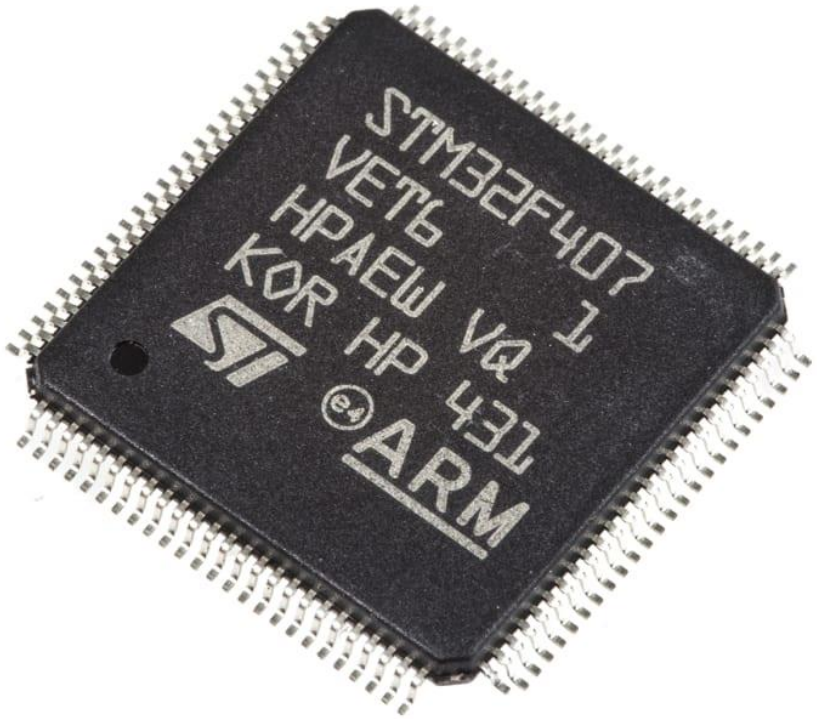
جامعة بيرزيت
BIRZEIT UNIVERSITY

GENERAL PURPOSE INPUT OUTPUT (GPIO)

Shadi Daana

STM32F4

- In this course, we will be using the STM32F4 microcontrollers as our reference platform
- STM32F4 series is based on RM® Cortex®-M4 Core
- Why STMF4?
  - because it offers a powerful, versatile, and well-documented architecture, making it suitable for a wide range of embedded systems applications.



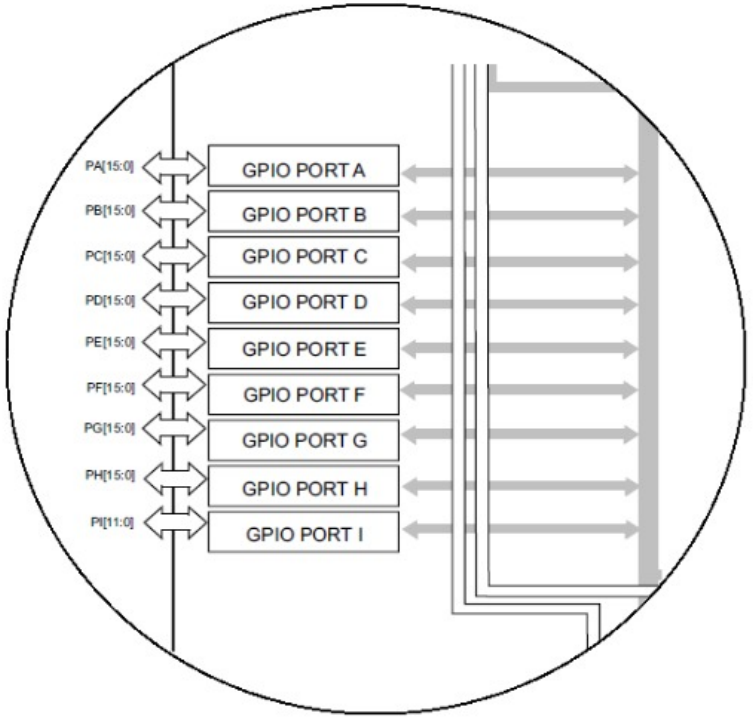
Ports and Pins in MCU

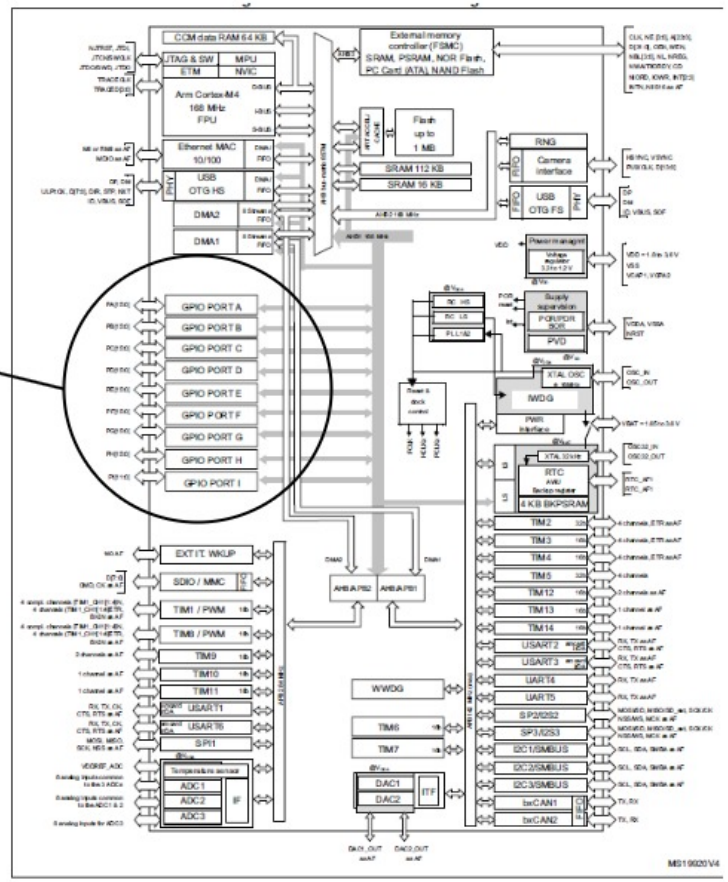
- The microcontroller needs a medium to transfer data to and from the outside world. This is done by using the **ports** of the microcontroller.
- A port is a collection of **pins** grouped together.
- The **pins** are what stick out of an IC, and connect electrically to the outside world
- **Ports** are represented by **registers** inside the microcontroller and allow the **program (firmware)** to control the state of the pins, or conversely, read the state of the pins if they are configured as inputs.

IC Packages

- Pins can take many styles depending on the type of IC package

What is a general-purpose input/output (GPIO)?





What is a general-purpose input/output (GPIO)?

- The STM32F4 microcontroller has a number of input and output ports assigned with initials (Port A, Port B, Port C, Port D, etc.)
- Each port has a specific number of pins (most ports have 16 pins each)
- Each pin can be used for both **Input** and **Output** operations. Besides, a pin can be used for **Analog** or **Alternate Functions**.
- In the Alternate Function, the pins can be used by the other peripherals within the microcontroller
- Hence, the pins of the STM32F4 microcontroller are called **general-purpose input and output (GPIO)**.

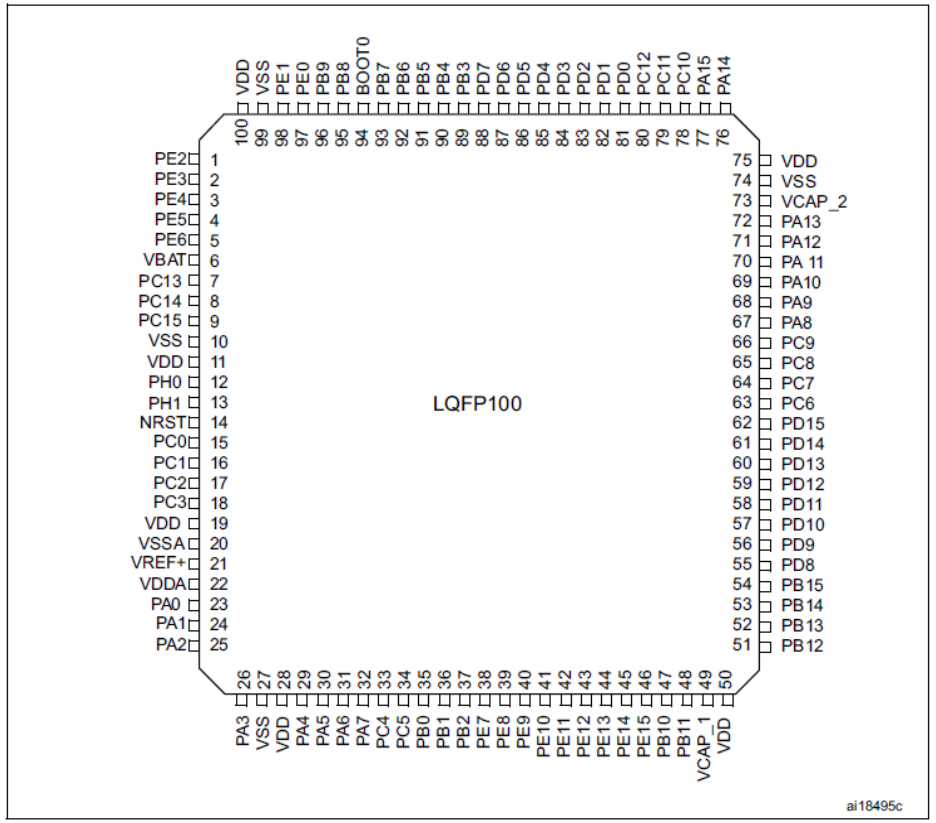
GPIO Structures

- Several GPIO structures are available across the range of STM32 devices. Each structure is associated with a list of options

