PART-1

Mastering Microcontroller with Embedded Driver Development

FastBit Embedded Brain Academy
Check all online courses at
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
Email : contact@fastbitlab.com

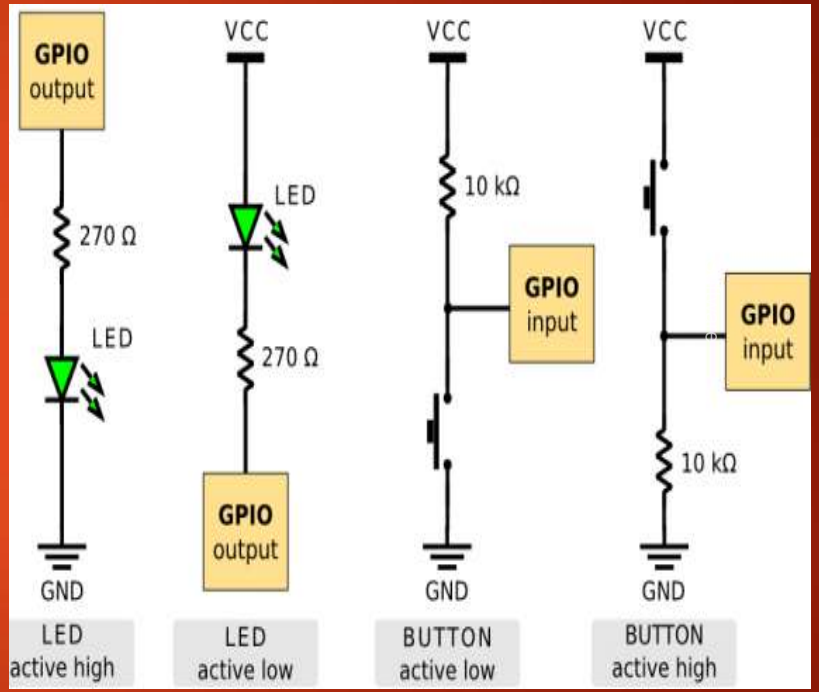
GPIO Must Know Concepts !

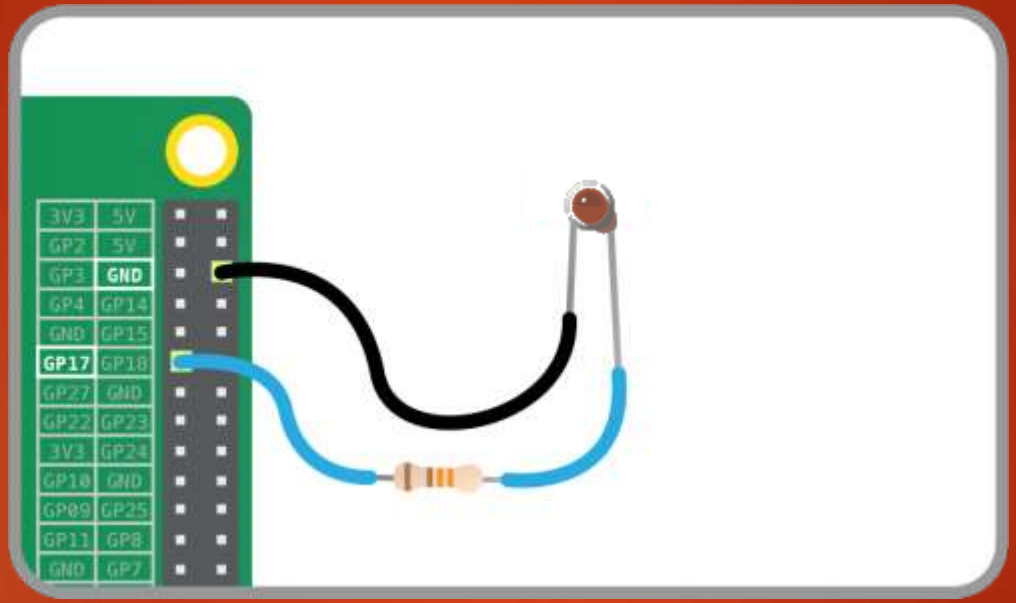
Section Takeaways !

- 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





GPIOs typically used for

Reading digital signal

Generating triggers for external components

Issuing interrupts

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***

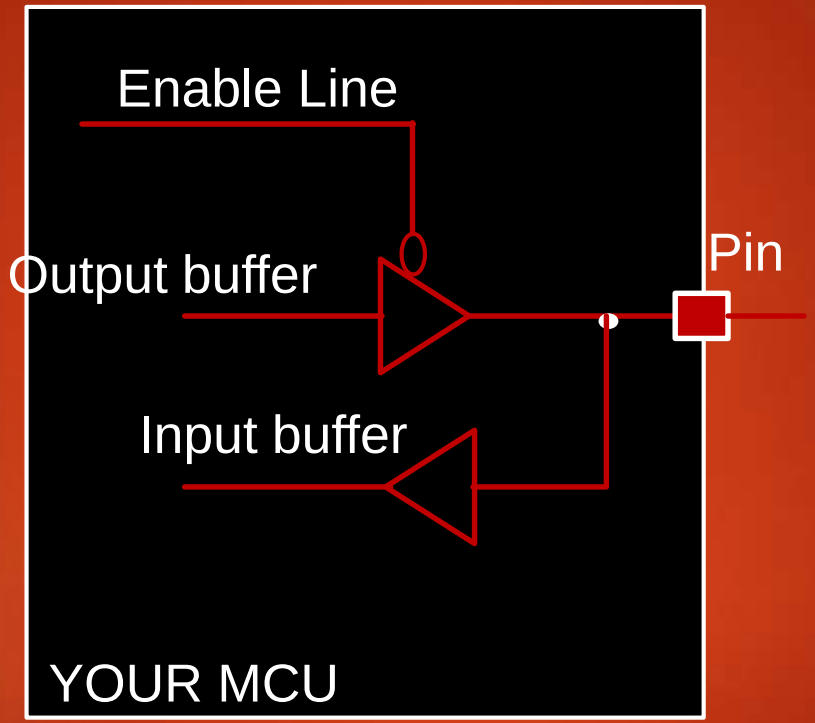
Each port have
8 i/o pins

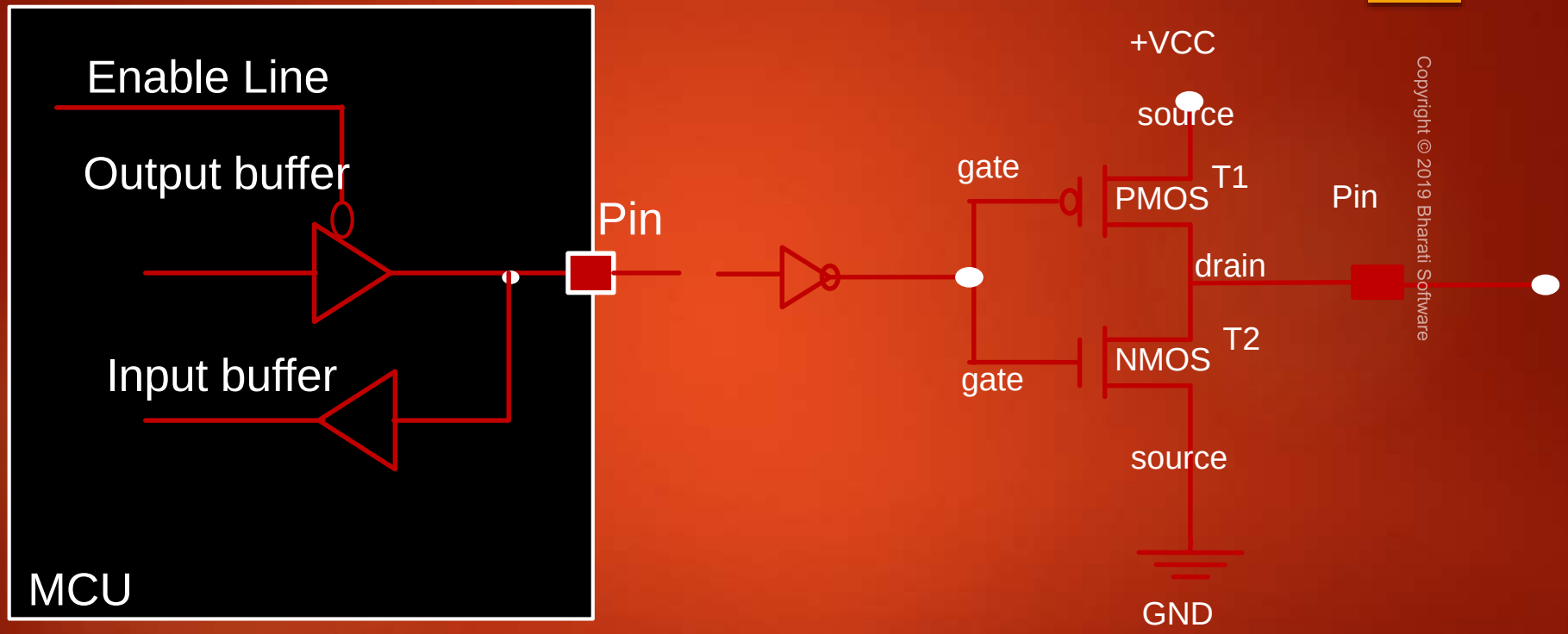
Each port have
32 i/o pins

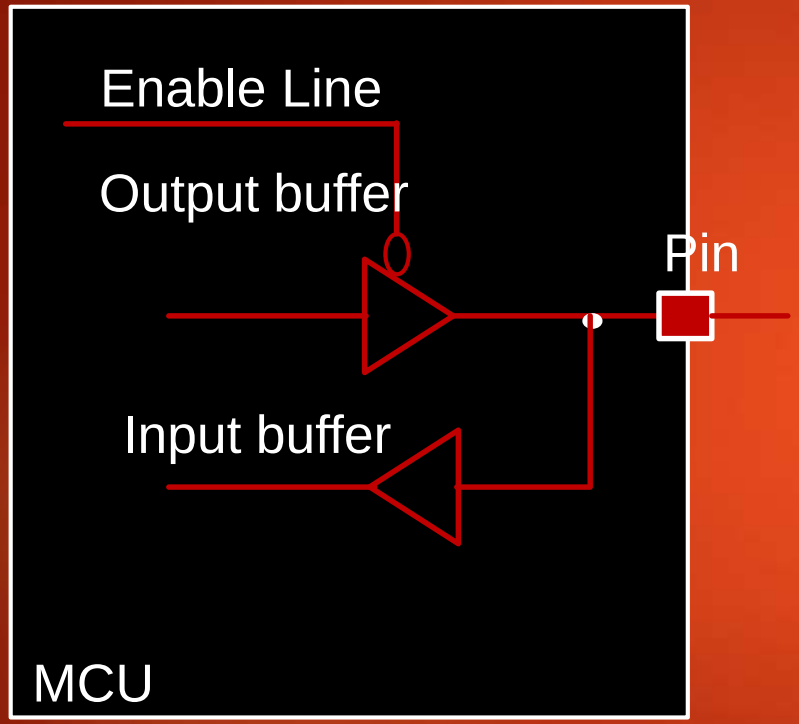
Each port have
16 i/o pins

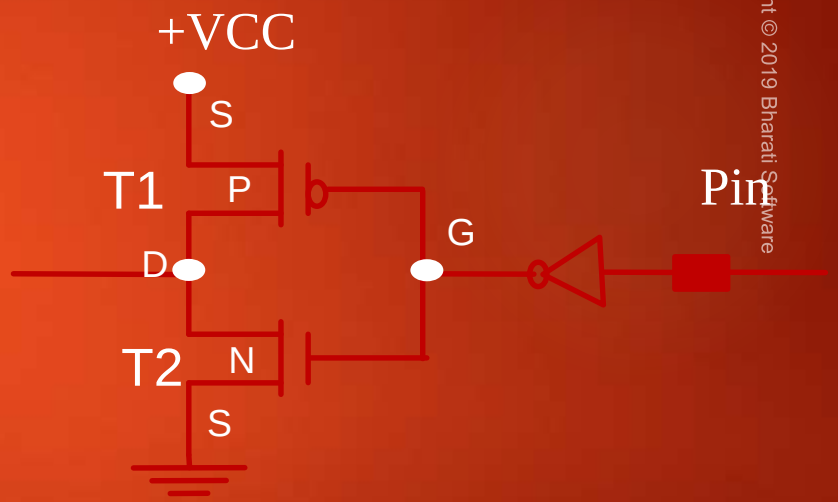
GPIO Pin Behind the Scene

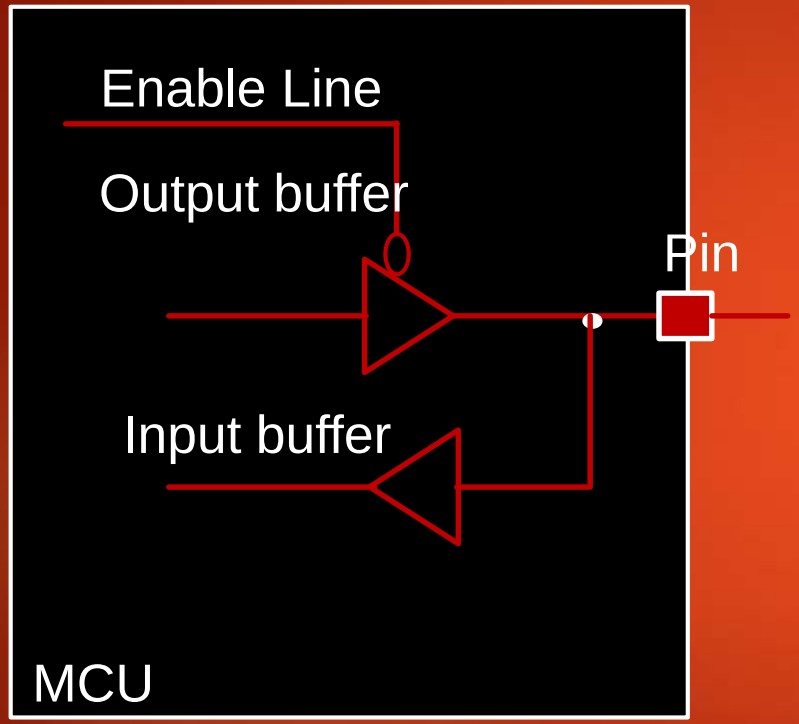
How does a GPIO pin is actually implemented inside the MCU?

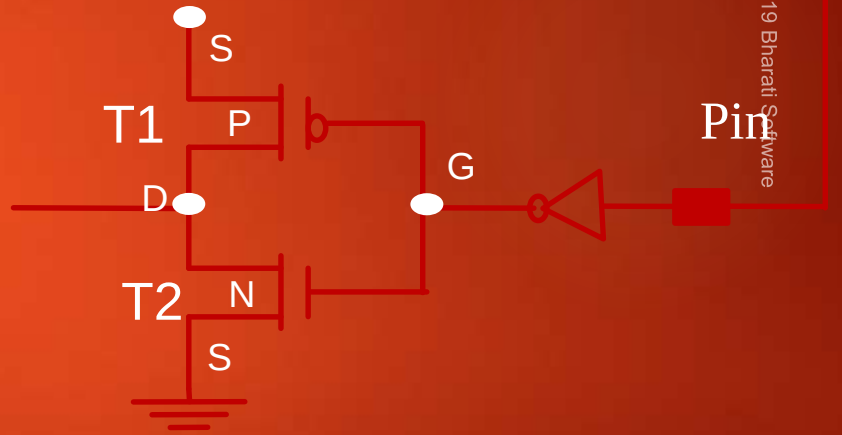


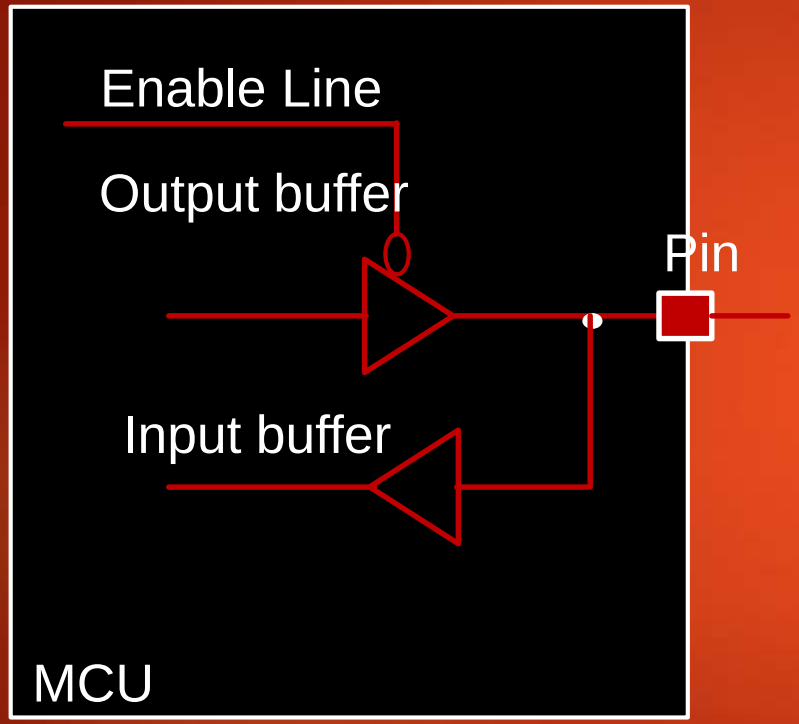


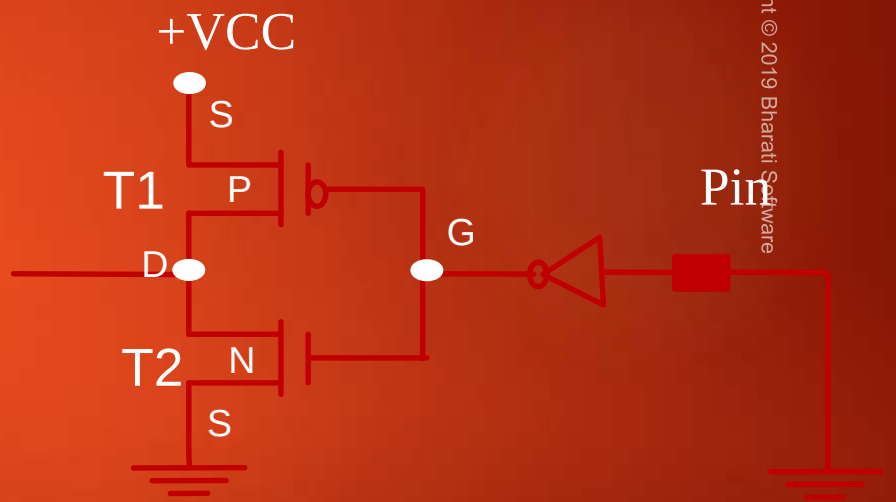










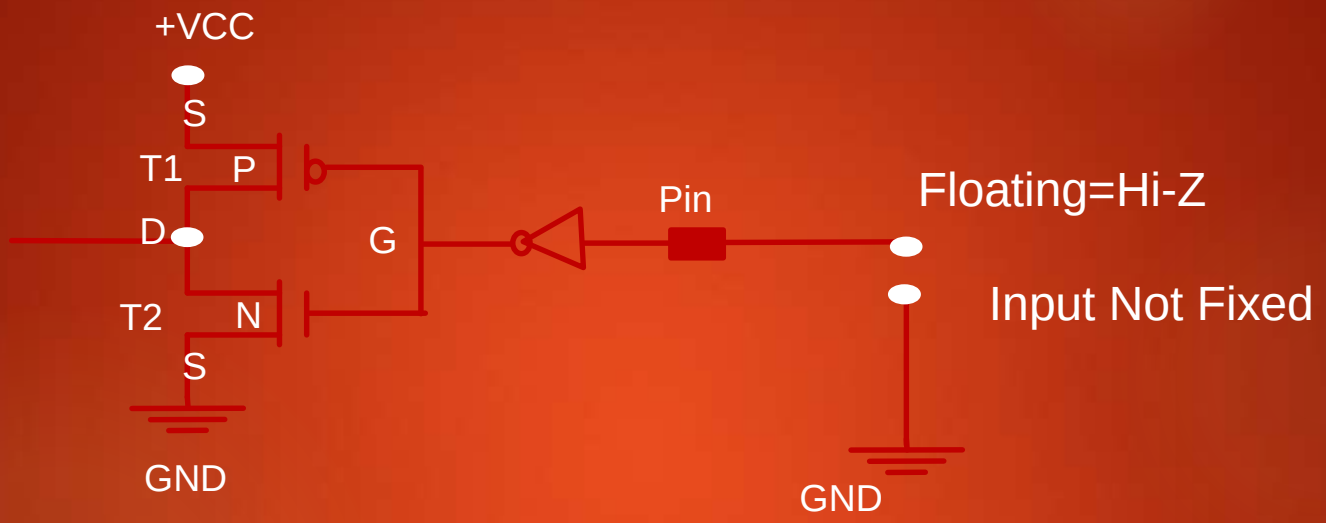


GPIO INPUT MODE

High Impedance(HI-Z) State

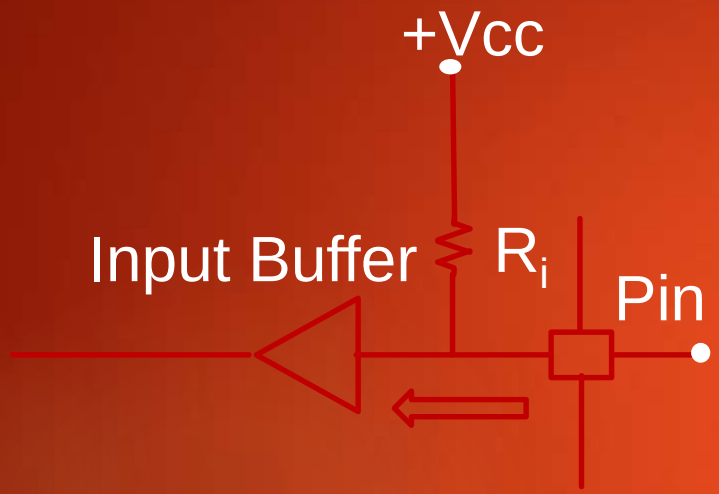
Let’s understand, what exactly is this High Impedance state, that people talk about when the pin is in input mode ??

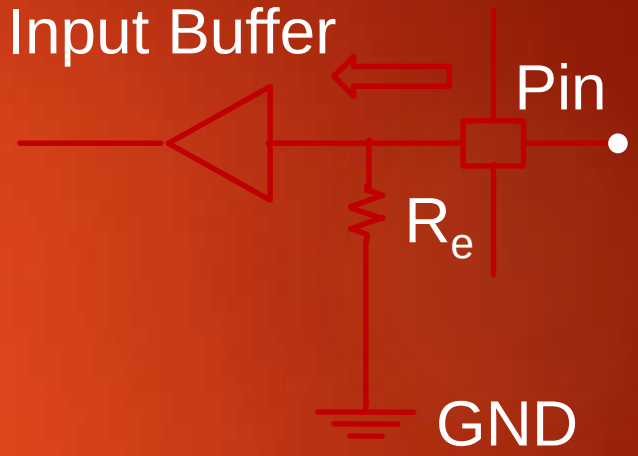
High impedance is also called as HI-Z state



