a 12-bit mantissa and 4-bit fraction**



### 30.6.3 Baud rate register (USART\_BRR)

Note: *The baud counters stop counting if the TE or RE bits are disabled respectively.*

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value

Bits 15:4 **DIV\_Mantissa[11:0]**: mantissa of USARTDIV

These 12 bits define the mantissa of the USART Divider (USARTDIV)

Bits 3:0 **DIV\_Fraction[3:0]**: fraction of USARTDIV

These 4 bits define the fraction of the USART Divider (USARTDIV). When OVER8=1, the DIV\_Fraction3 bit is not considered and must be kept cleared.

# USARTDIV = 104.1875

(For baudrate 9600bps with Fck = 16MHz and OVER8=0)

$$\text{DIV\_Fraction} = 0.1875 \times 16 = 3$$

$$\text{Div\_Mantissa} = 104 = 0x68$$

# USARTDIV = 0x683

(Program this value into USART\_BRR register to generate baudrate of 9600bps)

Now lets take another example of generating baud rate of 115200 bps.

For ,  $F_{CK} = 16\text{MHz}$  and  $\text{OVER8}=1$

# USARTDIV = 17.361

(For baudrate 115200bps with Fck = 16MHz and OVER8=1)

$$\text{DIV\_Fraction} = 0.361 \quad \times \quad 8 \quad = \quad 2.88 = 3$$

Div\_Mantissa = 17

# USARTDIV = 0x113

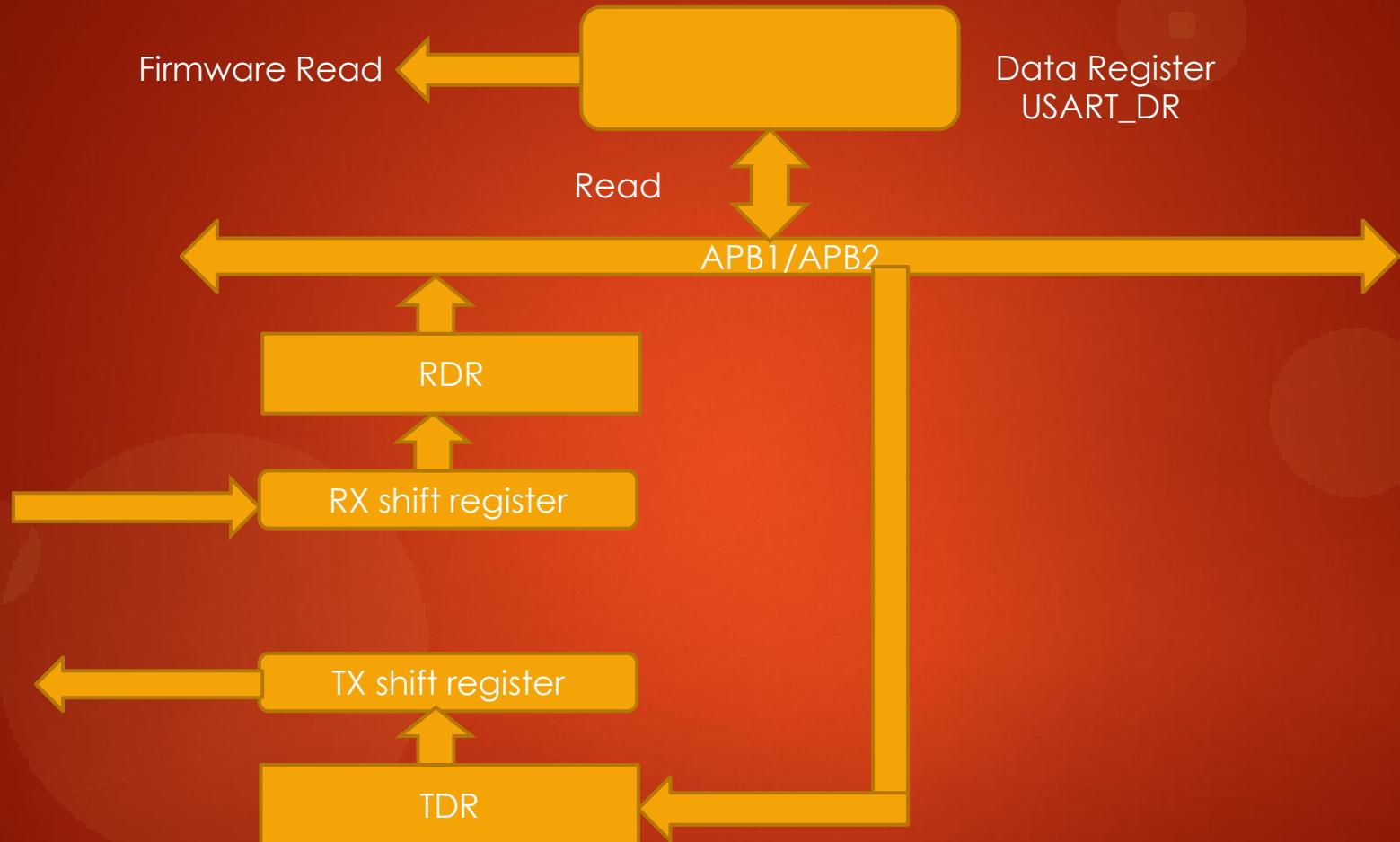
(Program this value into USART\_BRR register to generate baudrate of 115200bps)