List of GPIO structures

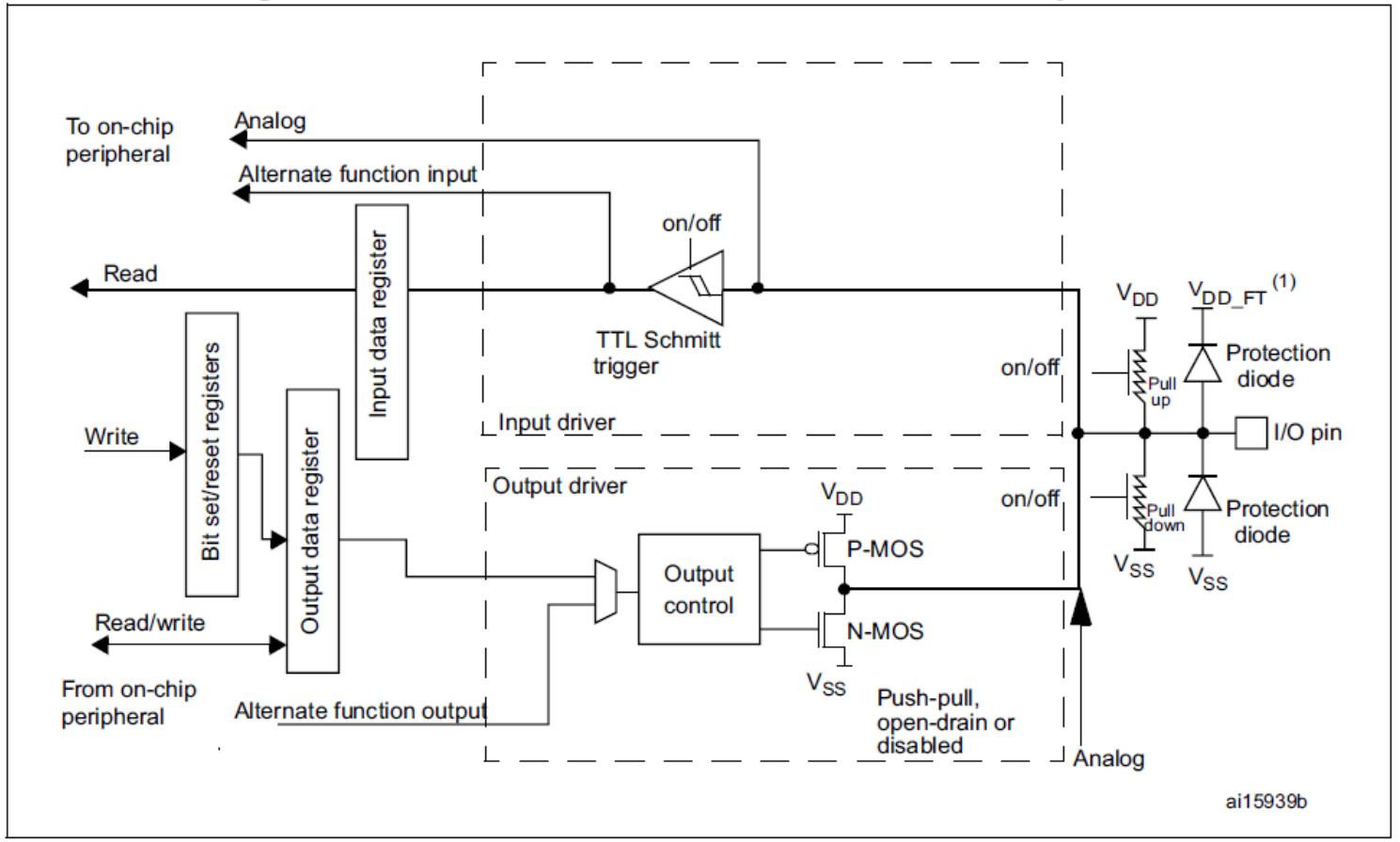
Name		Abbreviation	Definition
Pin Type		S	Supply pin
		I	Input only pin
		I/O	Input / output pin
I/O structure		FT(1)	Five-volt tolerant I/O pin
		TT(1)	Three-volt tolerant I/O pin
		TC	Three-volt capable I/O pin (Standard 3.3 V I/O)
		B	Dedicated boot pin
		RST	Bidirectional reset pin with embedded weak pull-up resistor
Pin functions	Alternate functions	Functions selected through **GPIOx_AFR** registers	
	Additional functions	Functions directly selected and enabled through peripheral registers	

1. FT and TT I/Os have options depending on the device. The user must refer to the datasheet for their definitions.



What's inside a port?

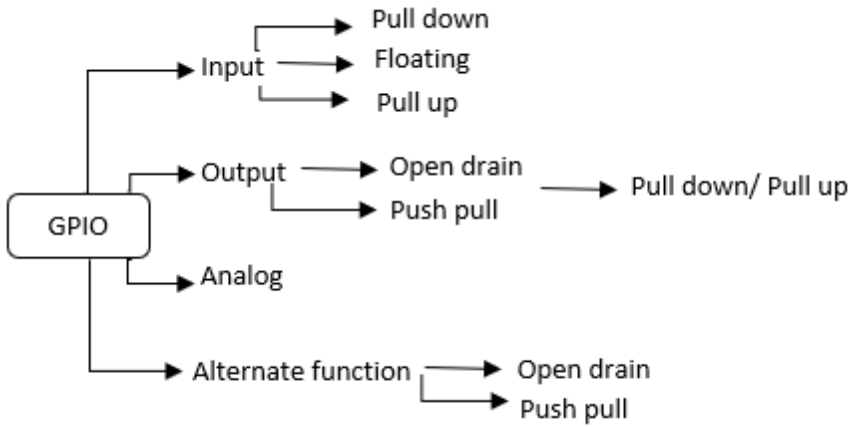
Figure 25. Basic structure of a five-volt tolerant I/O port bit



1. V_{DD_FT} is a potential specific to five-volt tolerant I/Os and different from V_{DD}.

Configuring GPIO Pins

- **Pin Mode**: Each port bit of the general-purpose I/O (GPIO) ports can be individually configured by software in several modes:
  - Digital input or output
  - Analog
  - Alternate function (AF).
- **Pin characteristics** :
  - **Input**: no pull-up and no pull-down or pull-up or pull-down
  - **Output**: push-pull or open-drain with pull-up or pull-down capability
  - **Alternate function**: push-pull or open-drain with pull-up or pull-down capability.



GPIO Registers

- In microcontroller programming, General-Purpose Input/Output (GPIO) registers are hardware registers that control the behavior of individual GPIO pins.
- These registers are used to configure, read from, and write to GPIO pins. GPIO registers vary from one microcontroller to another, but they typically include the following types of registers:
  - Mode Register (MODER)
  - Output Type Register (OTYPER)
  - Output Speed Register (OSPEEDR)
  - Pull-up/Pull-down Register (PUPDR)
  - Alternate Function Register (AFR)
  - Input Data Register (IDR)
  - Output Data Register (ODR)
  - Bit Set/Reset Register (BSRR)
  - Lock Register (LCKR)

GPIO Registers

- **Mode Register (MODER)**: This register is used to specify the mode of the GPIO pins, whether they should function as inputs, outputs, or alternate function pins. Each pin is typically represented by 2 bits in this register. Common mode values include:
  - 00: Input mode.
  - 01: Output mode.
  - 10: Alternate function mode.
  - 11: Analog mode.
- **Output Type Register (OTYPER)**: This register determines the output type of the GPIO pins when configured as outputs. It specifies whether the pin should be open-drain or push-pull. Common values include:
  - 0: Push-pull output.
  - 1: Open-drain output.

GPIO Registers

- **Pull-up/Pull-down Register (PUPDR):** This register configures the pull-up and pull-down resistors for the GPIO pins when they are configured as inputs. It can be used to enable or disable pull-up or pull-down resistors. Common values include:
  - 00: No pull-up, no pull-down.
  - 01: Pull-up.
  - 10: Pull-down.
- **Input Data Register (IDR):** This register holds the current state of the input pins. You can read from this register to determine whether a pin is high (1) or low (0).
- **Output Data Register (ODR):** This register holds the data to be written to output pins. Writing to this register sets the state of output pins as high or low.