GPIO INPUT MODE

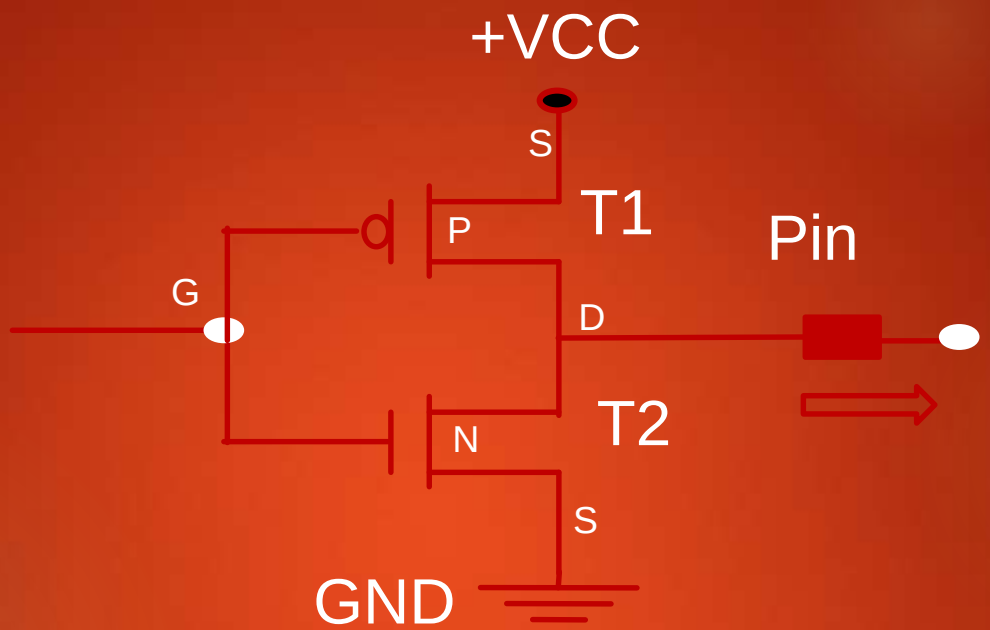
Pull-Up/Pull-Down State

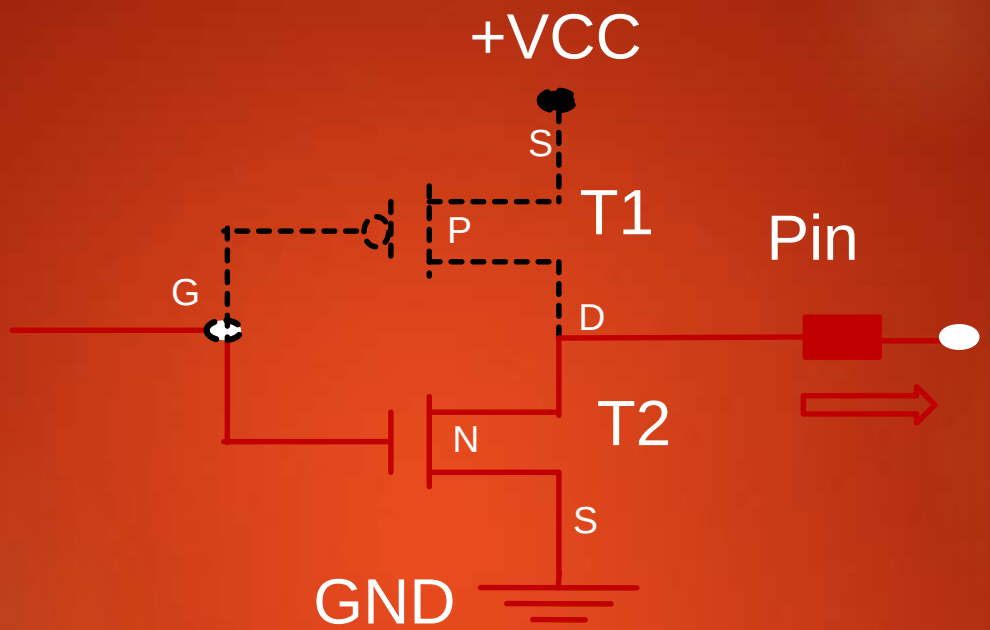




GPIO OUTPUT MODE

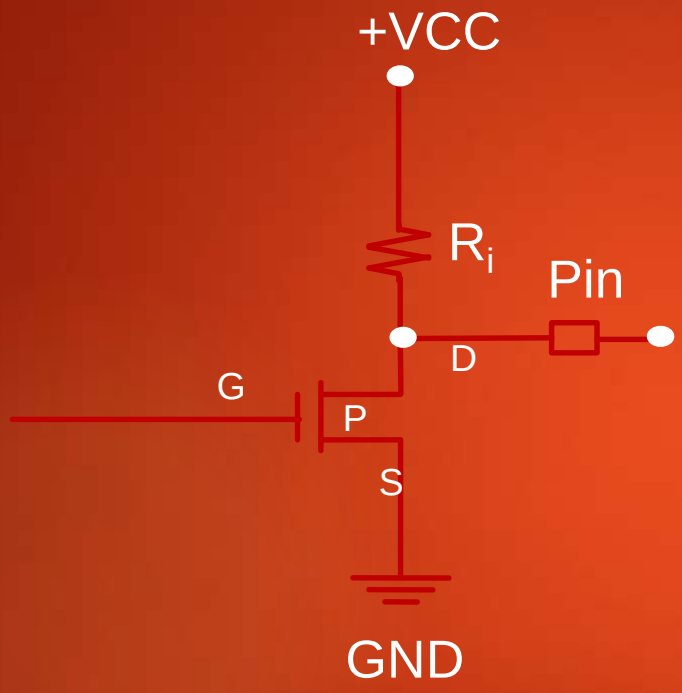
Open-drain State



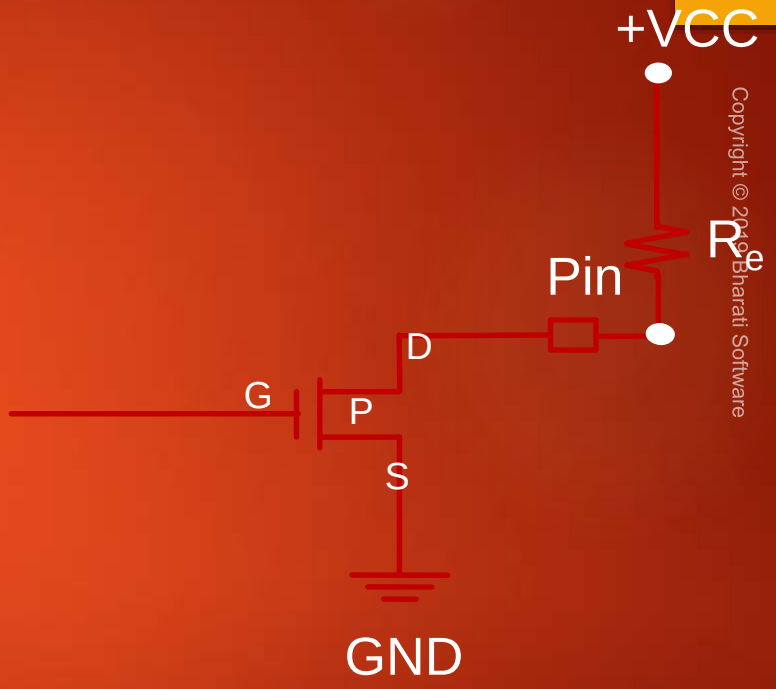


GPIO OUTPUT MODE

Open drain with Pull-Up



Open drain with internal pull up

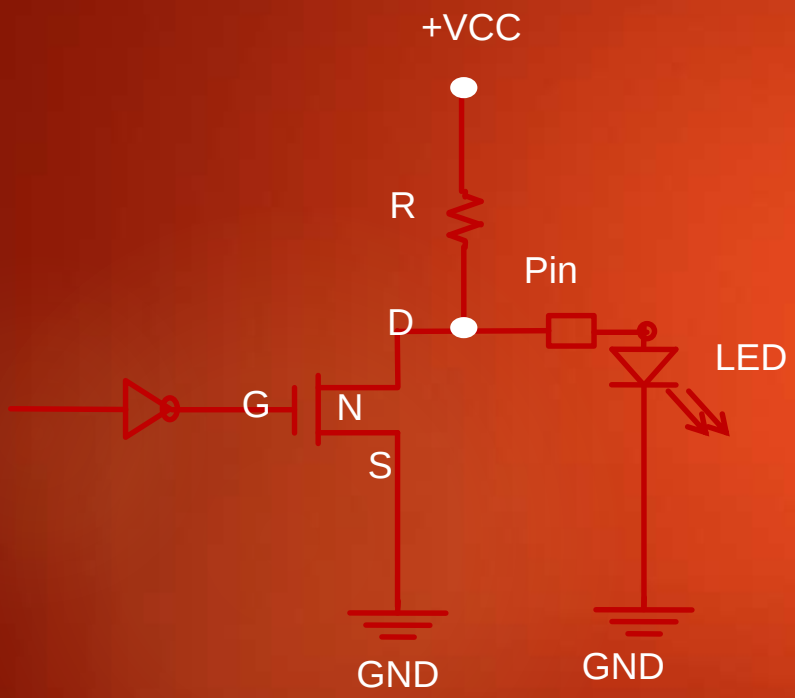


Open drain with External pull up

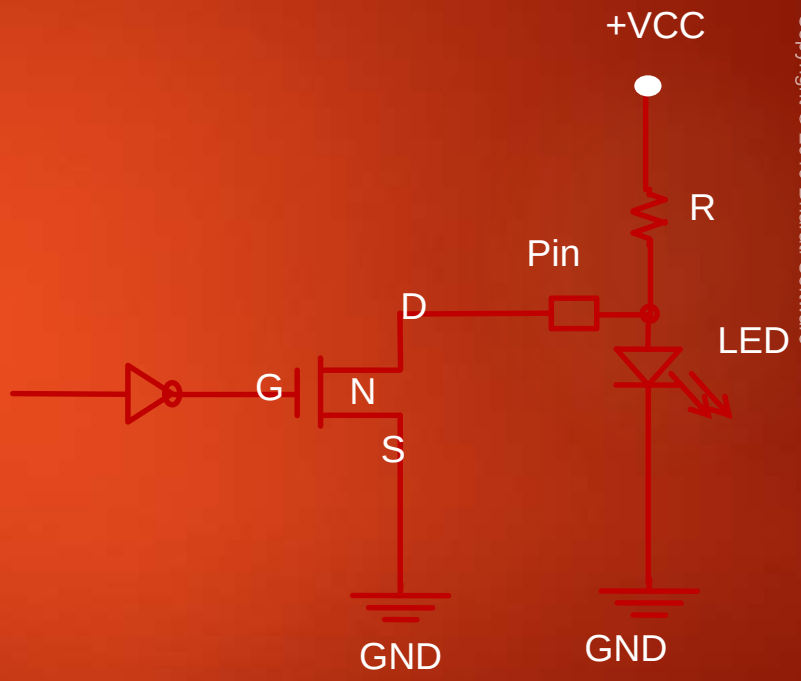
Practical Usage -1

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

Driving LED's



Using Internal Pull-up

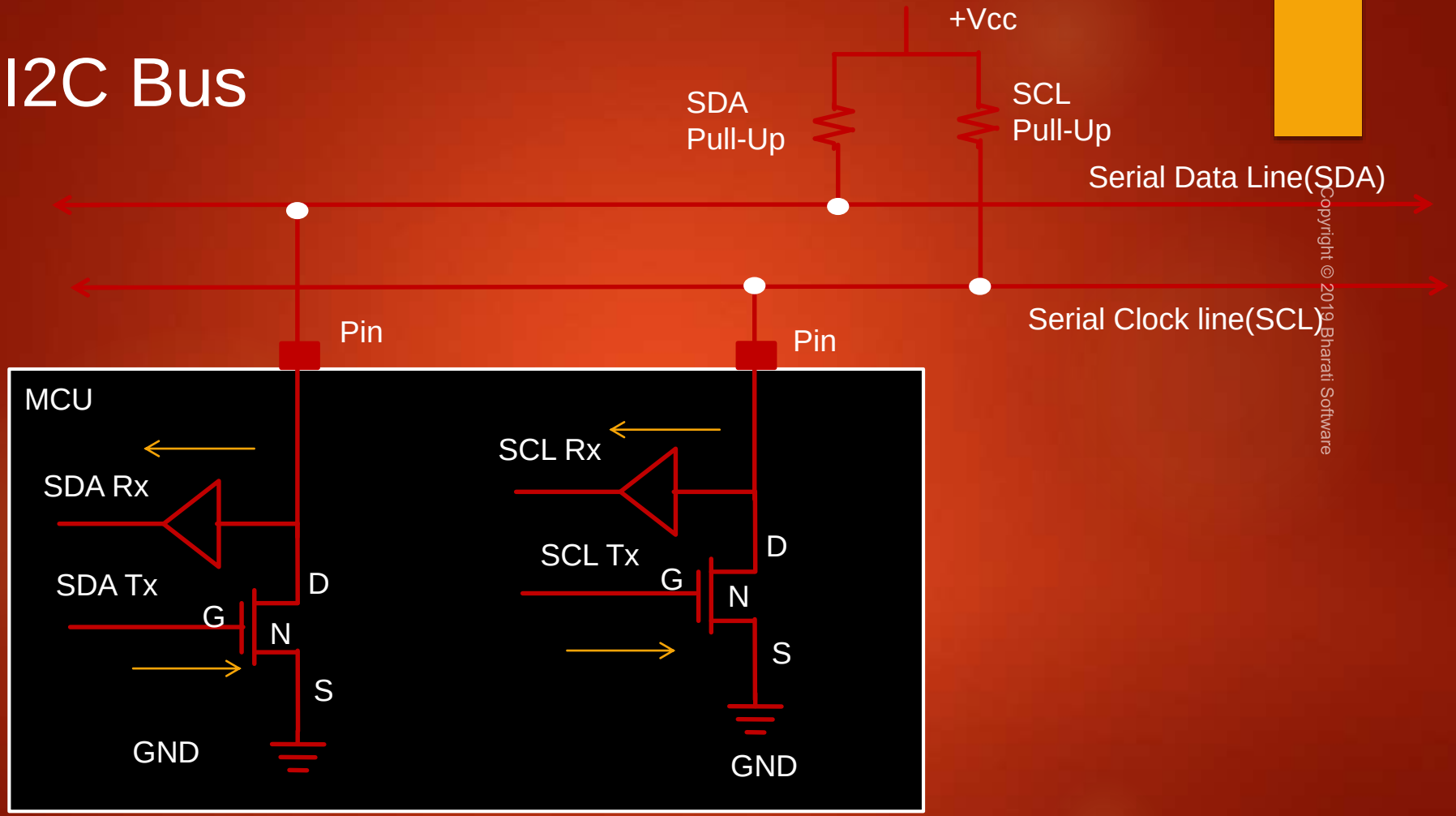


Using External Pull-up

Practical Usage -2

Driving I2C bus

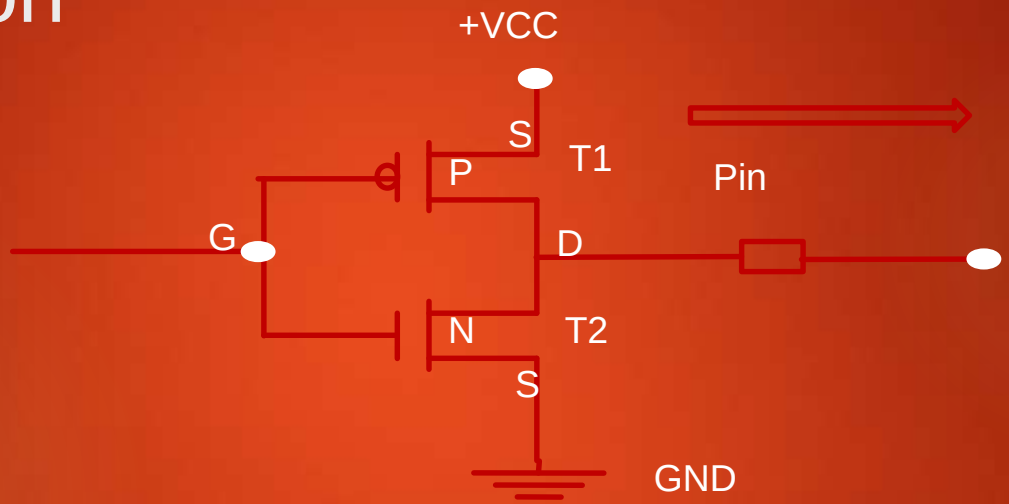
I2C Bus

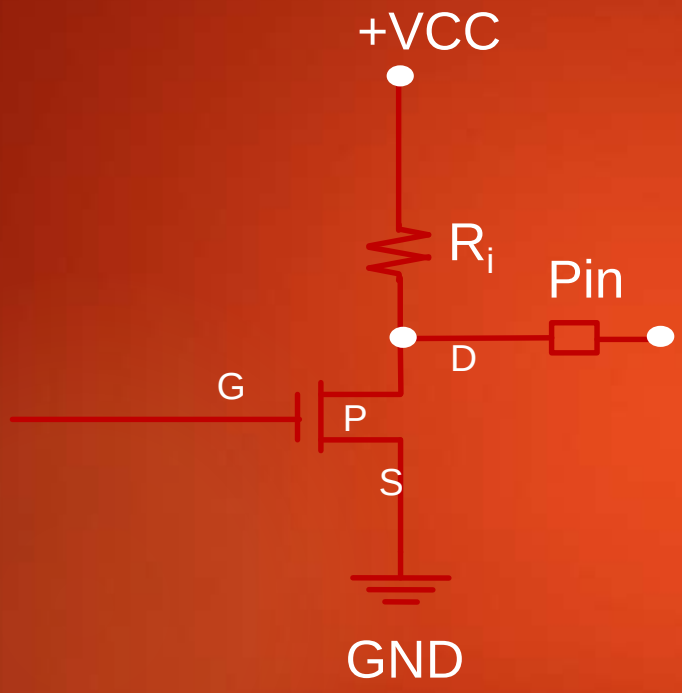


GPIO OUTPUT MODE

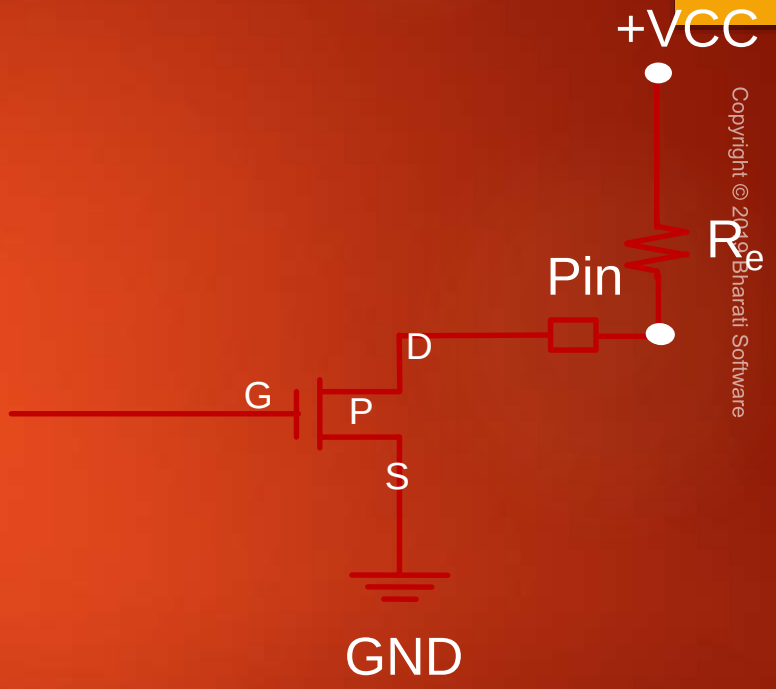
Push-Pull State

Output Mode with Push-Pull Configuration



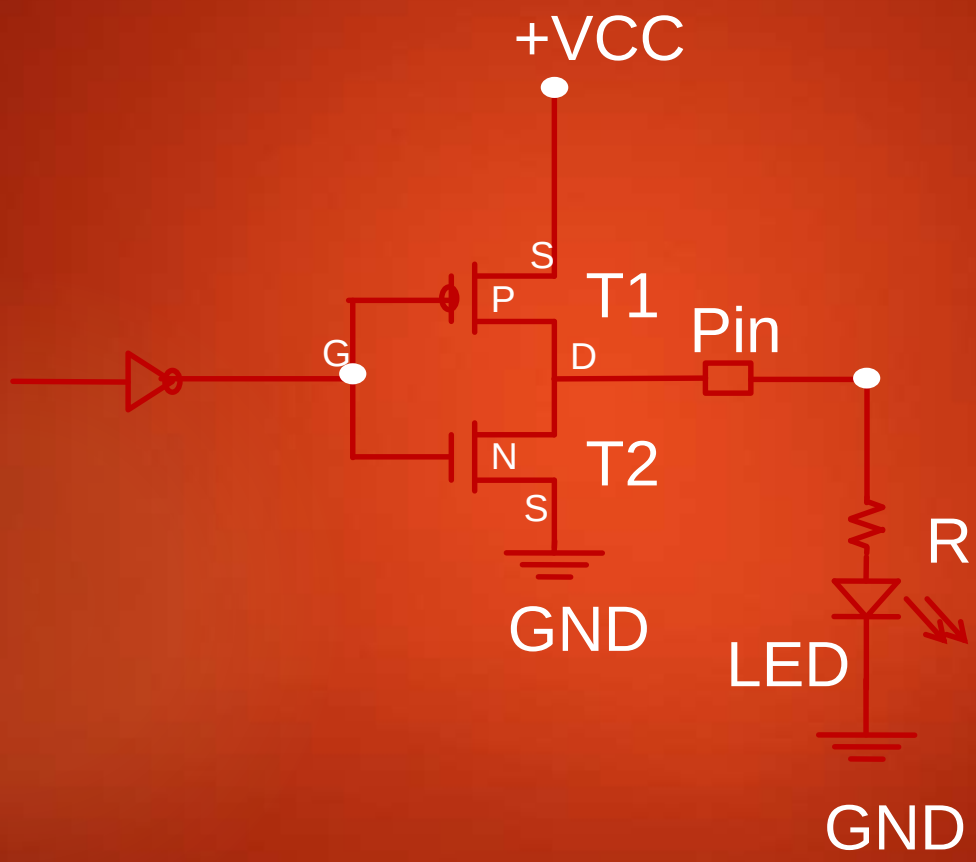


Open drain with internal pull up

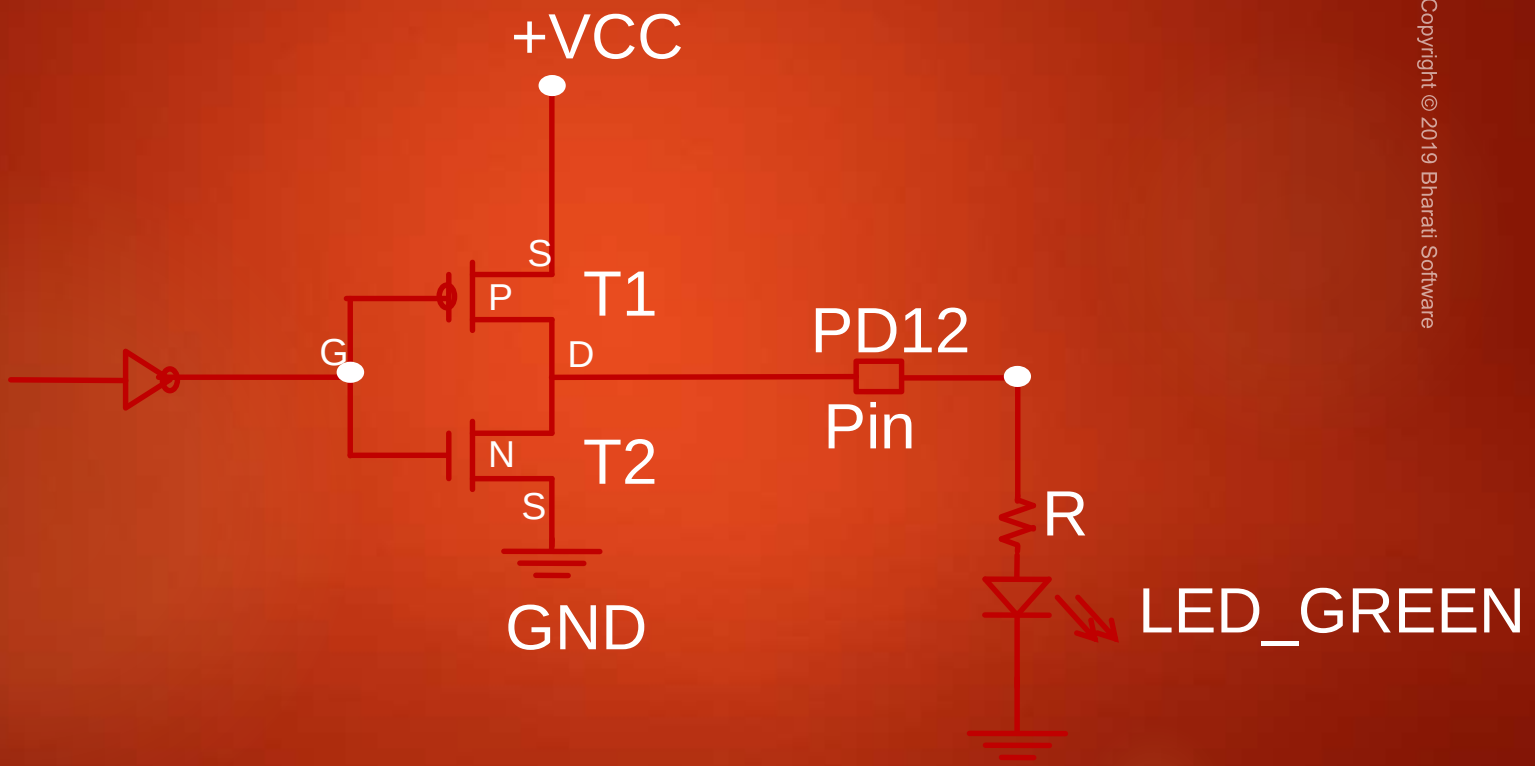


Open drain with External pull up

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

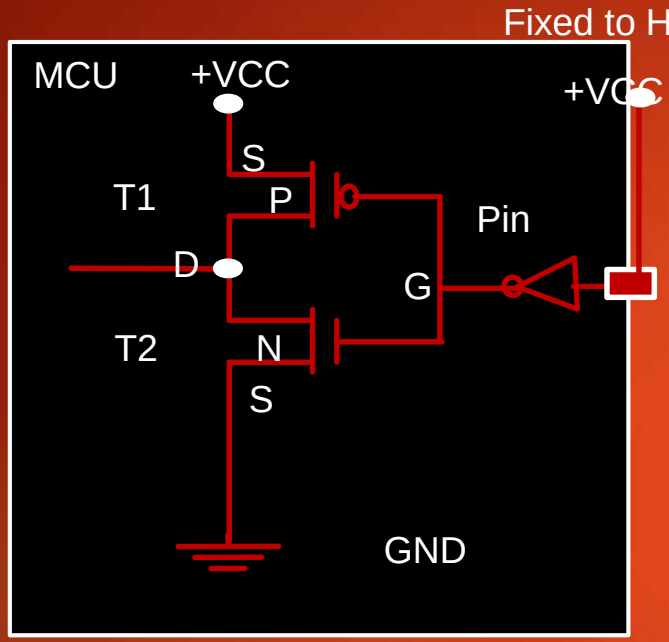


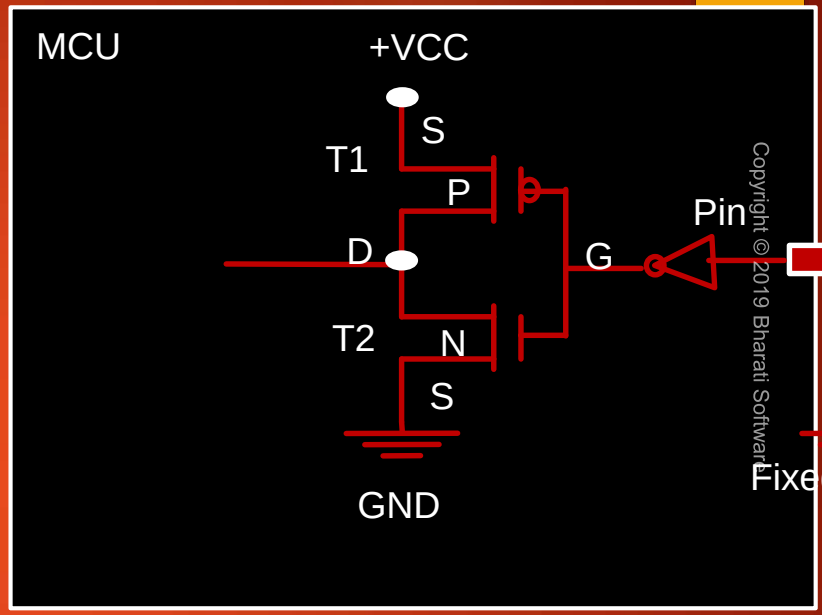
Discovery Board LED Connection

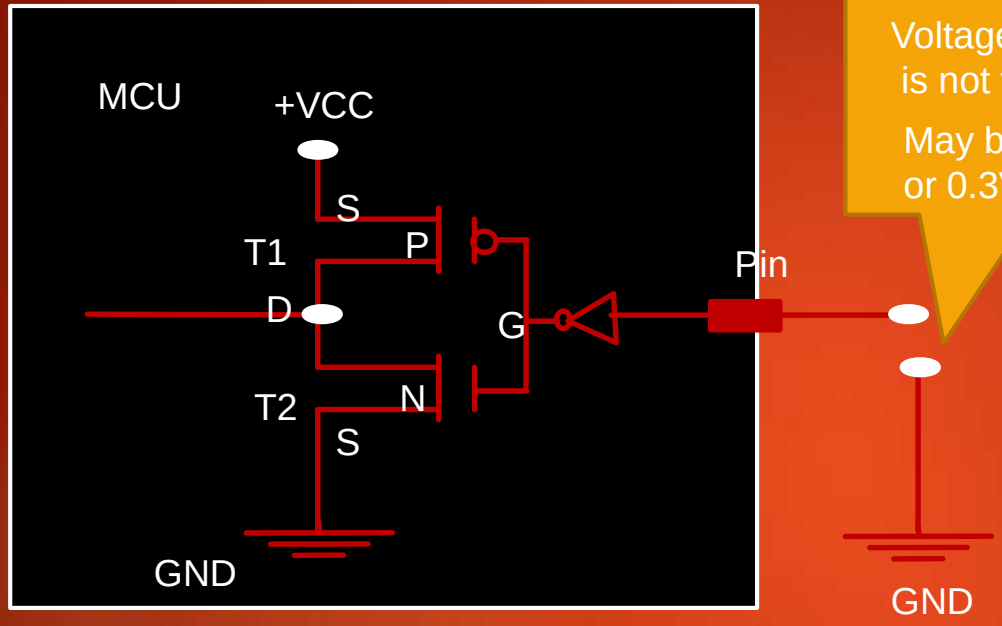


Optimizing IO Power Consumption

Leakage Mechanism By Input Pin Floating


Fixed to High
MCU
+VCC
+VCC
S
T1
P
Pin
D
G
T2
N
S
GND

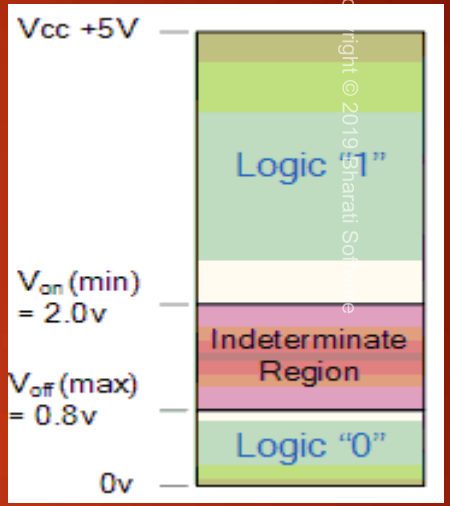

MCU
+VCC
S
T1
P
Pin
D
G
T2
N
S
GND
GND
Fixed to Low
Copyright © 2019 Bharati Software



Voltage of input is not fixed

May be 0.5Vcc or 0.3Vcc

Input Logic Level



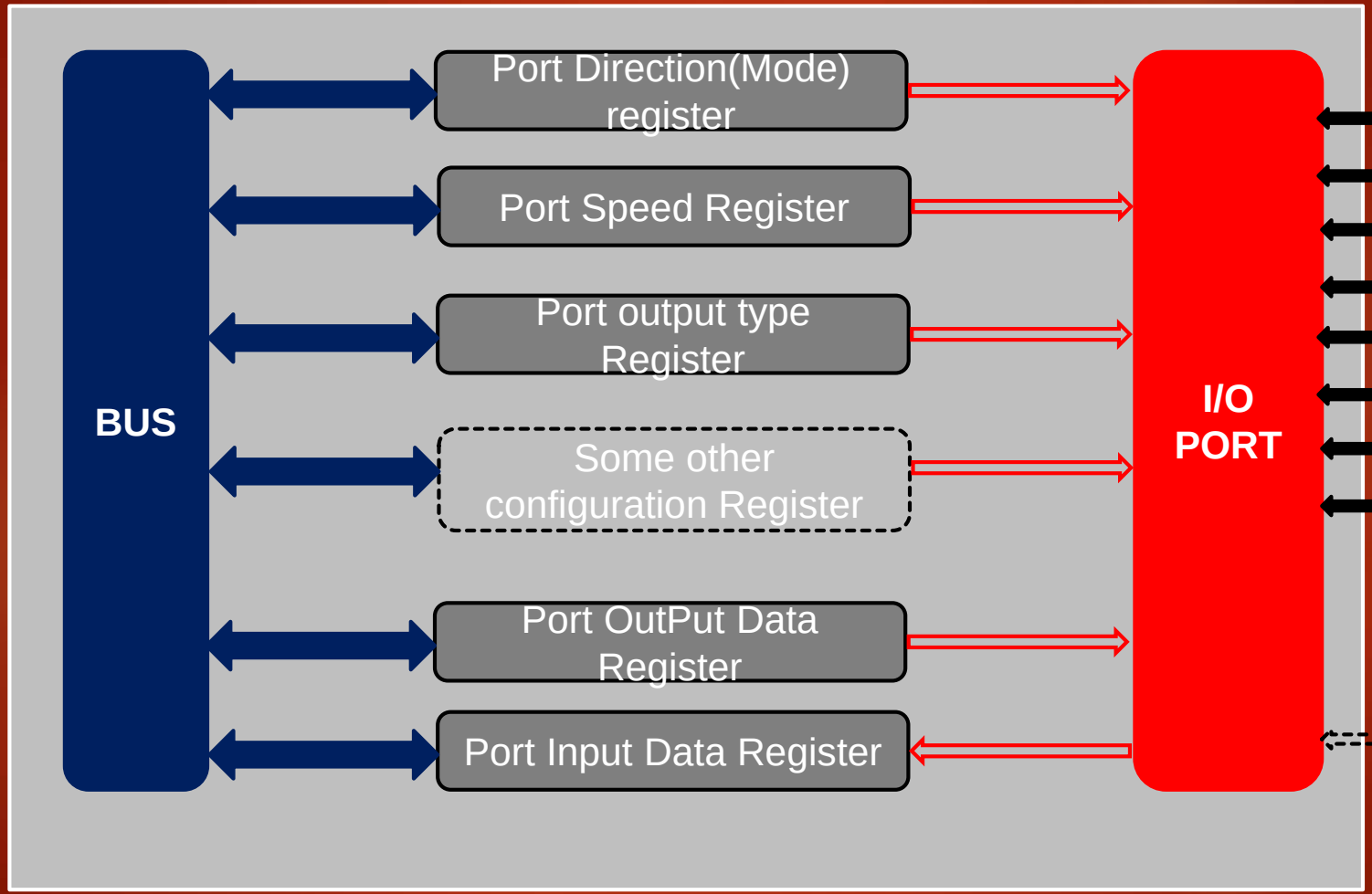
Valve Closed

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

- GPIOA
- GPIOB
- GPIOC
- GPIOD
- GPIOE

GPIO Driver development

GPIO Port MODE Register

32 bit reg.

GPIOA_MODER

32 bit reg.

GPIOB_MODER

32 bit reg.

GPIOC_MODER

32 bit reg.

GPIOD_MODER

32 bit reg.

GPIOE_MODER

32 bit reg.

GPIOF_MODER

32 bit reg.

GPIOG_MODER

32 bit reg.

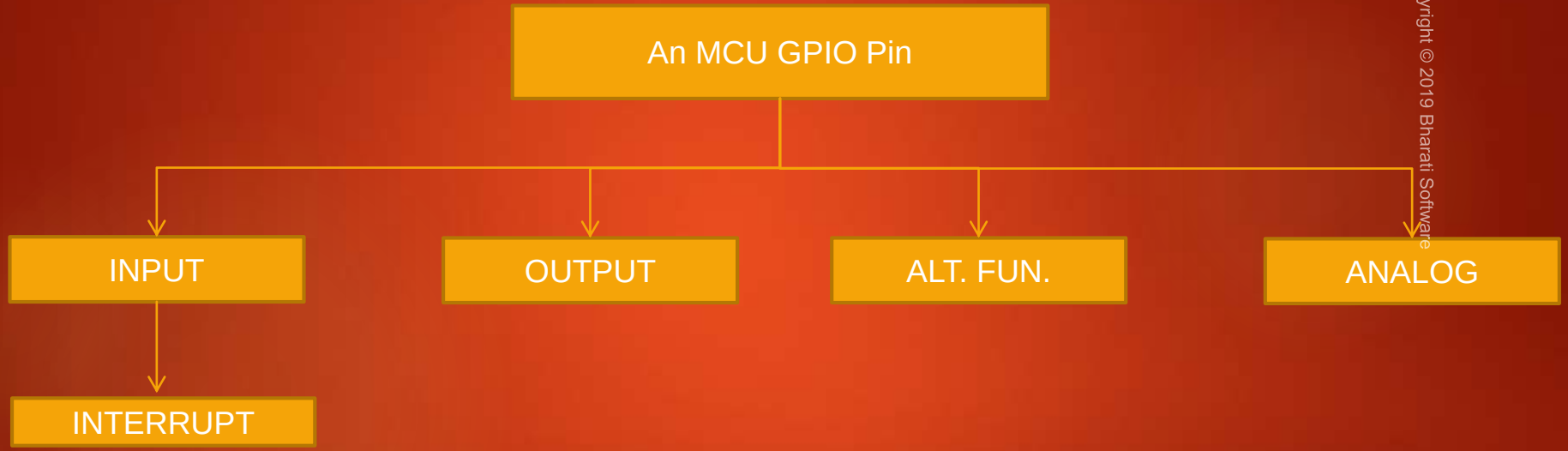
GPIOH_MODER

32 bit reg.

GPIOI_MODER

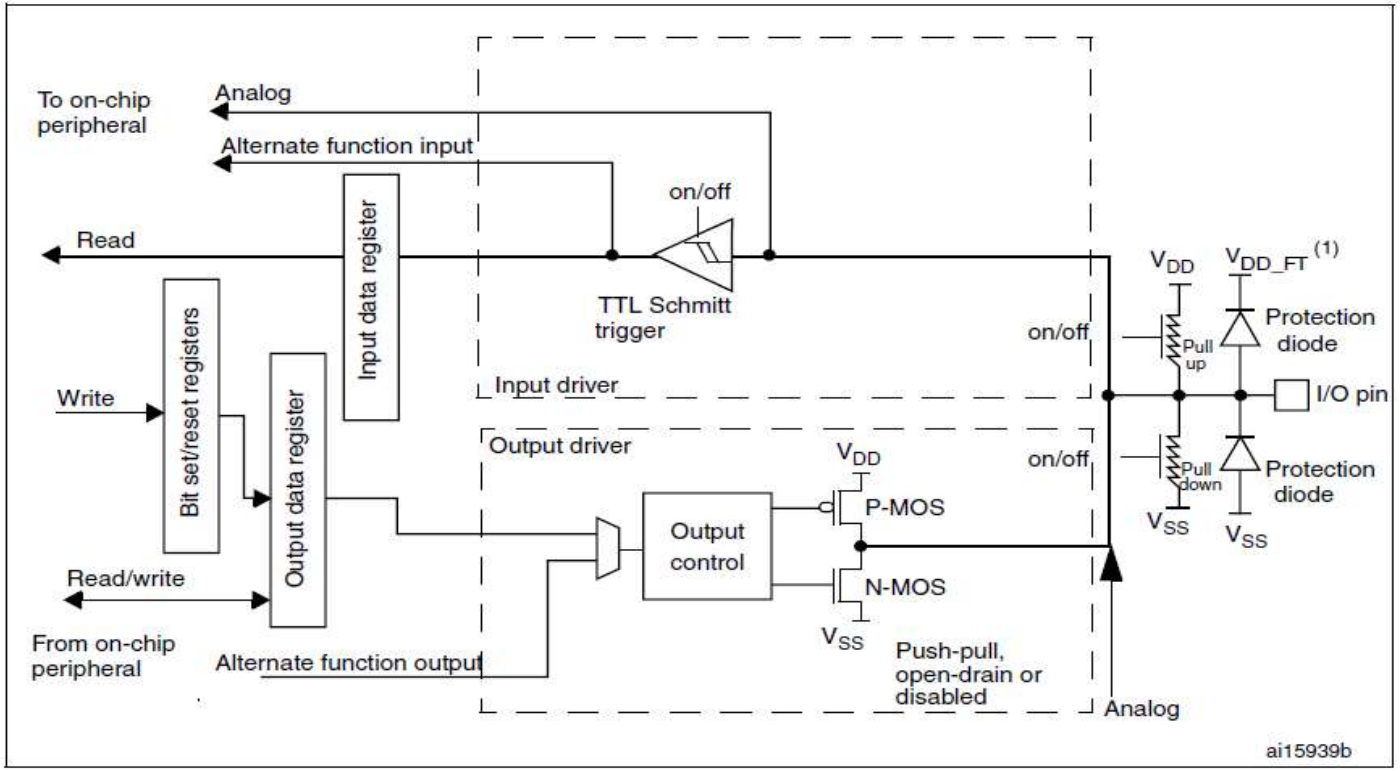
Before using any GPIO, you must decide its MODE. Whether you want to use it as an Input, Output or for any analog purposes

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

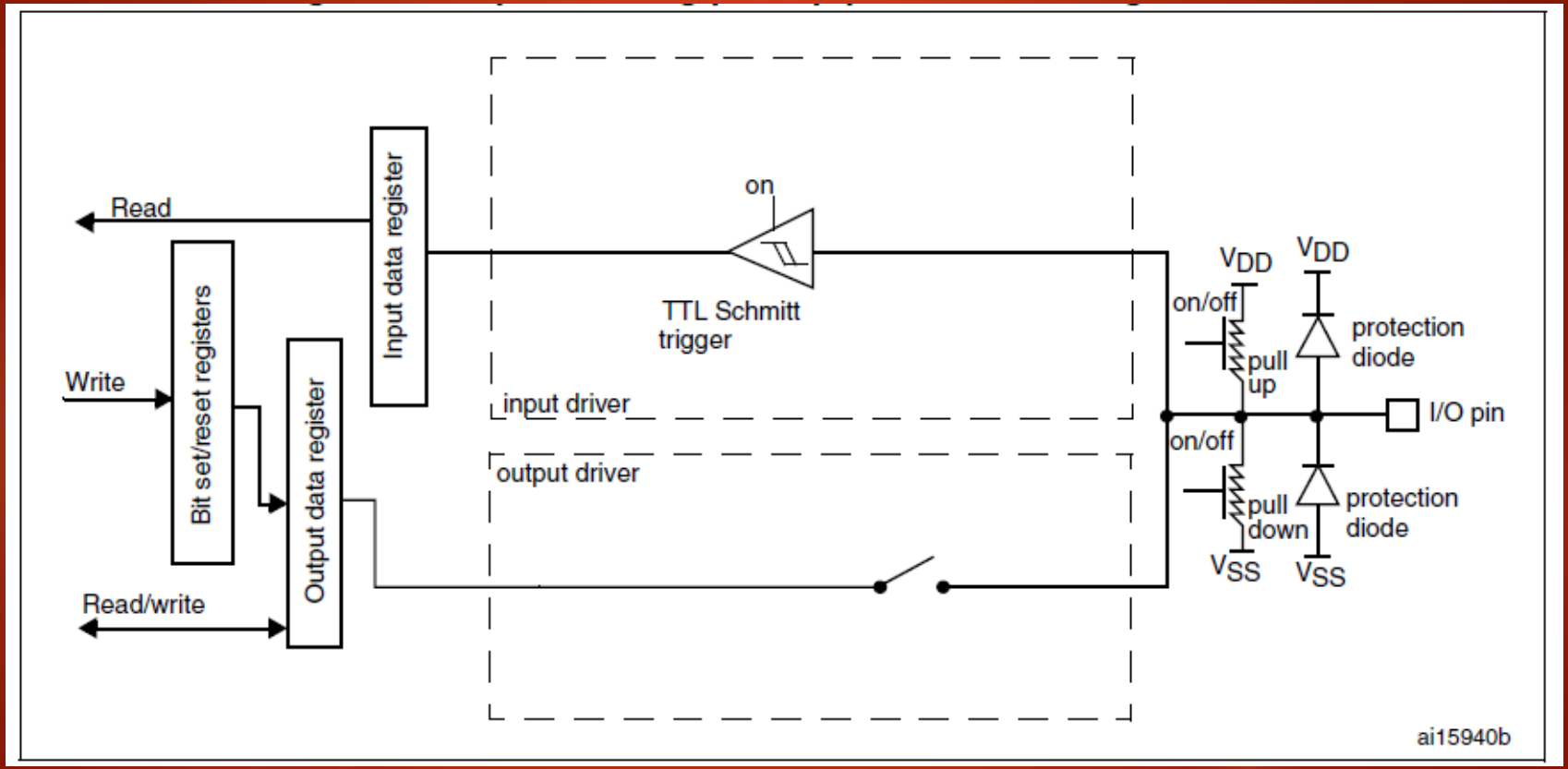
Input configuration

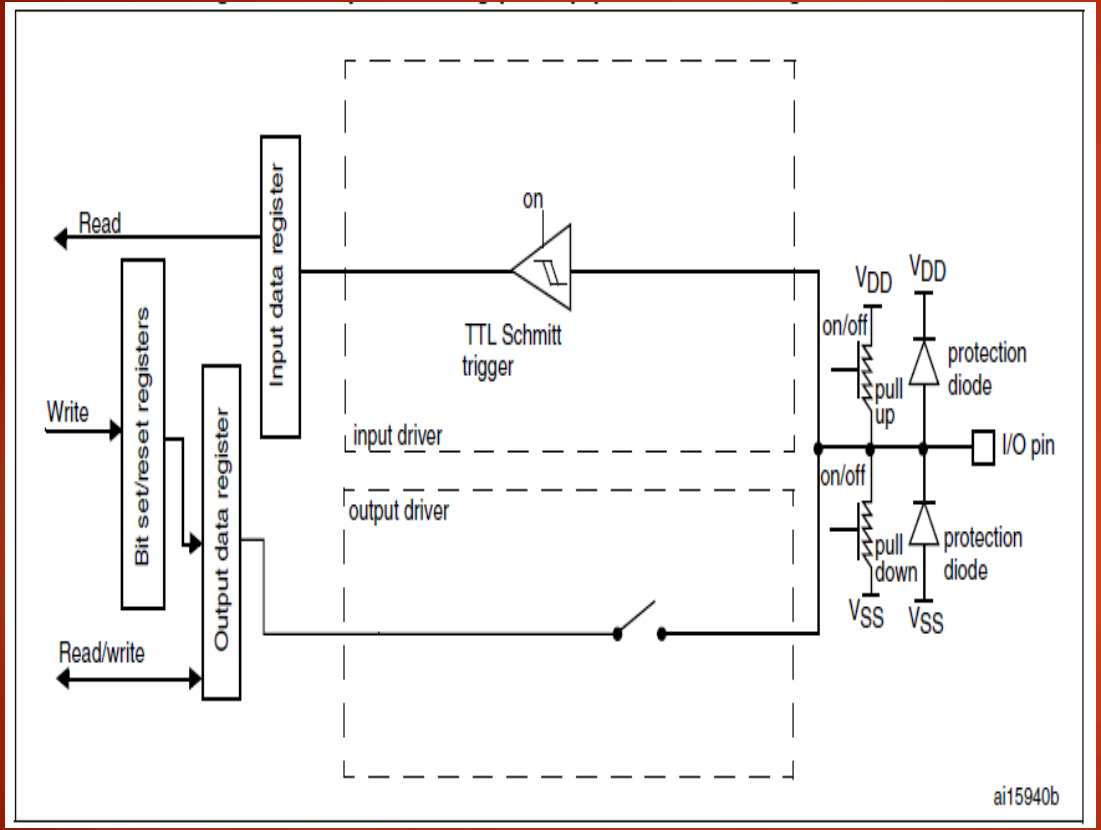


Input mode : Summary

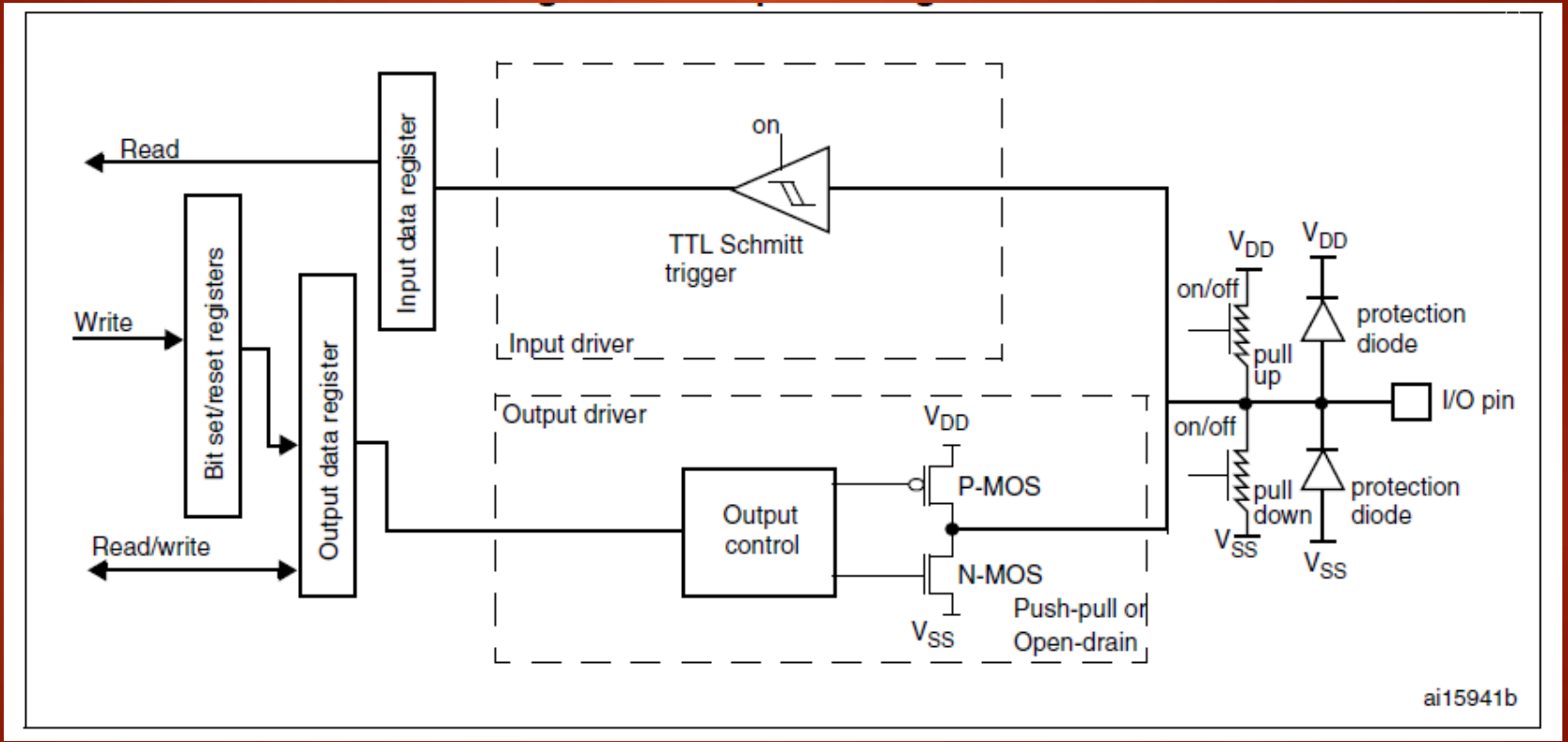
- When the I/O port is programmed as Input:
  - the output buffer is disabled
  - the Schmitt trigger input is activated
  - the pull-up and pull-down resistors are activated depending on the value in the GPIOx_PUPDR register
  - The data present on the I/O pin are sampled into the input data register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

Floating input

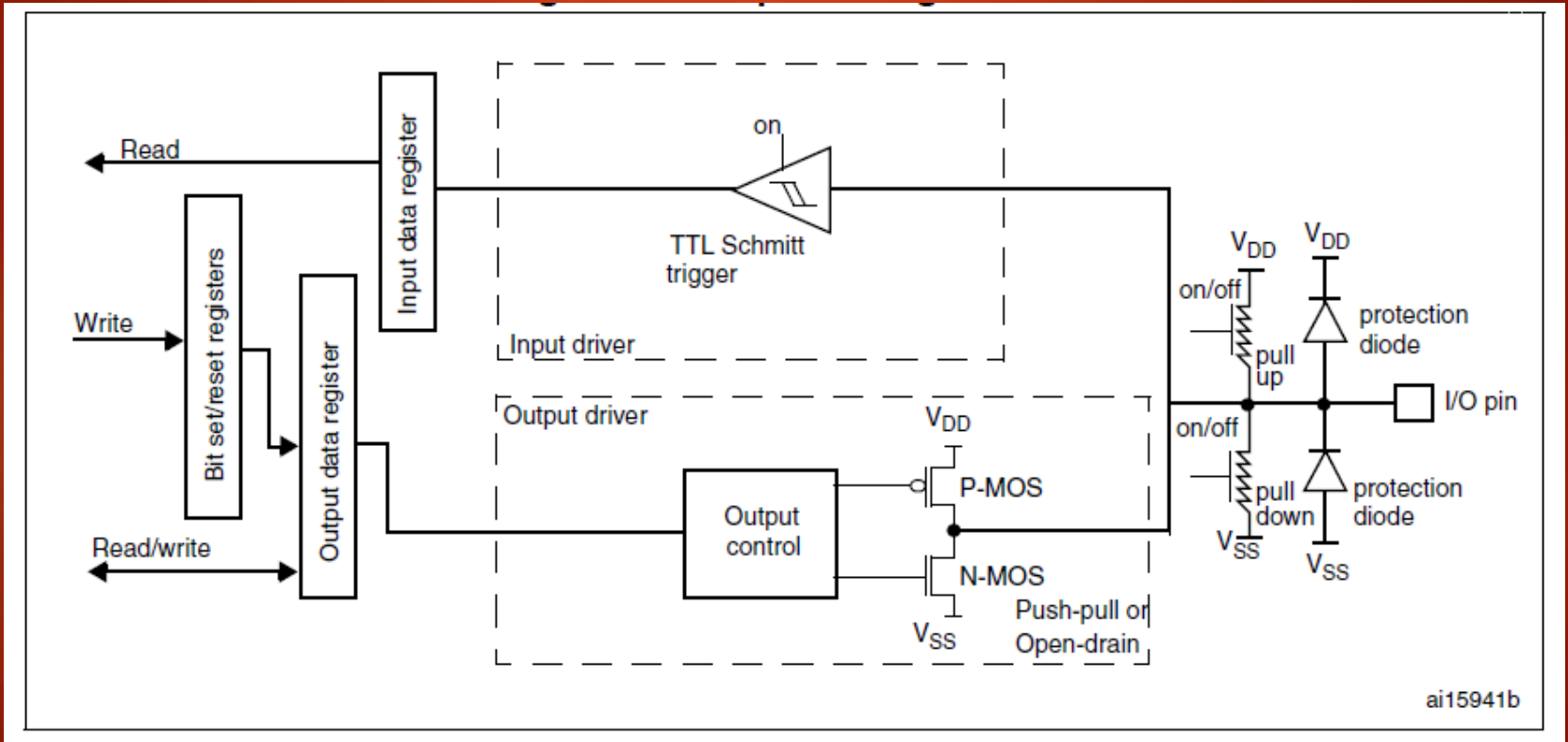




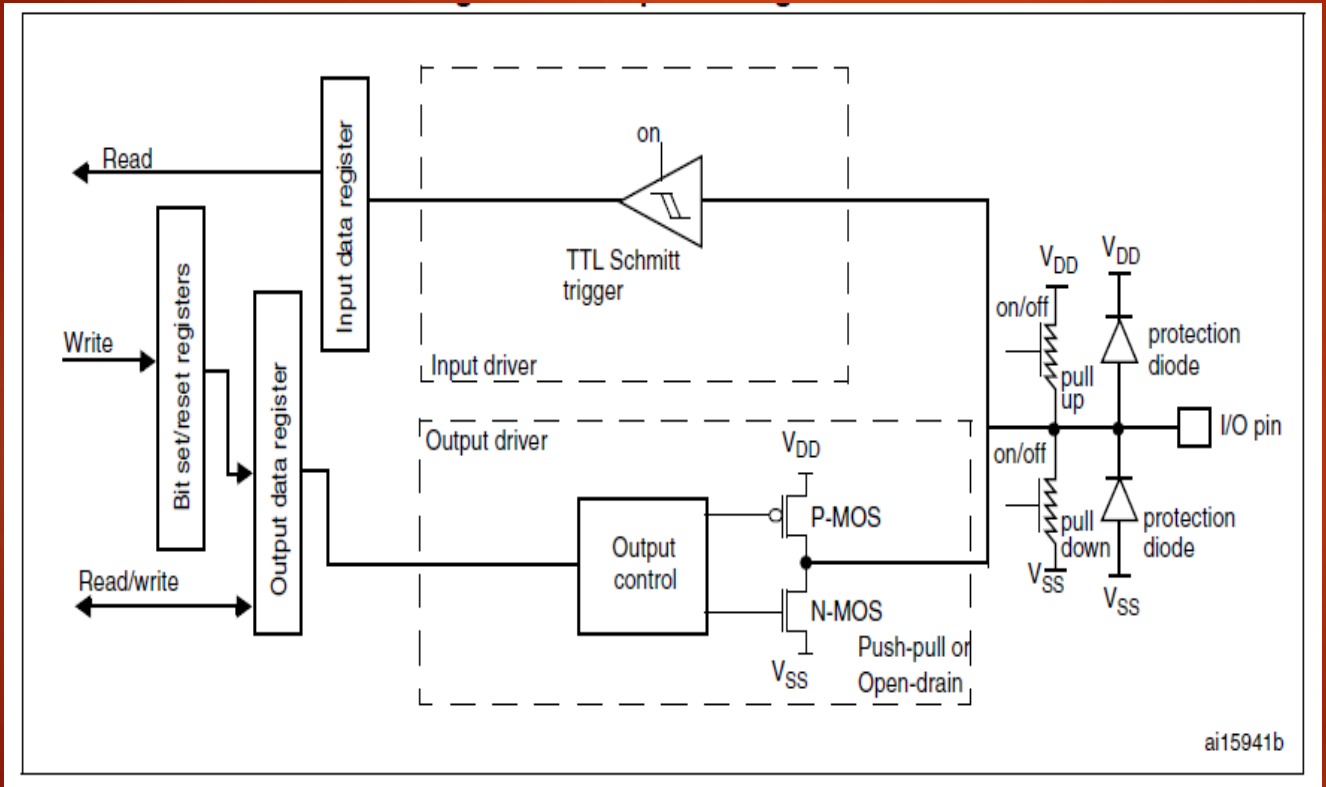
Output configuration



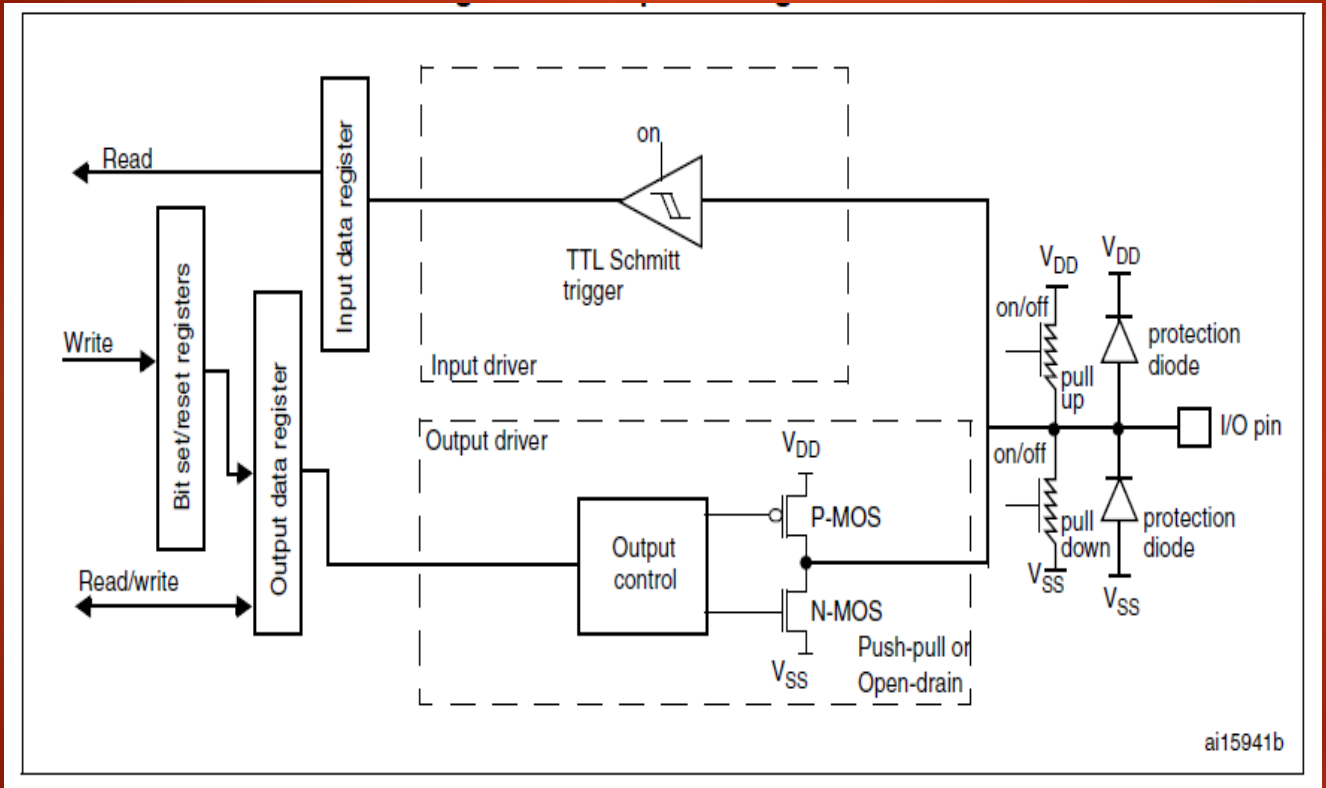
Output configuration (Push pull)