Copyright © 2019 Bharati Software

# Overrun Error

- ▶ An overrun error occurs when a character is received when RXNE has not been reset. Data can not be transferred from the shift register to the RDR register until the RXNE bit is cleared.
- ▶ When an overrun error occurs:
  - ▶ The ORE bit is set.
  - ▶ The RDR content will not be lost. The previous data is available when a read to USART\_DR is performed
  - ▶ The shift register will be overwritten. After that point, any data received during overrun is lost.
  - ▶ An interrupt will be generated if enabled.



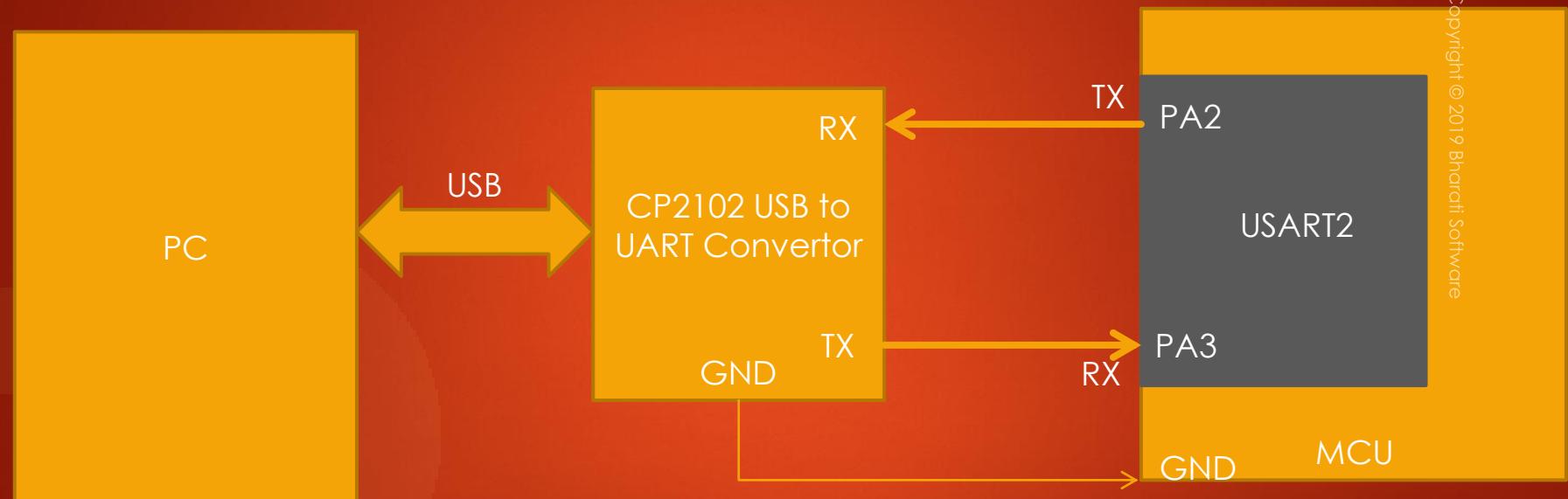
# USB to UART Converter cable



Copyright © 2019 Bharatiti Software

# USB to UART Converter module



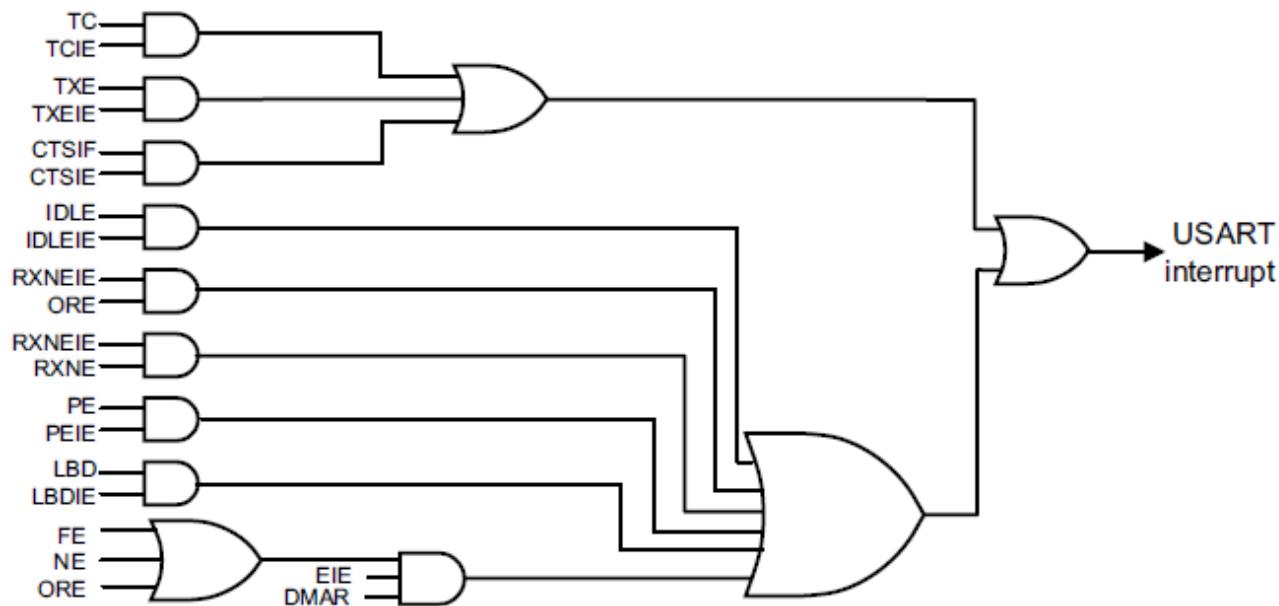


# USART interrupts

Table 147. USART interrupt requests

Interrupt event	Event flag	Enable control bit
Transmit Data Register Empty	TXE	TXEIE
CTS flag	CTS	CTSIE
Transmission Complete	TC	TCIE
Received Data Ready to be Read	RXNE	RXNEIE
Overrun Error Detected	ORE	
Idle Line Detected	IDLE	IDLEIE
Parity Error	PE	PEIE
Break Flag	LBD	LBDIE
Noise Flag, Overrun error and Framing Error in multibuffer communication	NF or ORE or FE	EIE

Figure 320. USART interrupt mapping diagram



MSv42089V1

# Exercise

Write a program to send some message over UART from STM32 board to Arduino board. The Arduino board will display the message (on Arduino serial monitor) sent from the ST board.

Baudrate : 115200 bps

Frame format : 1 stop bits, 8 bits , no parity

# Exercise

Write a program for stm32 board which transmits different messages to the Arduino board over UART communication.

For every message STM32 board sends , arduino code will change the case of alphabets(lower case to upper case and vice versa) and sends message back to the stm32 board .

The stm32 board should capture the reply from the arduino board and display using semi hosting.



Copyright © 2019 Bharati Software



Copyright © 2019 Bharati Software

# Arduino Sketch

## **001UARTRxString.ino**

Download the Arduino sketch into Arduino board