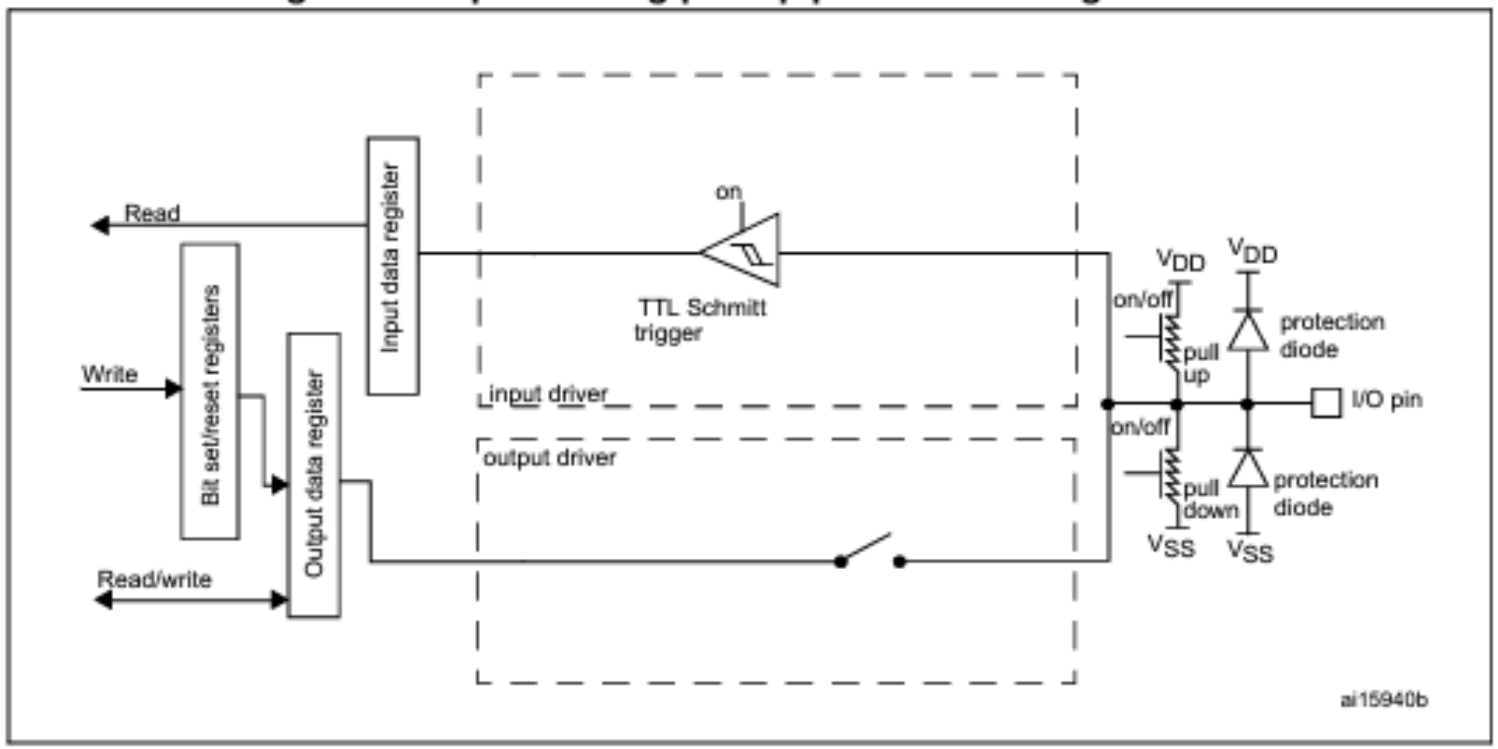
GPIO Registers

- **Output Speed Register (OSPEEDR):** This register sets the output speed of the GPIO pins when configured as outputs. It allows you to choose between various speed options, such as low, medium, or high speed.
- **Bit Set/Reset Register (BSRR):** This register is used to set or reset individual bits in the ODR. It is a quick way to change the state of specific pins without affecting the others.
- **Lock Register (LCKR):** The lock register can be used to lock or unlock the GPIO configuration, preventing accidental changes to pin configuration.
- **Alternate Function Register (AFR):** It is an important register in microcontrollers, particularly in STM32 microcontrollers, which use the Alternate Function to extend the capabilities of GPIO pins. These alternate functions can include communication interfaces (UART, SPI, I2C), timers, PWM, and more.

GPIO modes description

Input mode configuration

Figure 28. Input floating/pull up/pull down configurations



GPIO modes description

Programmed as input, an I/O port exhibits the following characteri stics:

- The output buffer is disabled
- The Schmitt trigger input is activated
- The pull-up or pull-down resistors are activated depending on the value in the **GPIOx_PUPDR** register
- The data present on the I/O pin is sampled into the input data register at each AHB clock cycle
- The I/O state is obtained by reading the **GPIOx_IDR** input data register

GPIO modes description

Input mode configuration

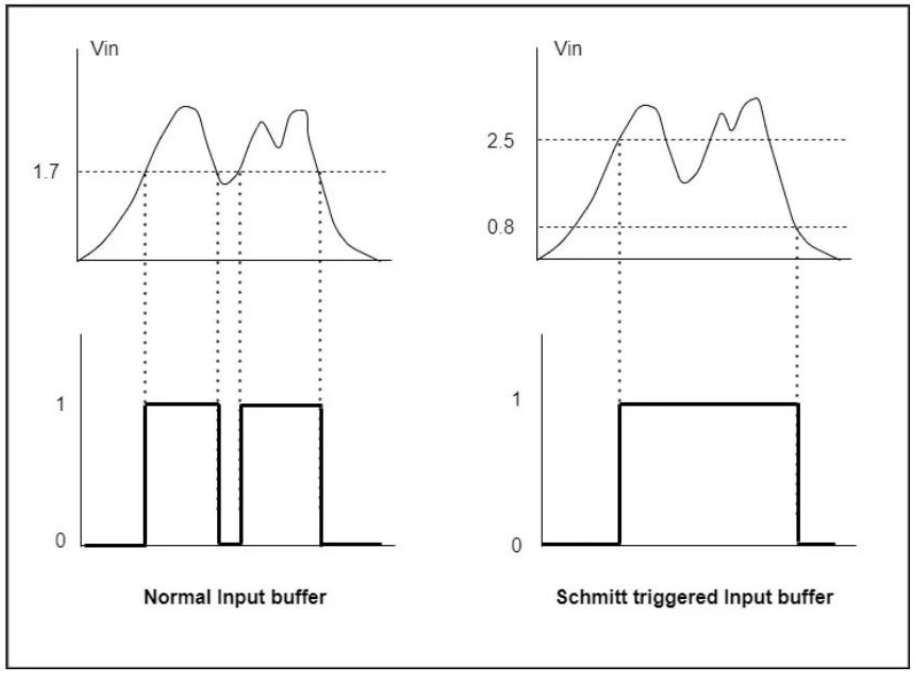
- When a STM32 device I/O pin is configured as input, one of three options must be selected:
  - Input with internal **pull-up**
  - Input with internal **pull-down**

  } to ensure a well-defined logical level in case of floating input signal
  External pull-up/pull-down can be used instead

  - **Floating** input: signal level follows the external signal. When no external signal is present, the Schmitt trigger randomly toggles between the logical levels induced by the external noise.
- *Note: It is important to note that it is not possible to activate pull-up and pull-down at the same time on the same I/O pin.*