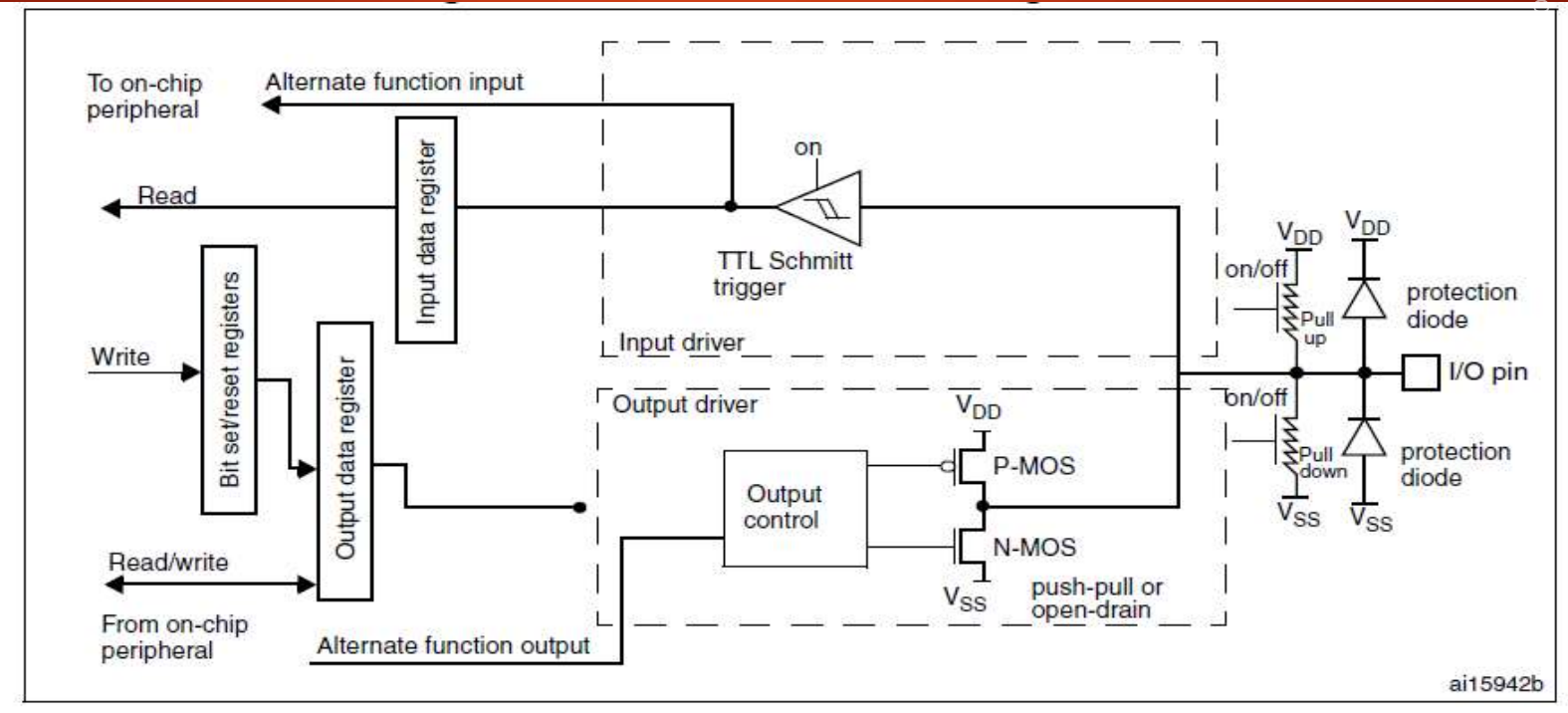
Output configuration (Push pull)



Output configuration (Open Drain)



Alternate function configuration



GPIO Port Output Type Register

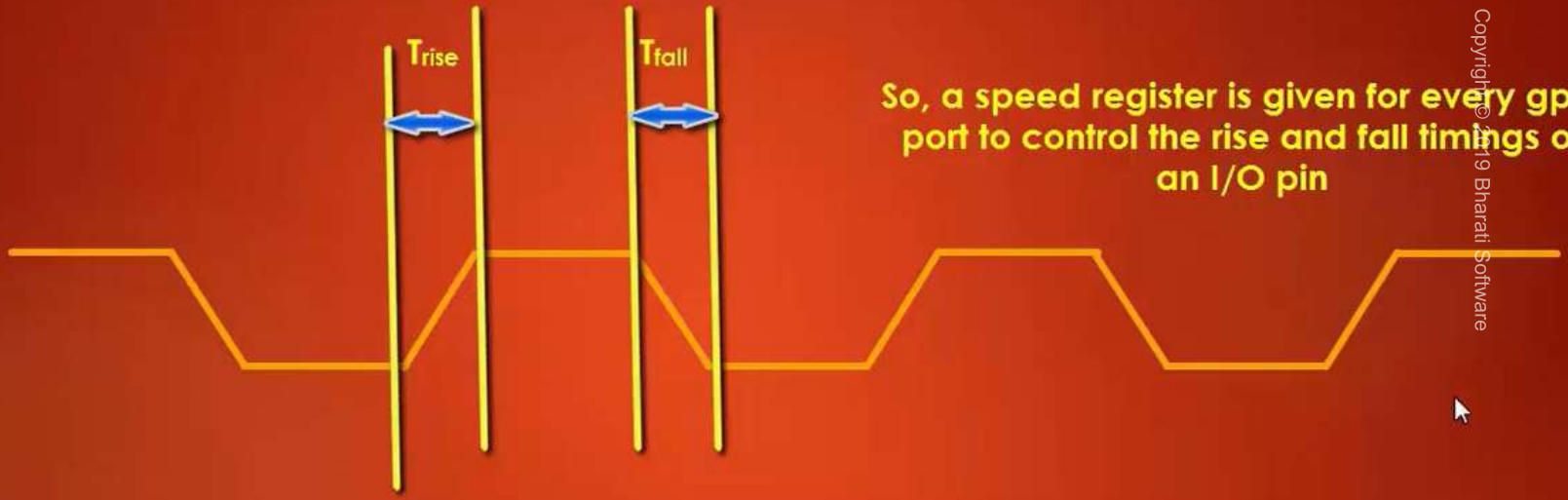
When a GPIO pin is in the output mode, this register is used to select the output type of that pin

GPIO Port Speed Register

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

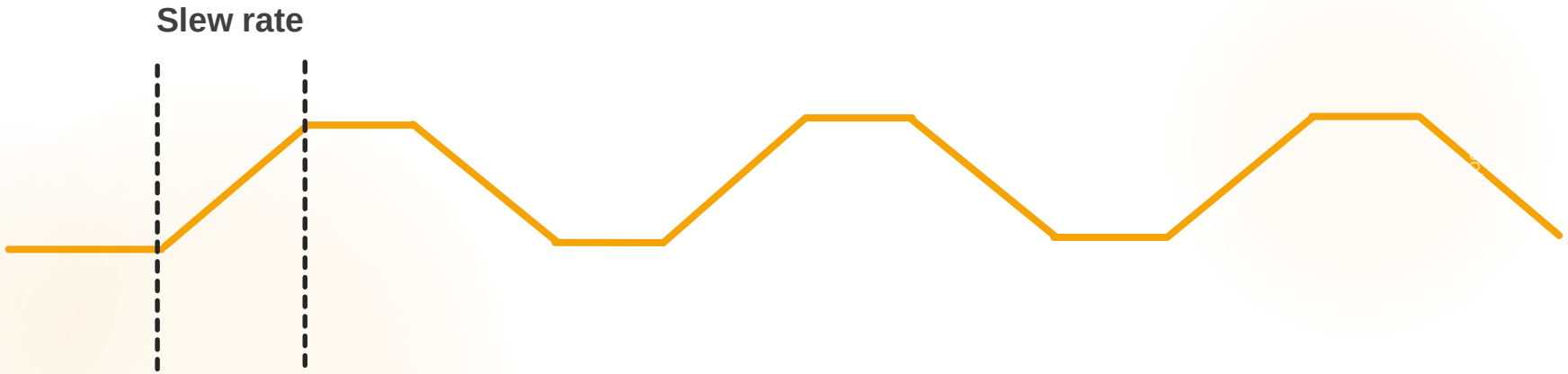
In other words you can control the slew rate of a pin

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

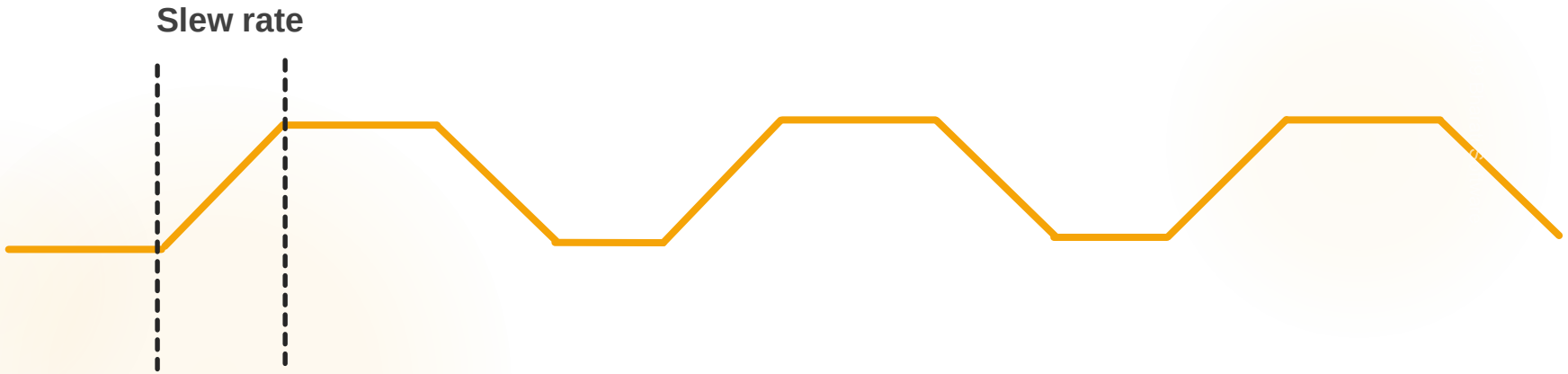


So, a speed register is given for every gpio port to control the rise and fall timings of an I/O pin

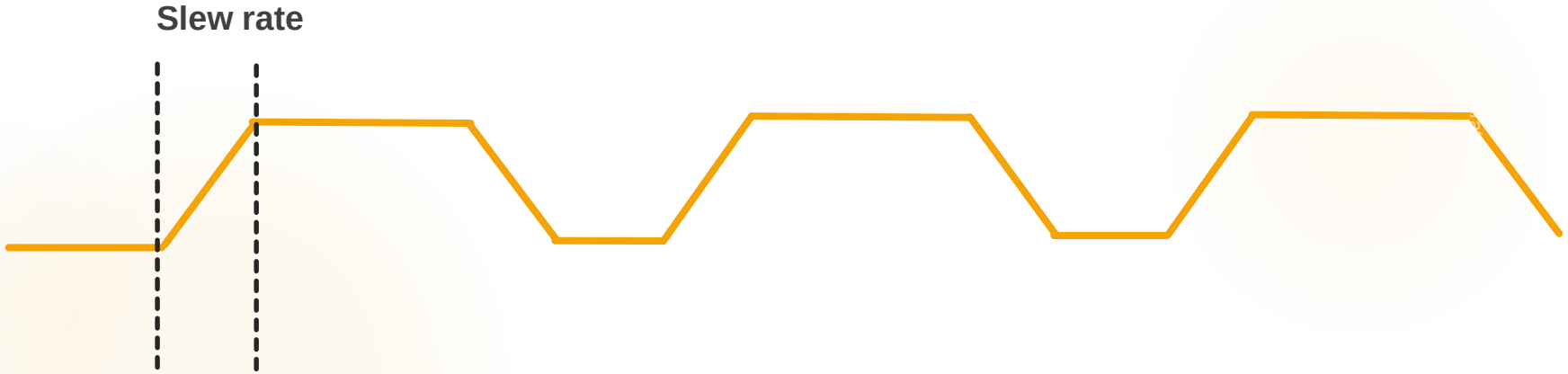
LOW



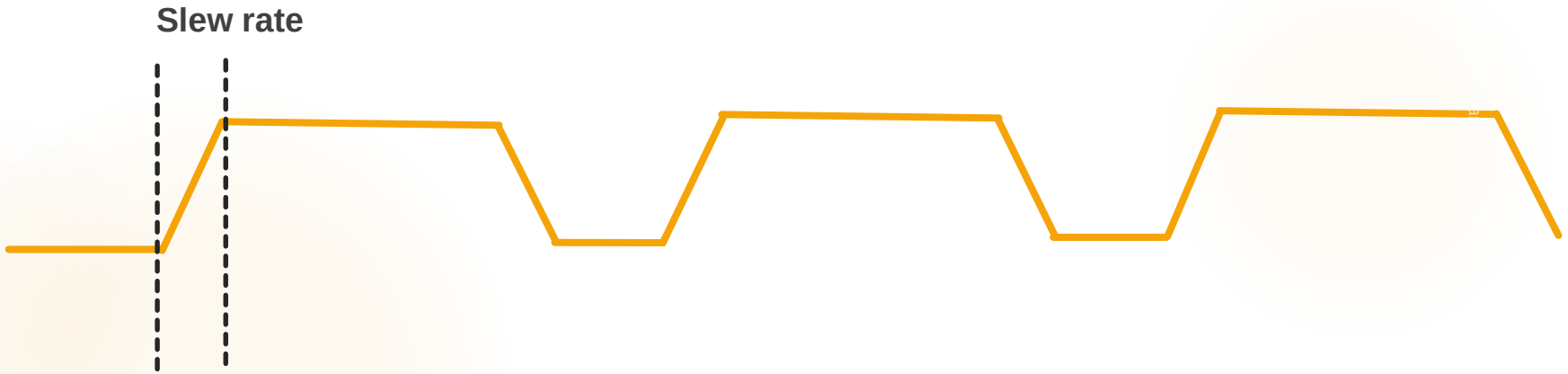
MEDIUM



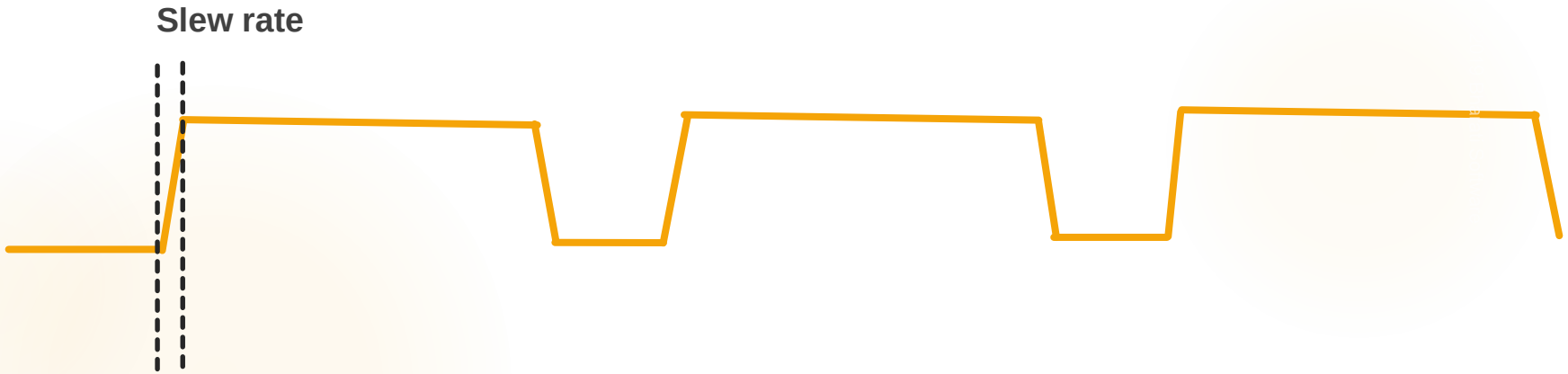
HIGH



VERY HIGH



VERY HIGH



Applicability of SPEED register configuration

MODER(i)[1:0]	OTYPER(i)	OSPEEDR(i)[B:A]	PUPDR(i)[1:0]		I/O configuration	
01 **Pin Mode = OUTPUT**	0	SPEED[B:A] **Applicable**	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 Pin Mode = INPUT	x	x	x	0	0	Input	Floating
	x	x	x	0	1	Input	PU
	x	x	x	1	0	Input	PD
	x	x	x	1	1	Reserved (input floating)	

Don't care

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

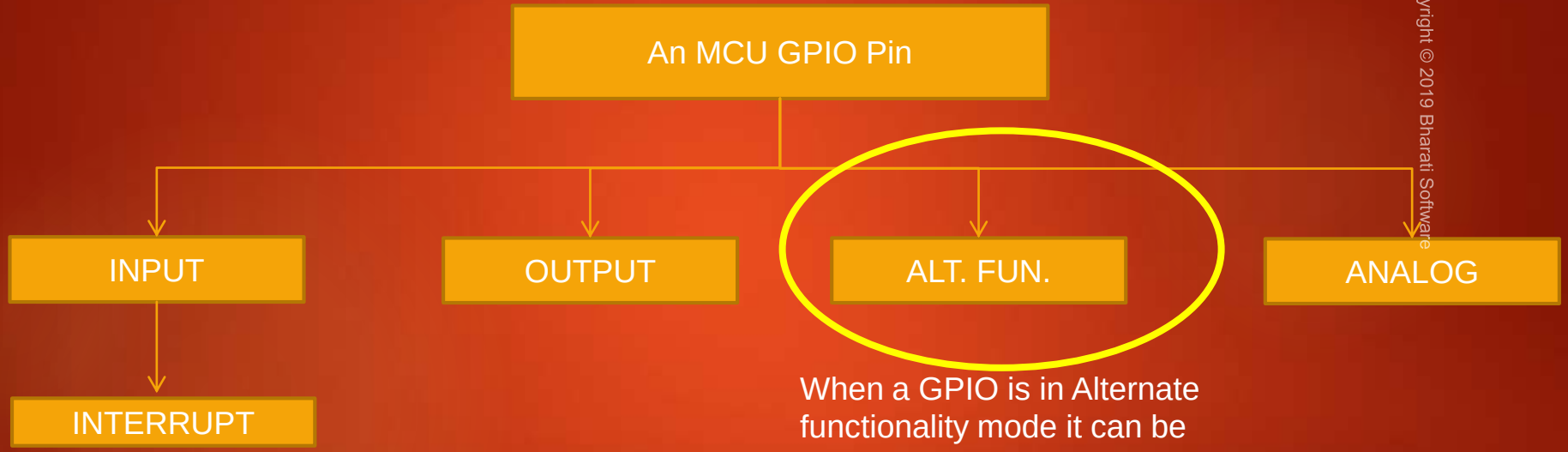
GPIO Port Output Data Register

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(1))
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

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	rw	rw	rw	rw	rw	rw	rw	rw	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	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
Pin 3				Pin 2				Pin 1				Pin 0			

Solution: PA1 as UART4_RX

AFx = AF8

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	rw	rw	rw	rw	rw	rw	rw	rw	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	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
Pin 3				Pin 2				Pin 1				Pin 0			

Solution: PA10 as TIM1_CH3

AFx = AF1

GPIO alternate function high register (GPIOx_AFRH) (x = A..I/J) GPIOA_AFRH

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	rw	rw	rw	rw	rw	rw	rw	rw	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	rw	rw	rw	rw	rw	rw	rw	rw	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	rw	rw	rw	rw	rw	rw	rw	rw	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	rw	rw	rw	rw	rw	rw	rw	rw	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	rw	rw	rw	rw	rw	rw	rw	rw	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	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Pin 3 Pin 2 Pin 1 Pin 0

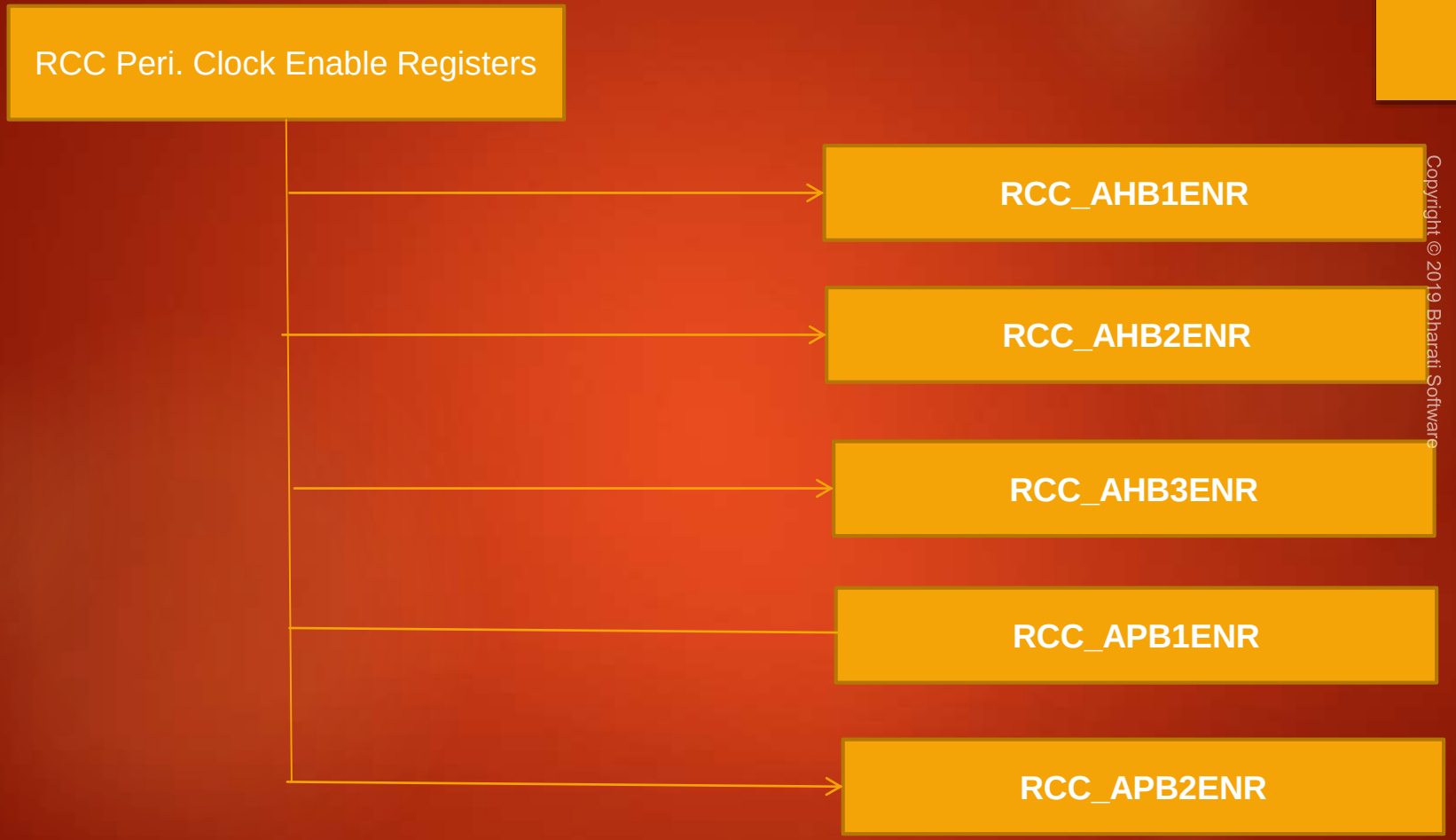
8.4.5 GPIO port input data register (GPIOx_IDR) (x = A..I/J/K)

Address offset: 0x10

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

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
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Enabling /Disabling GPIO Port Peripheral Clock



RCC AHB1 peripheral clock register (RCC_AHB1ENR)

Address offset: 0x30

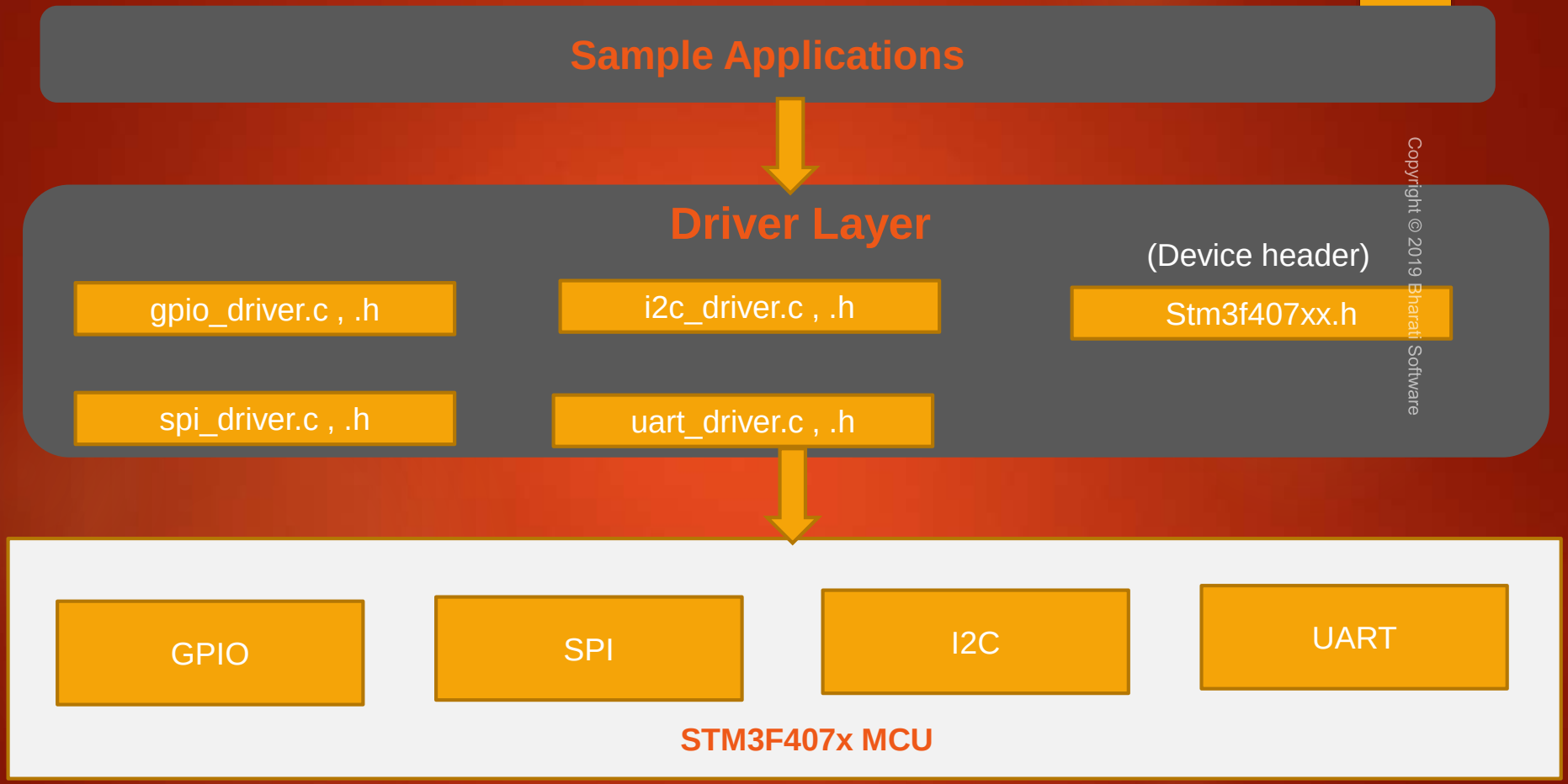
Reset value: 0x0010 0000

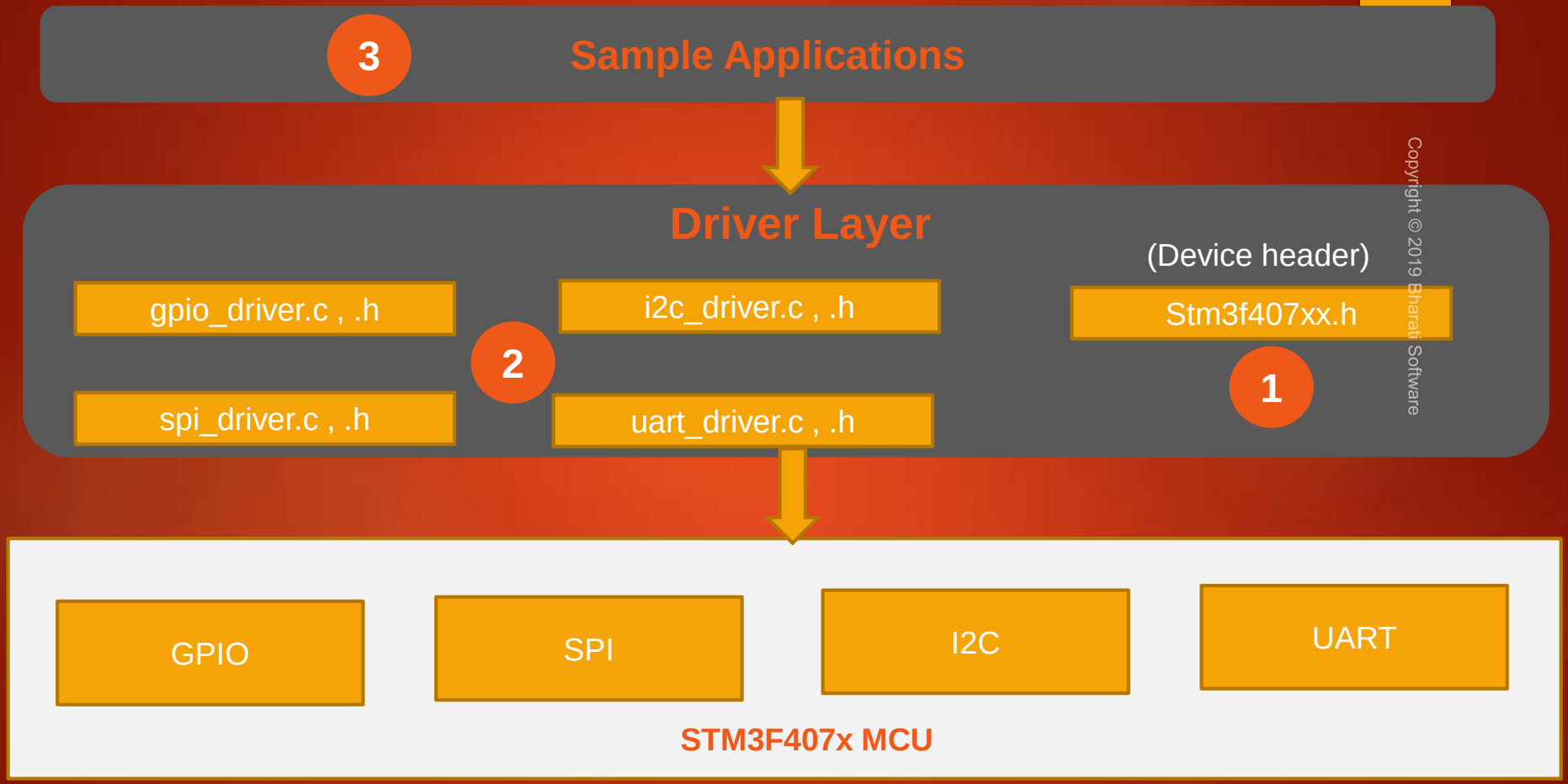
Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reser-ved	OTGHS ULPIEN	OTGHSEN	ETHMACPTPEN	ETHMACRXEN	ETHMACTXEN	ETHMACEN	Res.	DMA2DEN	DMA2EN	DMA1EN	CCMDATARAMEN	Res.	BKPSRAMEN	Reserved	
	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Res.	GPIOKEN	GPIOJEN	GPIOIEN	GPIOHEN	GPIOGEN	GPIOFEN	GPIOEEN	GPIODEN	GPIOCEN	GPIOBEN	GPIOAEN
			rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Peripheral Driver Development

High level project architecture

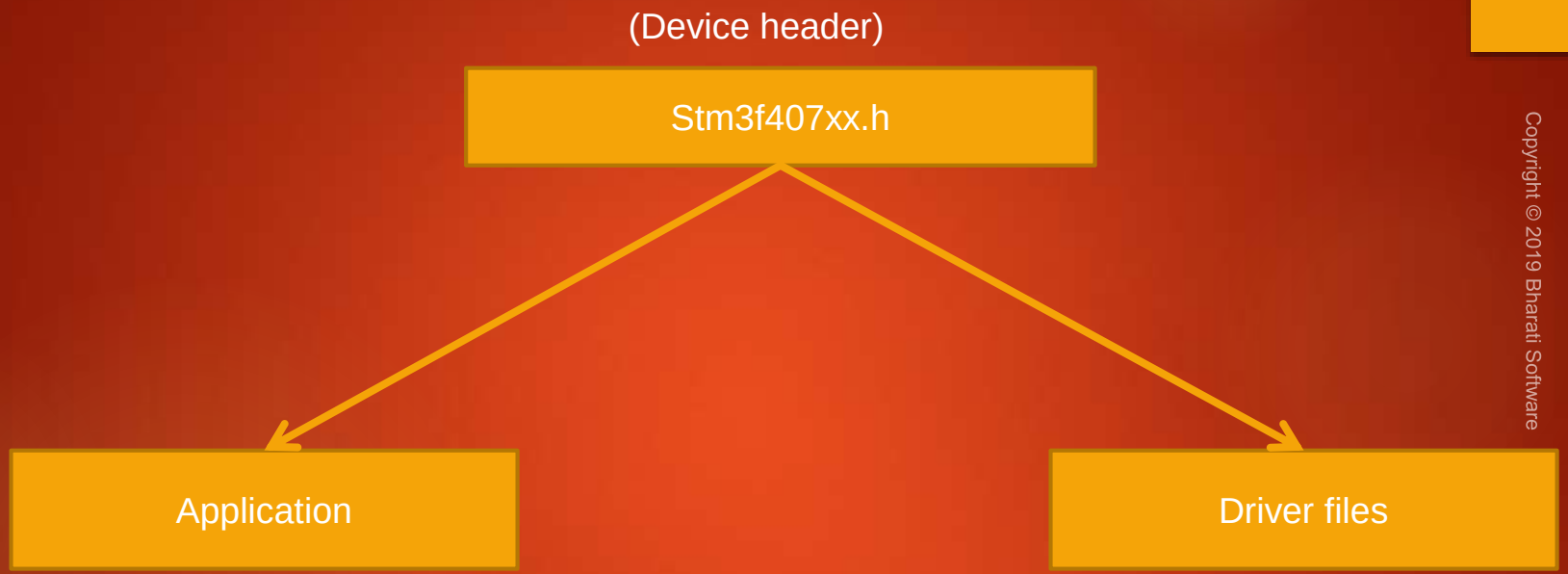




What is Device Header file and what it contains ?

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

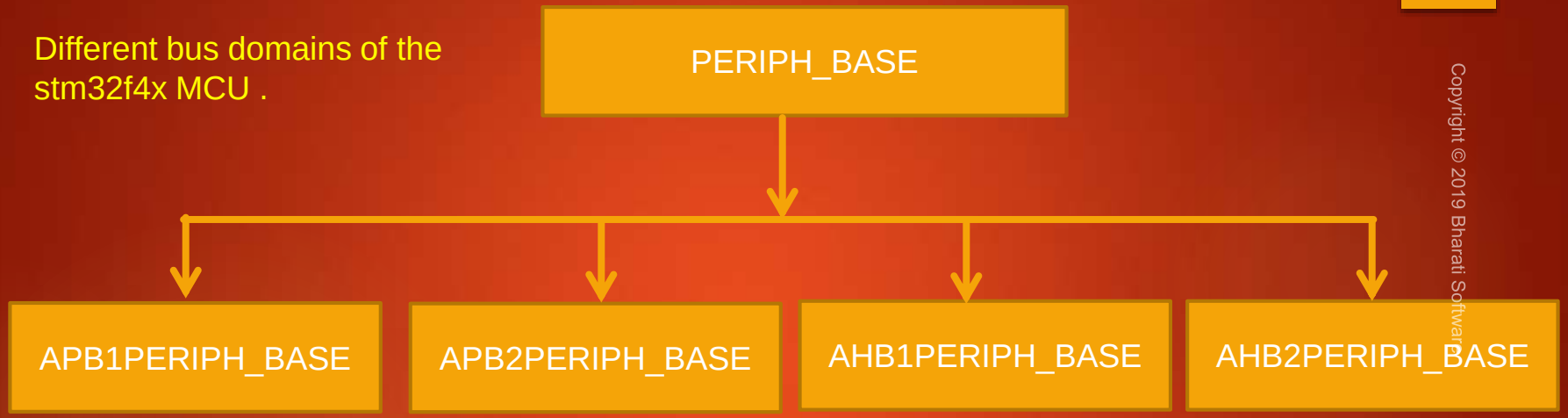
In the next lecture lets create MCU Device specific header file step by step

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 .



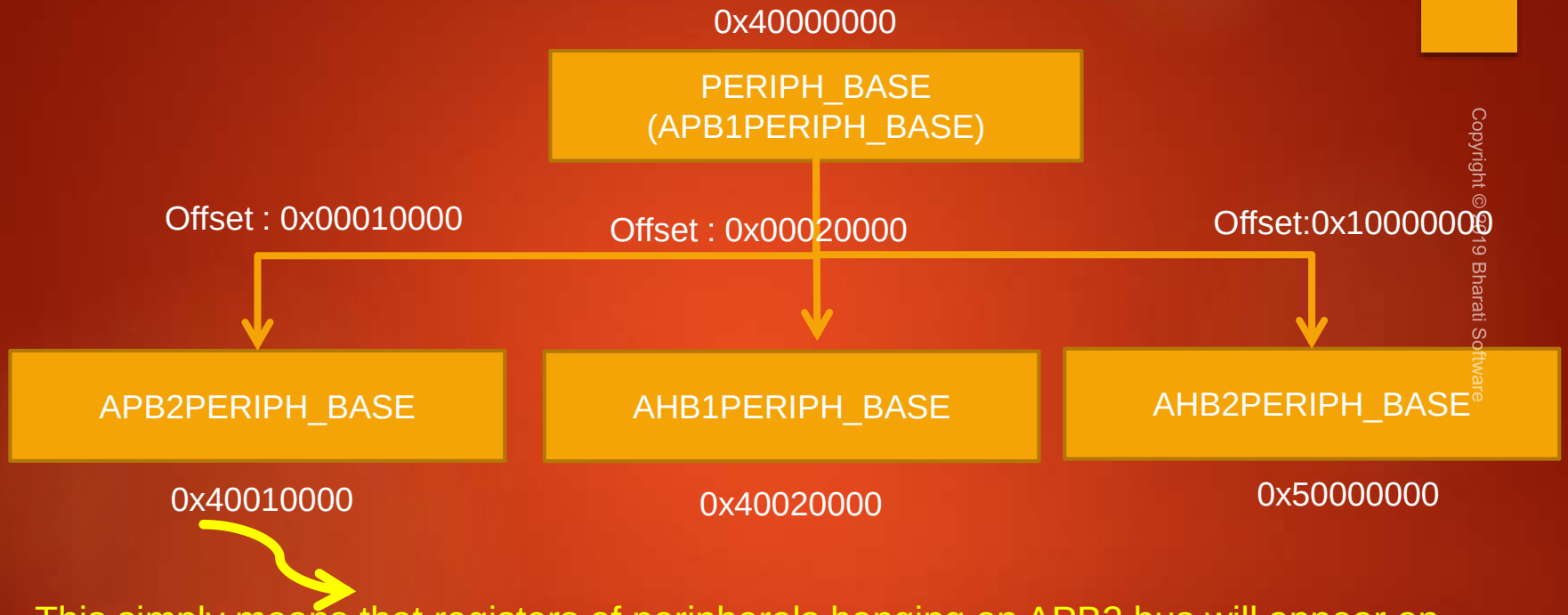
- Different peripherals are hanging on different busses .
- AHB bus is used for those peripherals which need high speed data communication (ex. Camera interfaces, GPIOs)
- APB bus is used for those peripherals for which low speed communication would suffice .

Full Memory map of the MCU

Full Memory map of the MCU

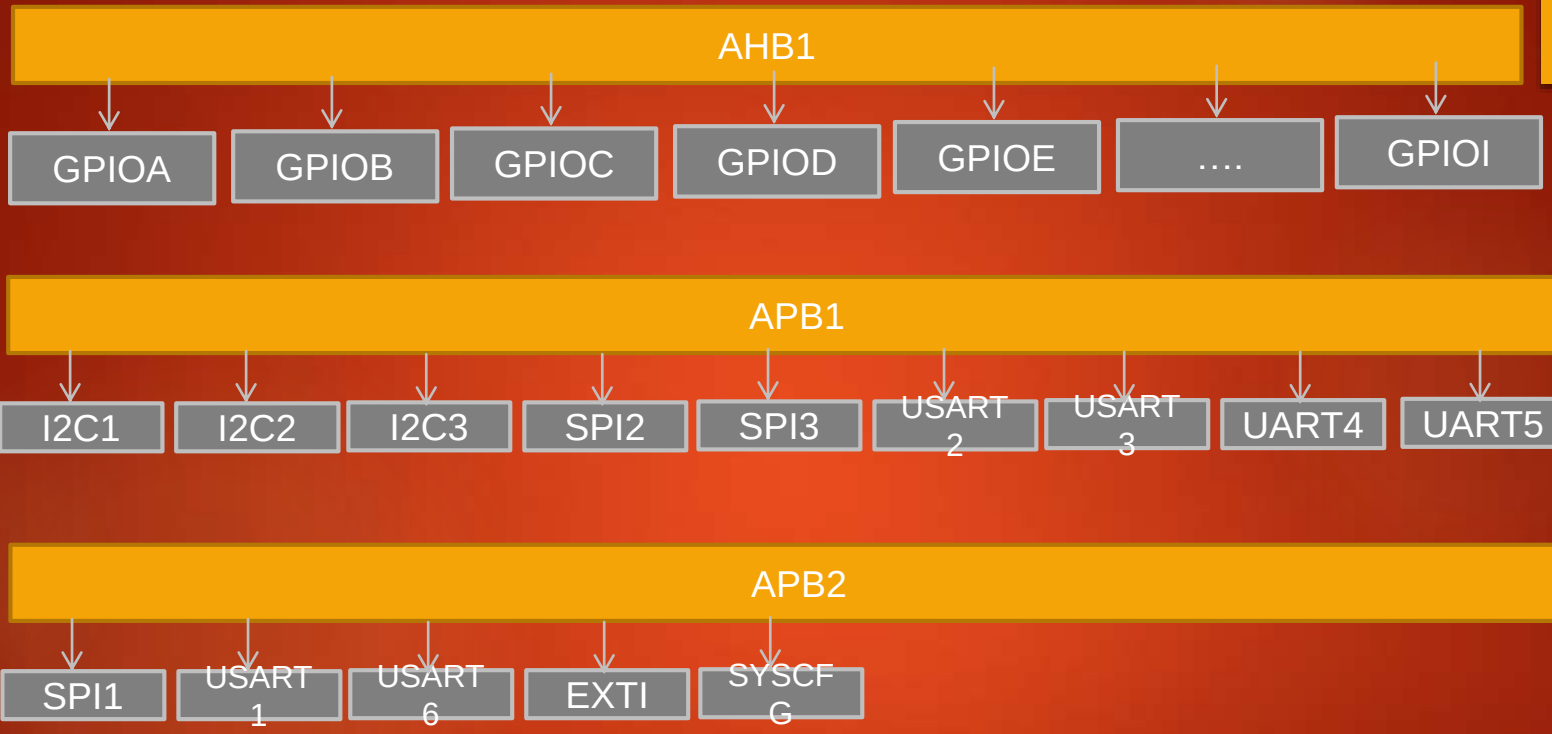
Find out 0x4000_0000 is the address of which register of which peripheral ????