GPIO modes description

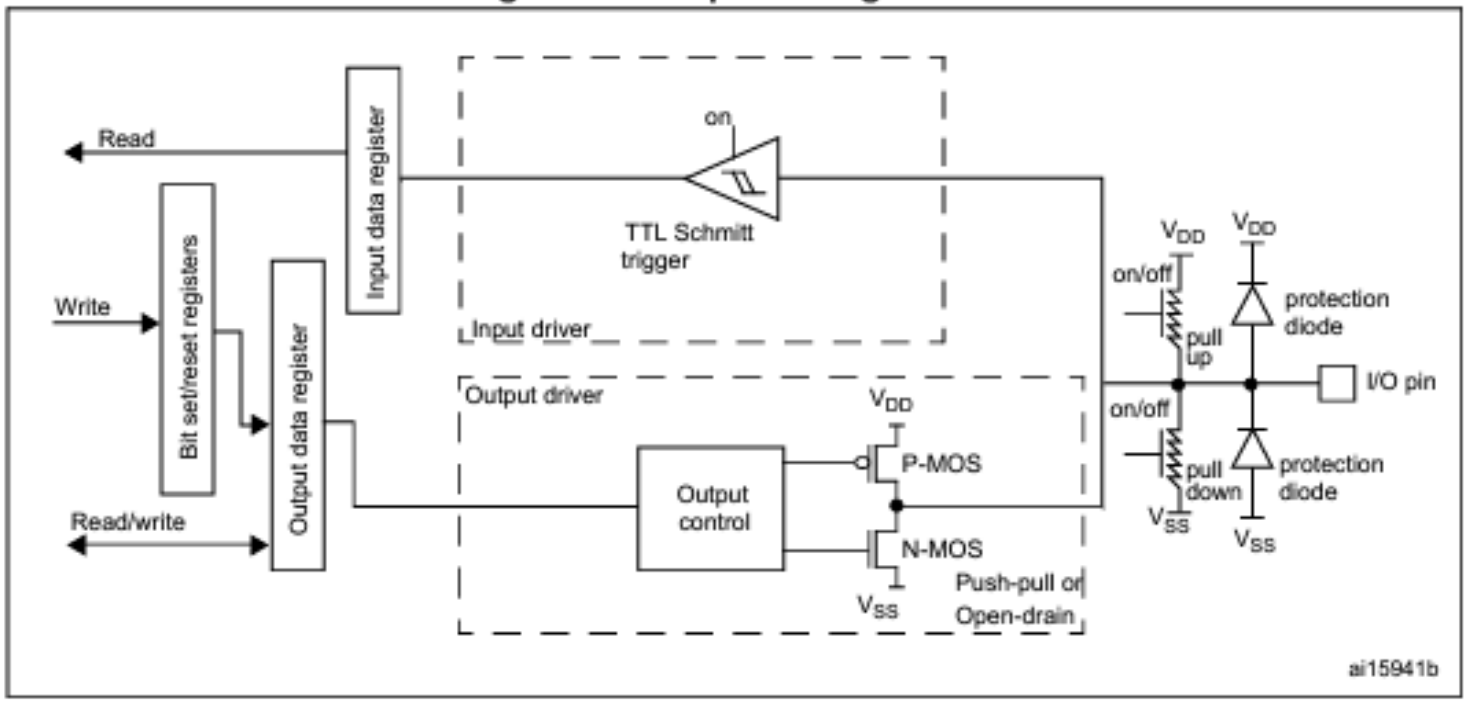
- A **Schmitt trigger** is a voltage comparator that turns **ON** only when the input voltage exceeds a certain threshold, and turns **OFF** when the input voltage falls below the lower threshold
- Schmitt trigger increases the noise immunity compared to the circuit with only a single input threshold.



GPIO modes description

Output mode configuration

Figure 29. Output configuration



GPIO modes description

Programmed as output, an I/O port exhibits the following characteristics:

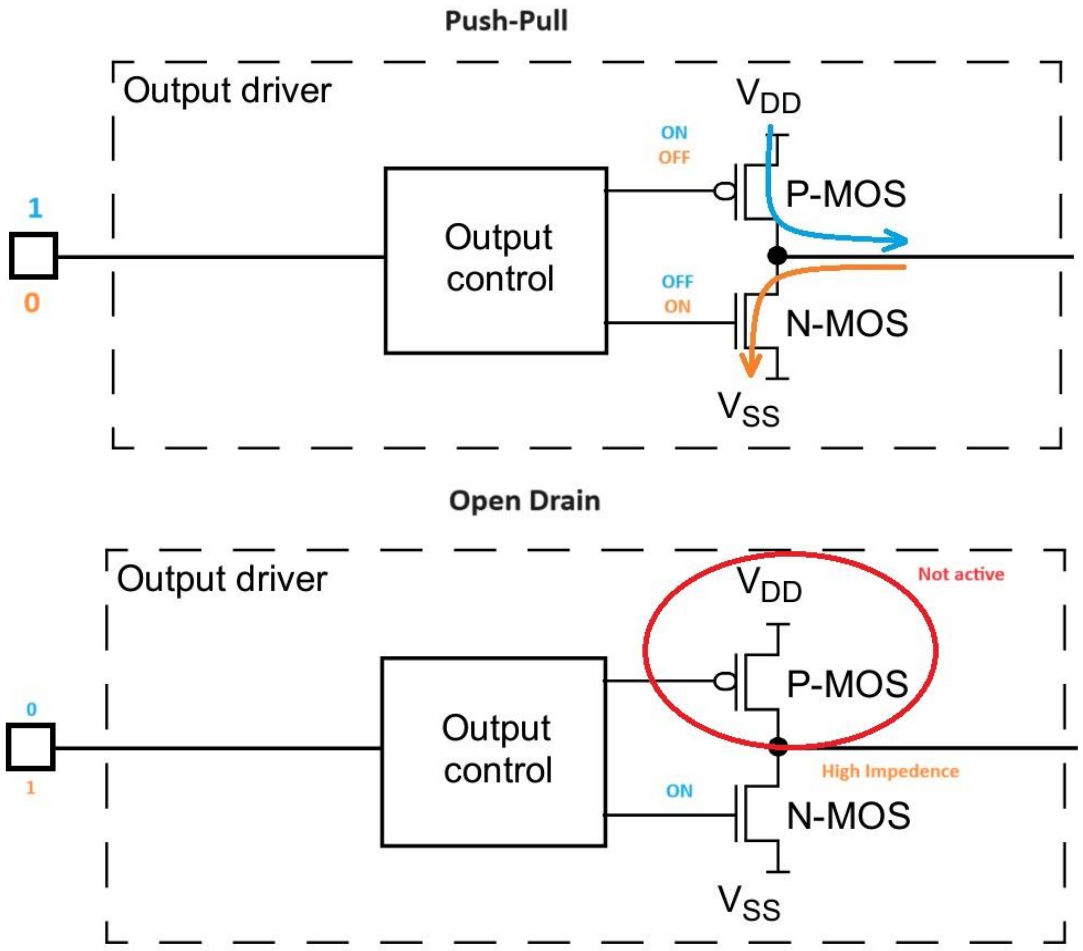
- The output buffer can be configured in open-drain or push-pull mode
- The Schmitt trigger input is activated
- The internal pull-up and pull-down resistors are activated depending on the value in the **GPIOx_PUPDR** register.
- The written value into the output data register **GPIOx_ODR** sets the I/O pin state
- The written data on **GPIOx_ODR** can be read from **GPIOx_IDR** register that is updated every AHB clock cycle

GPIO modes description

Output mode configuration

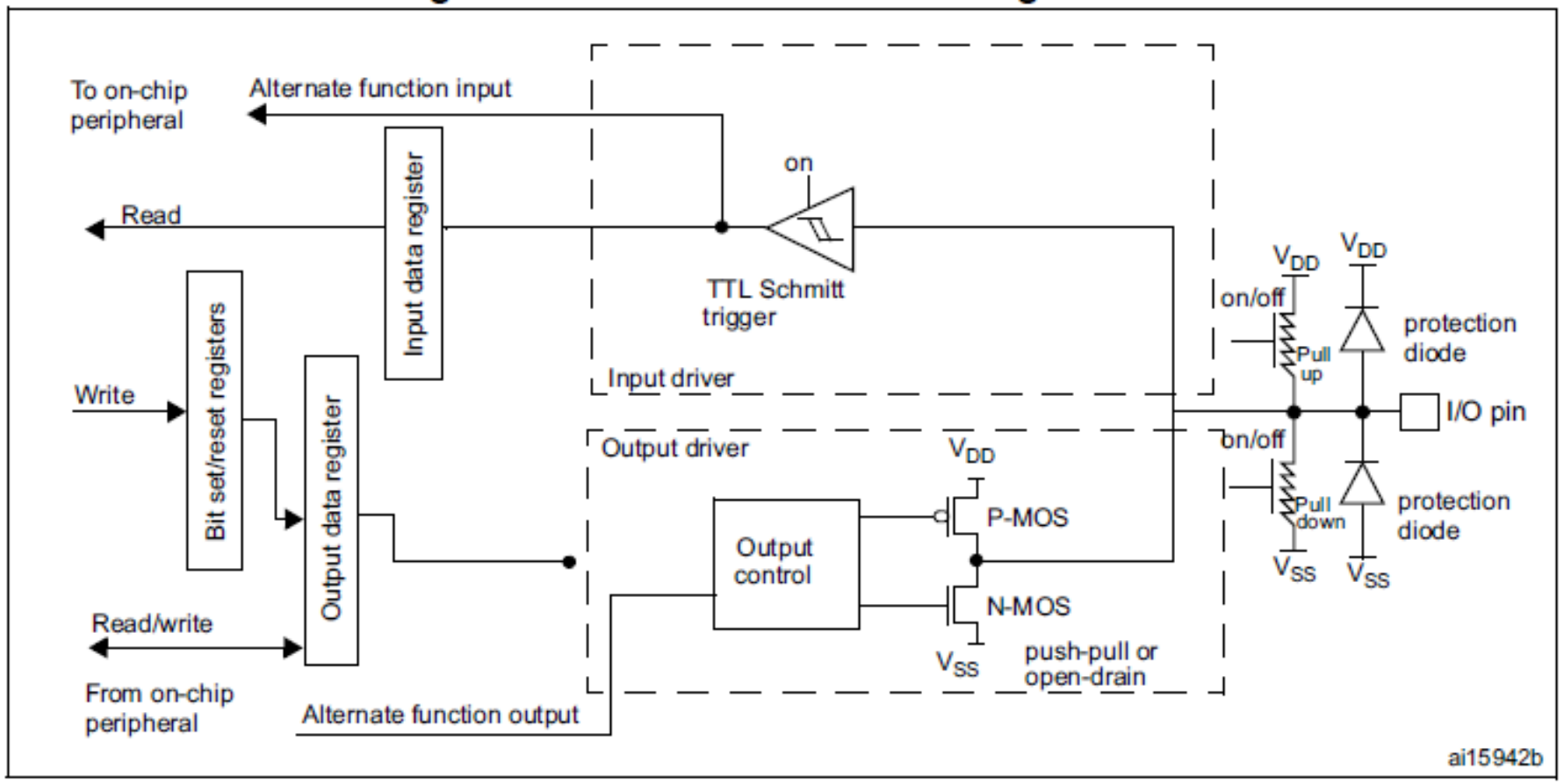
- When a STM32 device I/O pin is configured as output, one of two options must be selected:
  - **Push-pull** output mode: it uses two transistors, one PMOS and one NMOS. Each transistor is ON to drive the output to the appropriate level
  - **Open-drain** output mode: it does not use the PMOS transistor.
- The control of the two transistors is done through the GPIO port output type register (**GPIOx_OTYPER**)

GPIO modes description



GPIO modes description

Figure 30. Alternate function configuration



GPIO modes description

Alternate functions

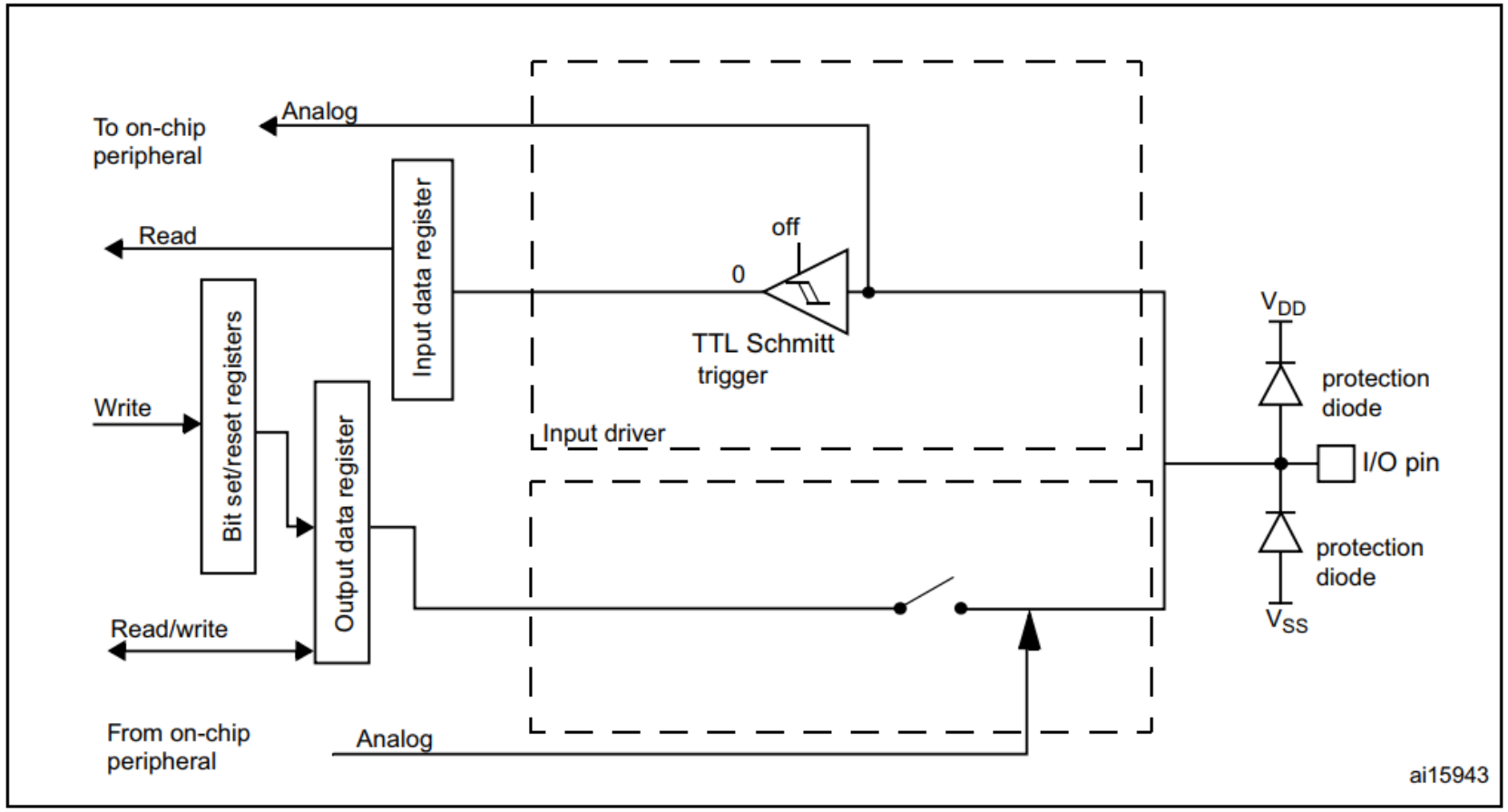
- On some STM32 GPIO pins, the user has the possibility to select alternate functions inputs / outputs.
- Each pin is multiplexed with up to sixteen peripheral functions such as communication interfaces (SPI, UART, I2C, USB, CAN, LCD and others), timers, debug interface, and others.
- The alternate function of the selected pin is configured through two registers:
  - **GPIOx_AFRL** (for pin 0 to 7)
  - **GPIOx_AFRH** (for pin 8 to 15)
- To know which functions are multiplexed on each GPIO pin, refer to the device datasheet.

GPIO modes description

- When the I/O port is programmed as alternate function mode:
  - The output buffer can be configured in open-drain or push-pull mode
  - The output buffer is driven by the signals coming from the peripheral (transmitter enable and data)
  - The Schmitt trigger input is activated
  - The pull-up and pull-down resistors activations depend on the value in the register **GPIOx_PUPDR**
- The data present on the I/O pin are sampled into the input data register at each AHB clock cycle.
- A read access to the input data register provides the I/O state.
- Alternate functions

GPIO modes description

Figure 31. High impedance-analog configuration



GPIO modes description

Analog configuration

- Few STM32 GPIO pins can be configured in analog mode which allows the use of ADC, DAC, OPAMP, and COMP internal peripherals. To use a GPIO pin in analog mode, the following register are considered:
- **GPIOx_MODER** to select the mode (Input, Output, Alternate, Analog)
- **GPIOx_ASCR** to select the required function ADC, DAC, OPAMP, or COMP

GPIO modes description

When the I/O port is programmed in an analog configuration:

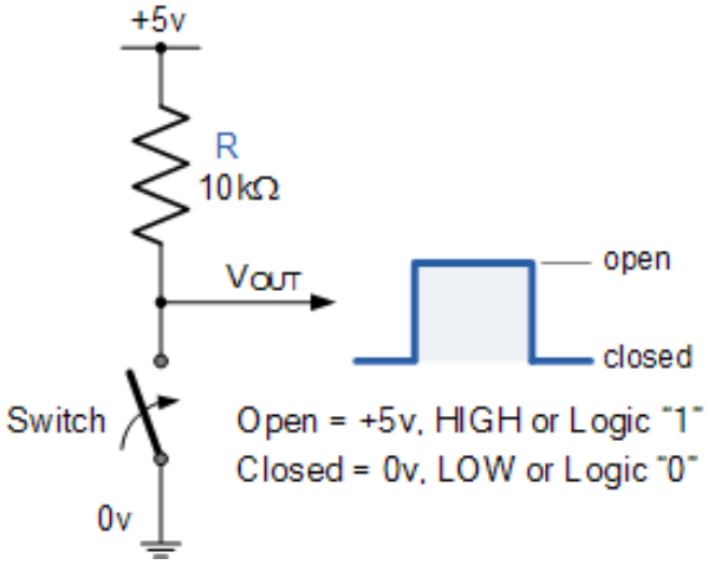
- The output buffer *is disabled*
- The Schmitt trigger input *is deactivated*, providing zero consumption for every analog value of the I/O pin. The output of the Schmitt trigger **is forced** to a constant value (0).
- The pull-up and pull-down resistors *are disabled* by hardware
- Read access to the input data register gets the value 0.