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



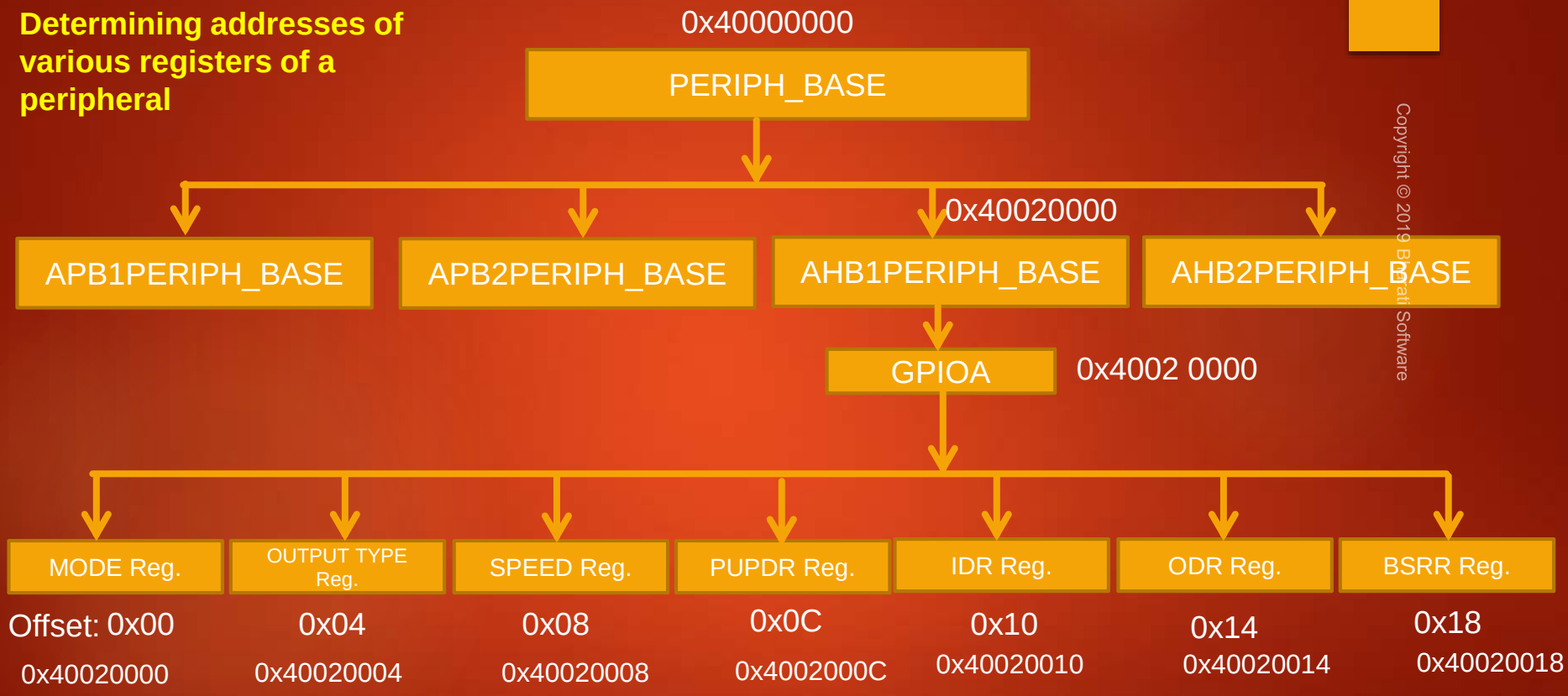
This simply means that registers of peripherals hanging on APB2 bus will appear on address 0x40010000 onwards in the memory map of the MCU.

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
control register 1	??
control register 2	??
status register	??
data register	??
CRC polynomial register	??
RX CRC register	??
TX CRC register	??
configuration register	??
prescaler register	??

Structuring Peripheral register details

Example : ‘C’ Structure for registers of GPIO peripheral


```c
/*
 * 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,                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;

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

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,                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;

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

Base address of GPIOA

```c
/*
 * 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      */
   uint16_t BSRRL;      /*!< TODO,                              Address offset: 0x18      */
   uint16_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;