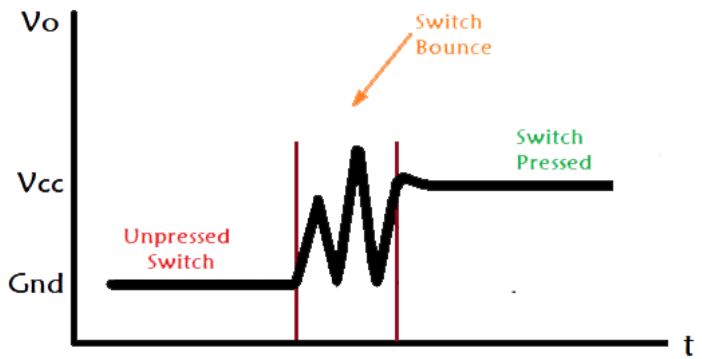
- *Note: Fore STM32F40XX MCUs, in the analog configuration, the I/O pins cannot be 5 Volt tolerant.*

Tips and Best Practices- Digital I/O

Switch Debouncing

- Switches and push-buttons are mechanical devices that have two or more sets of electrical contacts
- The most common way of input interfacing a switch (or push button) to an electronic circuit is via a pull-up resistor to the supply voltage as shown
- mechanical switches have a common problem called “contact bounce”

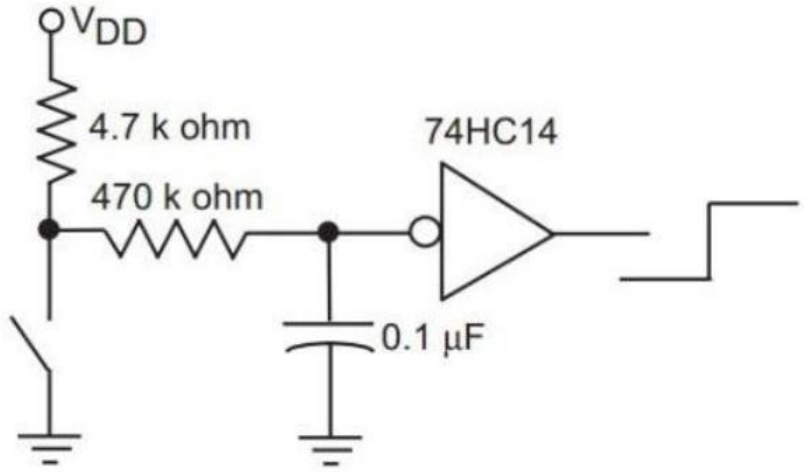




Tips and Best Practices- Digital I/O

Switch Debouncing

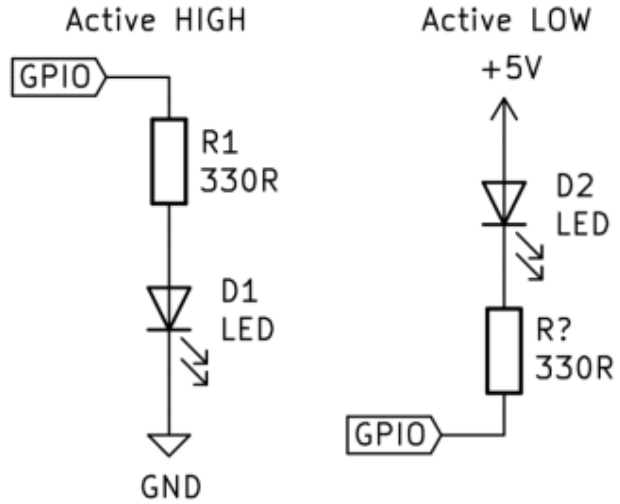
- To solve this issue there is two way:
  - Hardware
  - Software
- The usual hardware way is to build a low pass filter. Sometimes you will find a Schmitt-trigger after the RC filter, to avoid hysteresis.

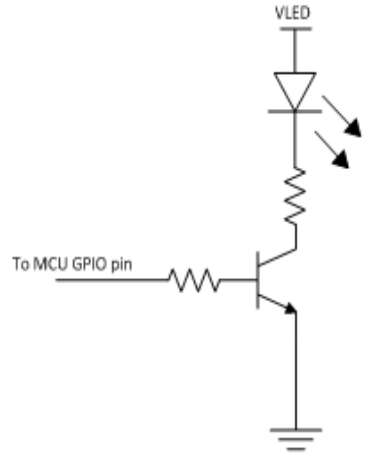


Tips and Best Practices- Digital I/O

Driving LEDs

- A simple method for driving a low-power LED can be wired directly to a GPIO
- A white LED needs a typical ~20 mA current under a typical 3.3 V supply
- As STM32 devices maximum sink current is 25 mA, there is not enough margin to directly drive a LED
- Usually, you need to use a transistor and to place a resistor in series with the LED to set the forward current

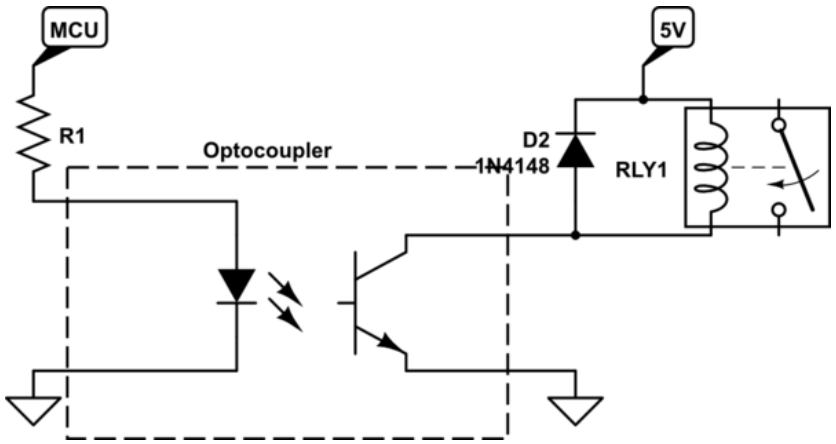




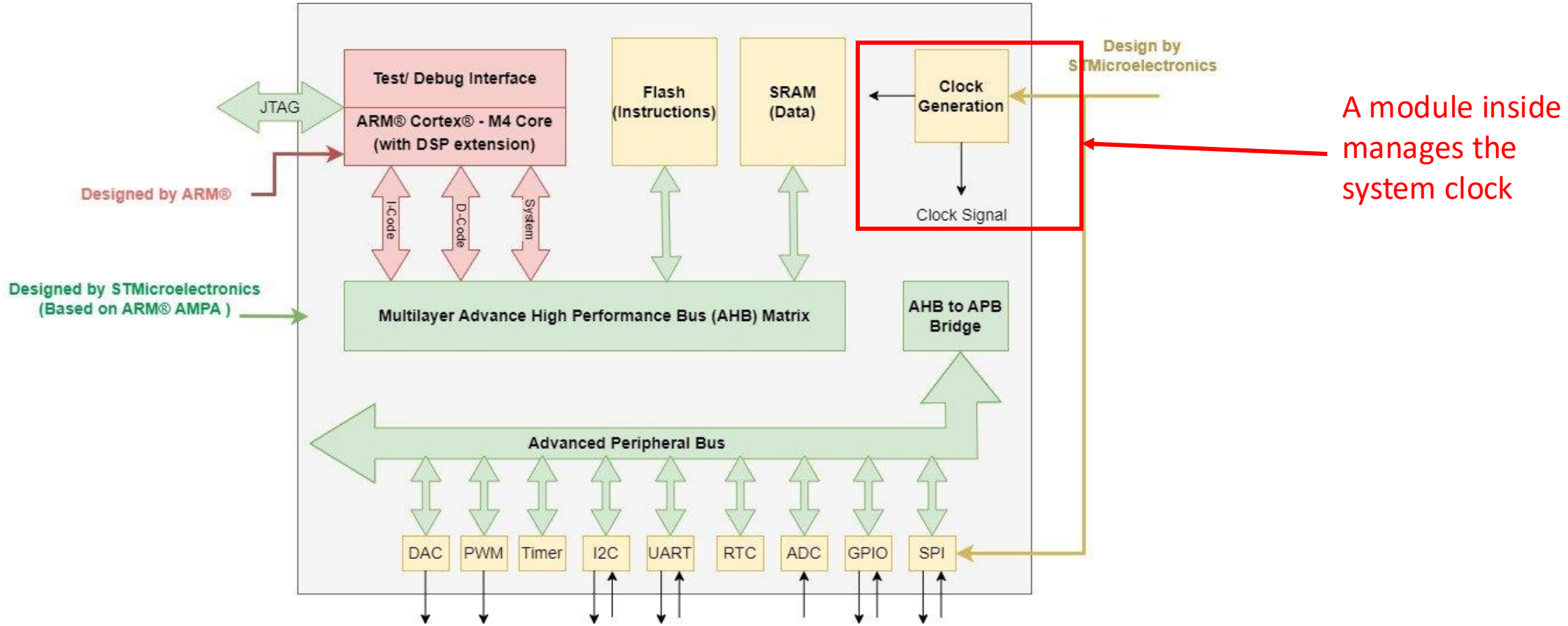
Tips and Best Practices- Digital I/O

Optocouplers

- Optocouplers (or optoisolator) create a safe connection between high-voltage equipment and microcontrollers with a means of complete electrical insulation.
- An Optocoupler is an electronic component with an LED and photo-sensitive device, such as a photodiode or phototransistor encased in the same package
- The advantage here is that the optical switch can be used for input interfacing harmful voltage levels onto the input pins of microcontrollers



Reset and Clock Control (RCC) in STM32



Reset and Clock Control (RCC) in STM32

- The **RCC block** in STM32 microcontrollers manages the system clock and reset operations.
- Provides the main clock sources and distributes clocks to various peripherals.
- **Clock Sources:**
  - **HSI** (High-Speed Internal) – internal RC oscillator.
  - **HSE** (High-Speed External) – external crystal oscillator.
  - **LSI** (Low-Speed Internal) – low-power internal oscillator for low-speed operations.
  - **LSE** (Low-Speed External) – external crystal for Real-Time Clock (RTC).
  - **PLL** (Phase-Locked Loop) – generates higher frequencies by multiplying input clock.