 GPIO_RegDef_t *pGPIOA = GPIOA;
```

Base address of GPIOA

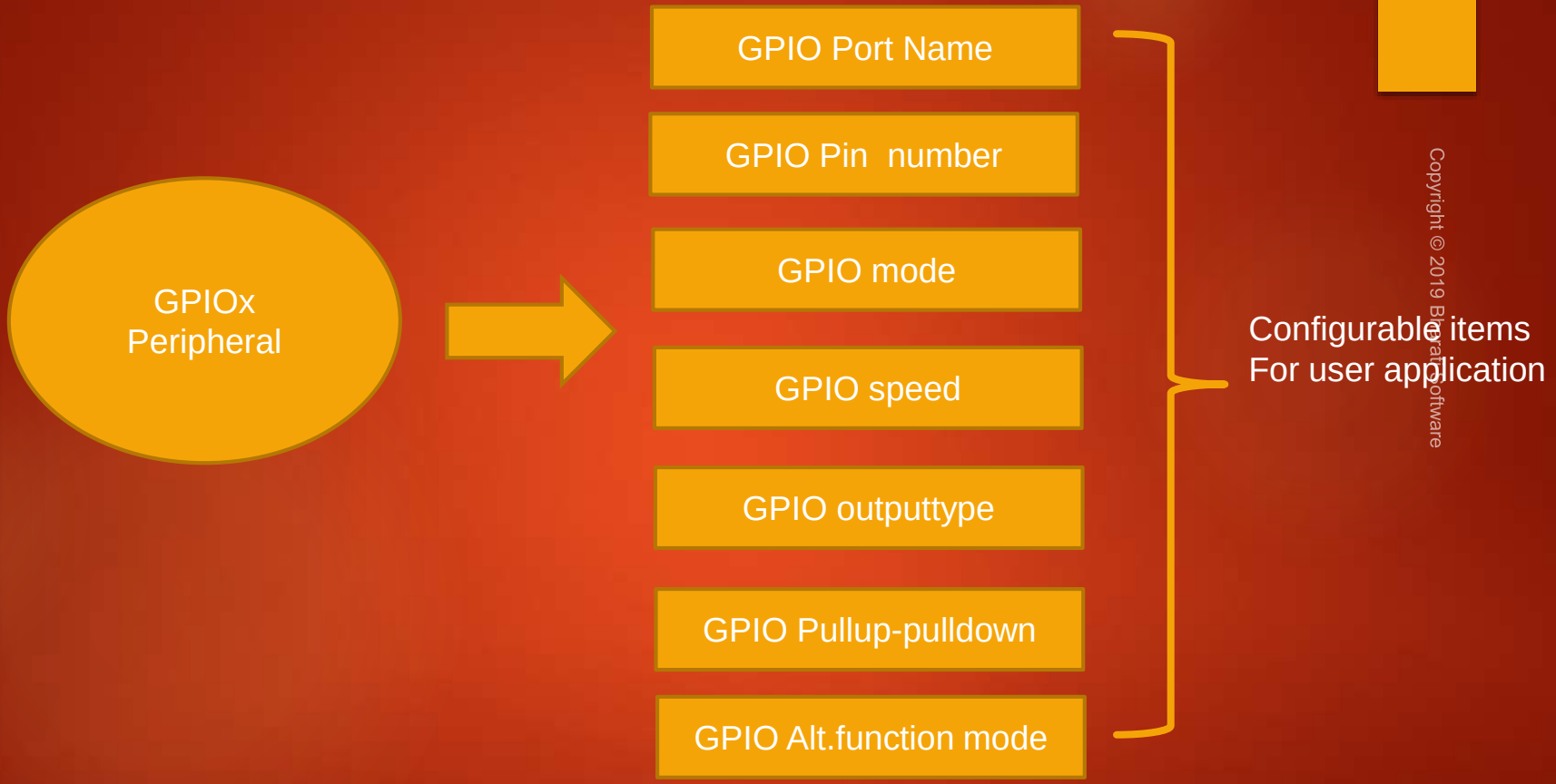
```c
#define GPIOA                    ((GPIO_RegDef_t*)GPIOA_BASE)
```

Peripheral clock enable and disable macros

Macros for IRQ(Interrupt Request) Numbers of the MCU

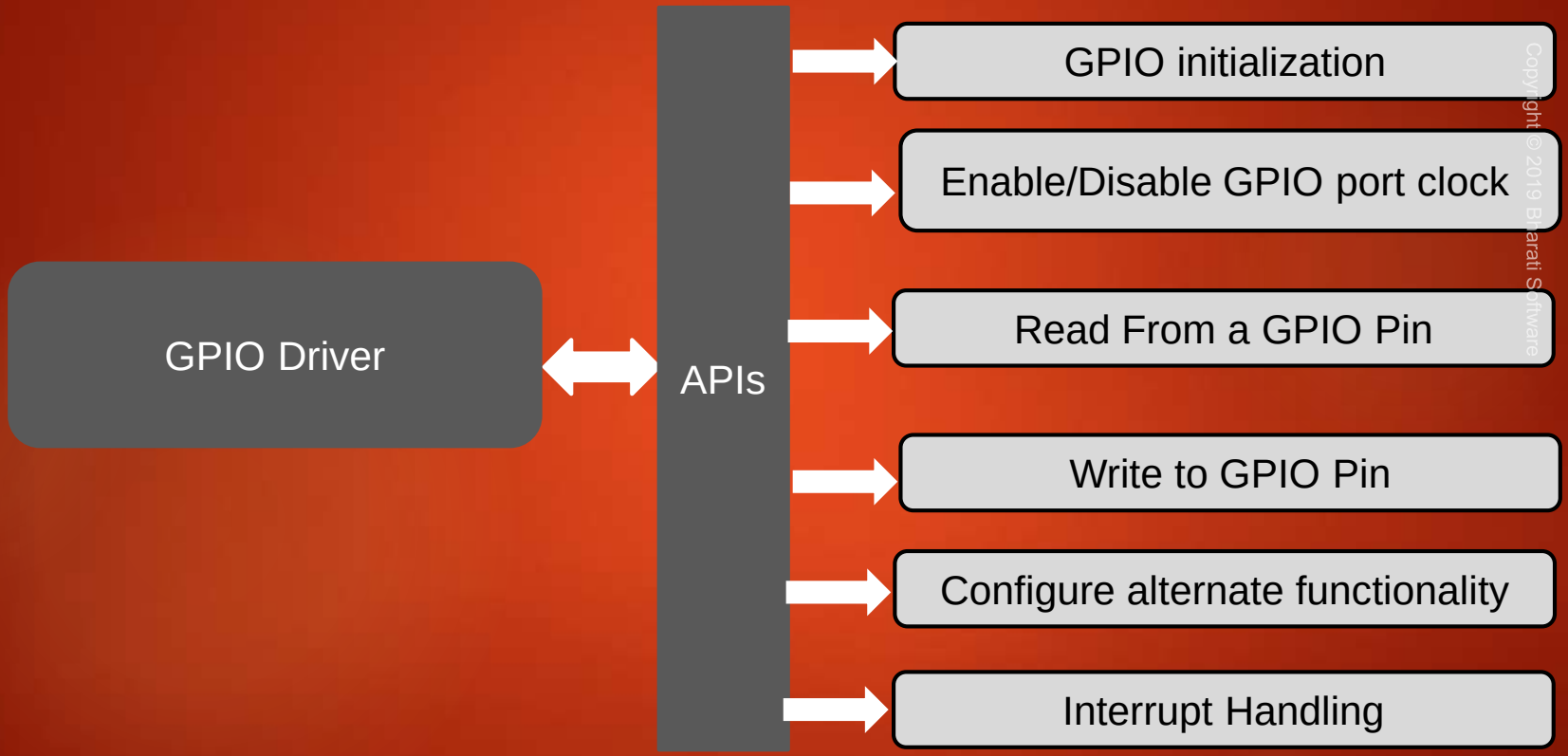
Creating GPIO Driver .c and .h files

GPIO Handle and Configuration structures



GPIO Driver APIs requirements

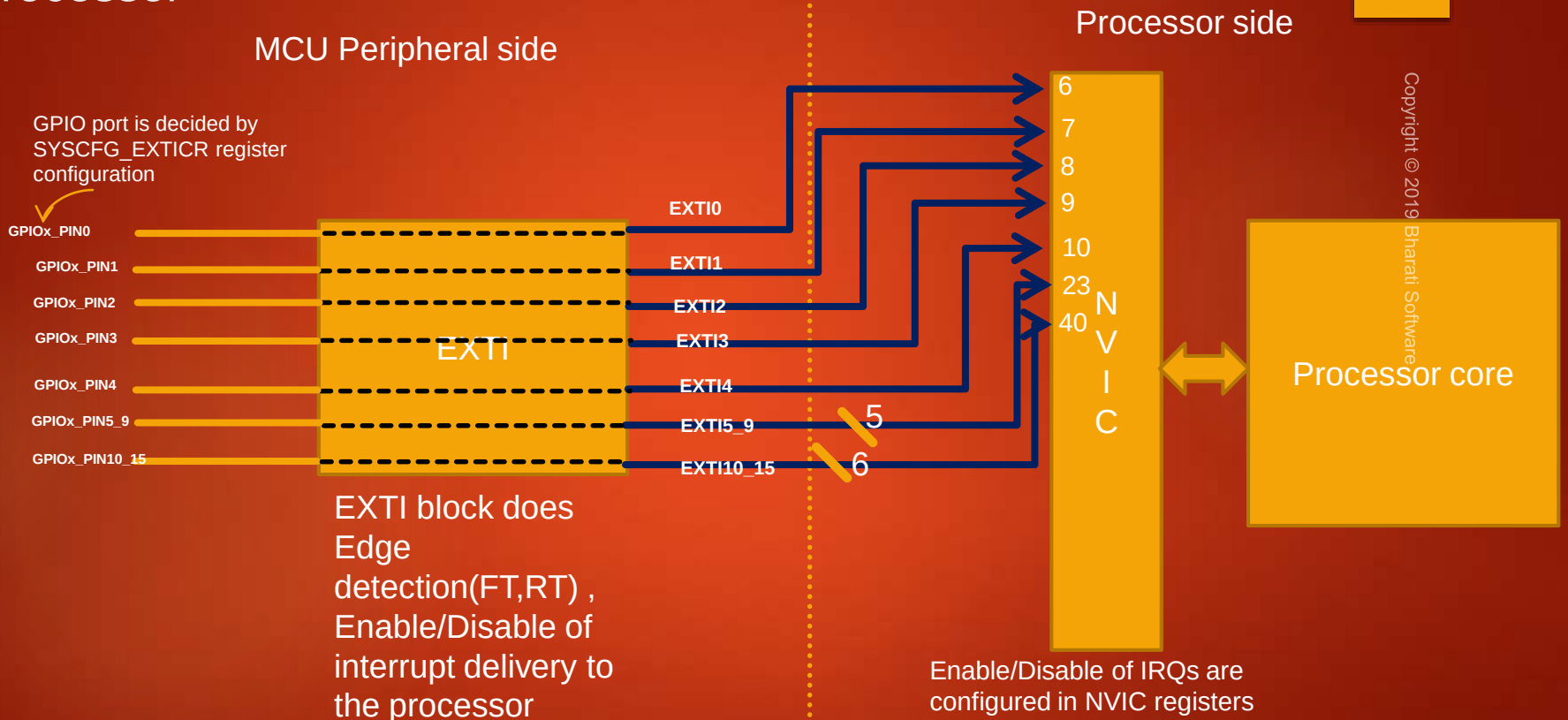
Driver API Requirements



GPIO Pin Interrupt Configuration

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	rw	rw	rw	rw	rw	rw	rw	rw	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

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	rw	rw	rw	rw	rw	rw	rw	rw	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	rw	rw	rw	rw	rw	rw	rw	rw	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

NVIC_ISER0

0 has no effect
1 enables the interrupt

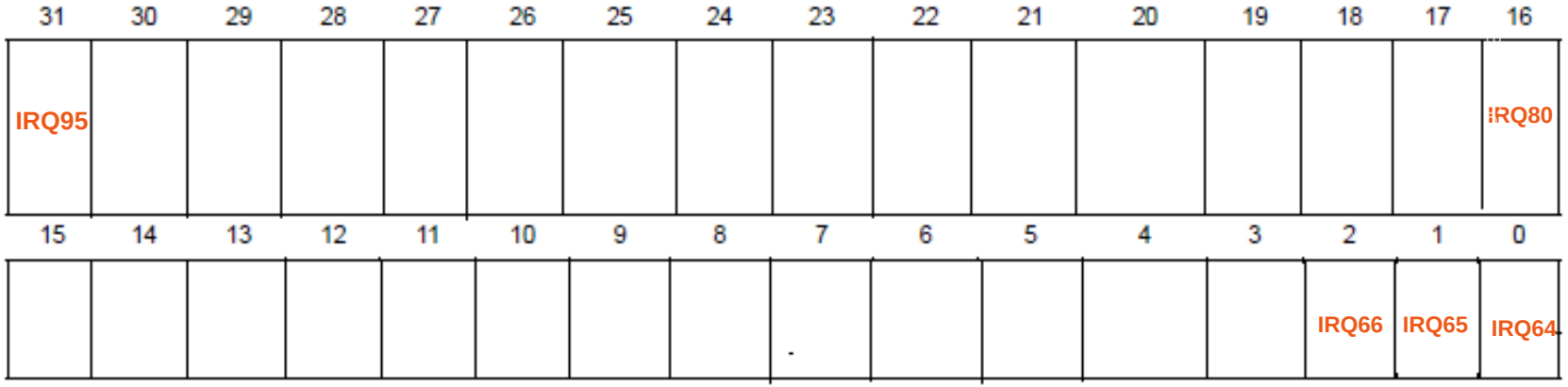
Interrupt Set-enable Registers

NVIC_ISER1

0 has no effect
1 enables the interrupt

Interrupt Set-enable Registers

NVIC_ISER2



0 has no effect
1 enables the interrupt

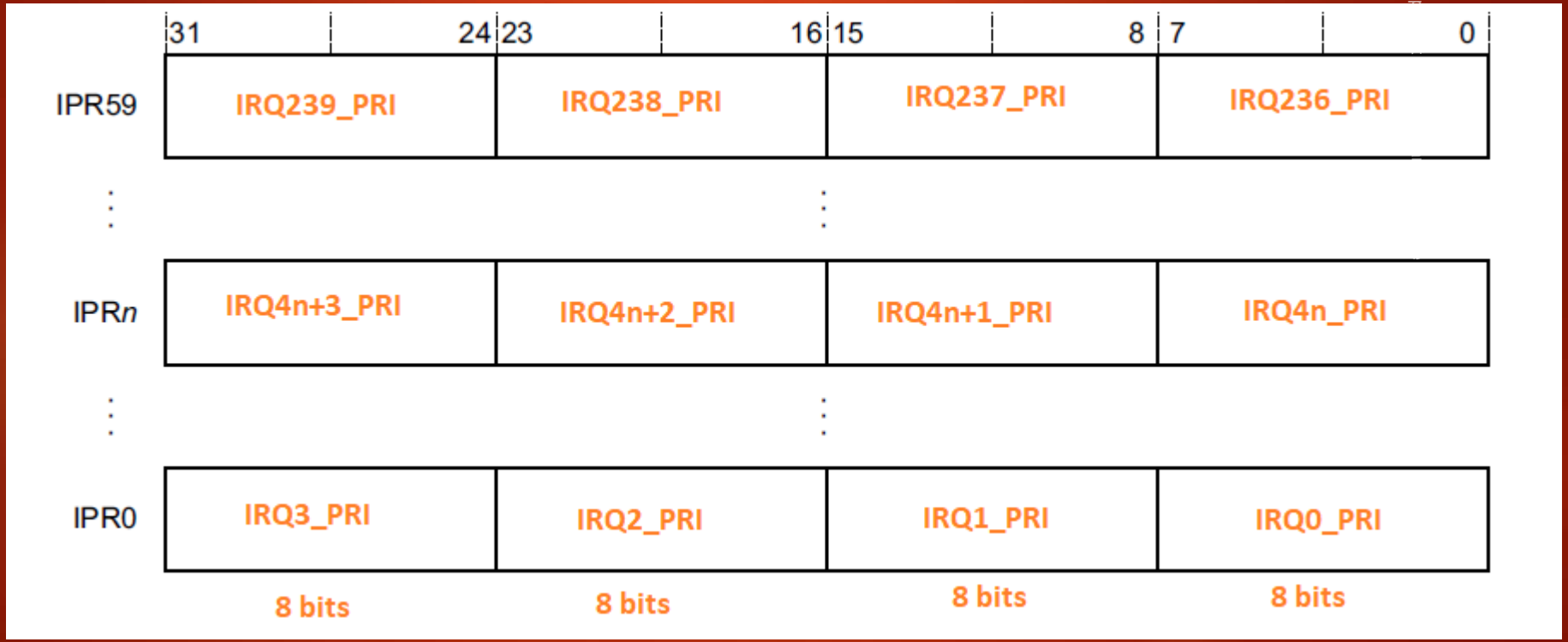
Interrupt Clear-enable Registers

NVIC_ICER0

0 has no effect
1 disable interrupt

Interrupt Priority Registers

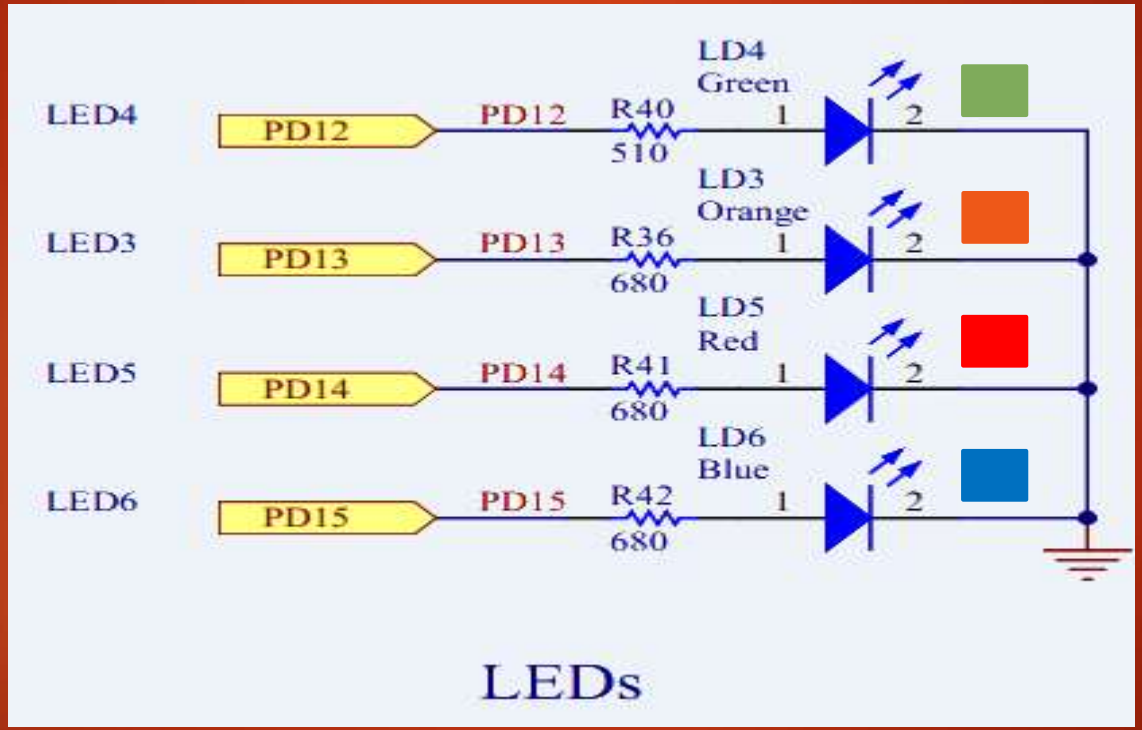
NVIC_IPR0-NVIC_IPR59

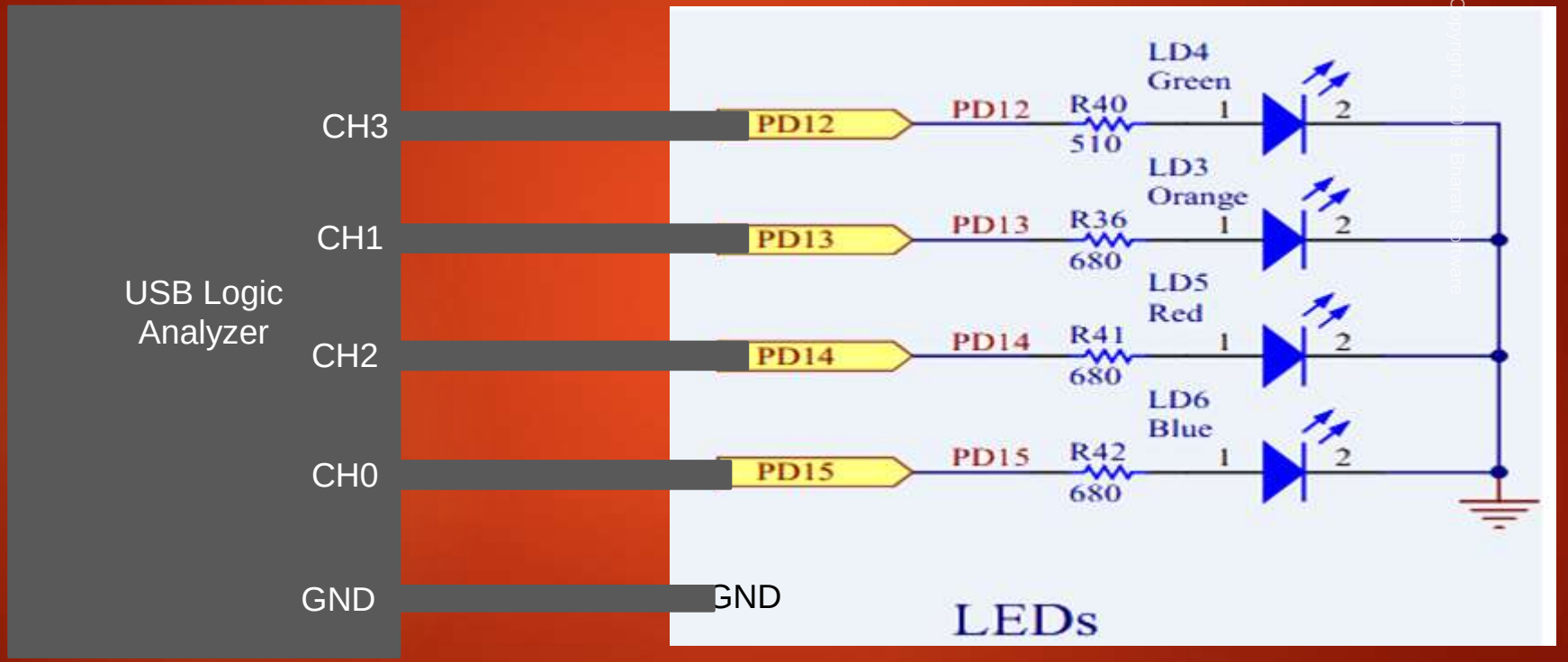


Exercise

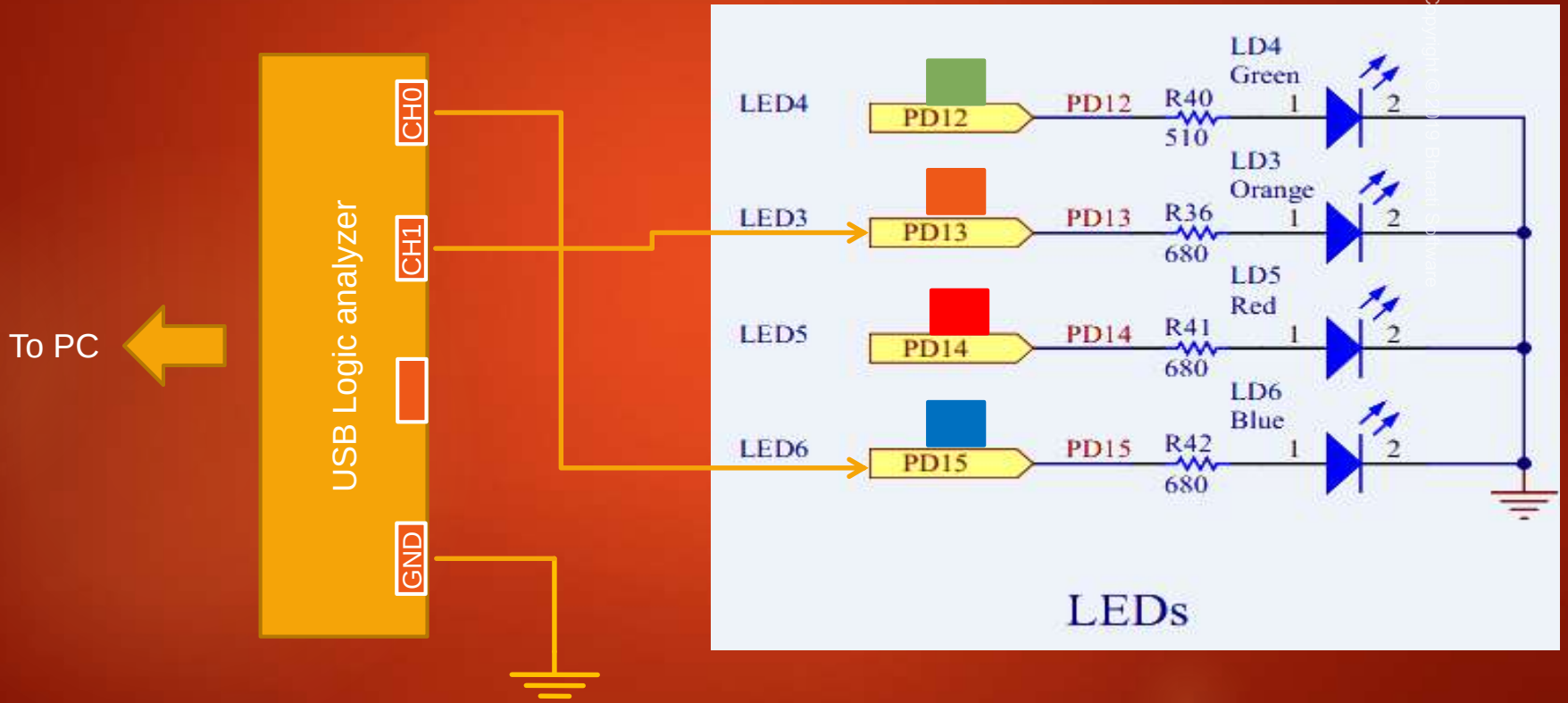
Write a program to toggle the on board LED with some delay .
Case 1 : Use push pull configuration for the output pin
Case 2 : Use open drain configuration for the output pin

Discovery Board LEDs



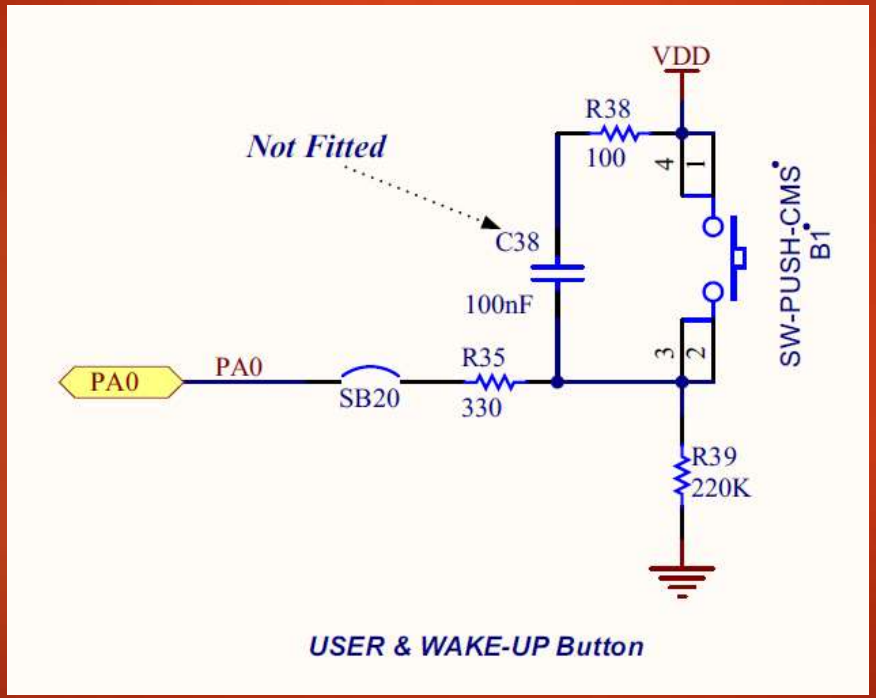

USB Logic Analyzer
CH3
CH1
CH2
CH0
GND
GND
PD12
PD13
PD14
PD15
PD12
R40
510
LD4
Green
PD13
R36
680
LD3
Orange
PD14
R41
680
LD5
Red
PD15
R42
680
LD6
Blue
1
2
LEDs

Probing LED Toggling using USB Logic Analyzer



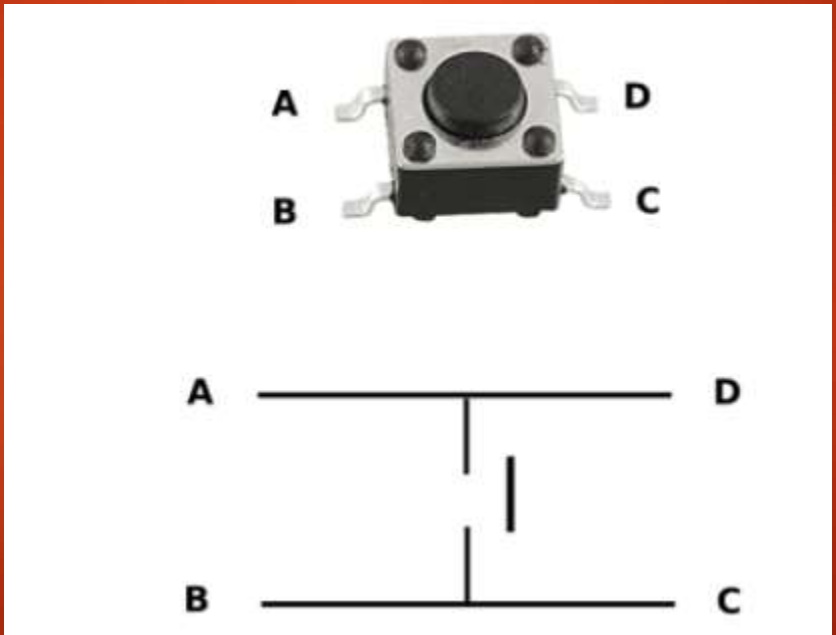
Exercise

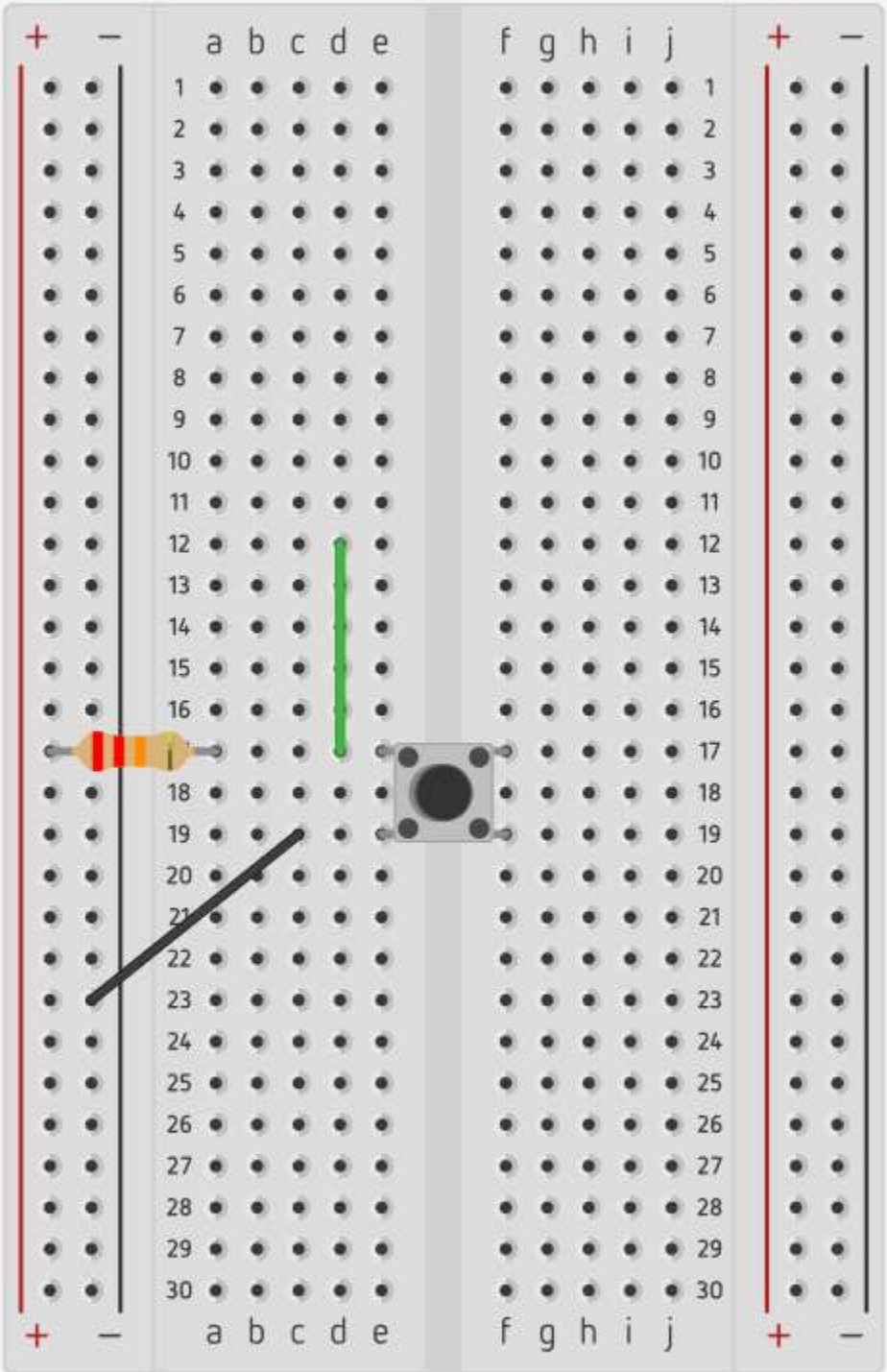
Write a program to toggle the on board LED whenever the on board button is pressed



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




+ −
a b c d e
f g h i j
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

Exercise

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

Exercise

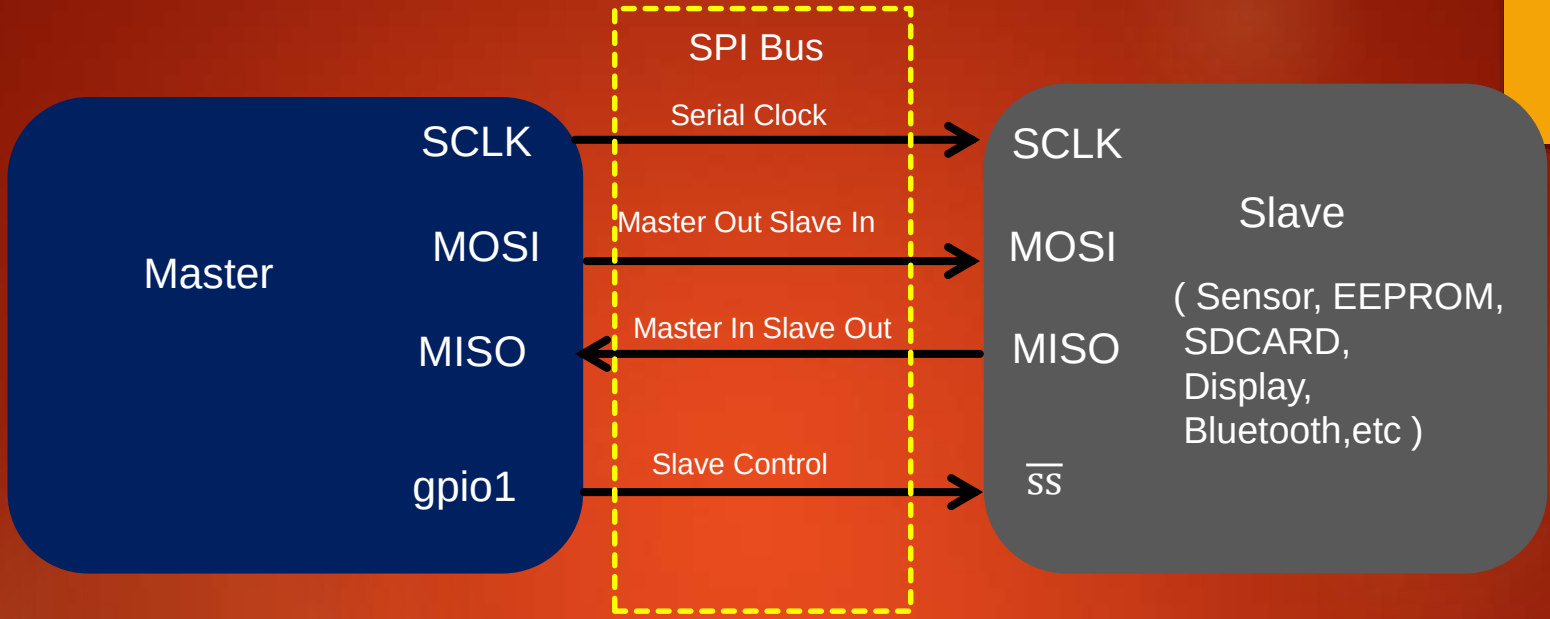
Connect an external button to PD5 pin and toggle the led whenever interrupt is triggered by the button press.

Interrupt should be triggered during falling edge of button press.

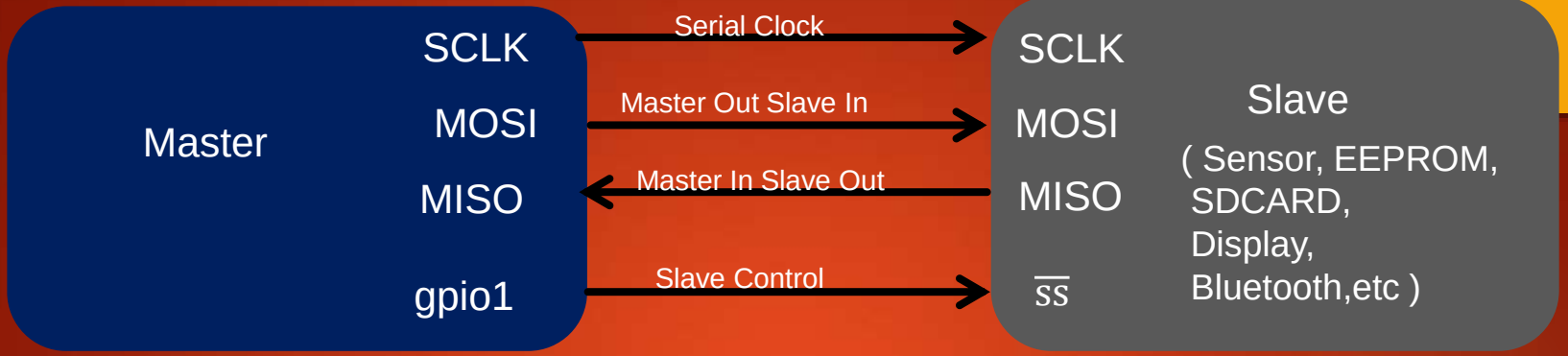
In the next lecture lets understand input mode of a pin

Introduction to SPI

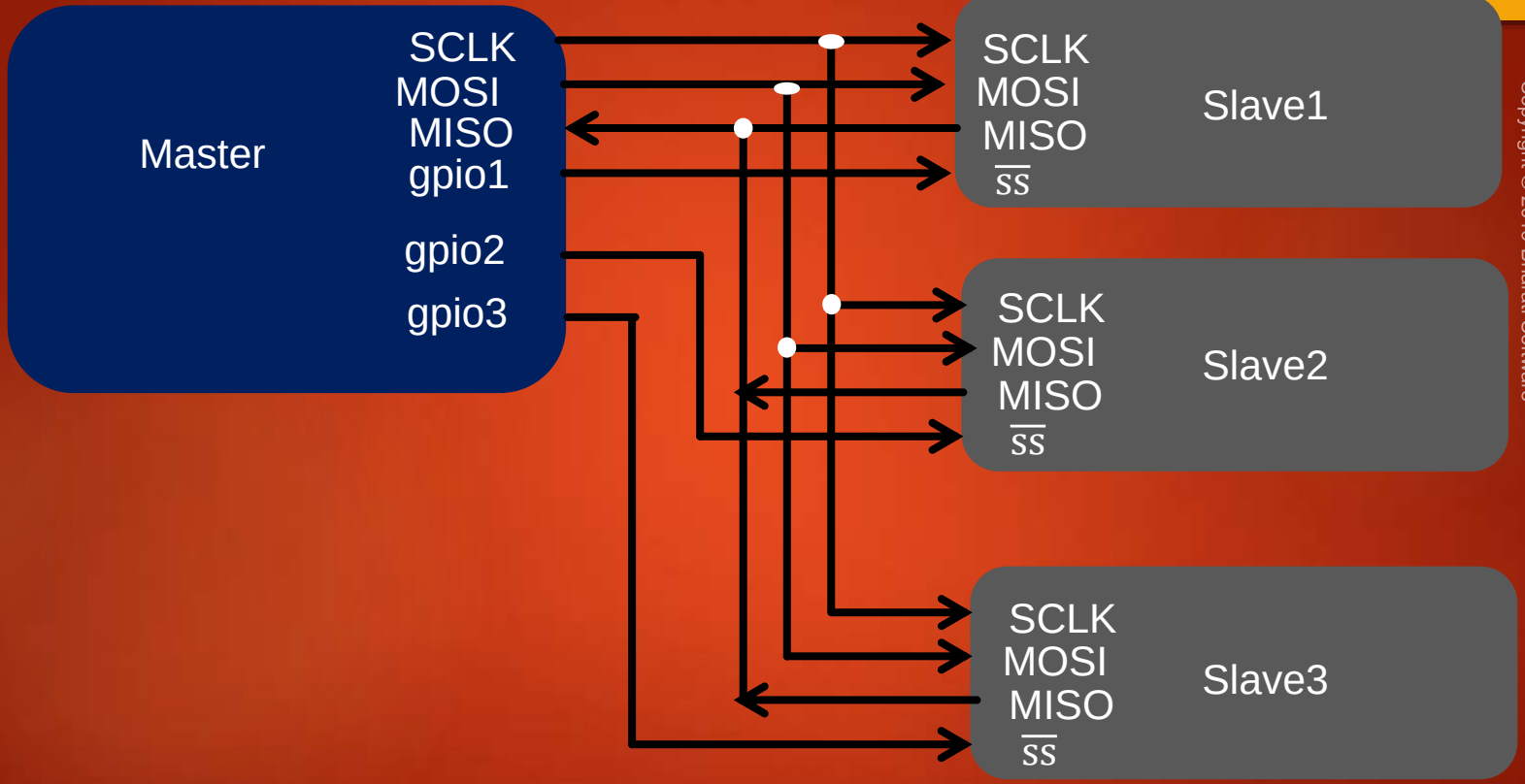
Serial Peripheral Interface



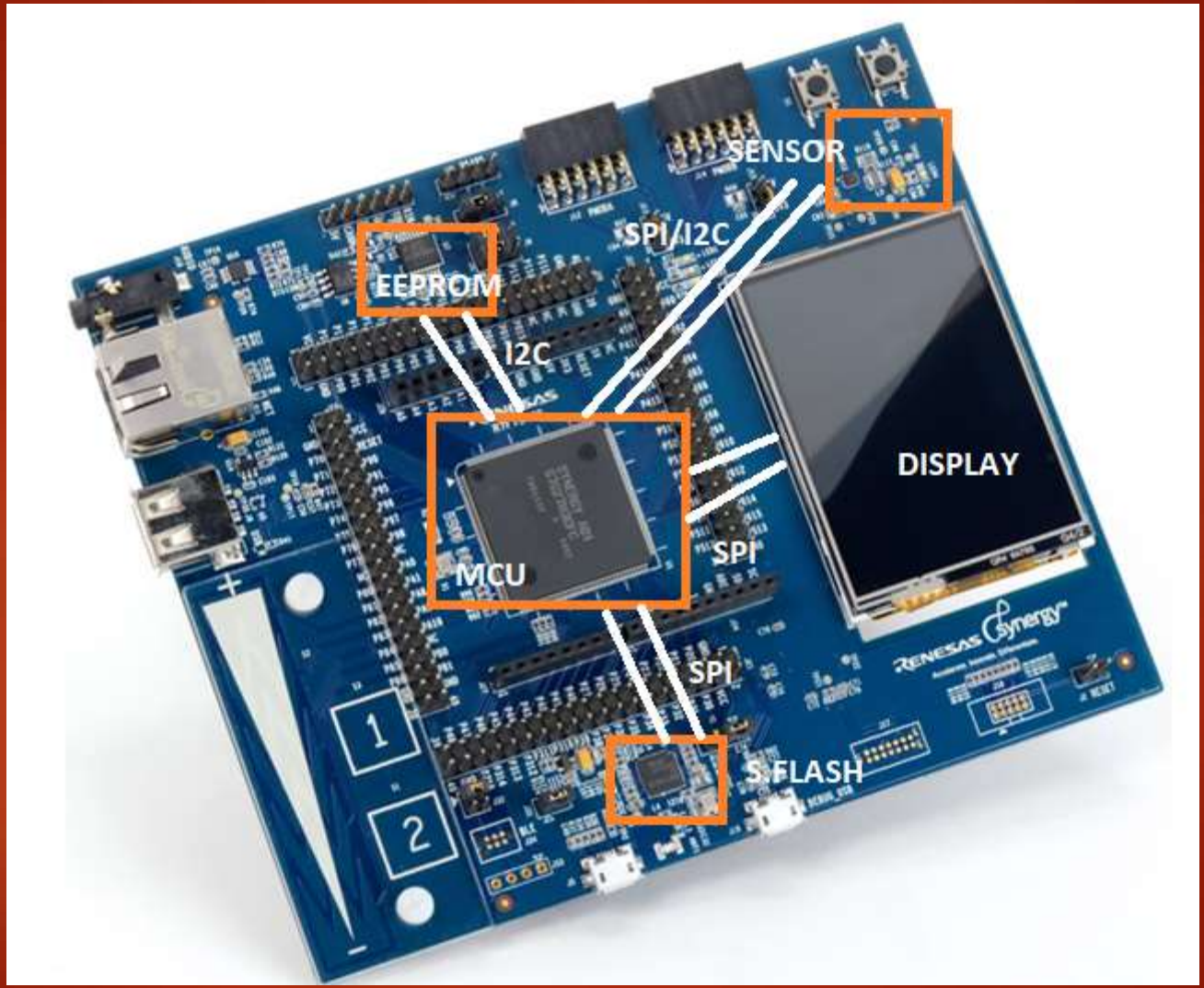
The SPI bus allows the communication between one master device and one or more slave devices.

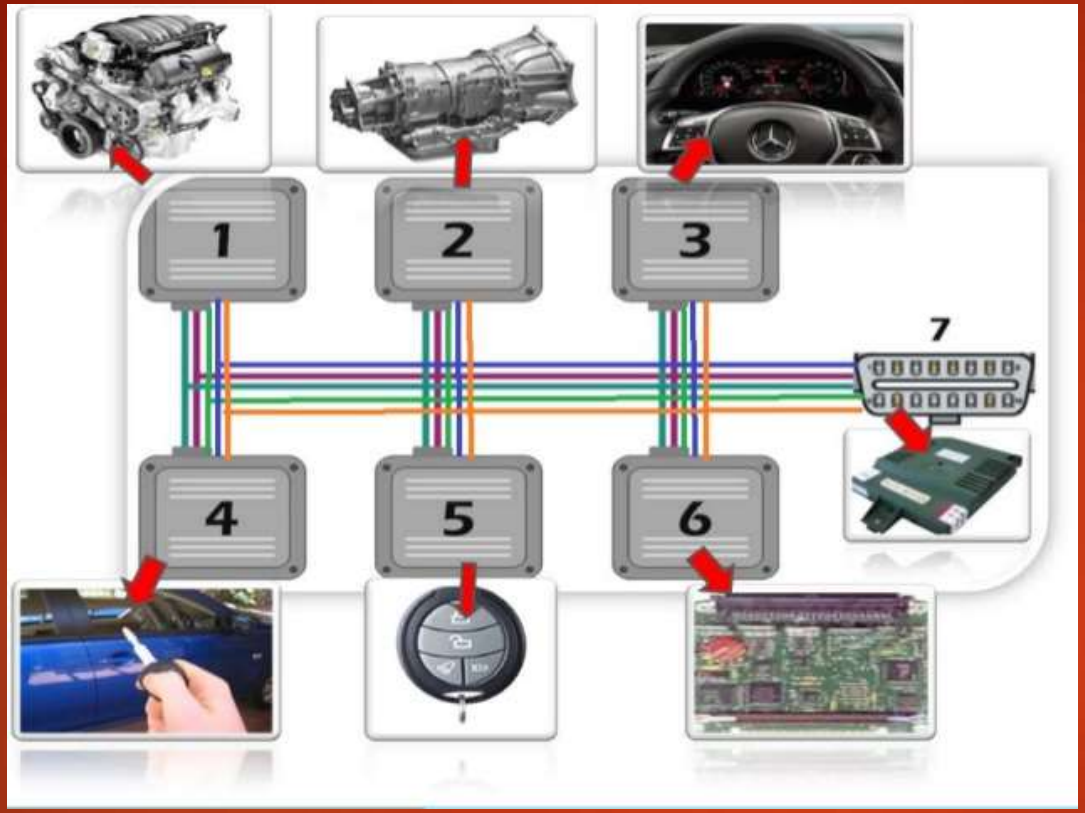


1. Four I/O pins are dedicated to SPI communication with external devices.
2. MISO: Master In / Slave Out data. In the general case, this pin is used to transmit data in slave mode and receive data in master mode
3. MOSI: Master Out / Slave In data. In the general case, this pin is used to transmit data 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

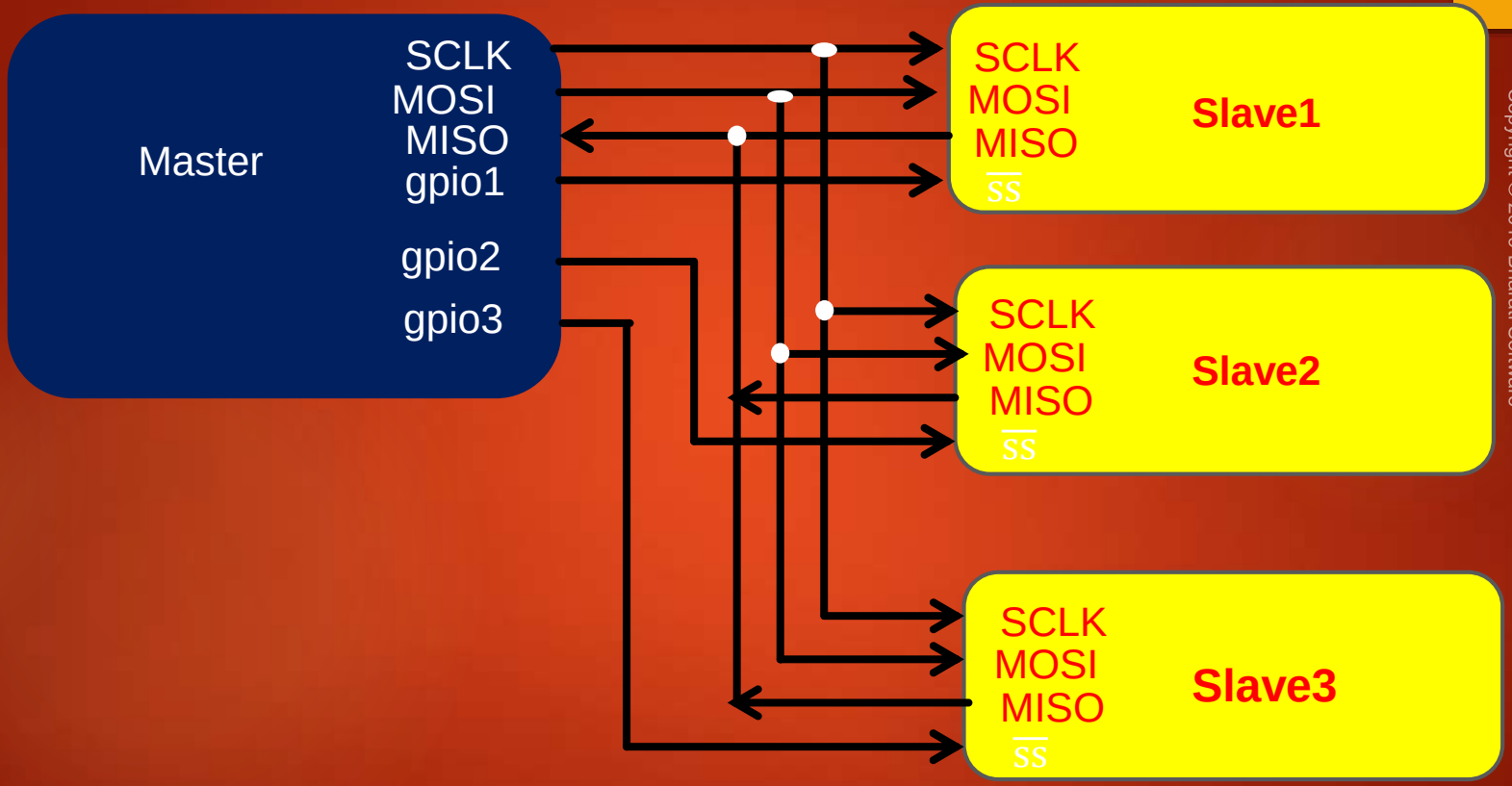


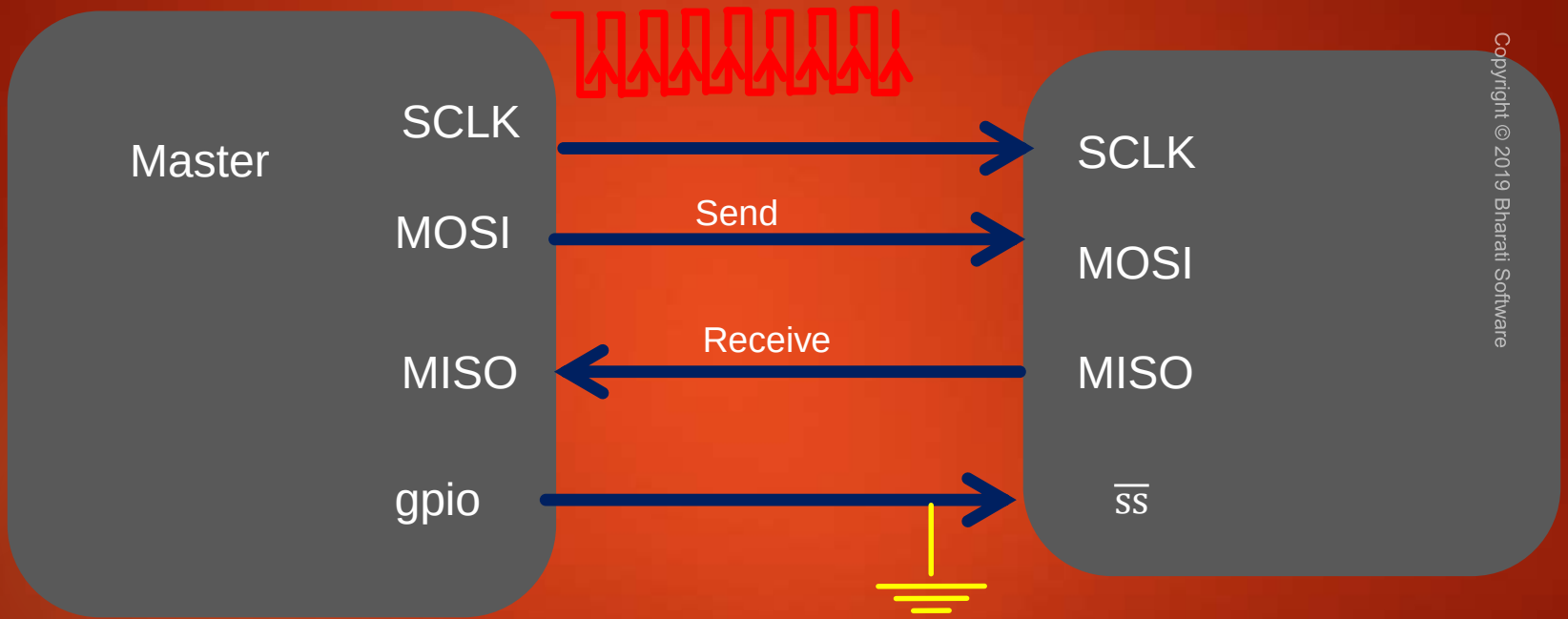
Protocol	Type	Max distance(ft.)	Max Speed (bps)	Typical usage
USB 3.0	dual simplex serial	9 (typical) (up to 49 with 5 hubs)	5 G	Mass storage, video
USB 2.0	half duplex serial	16 (98 ft. with 5 hubs)	1.5M, 12M, 480M	Keyboard, mouse, drive, speakers, printer, camera
Ethernet	serial	1600	10G	network communications
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 systemsSPI
SPI	synchronous serial	10	fPCLK/2	





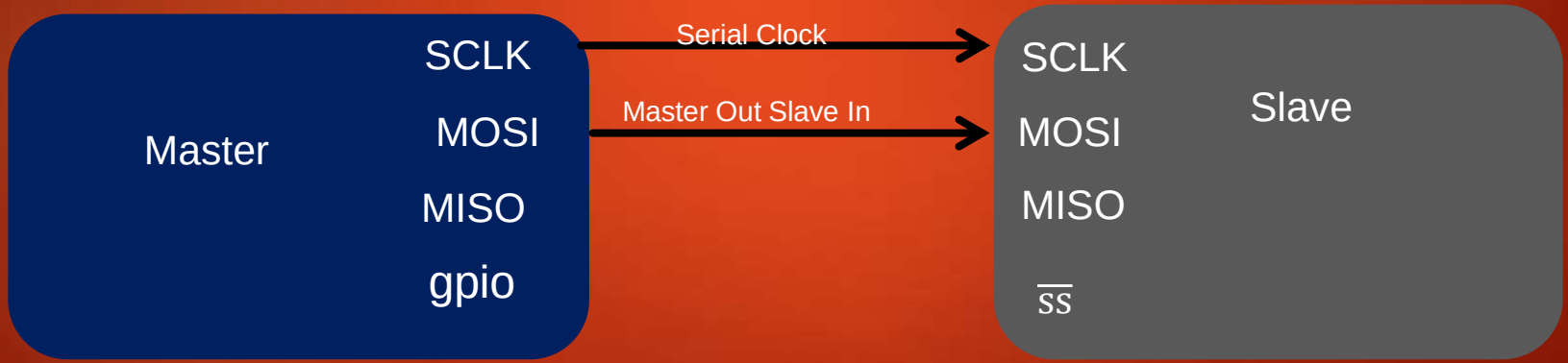
You can use CAN, ETHERNET , RS485, RS232 or combination of them , when you have to cover larger distances and want to achieve better quality of service.





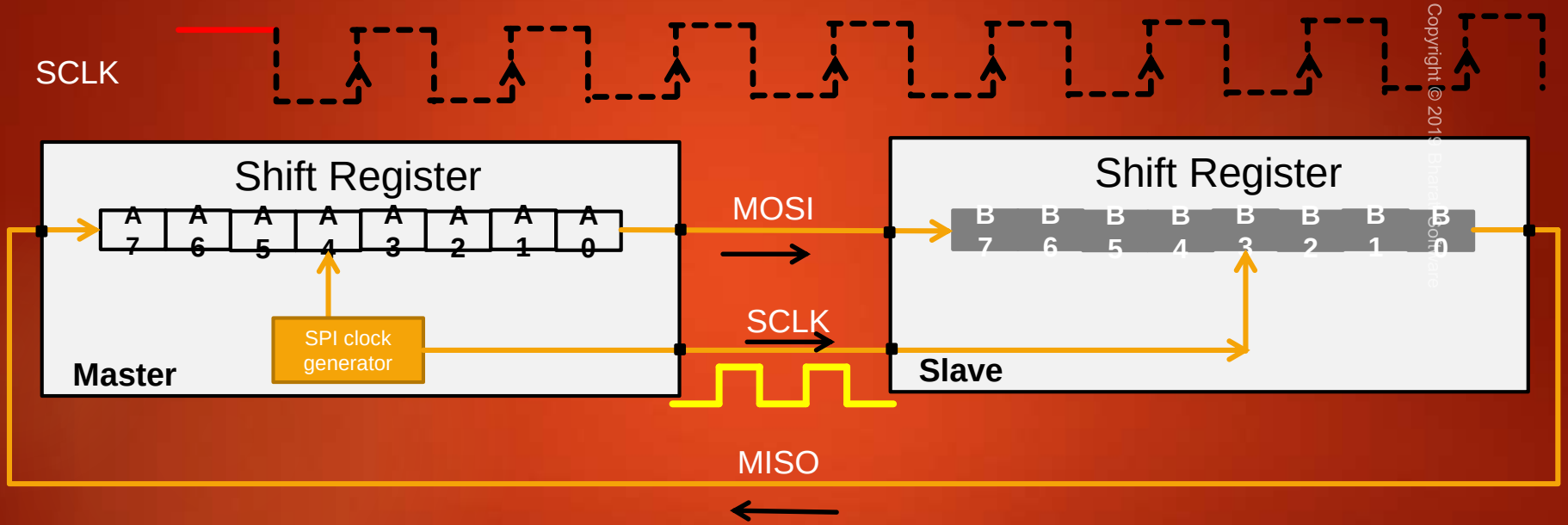
MINIMAL SPI BUS

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.

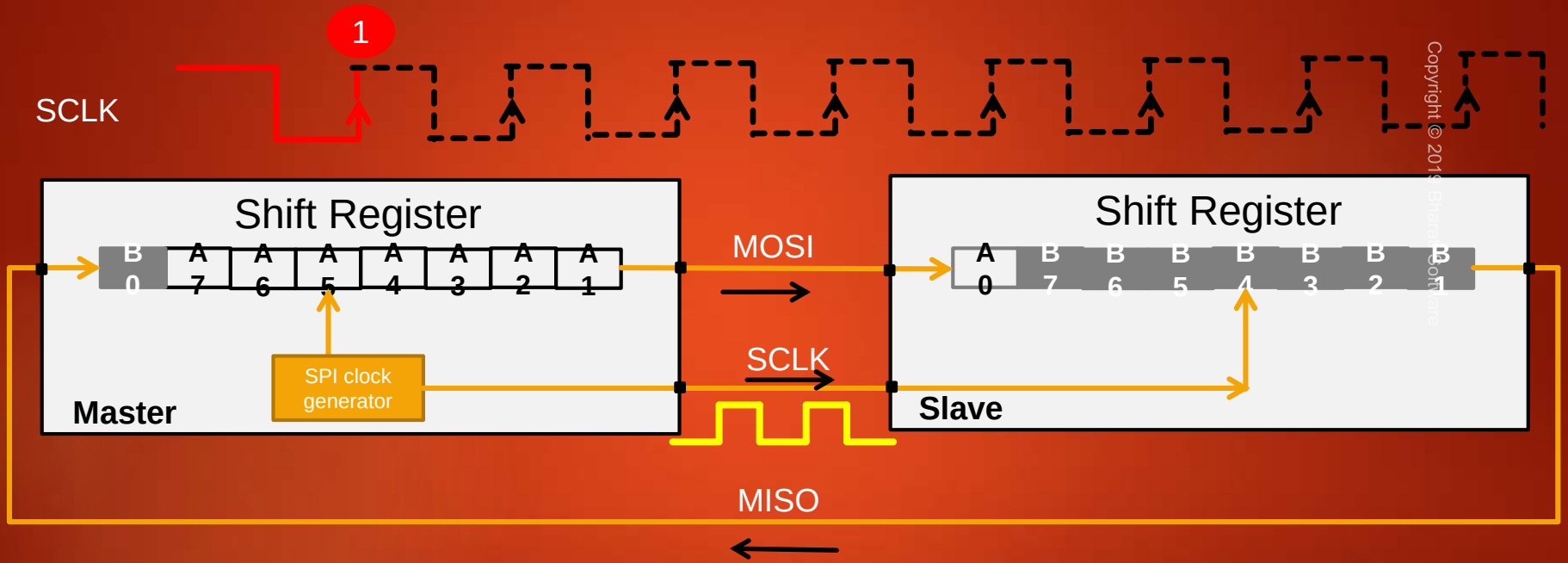


SPI Hardware :Behind the scenes

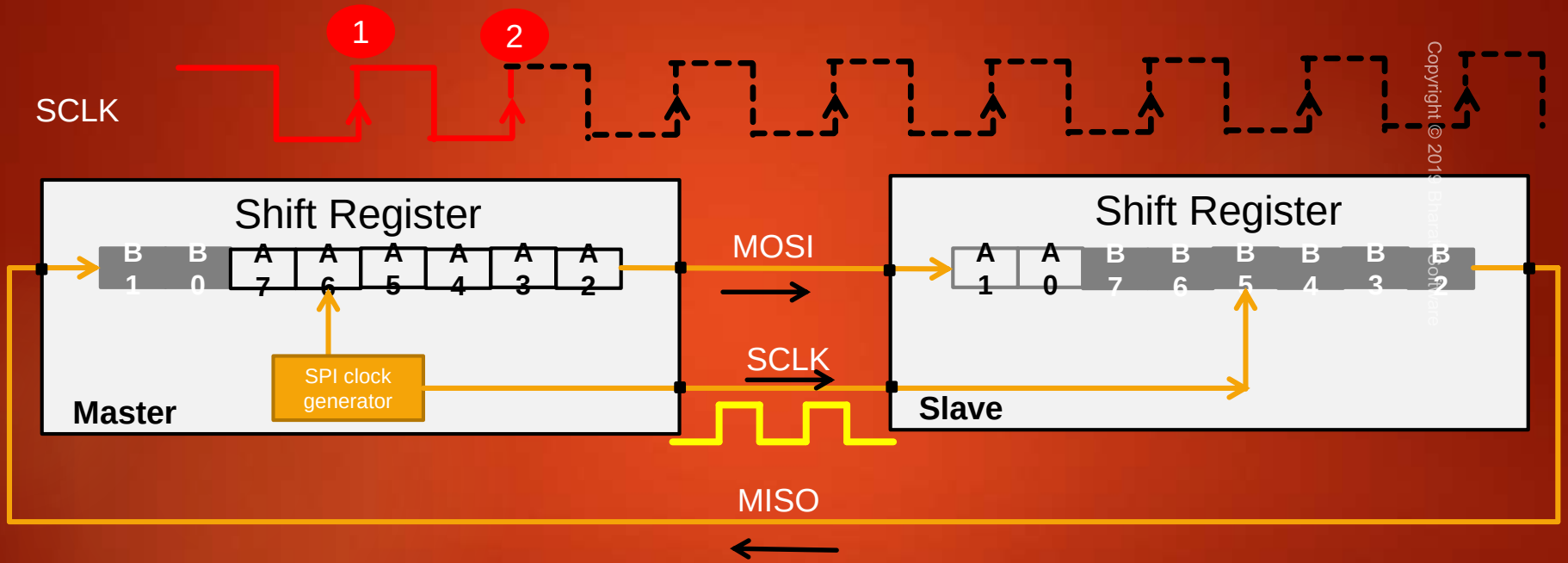
SPI Hardware :Behind the scenes



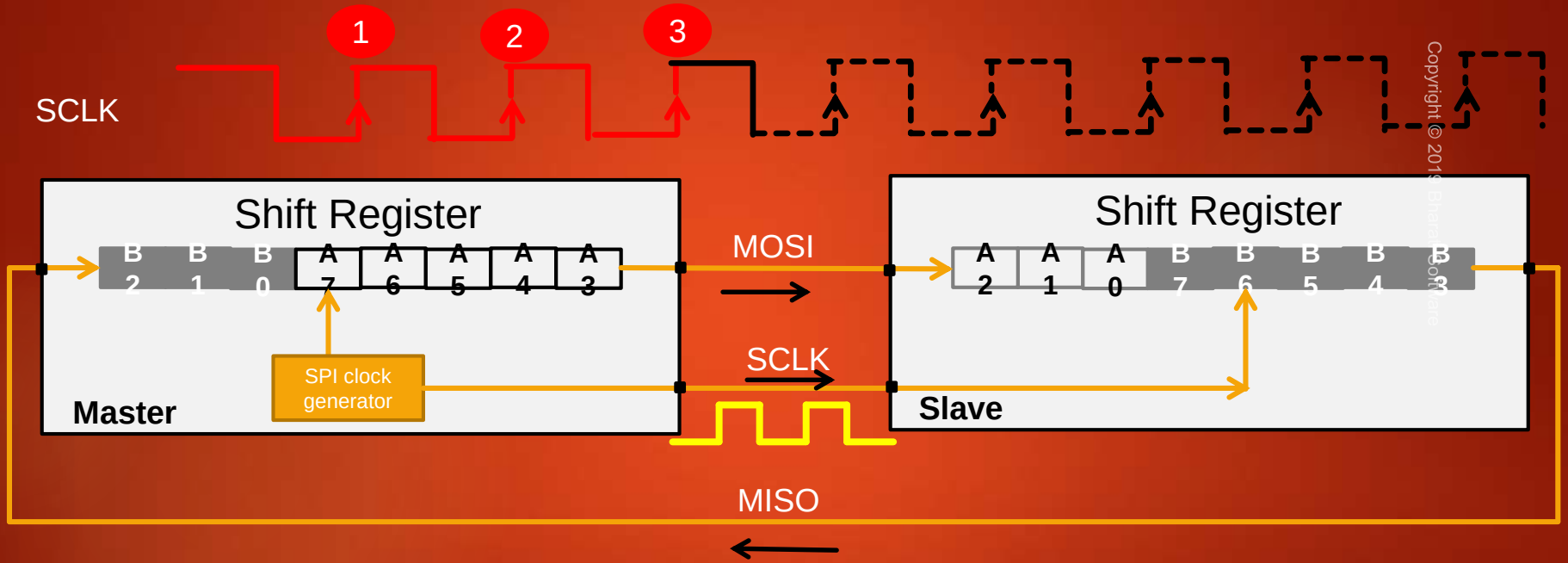
SPI Hardware :Behind the scenes



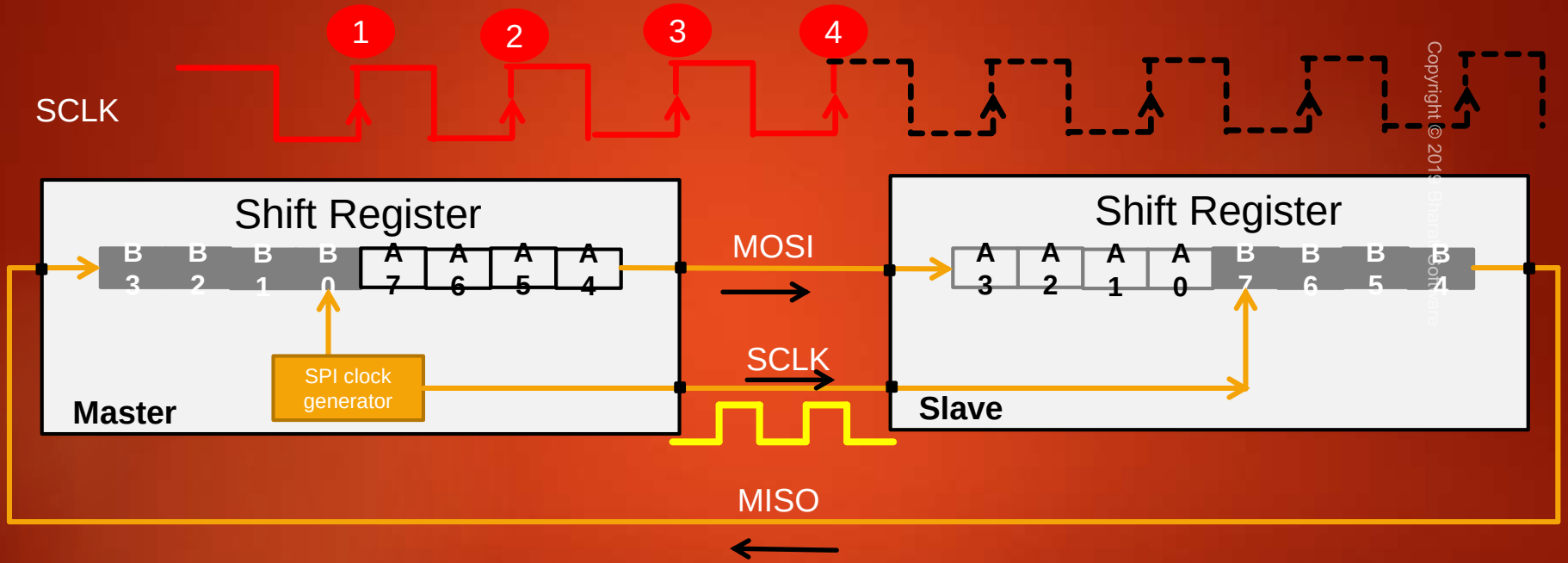
SPI Hardware :Behind the scenes



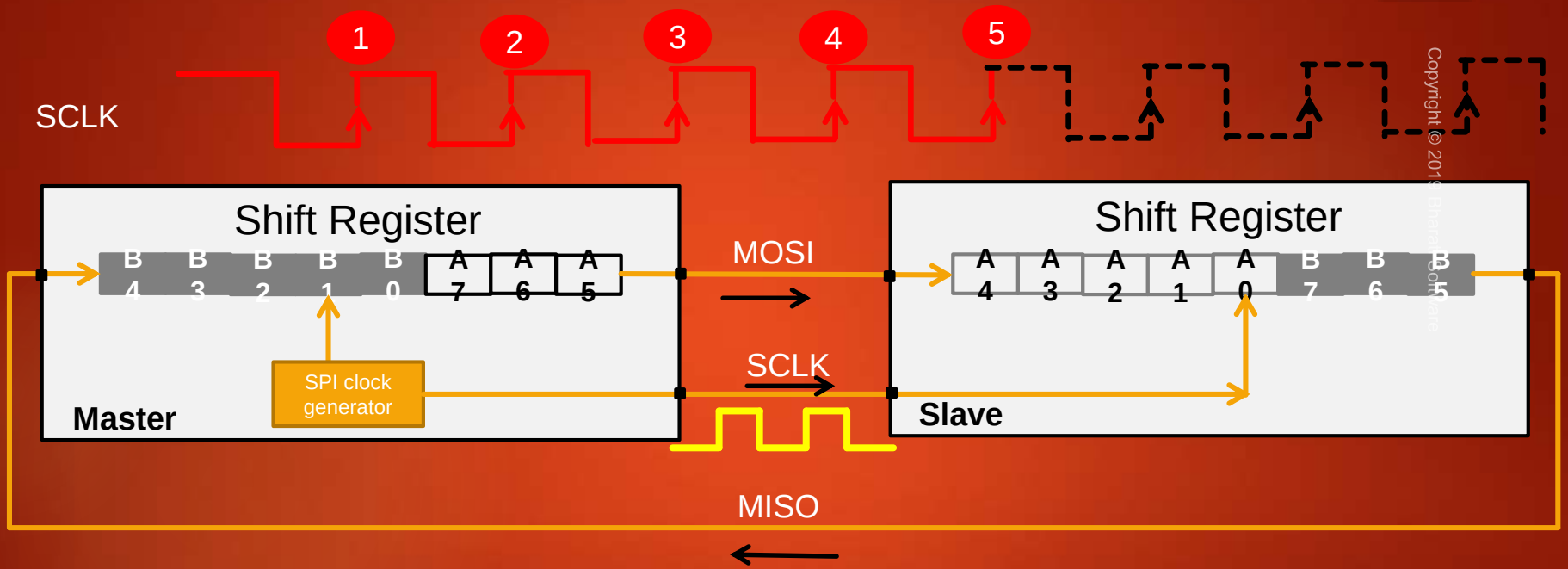
SPI Hardware :Behind the scenes



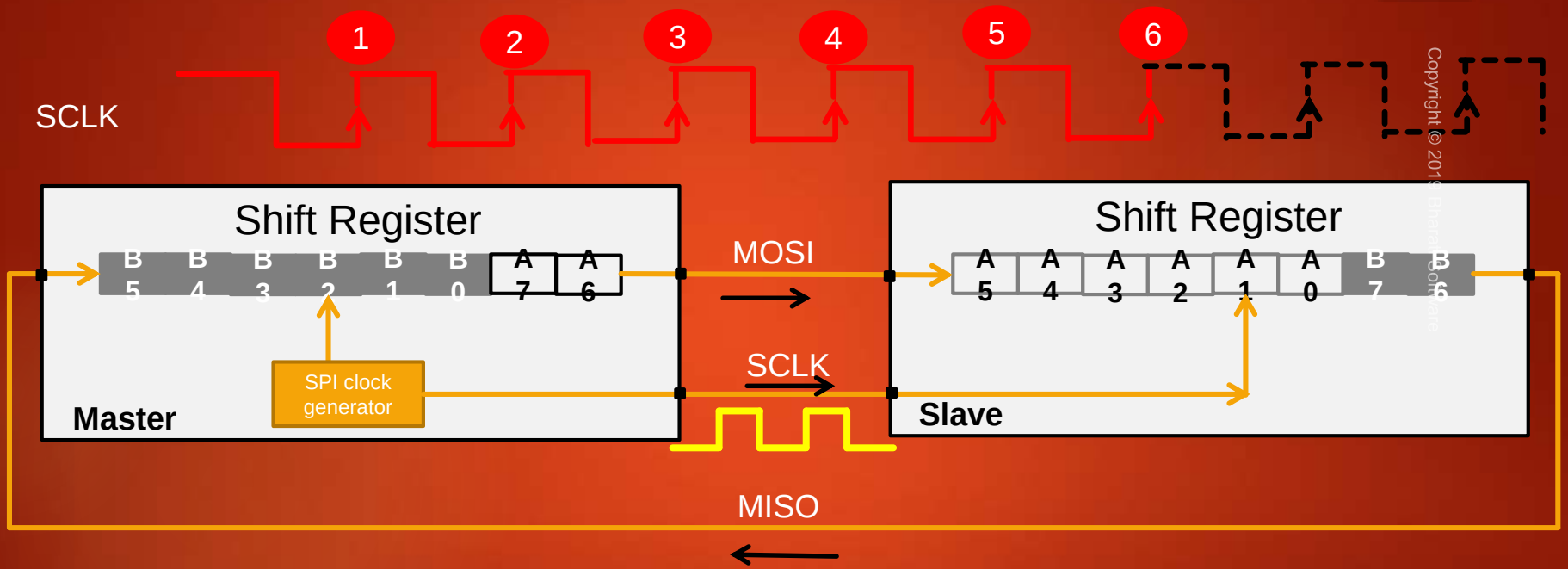
SPI Hardware :Behind the scenes



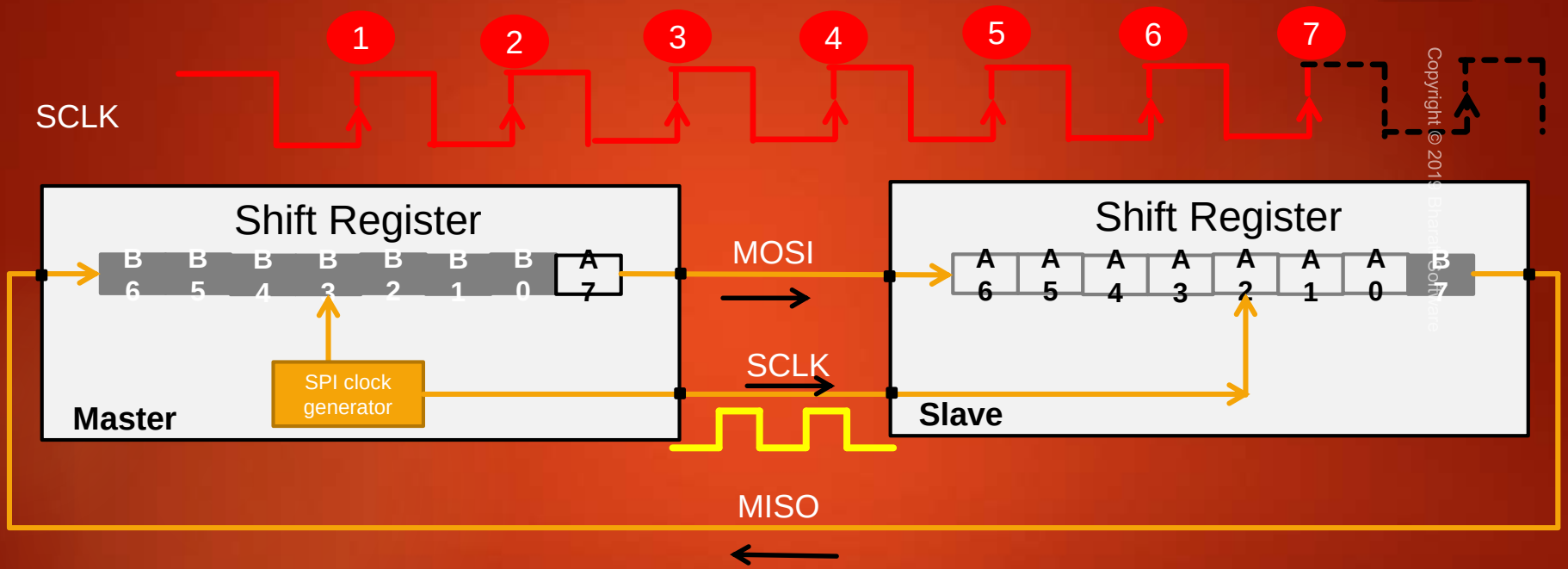
SPI Hardware :Behind the scenes



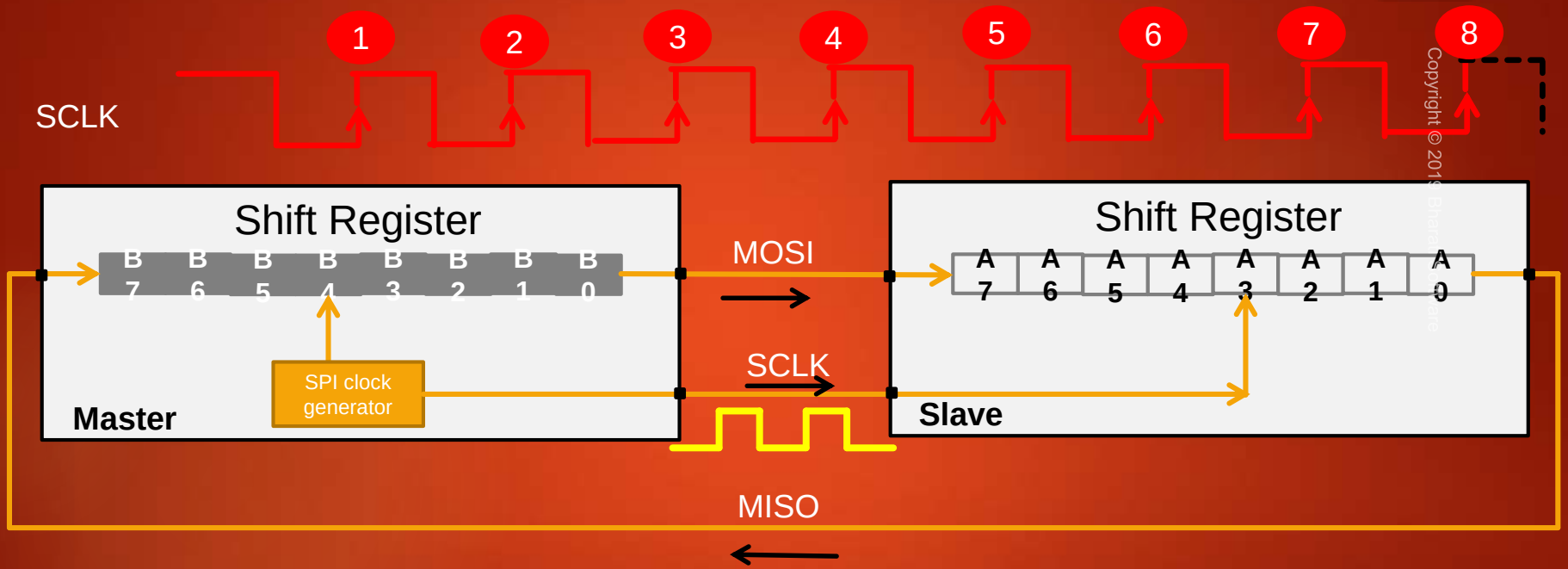
SPI Hardware :Behind the scenes



SPI Hardware :Behind the scenes



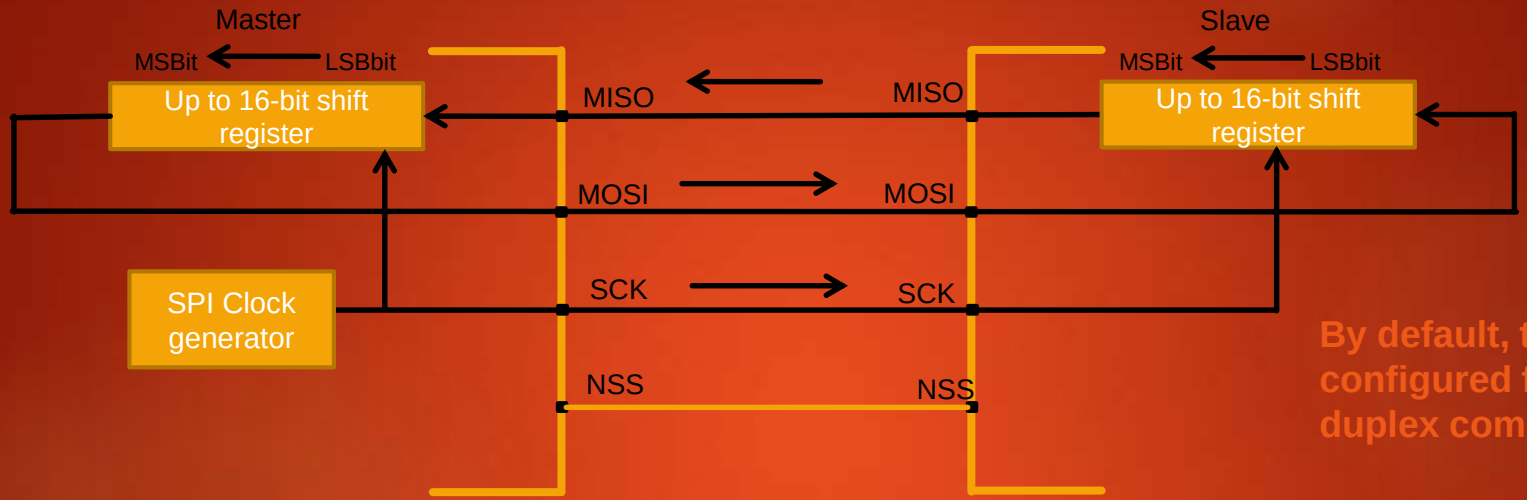
SPI Hardware :Behind the scenes



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

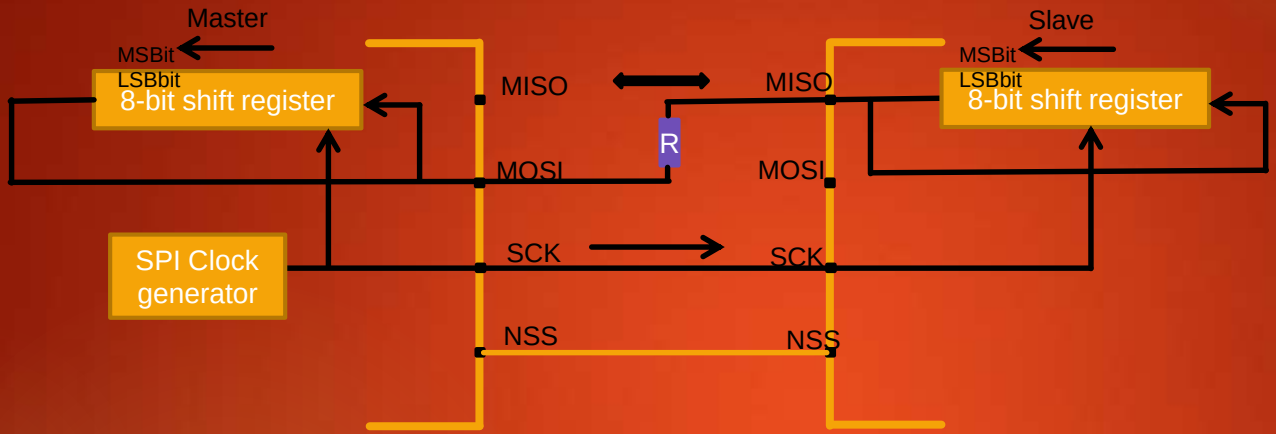


By default, the SPI is configured for full-duplex communication

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.

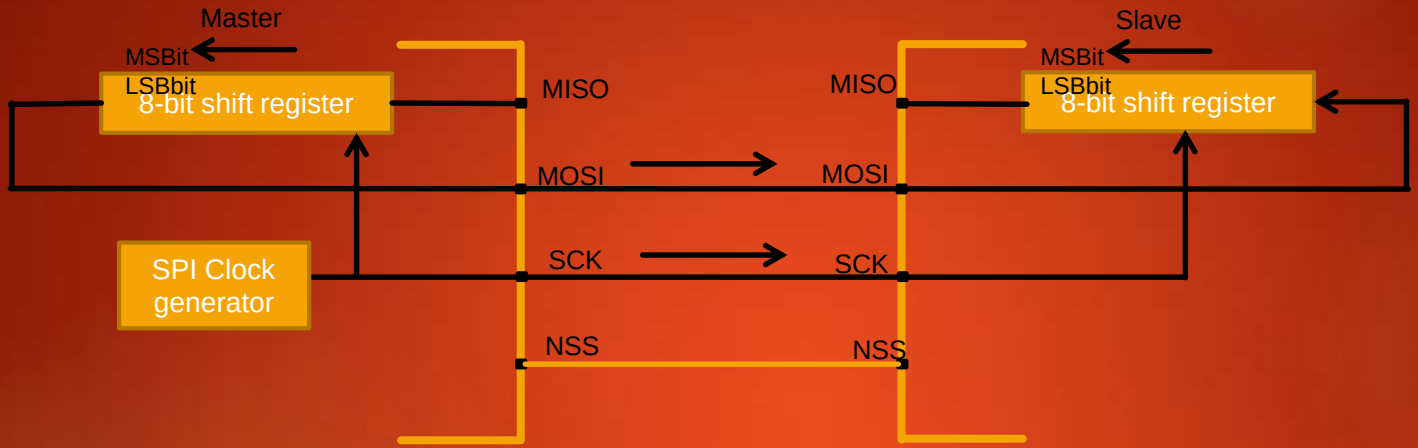
Remember that in SPI communication ,slave will not initiate data transfer unless master produces the clock

HALF -DUPLEX COMMUNICATION



In this configuration, one single cross connection line is used to link the shift registers of the master and slave together. During this communication, the data is synchronously shifted between the shift registers on the SCK clock edge in the transfer direction selected reciprocally by both master and slave

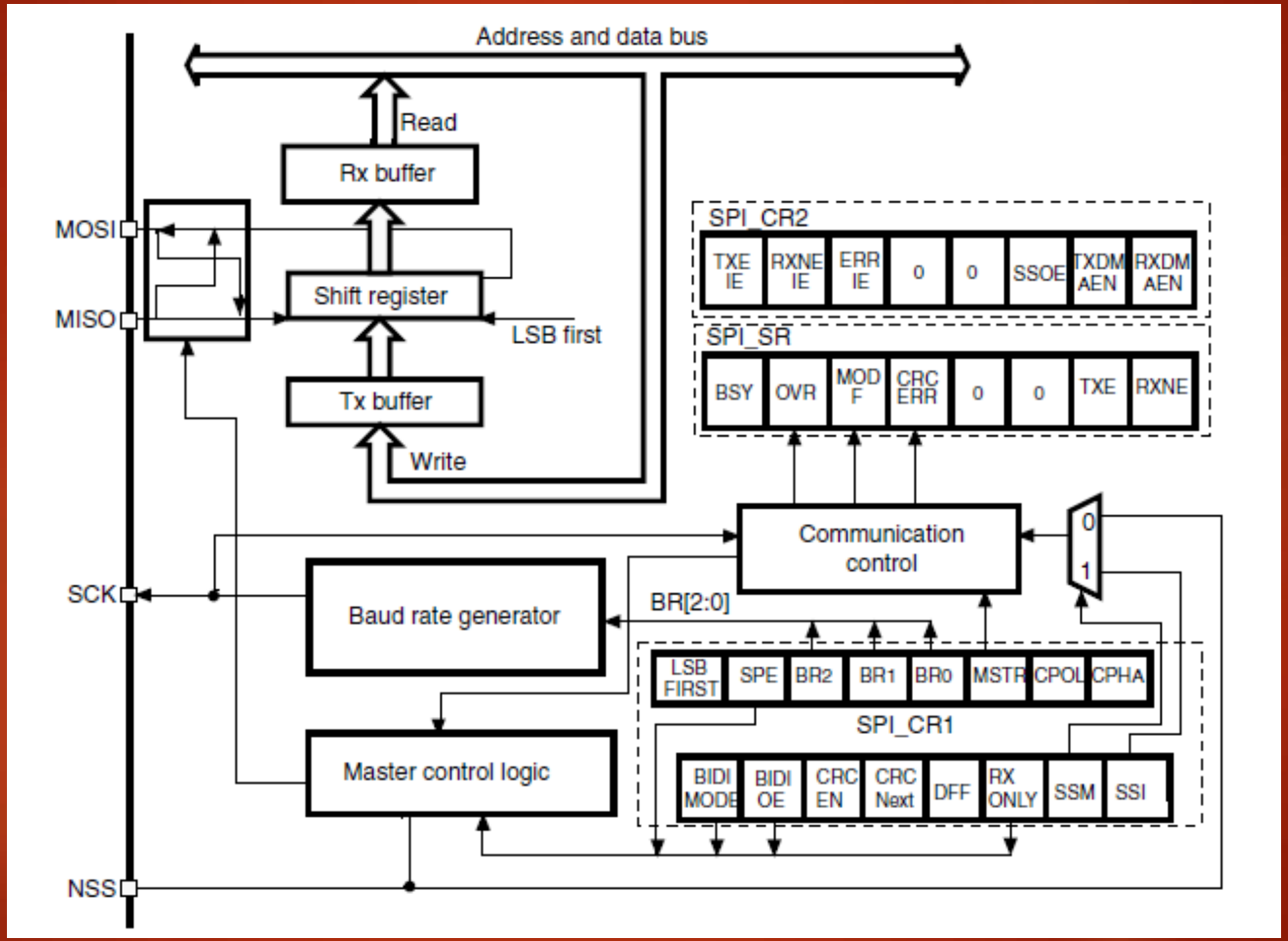
SIMPLEX COMMUNICATION



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



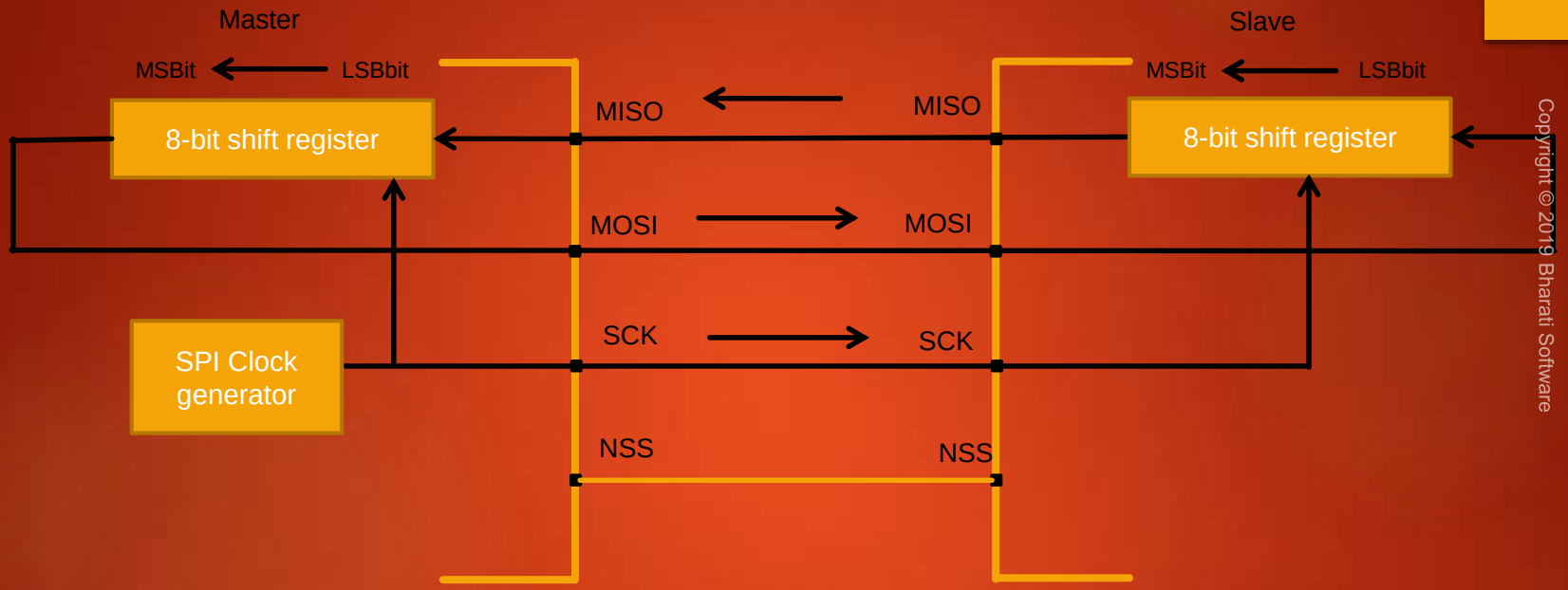
SLAVE SELECT (NSS) PIN MANAGEMENT

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.

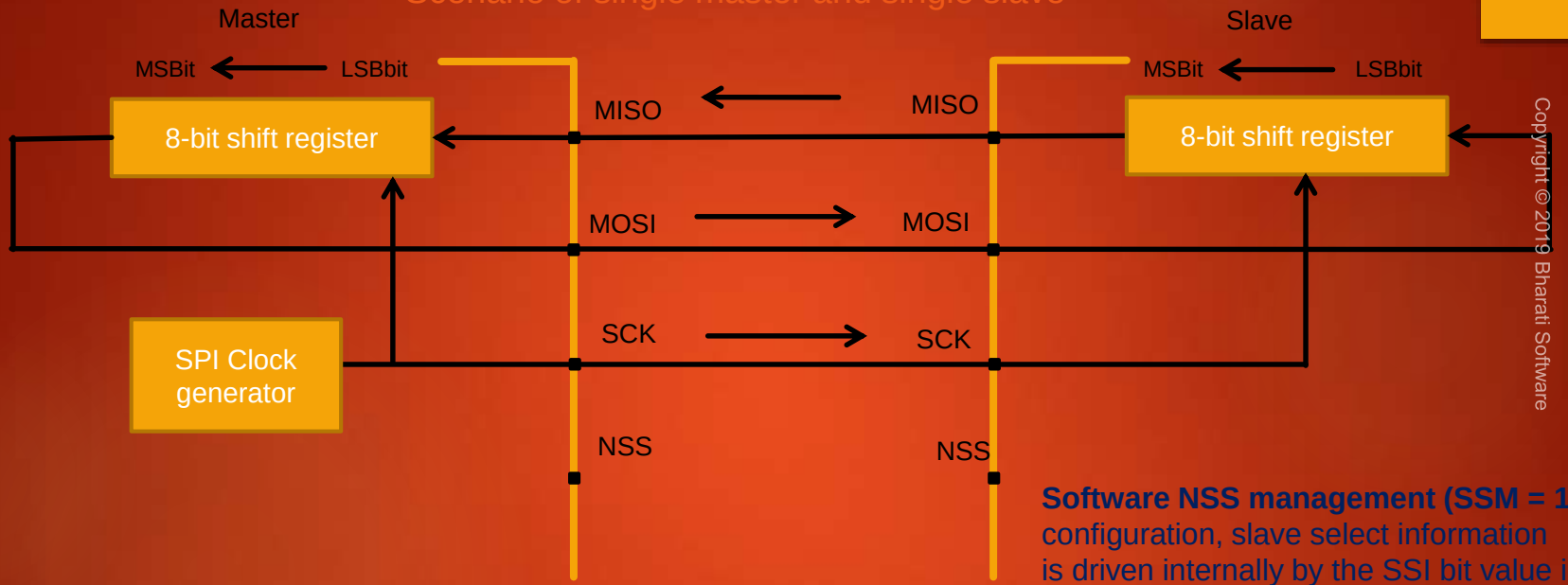


Scenario of single master and single slave

2 Types of slave managements

Hardware slave management
Software slave management

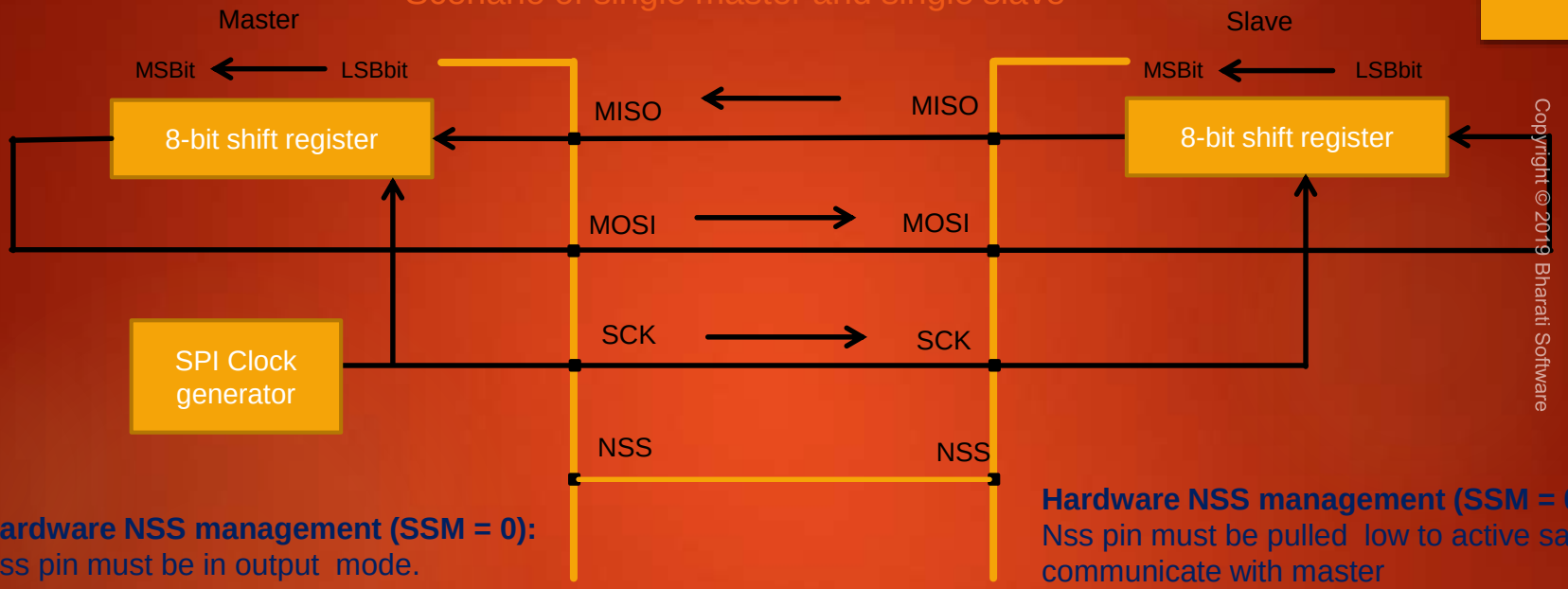
Scenario of single master and single slave



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.

Hardware or software slave select management can be set using the SSM bit in the SPIx_CR1 register:

Scenario of single master and single slave

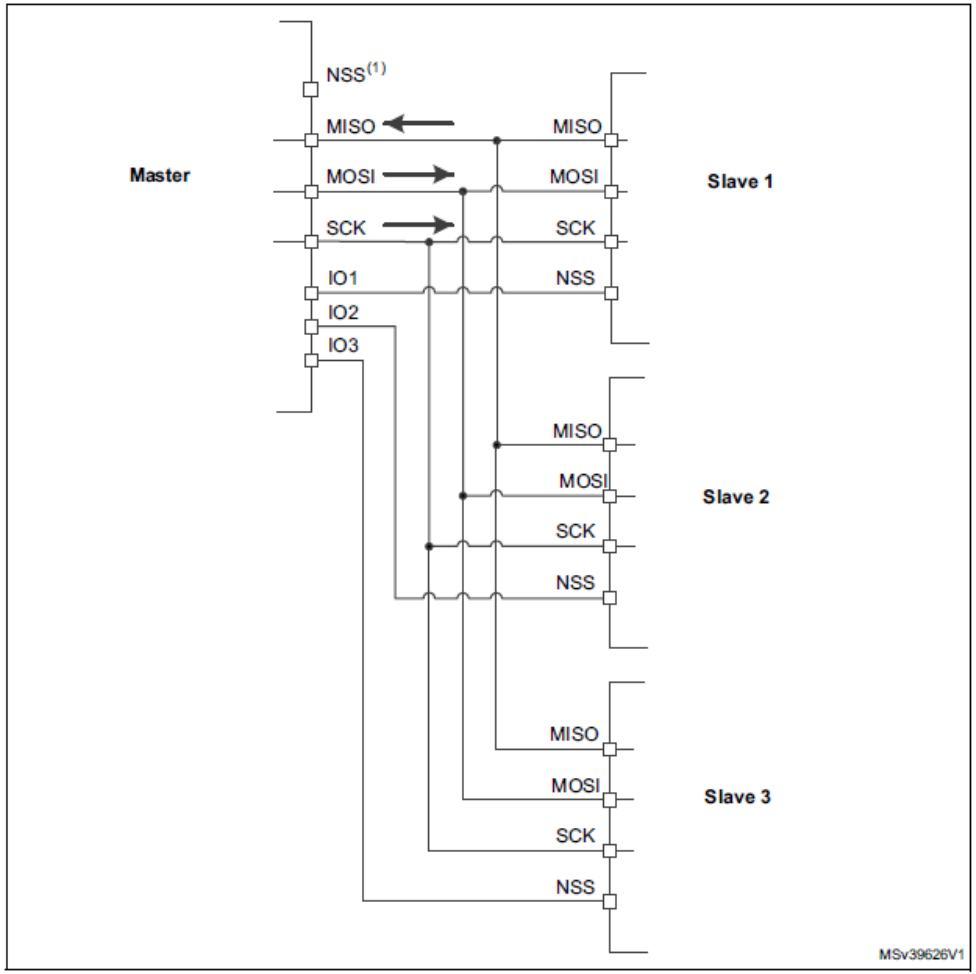


Hardware NSS management (SSM = 0):
Nss pin must be in output mode.
The NSS pin is managed by the hardware

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

Hardware or software slave select management can be set using the SSM bit in the SPIx_CR1 register:

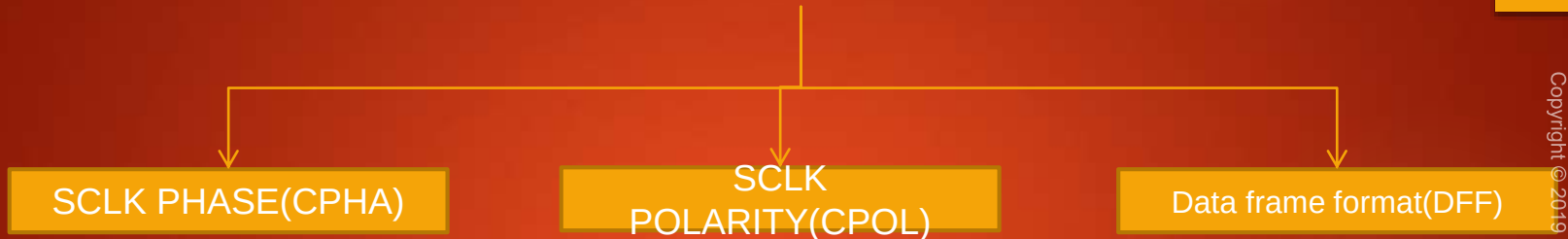
Master and three independent slaves



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

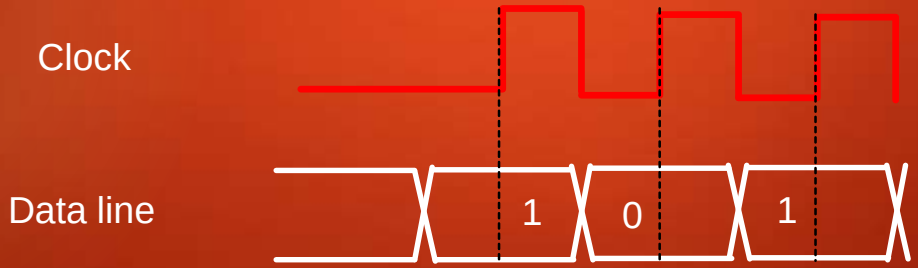
CPOL =1

CPHA(CLOCK PHASE)

- CPHA controls at which clock edge of the SCLK(1st or 2nd) 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.

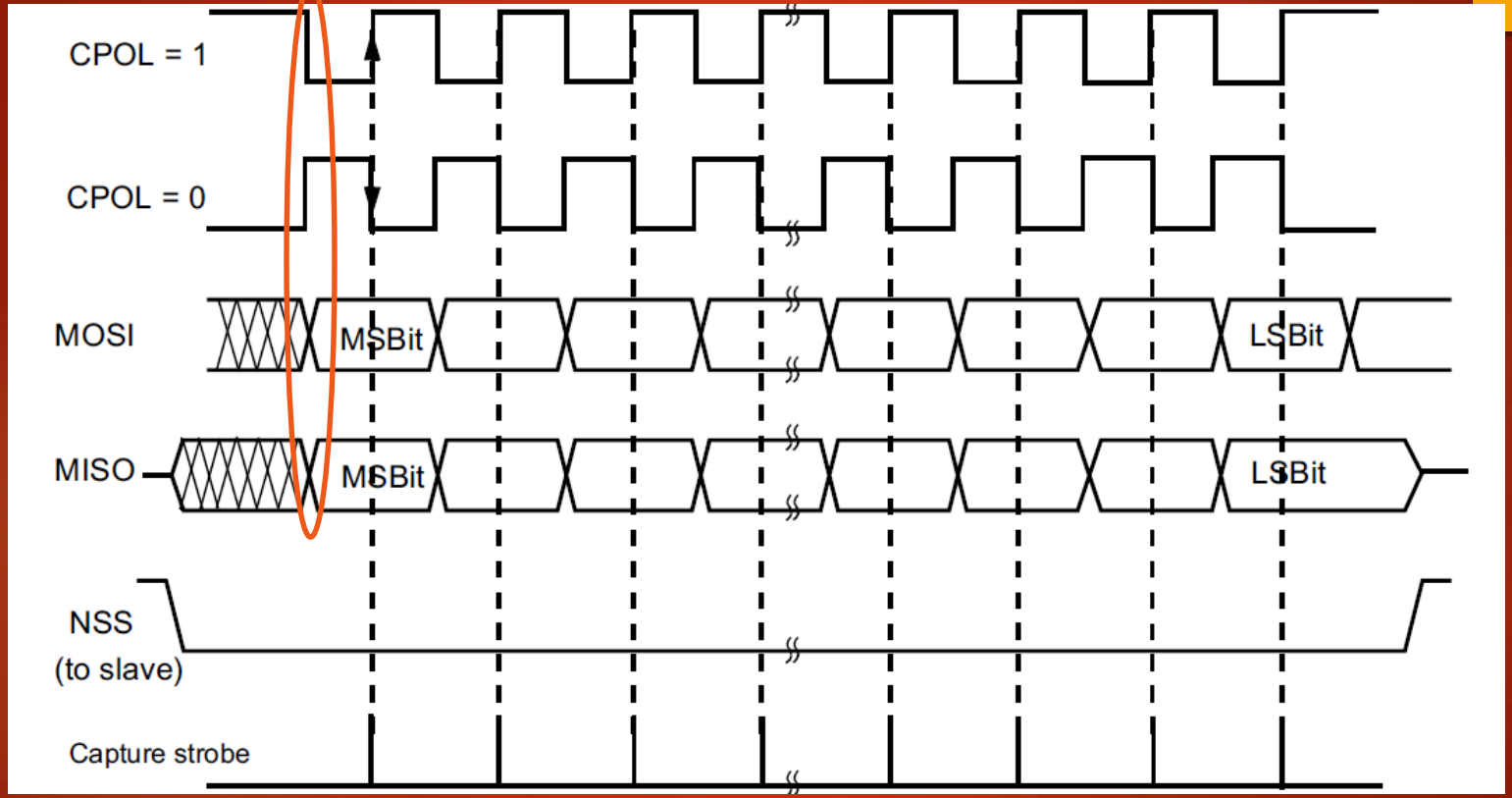
Data toggling means, data transition to the next bit.

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

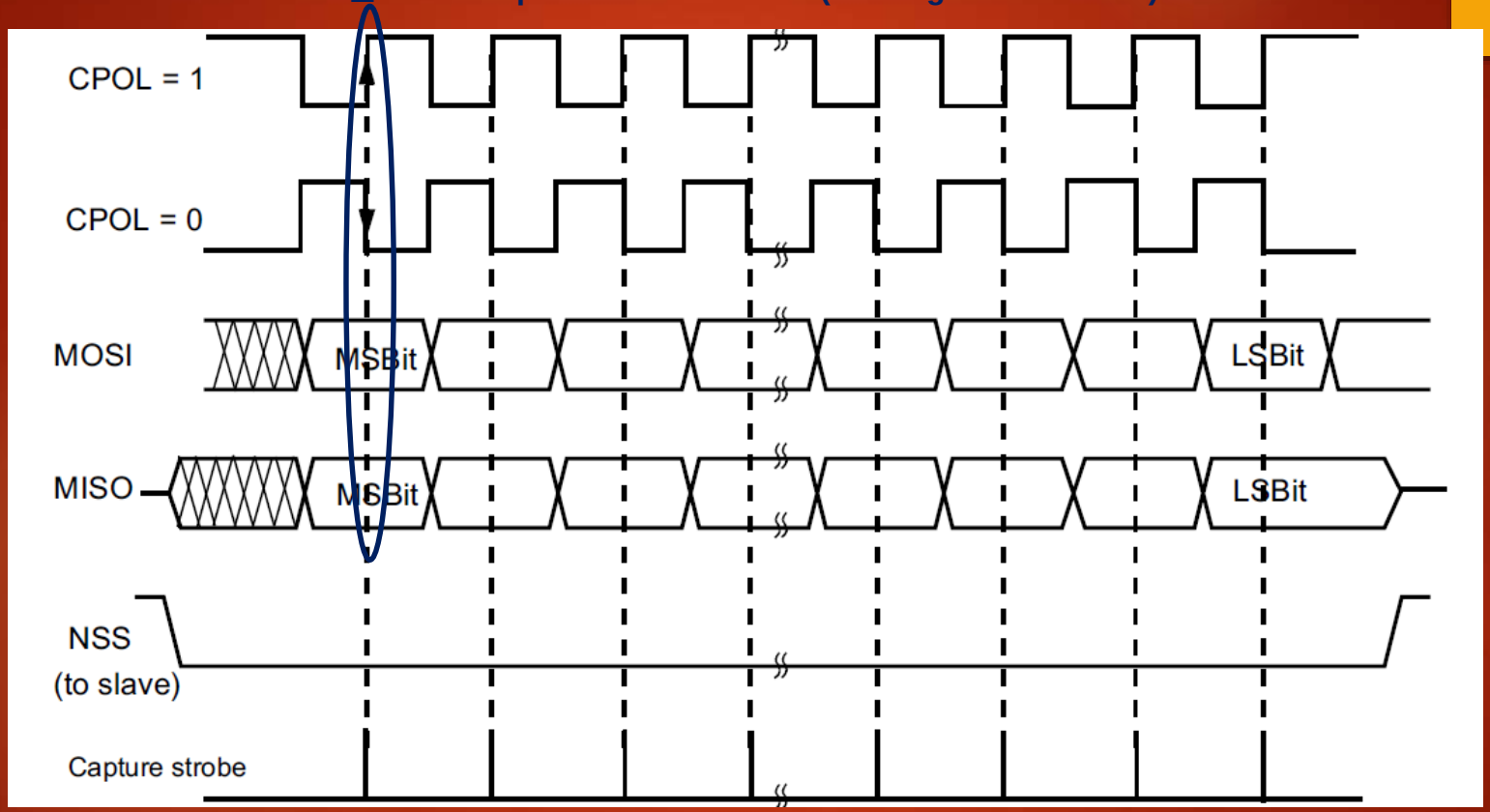


CPHA=1

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



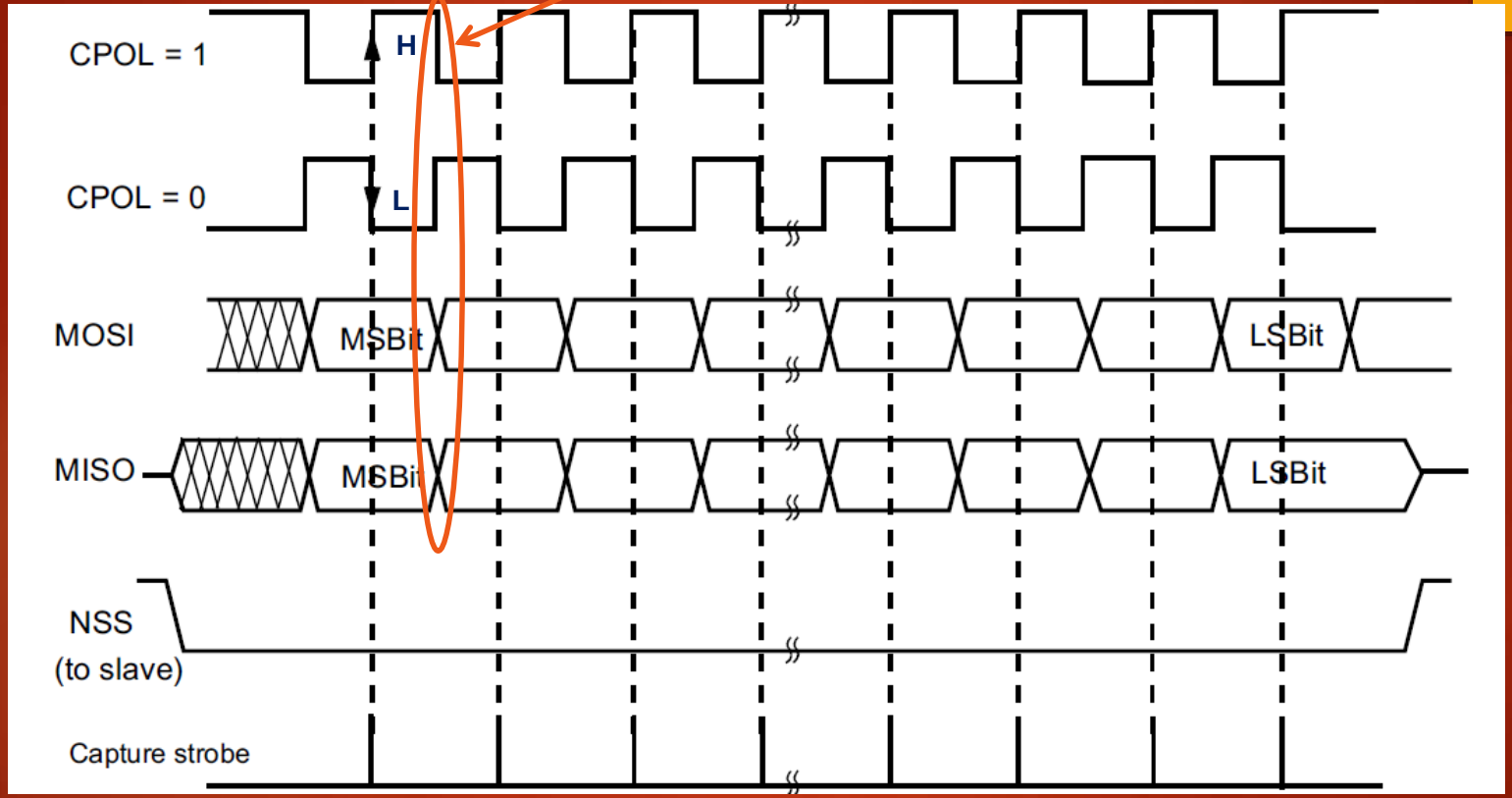
CPHA=1



CPHA=1

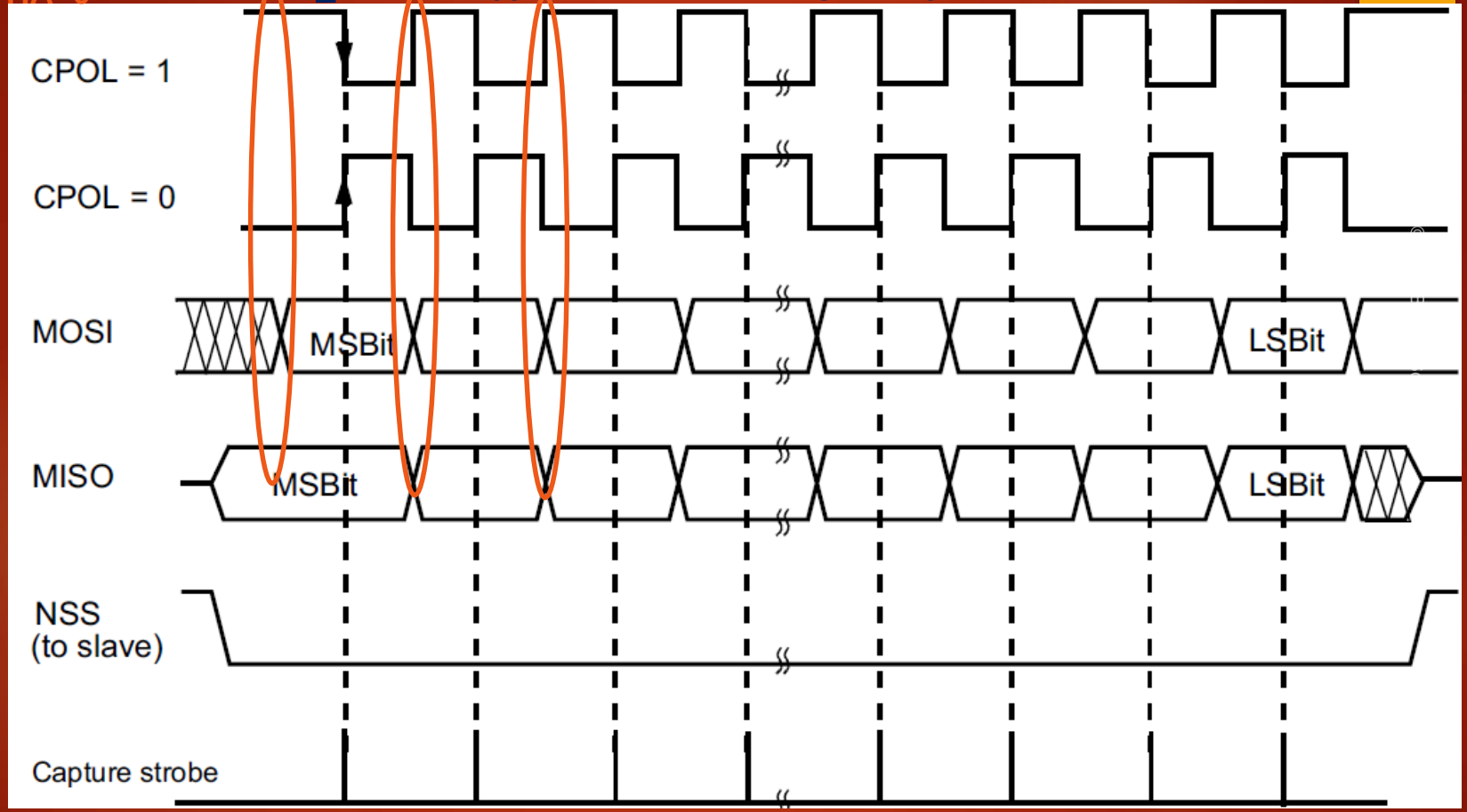
1

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



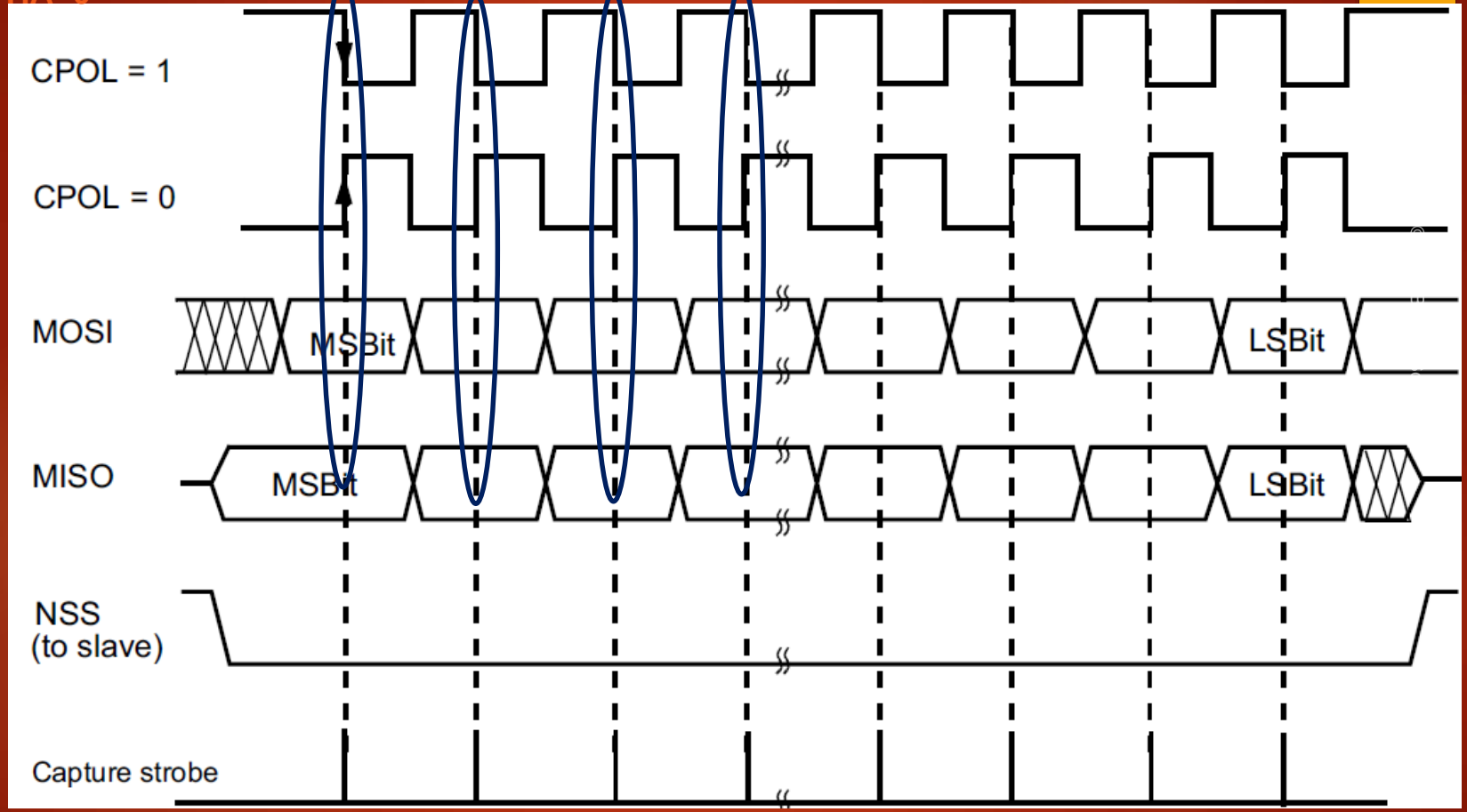
CPHA=0

1 Data will appear on the lines during 2nd edge of the lock



CPHA=0

2 Slave captures the data here (1st edge of the SCLK)



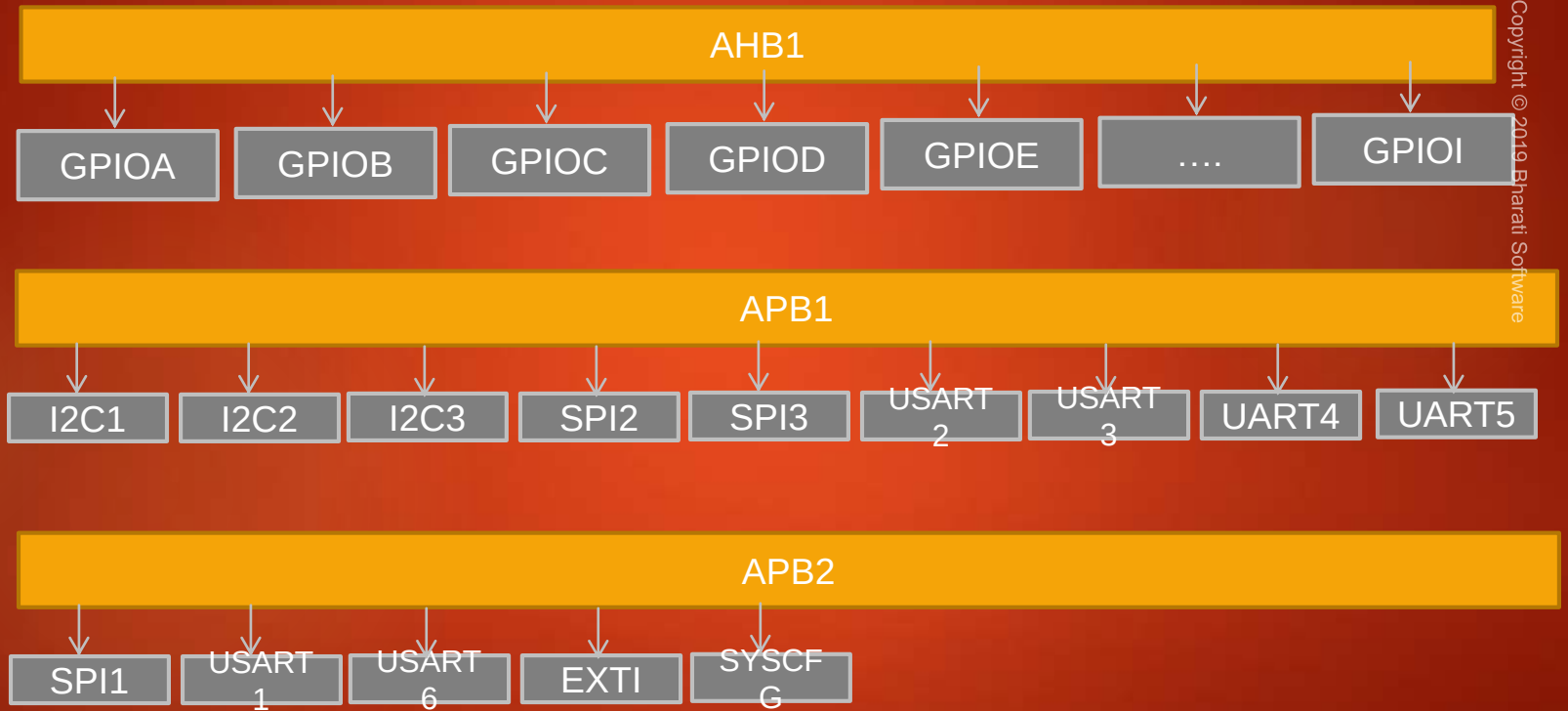
Different SPI Modes

Mode	CPOL	CPHA
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.

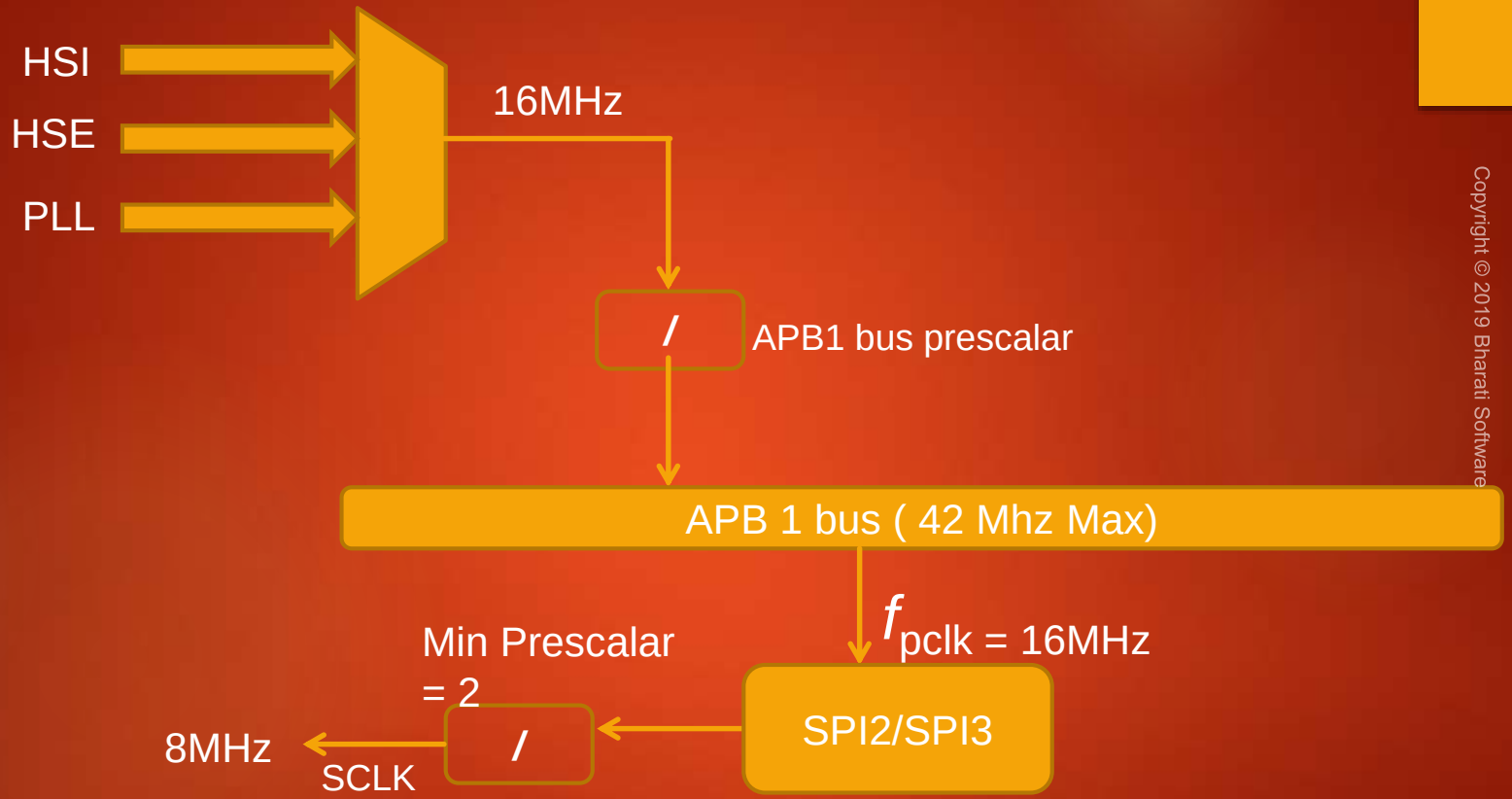
SPI peripherals of your MCU

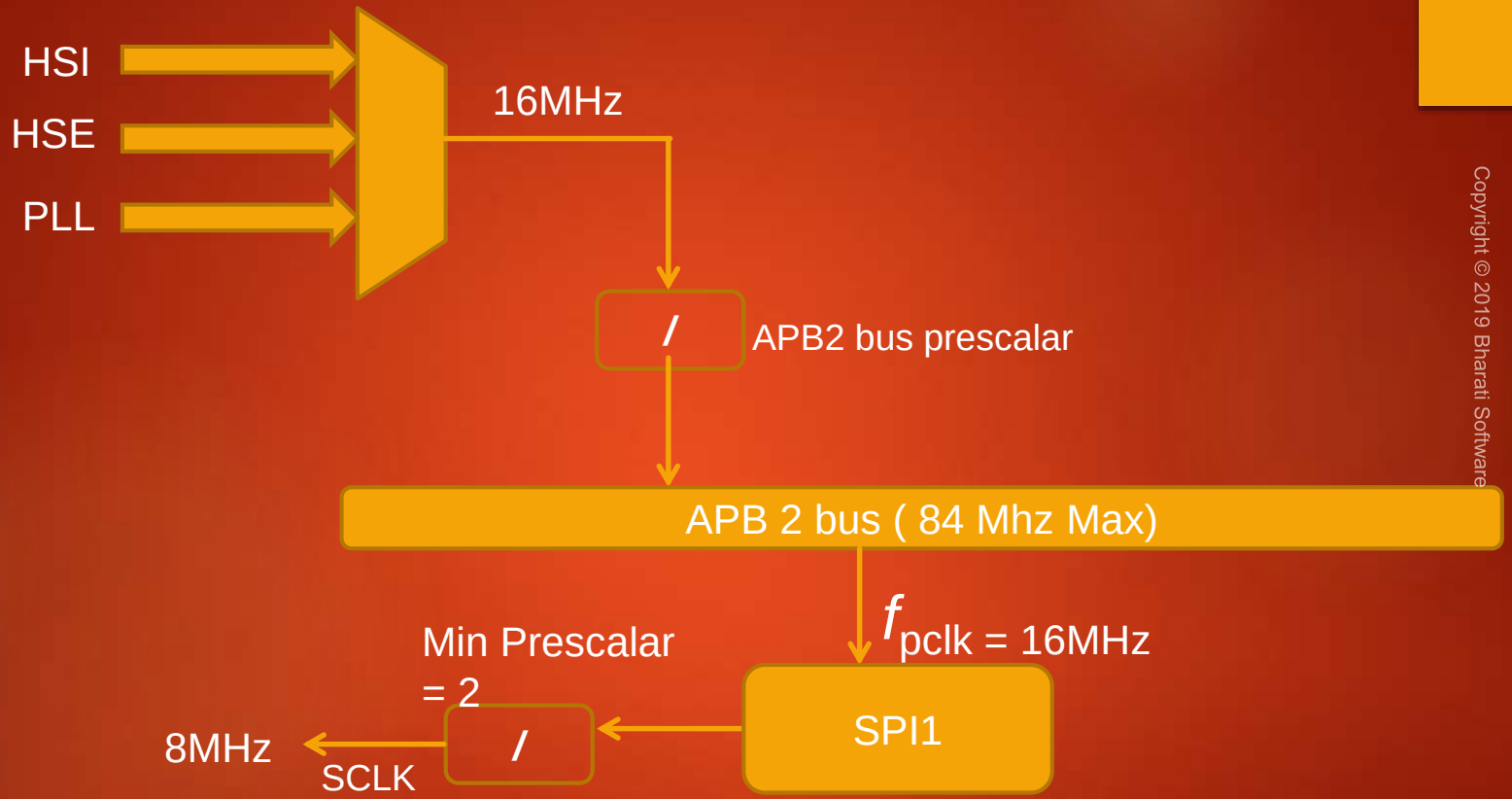


SPI serial clock (SCLK)

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

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



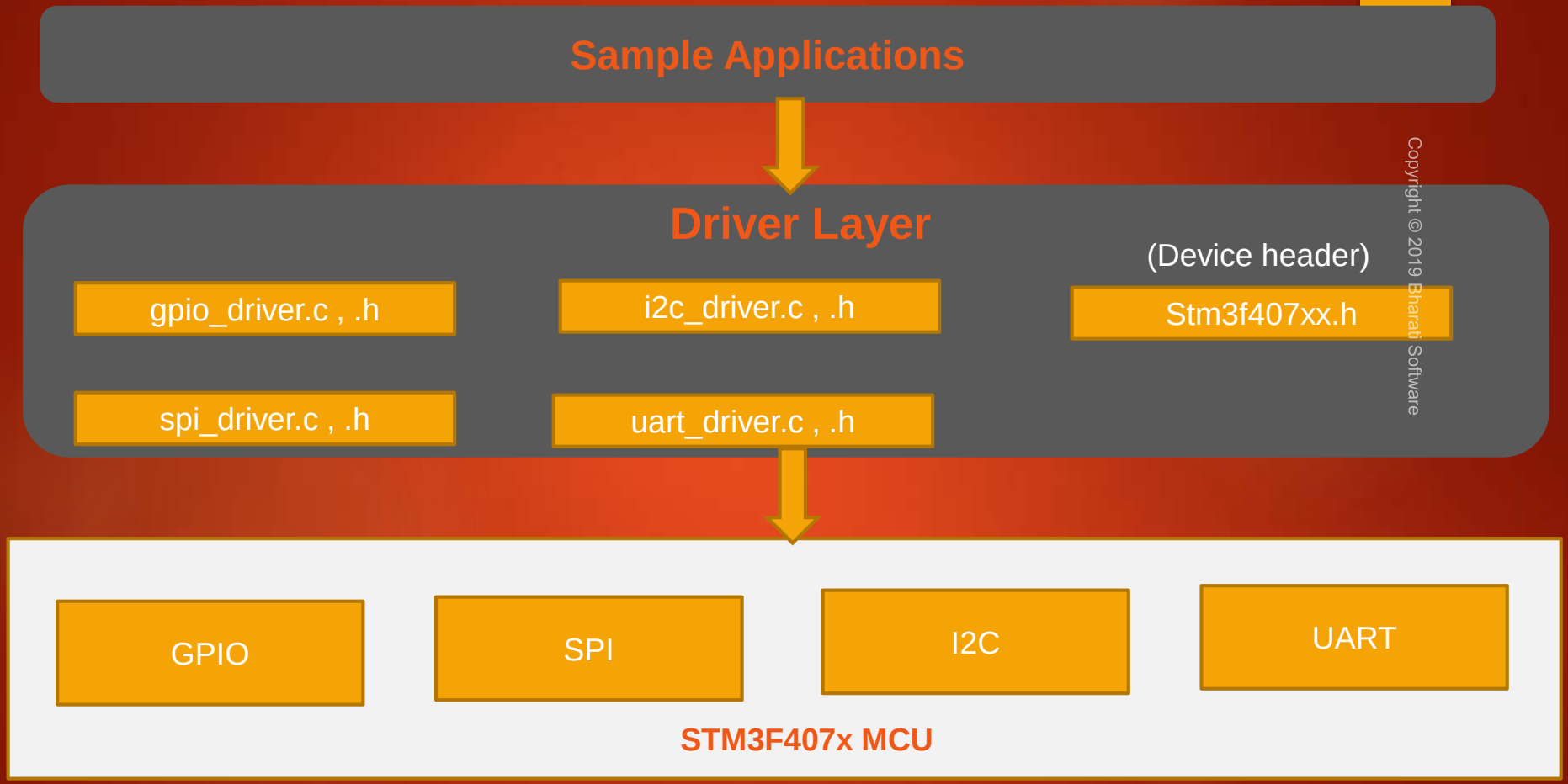


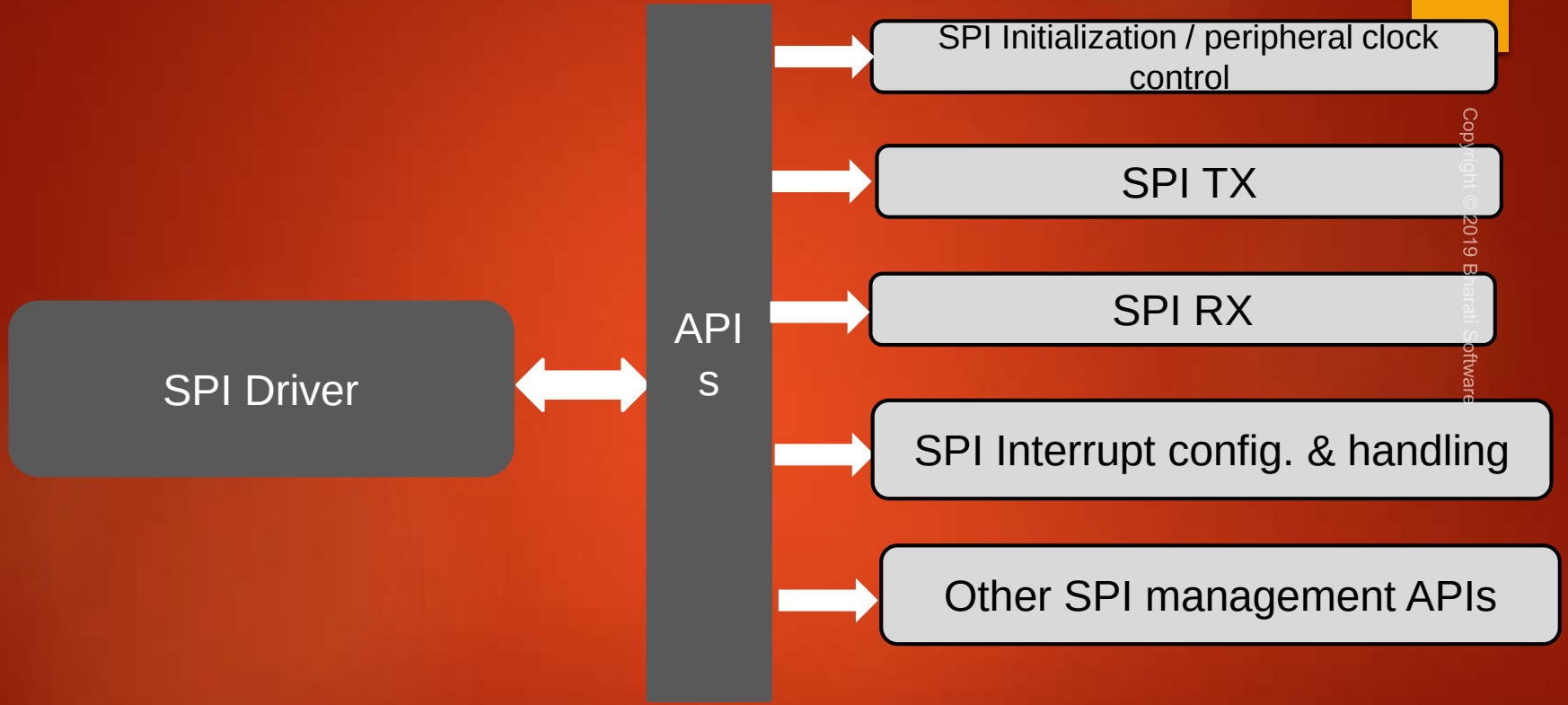
So, if we use the internal RC oscillator of *16Mhz as our system clock* then SPI1/SPI2/SPI3 peripherals can able to produce the serial clock of maximum 8MHz.

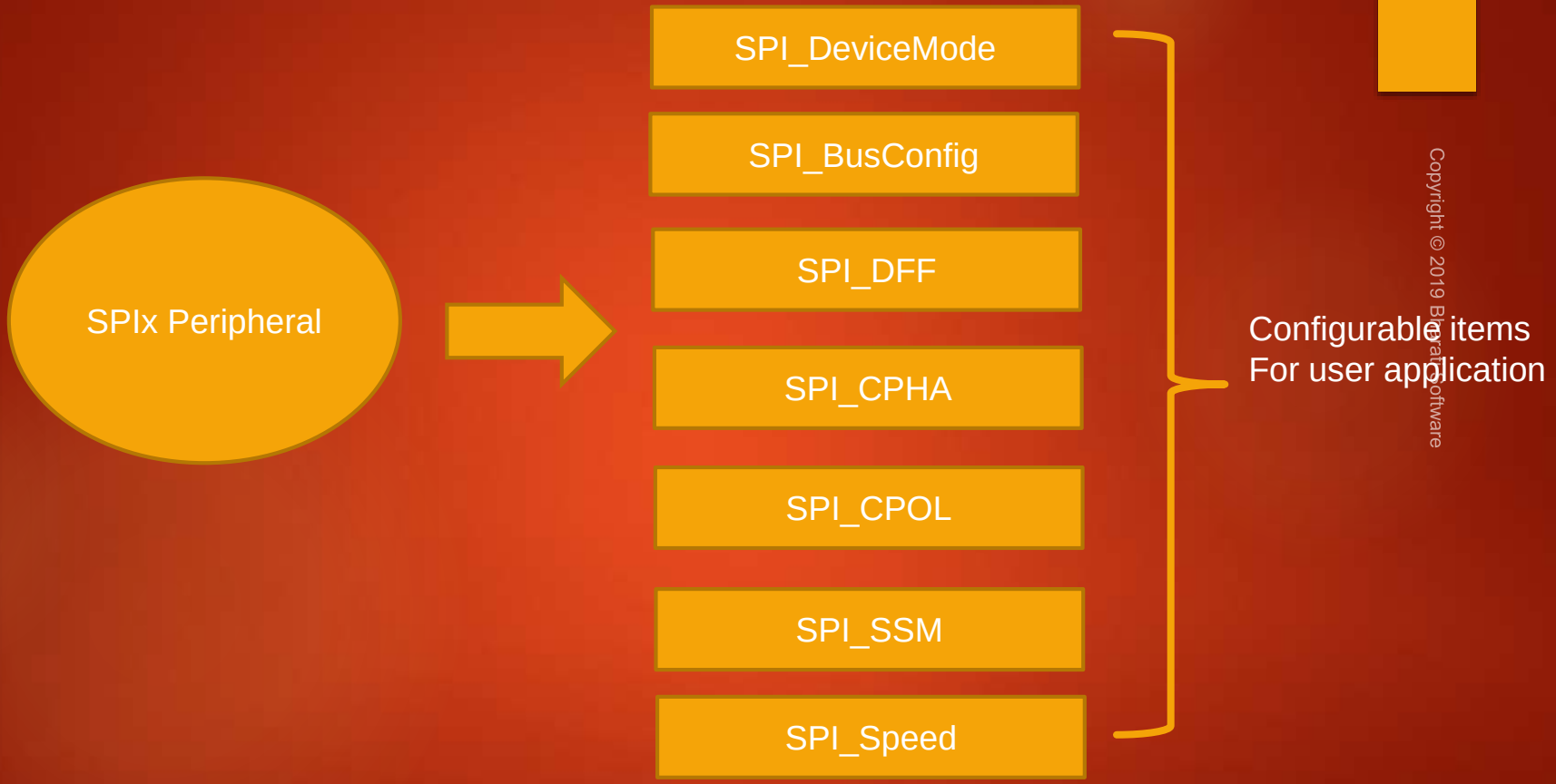
Remember that in SPI communication ,slave will not initiate data transfer unless master produces the clock

SPI Driver Development

Driver API requirements and user configurable items







SPI handle structure and configuration structure

SPI Configuration Structure