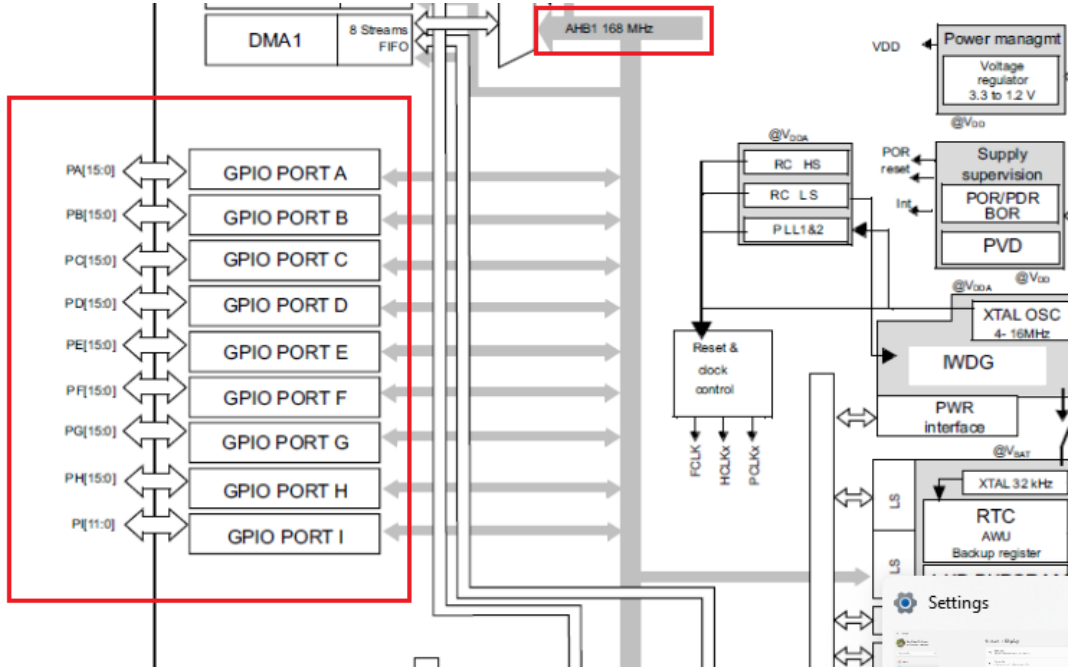
Reset and Clock Control (RCC) in STM32

Clock Trees:

- **AHB (Advanced High-Performance Bus):** for core and high-speed peripherals.
- **APB1 / APB2 (Advanced Peripheral Bus):** for general-purpose peripherals and low-speed peripherals.

- **Important Note!!**

- In STM32 microcontrollers, peripheral clocks, including those for GPIO ports, are **disabled by default** to save power.
- To use a peripheral like a GPIO port, you must explicitly enable its clock in the RCC before configuring or using it.



Reset and Clock Control (RCC) in STM32

7.3.10 RCC AHB1 peripheral clock enable 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	Reserved		DMA2EN	DMA1EN	CCMDATARAMEN	Res.	BKPSRAMEN	Reserved	
	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	Reserved			GPIOIEN	GPIOHEN	GPIOGEN	GPIOFEN	GPIOEEN	GPIODEN	GPIOCEN	GPIOBEN	GPIOAEN
			rw				rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 2 **GPIOCEN:** IO port C clock enable
Set and cleared by software.
0: IO port C clock disabled
1: IO port C clock enabled

Bit 1 **GPIOBEN:** IO port B clock enable
Set and cleared by software.
0: IO port B clock disabled
1: IO port B clock enabled

Bit 0 **GPIOAEN:** IO port A clock enable
Set and cleared by software.
0: IO port A clock disabled
1: IO port A clock enabled

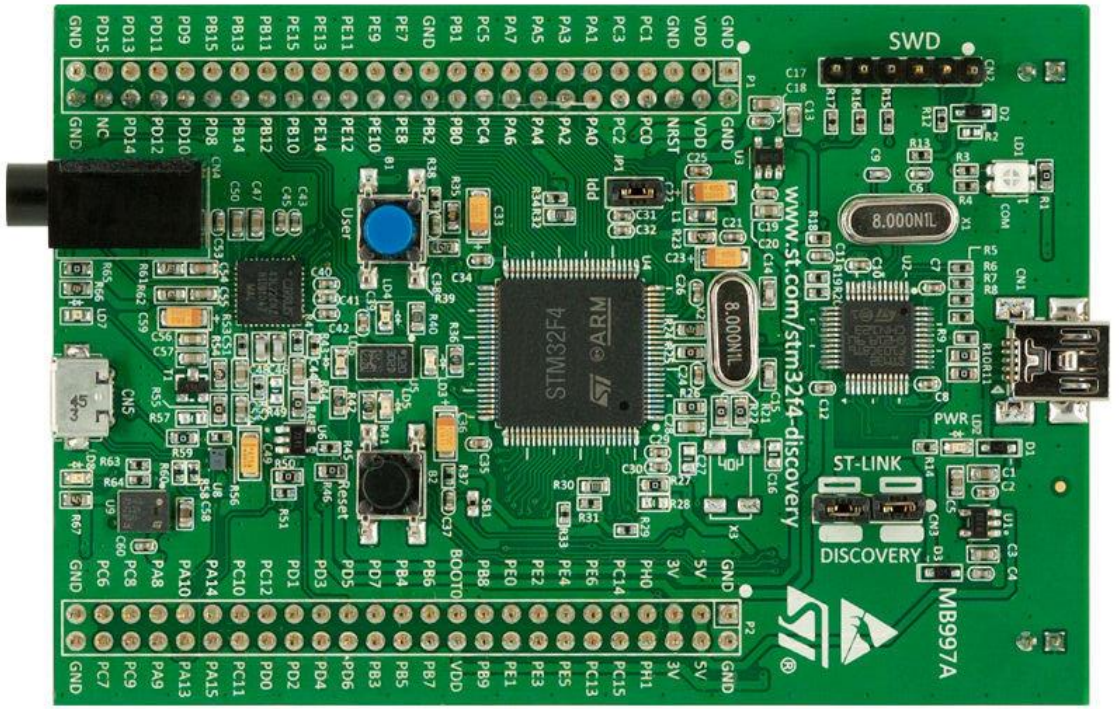
GPIO Configuration Example

For a better understanding of how microcontrollers function through programming, it’s crucial to explore different programming approaches.

- **Bare-metal programming (register-based)**
  - You need to carefully review the datasheet and reference manual in order to fully understand all the different hardware configurations and how they operate in this step.
- **Using CMSIS library**
  - Cortex Microcontroller Software Interface Standard (**CMSIS**) library can be used at this step
- **Using Low-level library**
  - When you want to work at a relatively low level but with a more organized and manufacturer-specific **API**.
  - Low-level libraries, sometimes provided by the microcontroller manufacturer, offer a set of functions and macros to configure and control the microcontroller's hardware.
- **Using High-level Library (HAL)**
  - HAL is a higher-level library that abstracts the microcontroller's hardware, providing a more user-friendly API for developers

GPIO Configuration Example

- The following slides will show an example of implementing the previous approaches to **blink a LED** using STM32F4DISCOVERY kit. However, you can use any STM32 board.