```c
/*
 *  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 Handle Structure

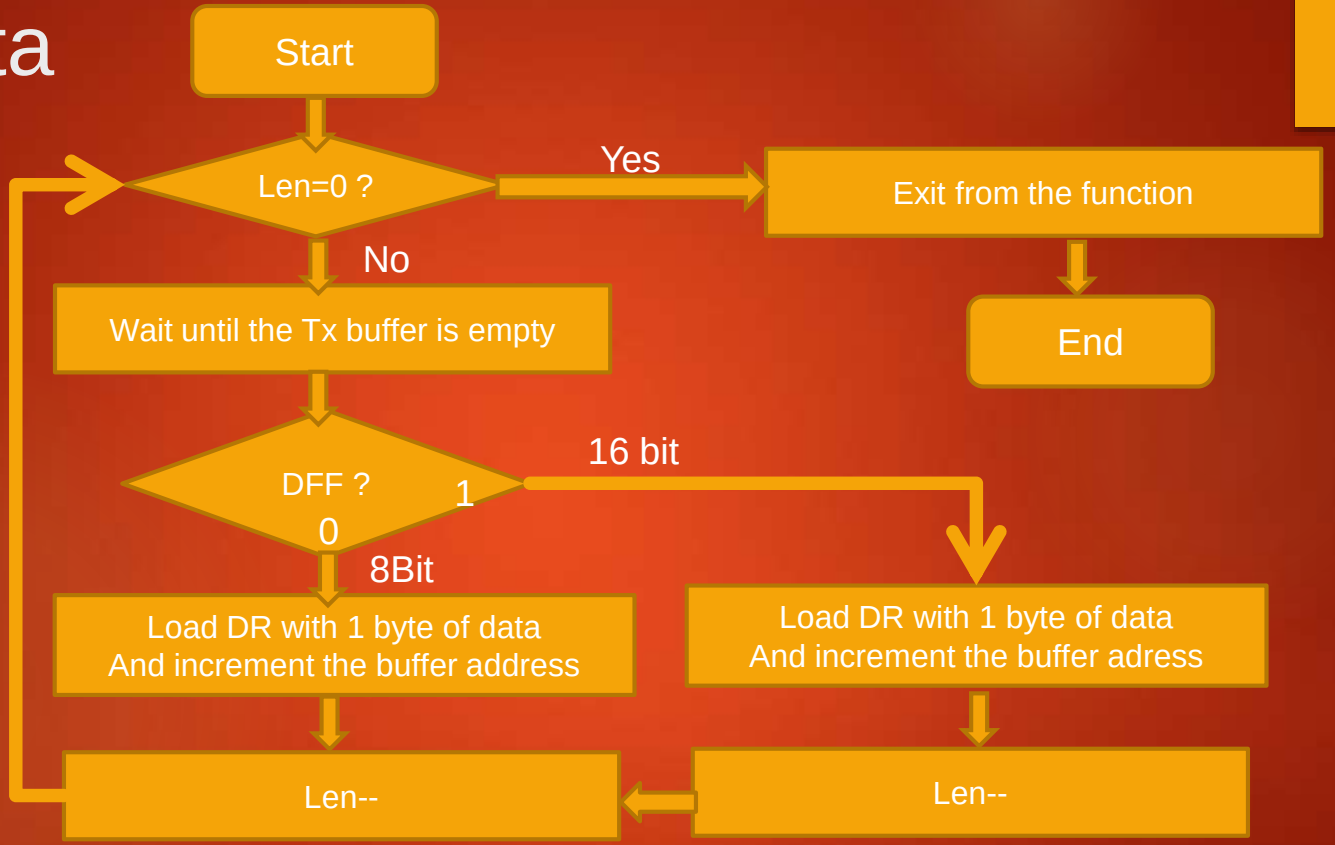
```c
/*
 *Handle structure for SPIx peripheral
 */
typedef struct
{
    SPI_RegDef_t    *pSPIx;    /*!< This holds the base address of SPIx(x:0,1,2) peripheral >*/
    SPI_Config_t    SPIConfig;
}SPI_Handle_t;
```

Exercise :

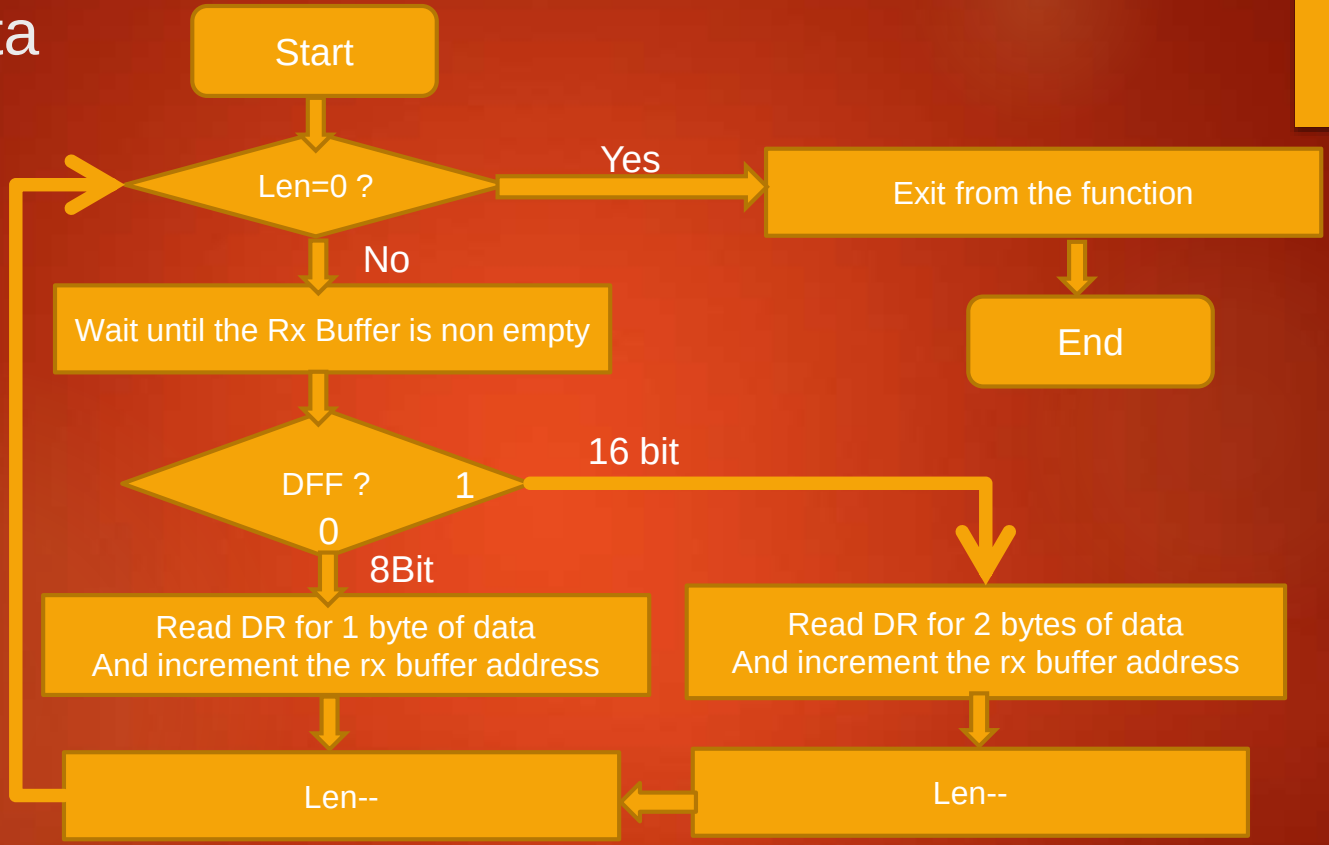
1. Complete SPI register definition structure and other macros (*peripheral base addresses, Device definition , clock en , clock di , etc*) in MCU specific header file
2. Complete SPI Configuration structure and SPI handle structure in SPI header file

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 for 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

Command formats

< command_code(1) > <arg1 > <arg2>

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

ON/OFF

1)CMD_LED_CTRL <pin no(1)> <value(1)>
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

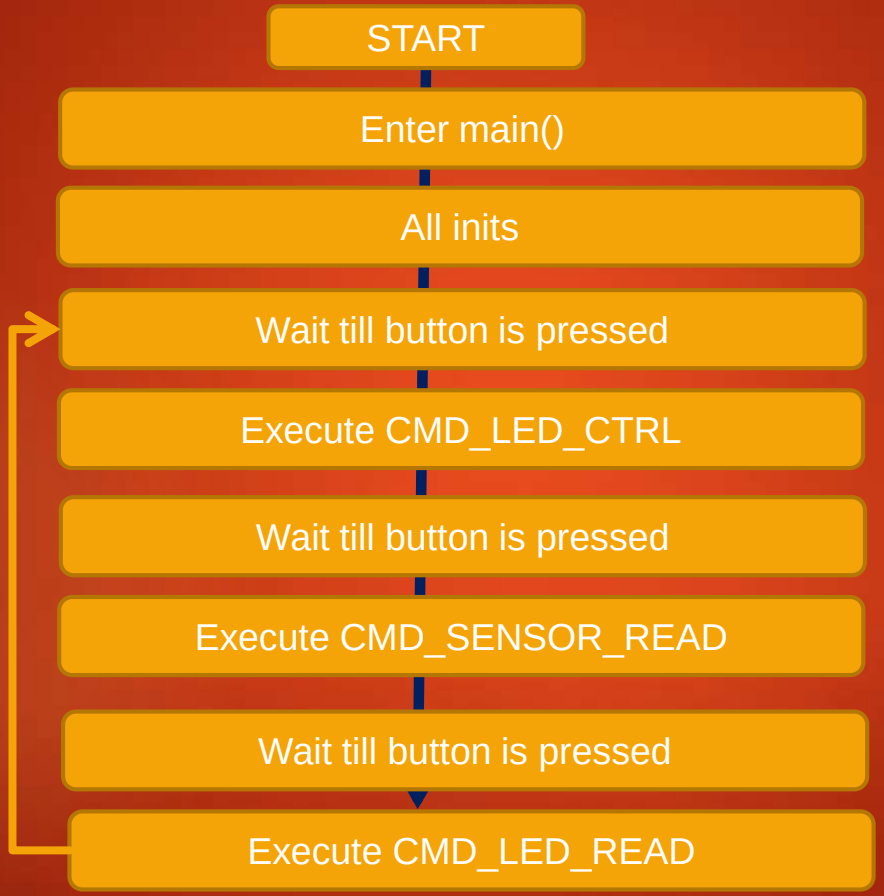
Slave returns : Nothing

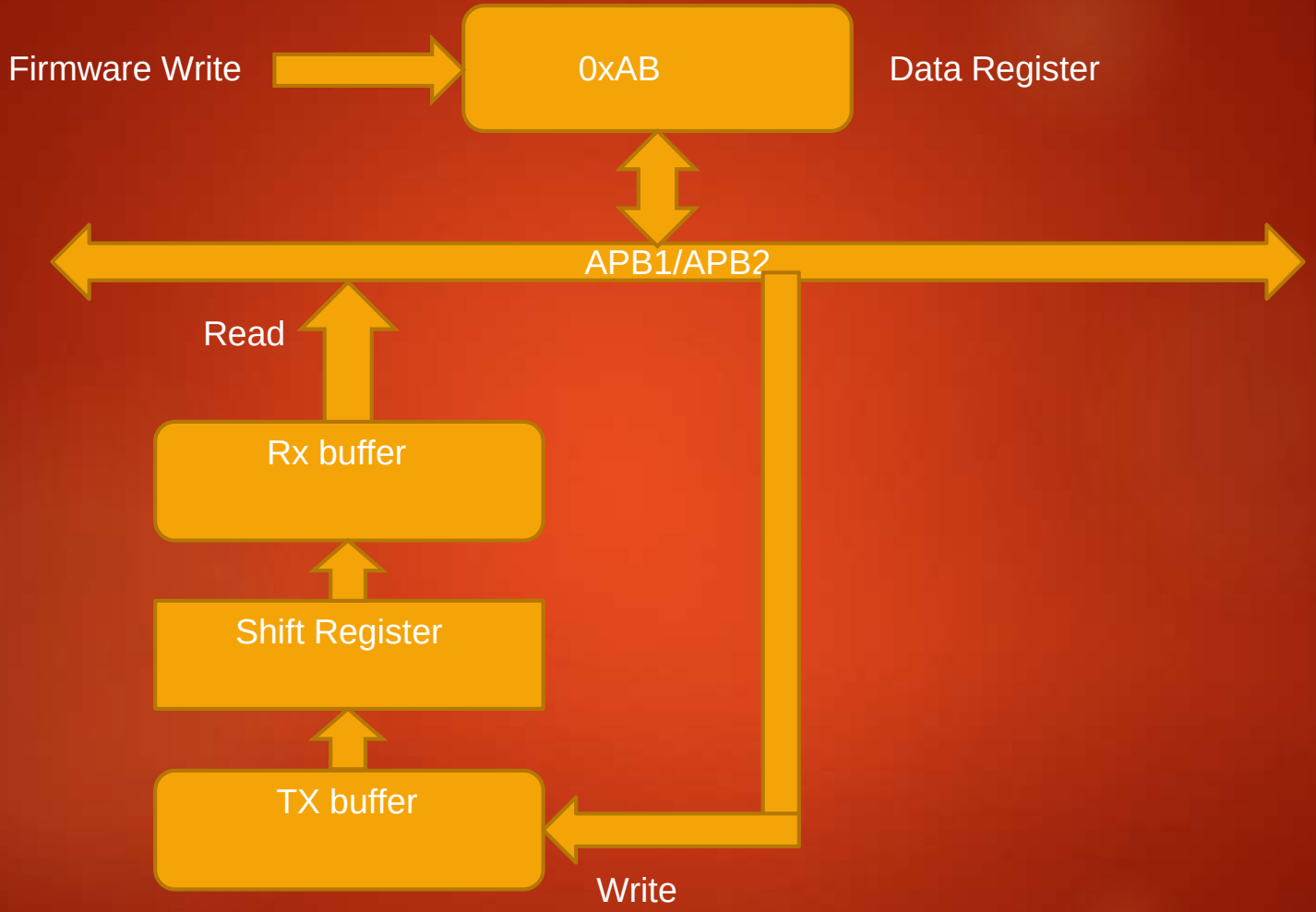
CMD_ID_READ

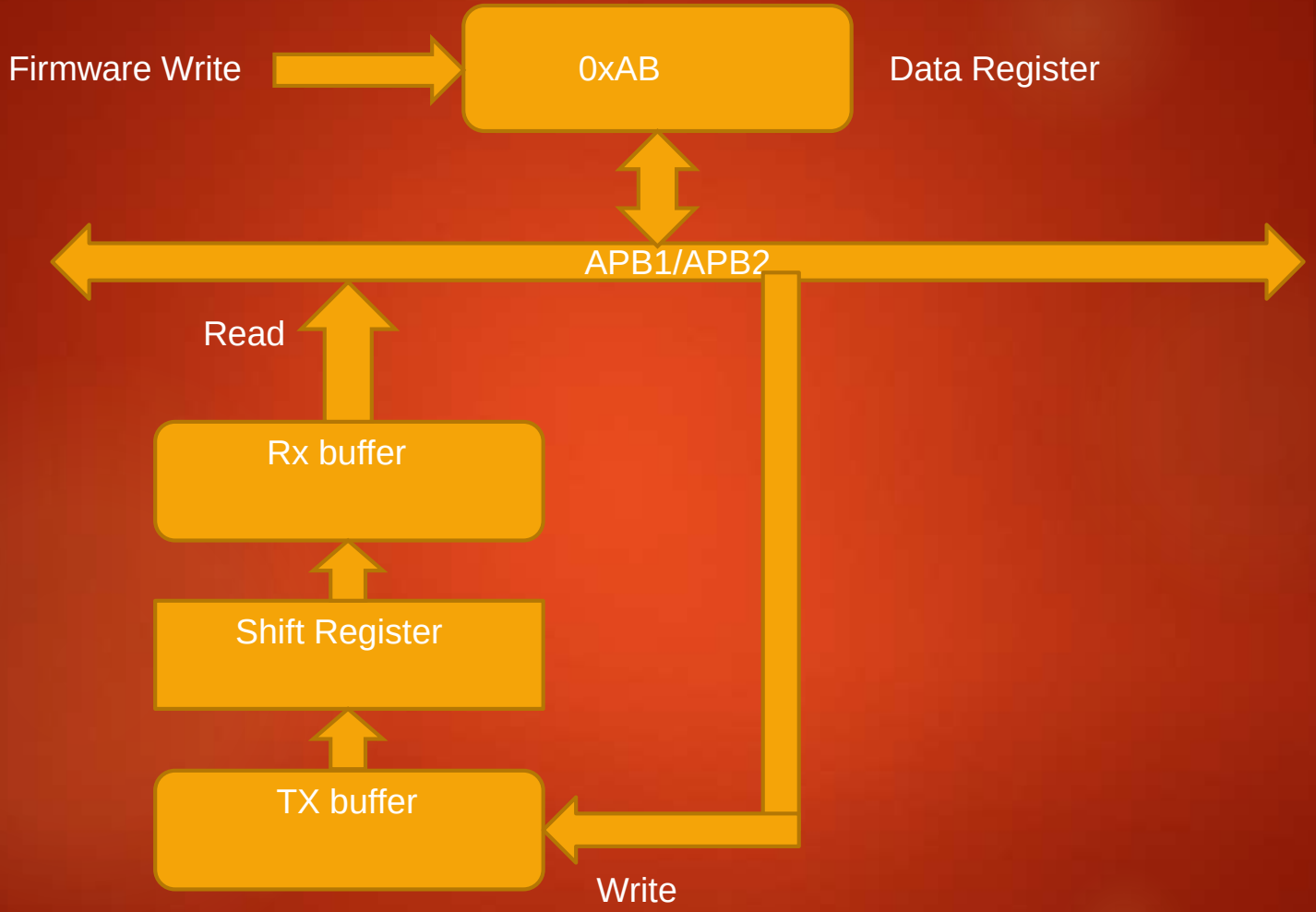
No arguments for this command

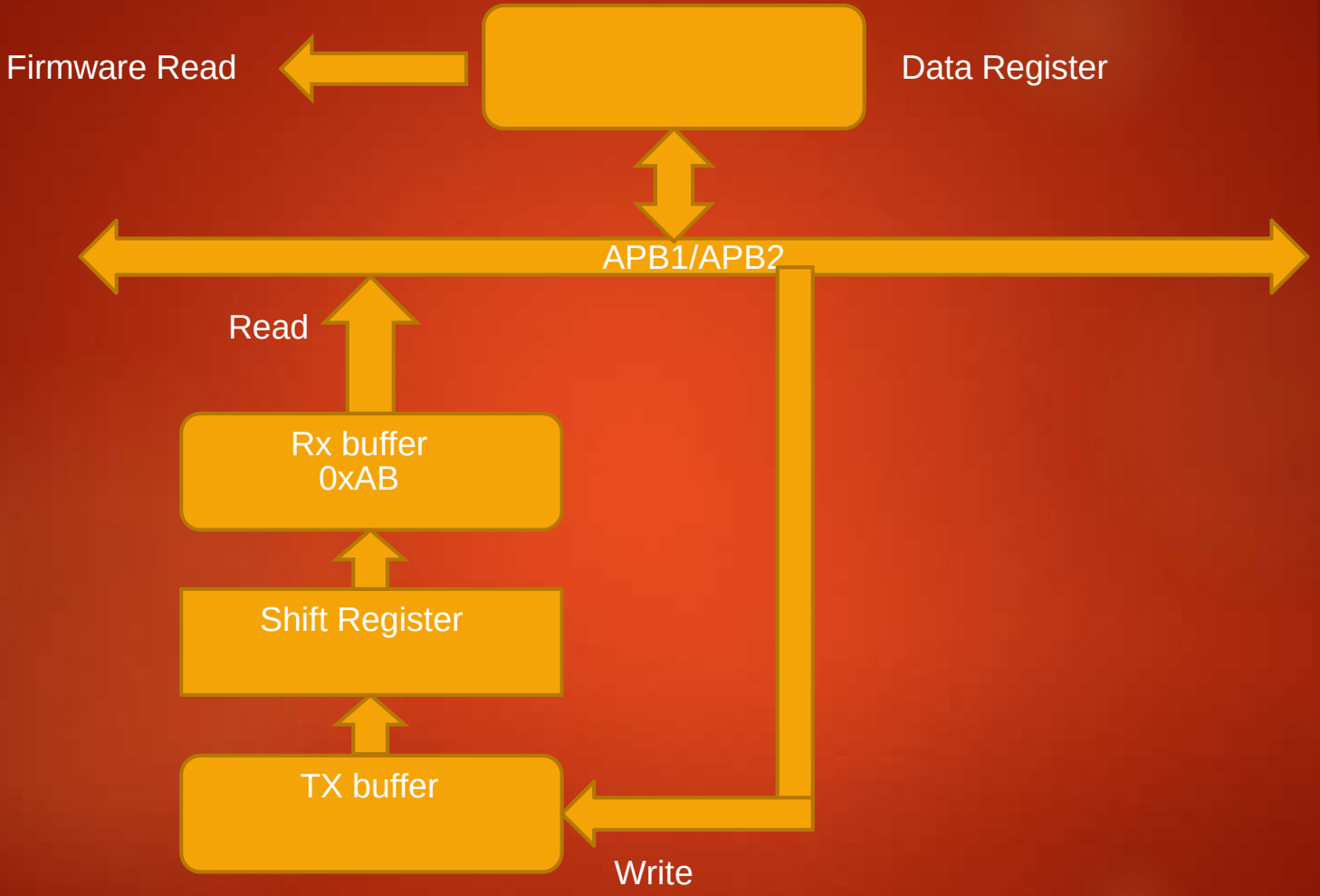
Slave returns : 10 bytes of board id string

Application flow chart









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

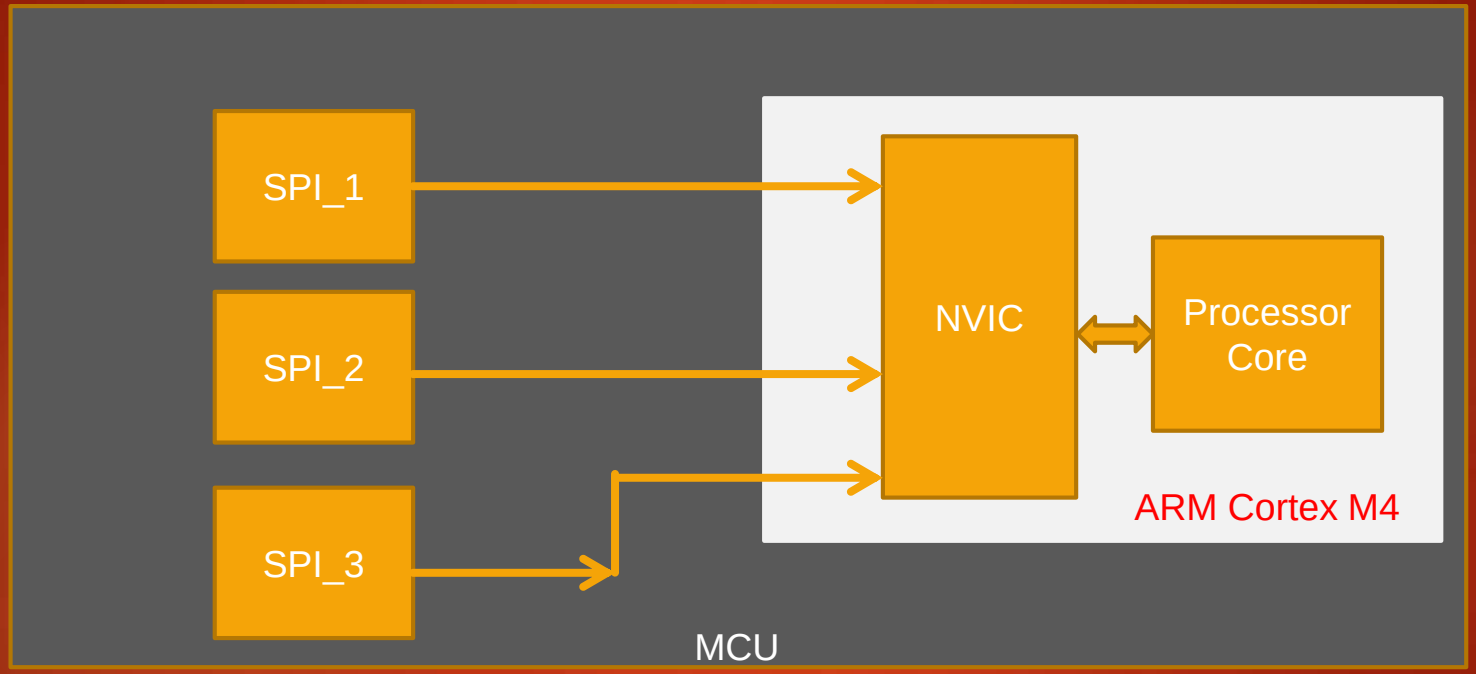
Interrupts can be enabled and disabled separately.

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	ERRIE
Overrun error	OVR	
CRC error	CRCERR	
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_Handle_t *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 Tx 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                0
#define SPI_BUSY_IN_RX           1
#define SPI_BUSY_IN_TX           2
```

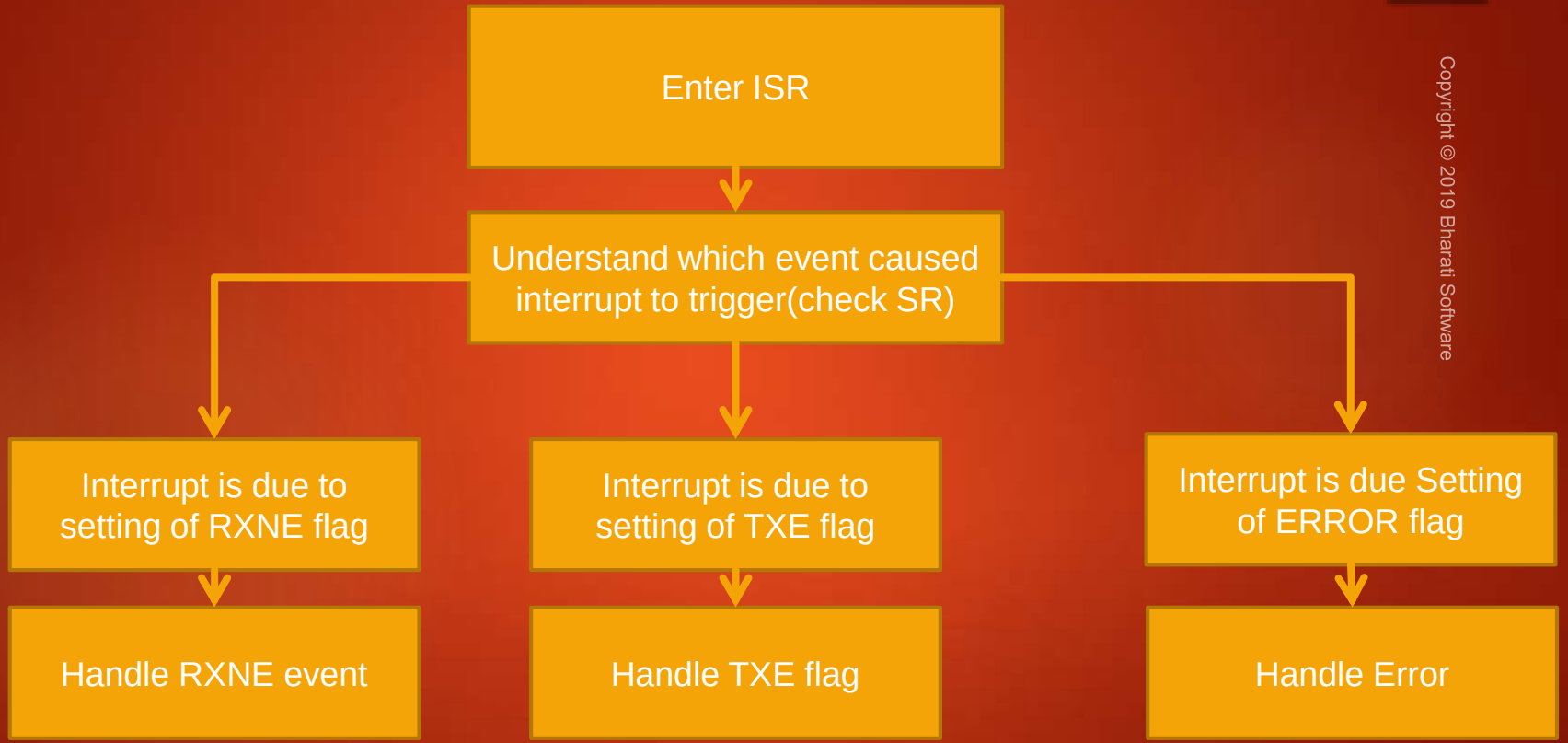
SPI receive data API (Interrupt mode)

SPI receive data with interrupt

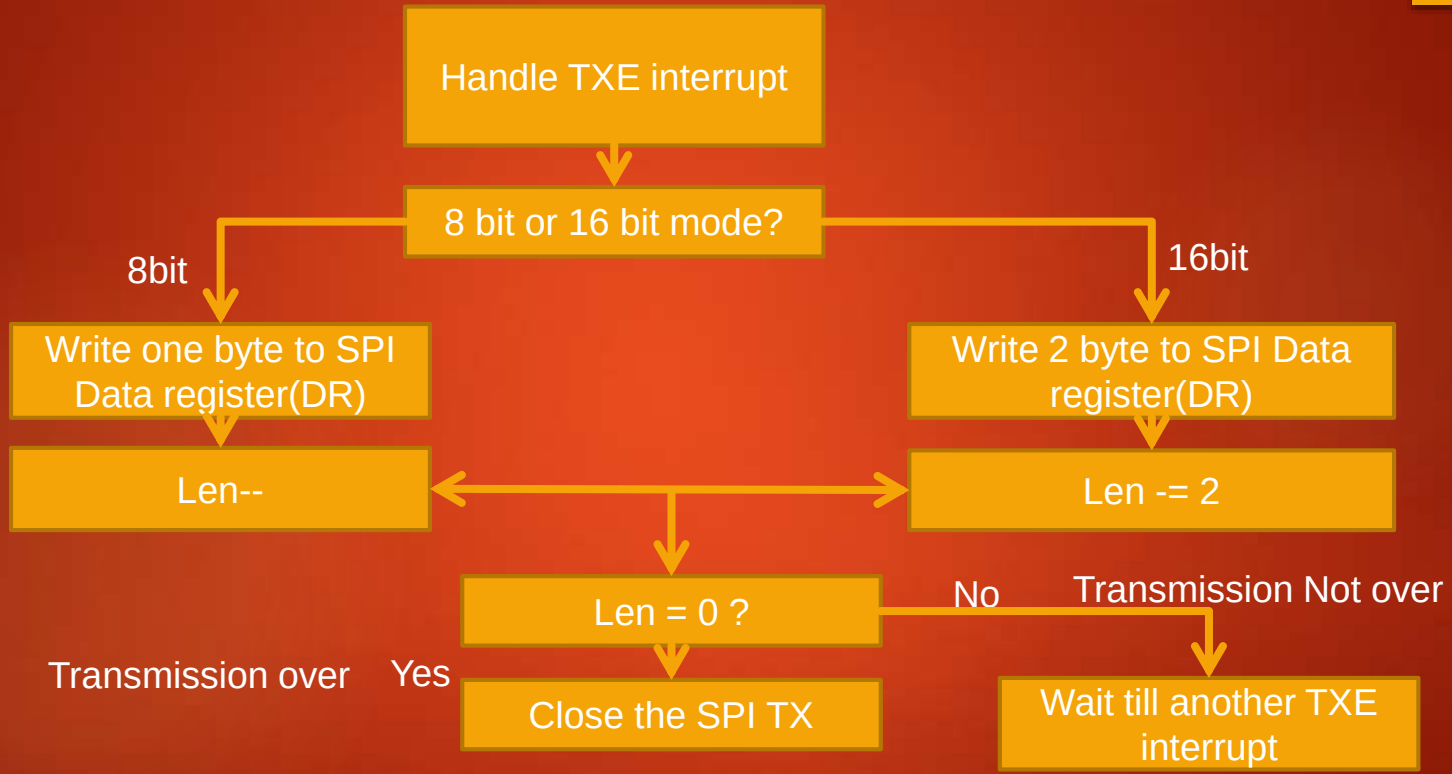
```
uint8_t SPI_ReceiveDataWithIT(SPI_Handle_t *pSPIHandle,uint8_t *pRxBuffer, uint8_t Len)
{

    //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

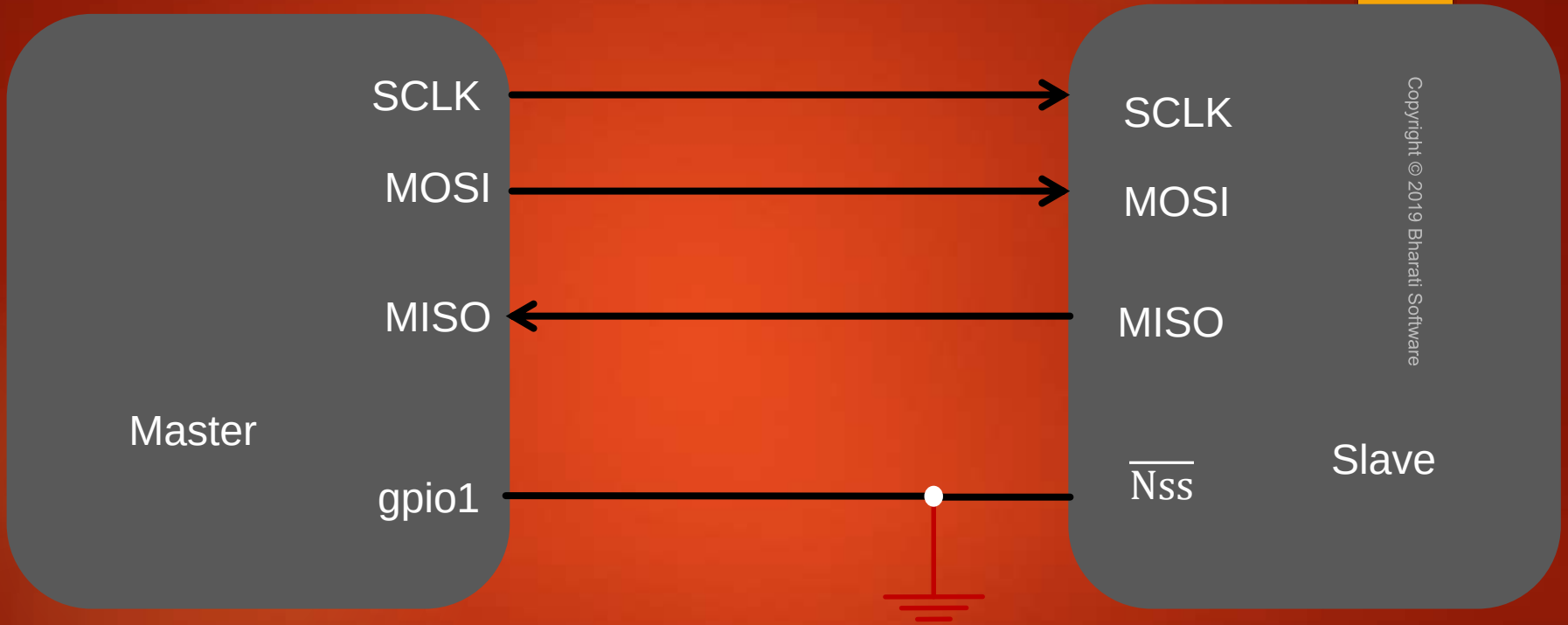
Handling SPI interrupt in the Code



Handling TXE interrupt



Configuring NSS



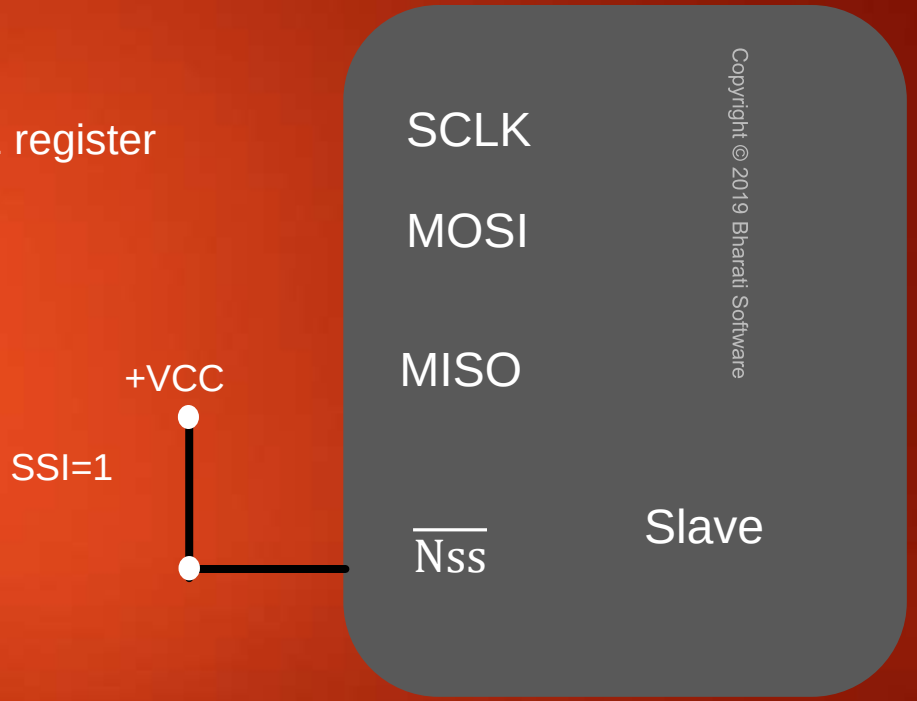
In STM32F4xx based microcontroller, the NSS pin can be handled by 2 ways

- Software Slave Management
- Hardware Slave Management

Software Slave Management

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

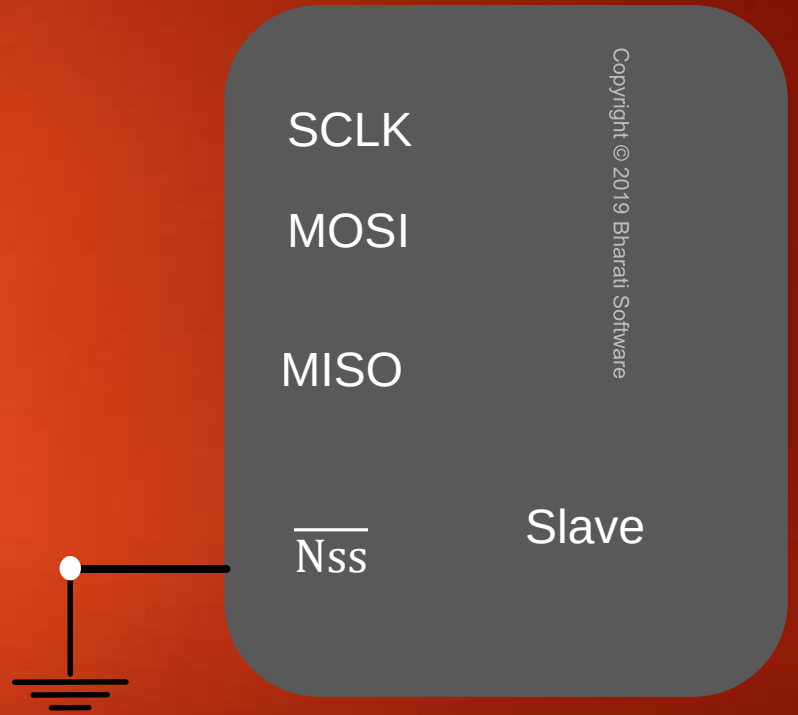
If **SSI** bit = 1 , then **NSS** goes **HIGH**



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**

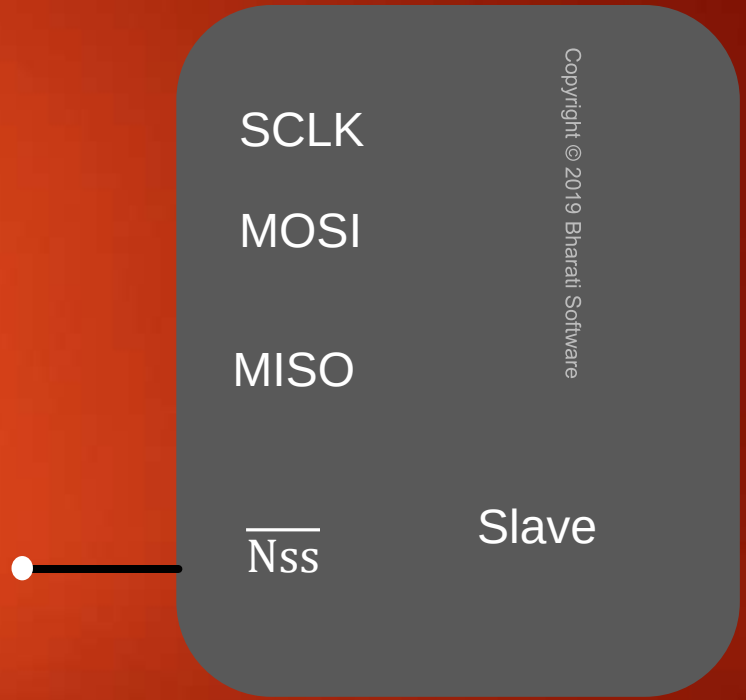


Software Slave Management

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

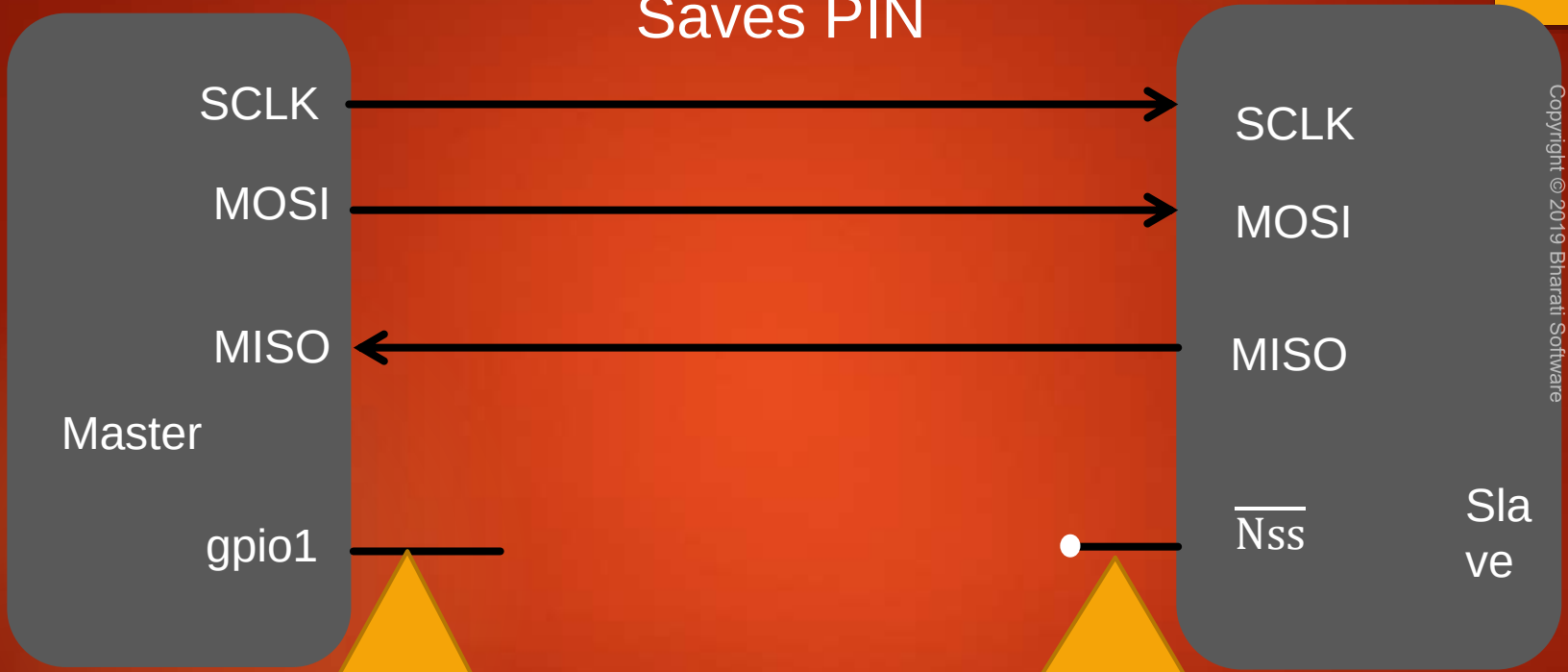
If **SSI** bit = 1 , then **NSS** goes **HIGH**
If **SSI** bit = 0, then **NSS** goes **LOW**

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



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

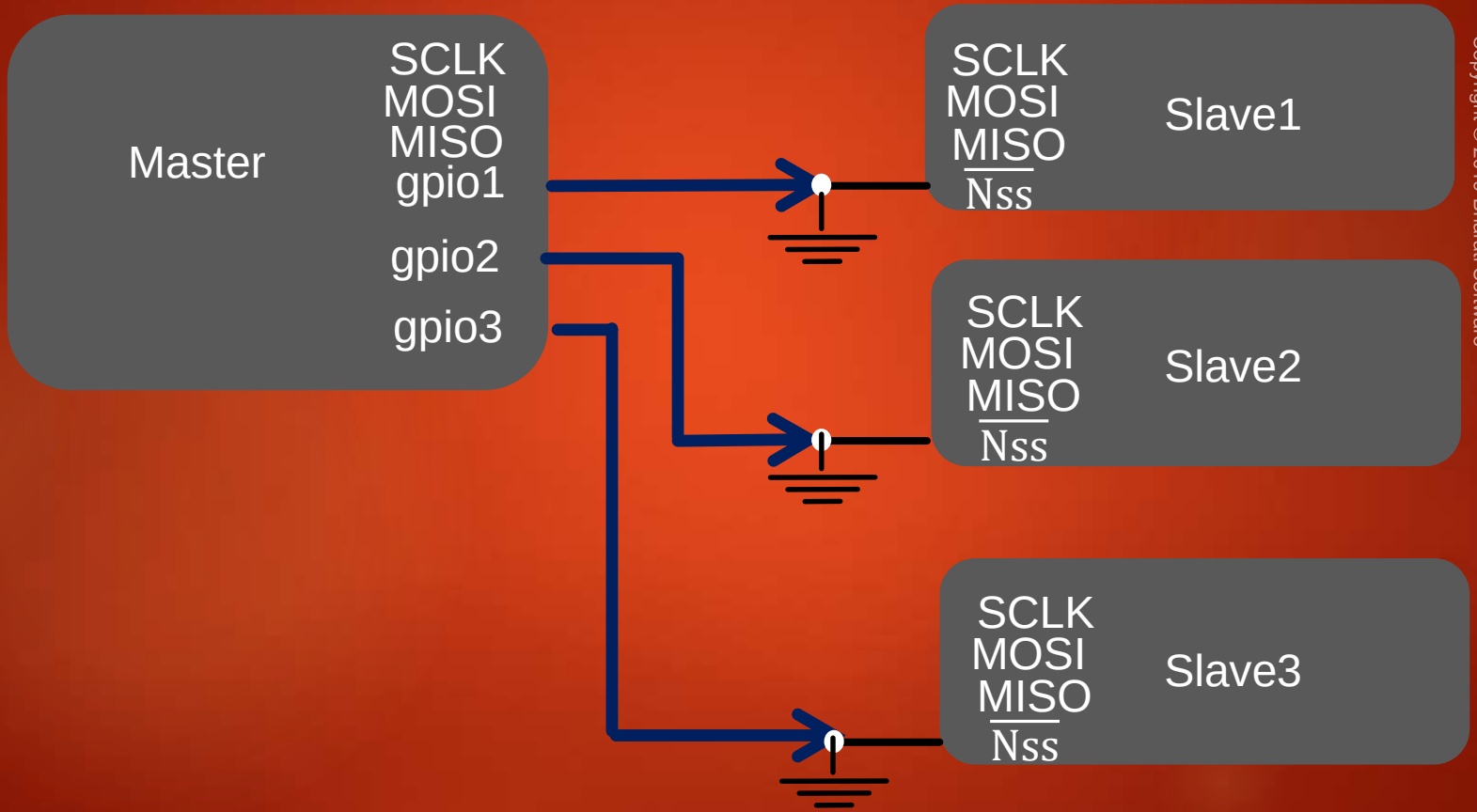
Saves PIN



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.

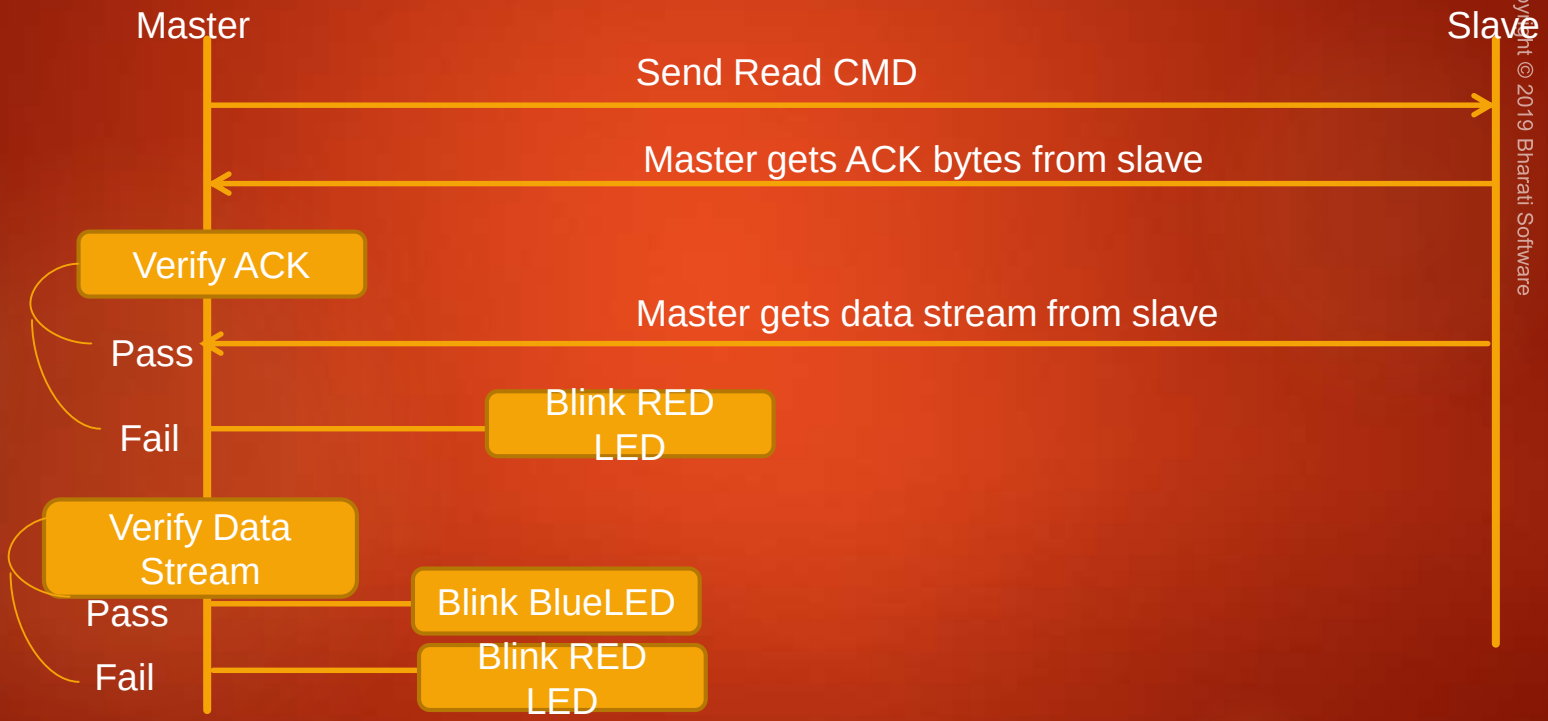
Implementing slave tx api

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

- 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

- ✓ 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

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

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

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

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

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

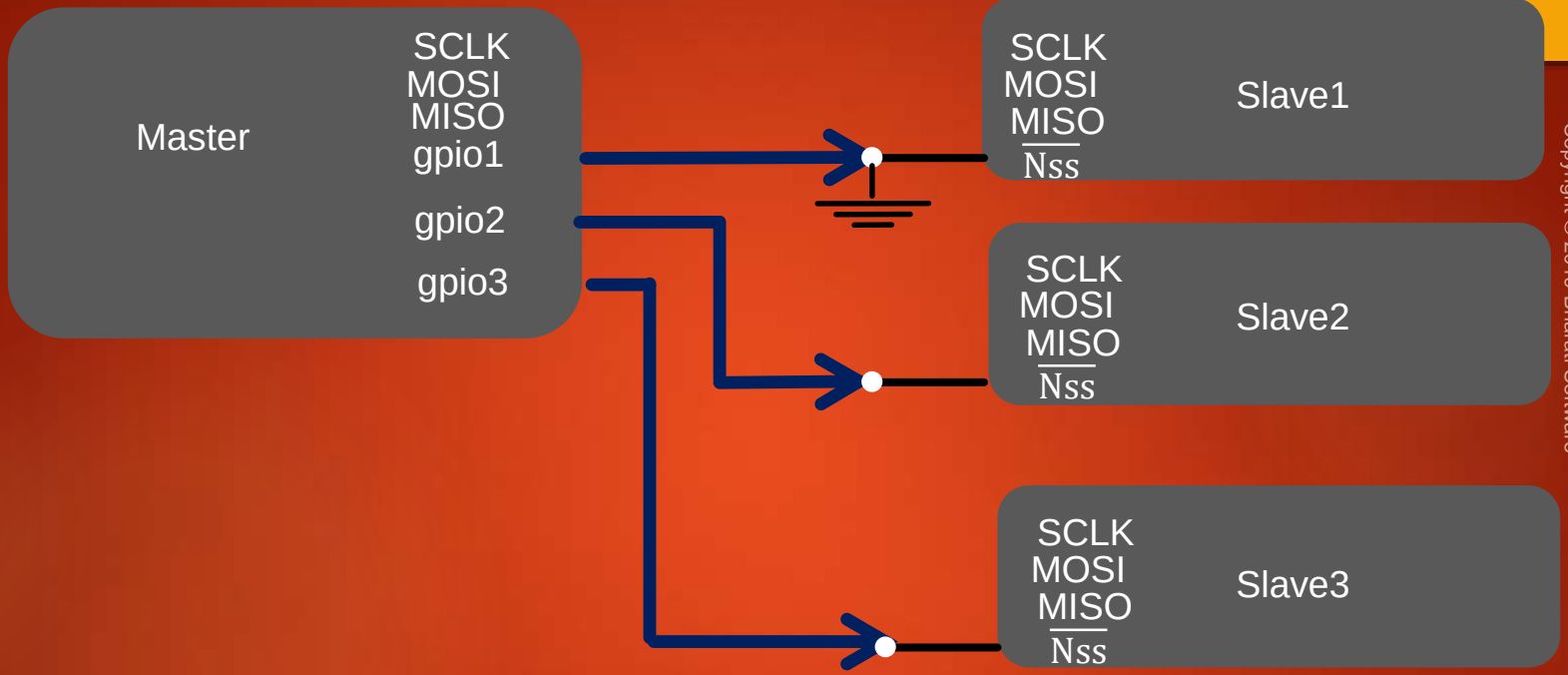
Case 4

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



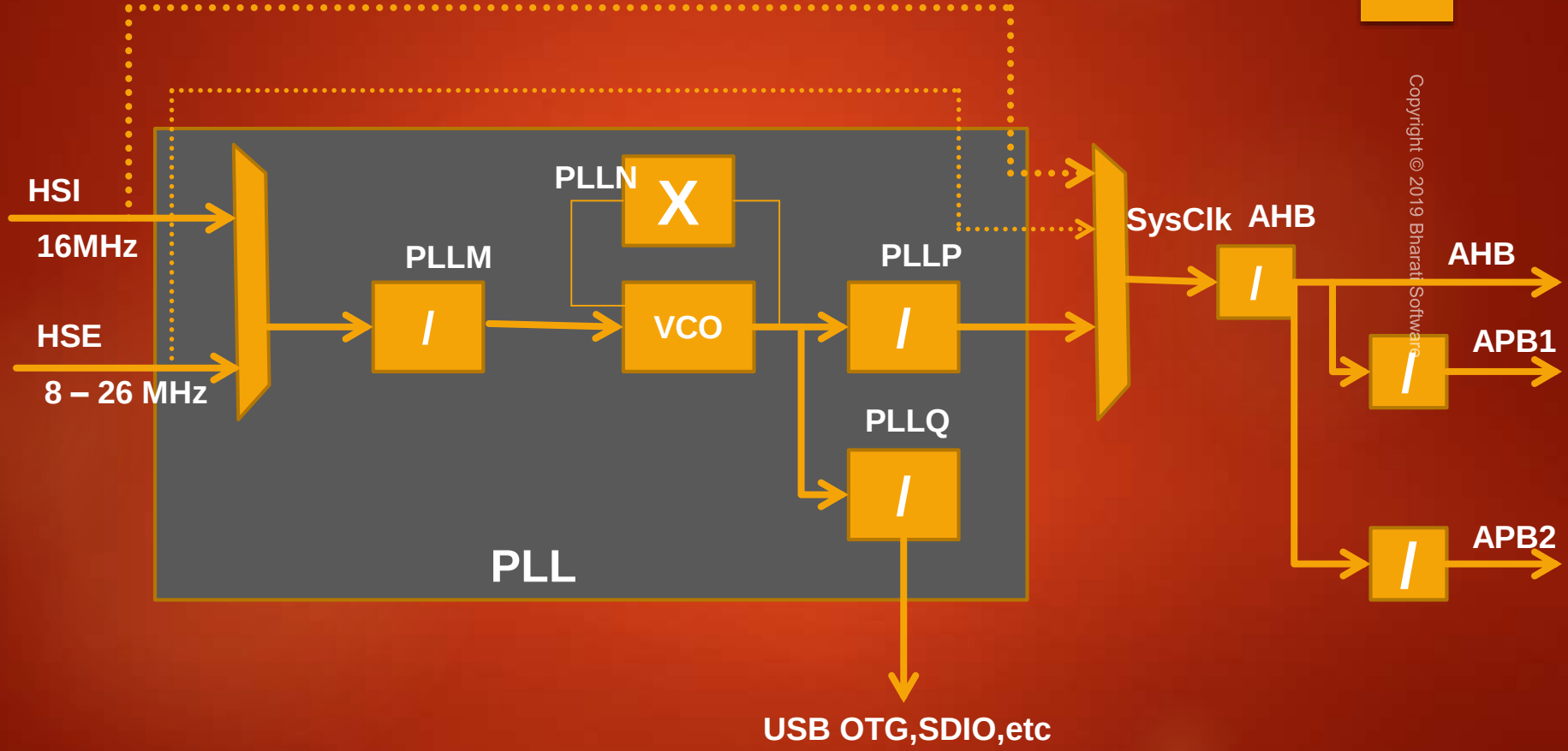
Cross Check These Settings

- 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

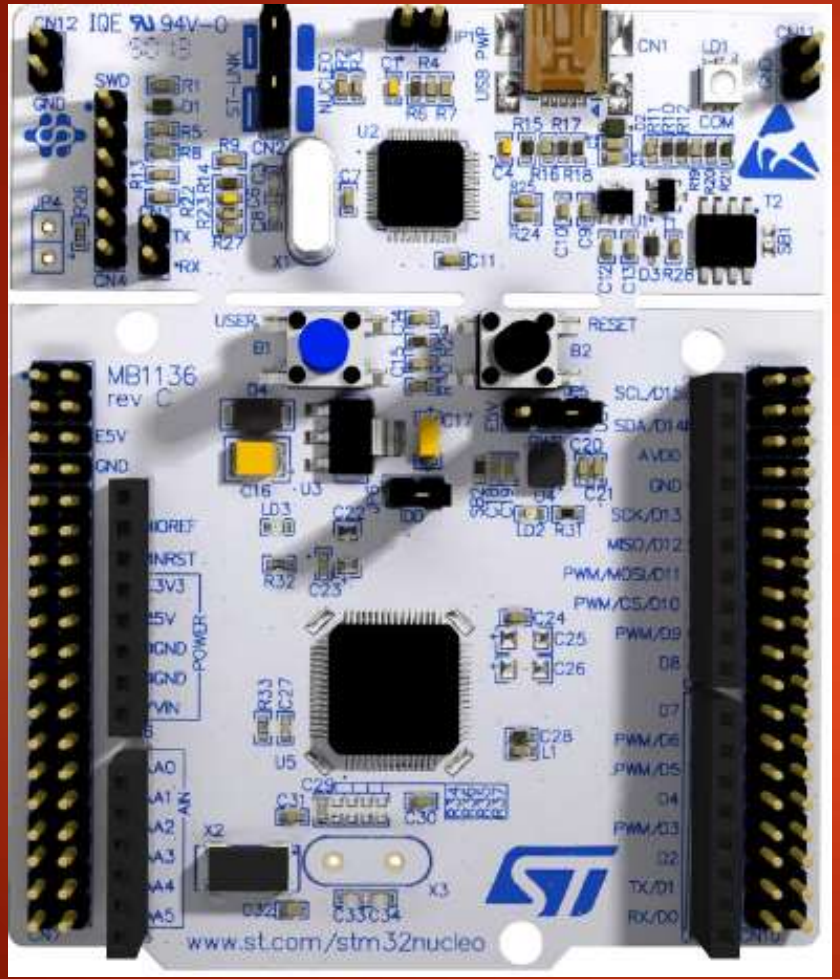
- High Speed Internal (HSI) oscillator
- High Speed External (HSE) oscillator
- Phase Locked Loop (PLL)
- Low Speed Internal (LSI) clock
- Low Speed External (LSE) clock

Clock Sources in the MCU



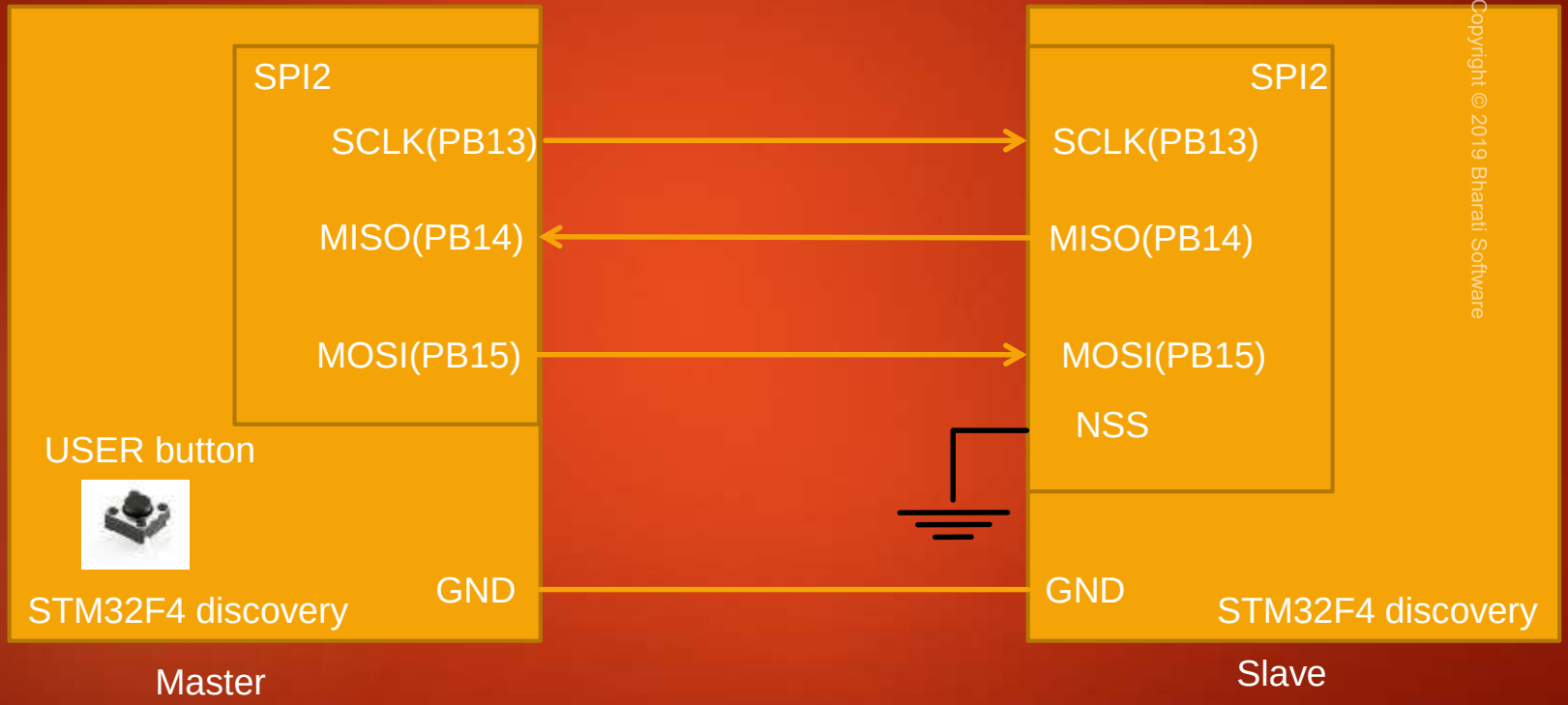
The HSI RC oscillator which is inside the MCU has the advantage of providing a clock source at low cost . Because there is no external components required to use this clock. It also has a faster start-up time than the external crystal oscillator however, the frequency is less accurate than an external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

STM32F4xx Discovery board



STM32F411RE Nucleo-64

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