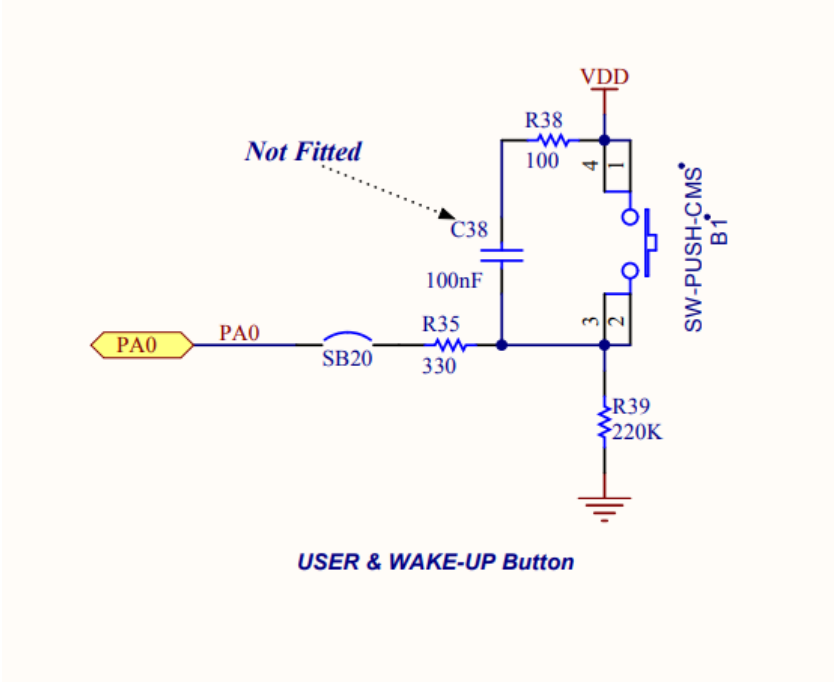


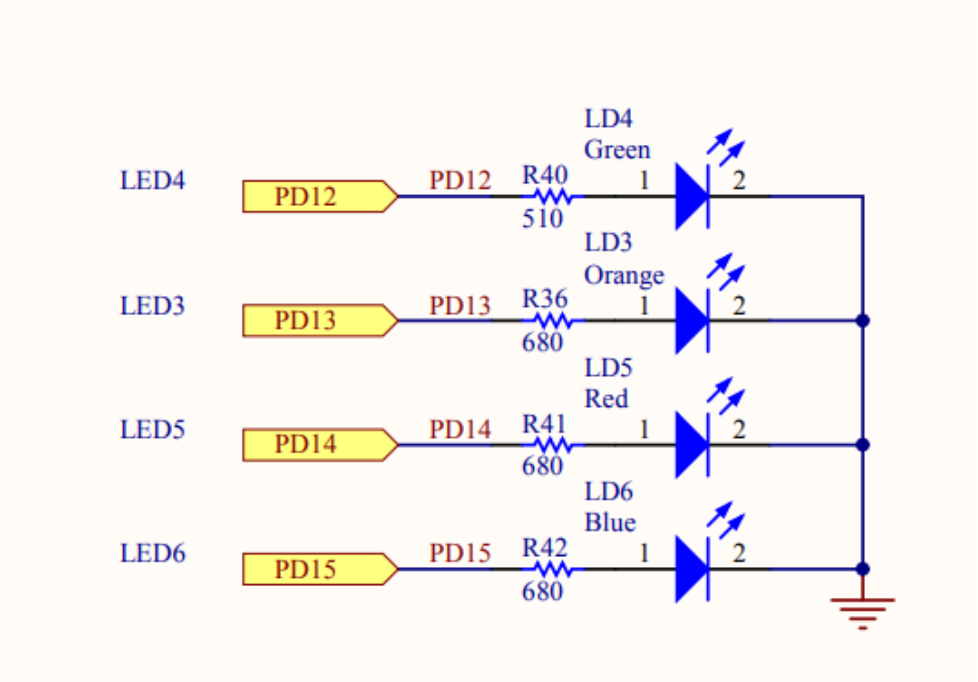
GPIO Configuration Example

- **Required documents:**
  - STM32F407 **datasheet**
  - STM32F407 **reference manual**
  - STM32F4DISCOVERY Discovery kit **user manual**
  - STM32F4DISCOVERY Discovery kit **schematic diagram**
- **Required IDE:**
  - STM32CubeIDE

GPIO Configuration Example

- STM32F4DISCOVERY has LEDs and a User-Button to demonstrate the digital I/O features and configurations





GPIO Configuration Example

- **Bare-metal programming (register-based)**

1. **Locate registers**
   - Check the Memory Map to know the address of Peripherals:

Bus	Boundary Address	Peripheral
AHB1	0x4002 3800 - 0x4002 3BFF	RCC (Clock)
AHB1	0x4002 0C00 - 0x4002 0FFF	GPIOD (I/O)

   - So, define the base addresses:

```
#define RCC_BASE         0x40023800UL
#define GPIOD_BASE       0x40020C00UL
```

GPIO Configuration Example

2. Configure Clock

- Check the Memory Map to know the address of Peripherals:

RCC clock control register (RCC_CR)

Address offset: 0x00

Reset value: 0x0000 XX83 where X is undefined.

- **HSION** is enabled by default. HSI is 16 MHz.

RCC clock configuration register (RCC_CFGR)

Address offset: 0x08

Reset value: 0x0000 0000

- HSI oscillator is used as the system clock, no divider activated. System is clocked at 16 MHz (equal to HSI's Frequency)

GPIO Configuration Example

- The GPIOD is on AHB1 bus, therefore, check the clock on AHB Bus:

RCC AHB1 peripheral clock enable register (RCC_AHB1ENR)

Address offset: 0x30

Reset value: 0x0010 0000

- GPIO Clock is disabled by default. Need to enable GPIODEN bit:

```
#define RCC_AHB1ENR      RCC_BASE + 0x30
*(uint32_t *)(RCC_AHB1ENR) |= (1 << 3);
```

GPIO Configuration Example

3. Configure GPIO Port and Pin

GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

GPIO port bit set/reset register (GPIOx_BSRR) (x = A..I/J/K)

Address offset: 0x18

Reset value: 0x0000 0000

- Do the same reading and understanding as we do with Clock registers, here are what we have to do:
  - Set **Pin 15** on **Port D** to Output mode on bit **[31:30]** of the **GPIOA_MODER** register
  - To output **HIGH** value, set **bit 15** of the **GPIOA_BSRR** register
  - To output **LOW** value, set **bit 31** of the **GPIOA_BSRR** register

```
#define GPIOA_MODER     GPIOD_BASE + 0x00
#define GPIOA_BSRR      GPIOD_BASE + 0x18


*(uint32_t *)(GPIOD_MODER)  &= ~(1 << 31);
*(uint32_t *)(GPIOD_MODER)  |=  (1 << 30);
*(uint32_t *)(GPIOD_BSRR)   |=  (1 << (15));
*(uint32_t *)(GPIOD_BSRR)   |=  (1 <<
(15+16));
```

GPIO Configuration Example

4. Delay

- Since we know that the MCU is operating at 16 MHz (HSI), For a delay of 50 ms, suppose that every loop requires 10 instruction cycles, the total number of iterations is

$$\frac{0.05 \times 16,000,000}{10}$$

```
#define DELAY_MAX 80000;
for(uint32_t i=DELAY_MAX; i--;);
```

GPIO Configuration Example

4. The complete program

- To wrap up all the previous slides, the code will be:

```
#include <stdint.h>


/* Register Addresses*/
/* Clock */
#define RCC_BASE         0x40023800UL
#define RCC_AHB1ENR      RCC_BASE + 0x30


/* GPIO */
#define GPIOD_BASE       0x40020C00UL
#define GPIOD_MODER      GPIOD_BASE + 0x00
#define GPIOD_BSRR       GPIOD_BASE + 0x18


/* delay counter */
#define DELAY_MAX        80000
```

```
int main(void){
  /* turn on clock on GPIOD */
  *(uint32_t *)(RCC_AHB1ENR) |= (1 << 3);

  /* set PD15 to output mode */
  *(uint32_t *)(GPIOD_MODER)  &= ~(1 << 31);
  *(uint32_t *)(GPIOD_MODER)  |=  (1 << 30);


  while(1) {
    /* set HIGH value on pin PD15 */
    *(uint32_t *)(GPIOD_BSRR)   |=  (1 << 15);
    for(uint32_t i=DELAY_MAX; i--;);


    /* set LOW value on pin PD15 */
    *(uint32_t *)(GPIOD_BSRR)    |=  (1 <<(15+16));
    for(uint32_t i=DELAY_MAX; i--;);
  }
}
```

GPIO Configuration Example

- **CMSIS Code Example**

1. Integrate the CMSIS library

- We need to attach the required libraries (provided by ARM) to our project
- Download CMSIS Core, the latest version is CMSIS 5.9.0 (Link)
- Copy the header files in `ARM.CMSIS.5.9.0.pack/CMSIS/Core/Include` to your project
- Download CMSIS-Pack for the target MCU from (Link)
- Copy the header files in the following folder to your project file:

  `Keil.STM32F4xx_DFP.2.17.1\Drivers\CMSIS\Device\ST\STM32F4xx\Include`
- Add the following header paths to the project **path and symbols**

  `CMSIS/Core/Include and CMSIS/Device/ST/STM32F4xx/Include`

GPIO Configuration Example

1. Configure Clock

- Use the RCC peripheral, set the bit RCC_AHB1ENR_GPIOAEN in the AHB1ENR register to enable clock for GPIOA

```
RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN;
```

2. Configure GPIO Port and Pin

```
GPIOD->MODER &= ~GPIO_MODER_MODER15_1;
GPIOD->MODER |=  GPIO_MODER_MODER15_0;
GPIOD->BSRR |= GPIO_BSRR_BS_15;
GPIOD->BSRR |= GPIO_BSRR_BR_15;
```

GPIO Configuration Example

3. The complete program

```
#include <stdint.h>
#include <stm32f4xx.h>
/* delay counter */
#define DELAY_MAX       50000

int main(void){
  /* turn on clock on GPIOD */
  RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN;

  /* set PD15 to output mode */
  GPIOD->MODER &= ~GPIO_MODER_MODE15_1;
  GPIOD->MODER |=  GPIO_MODER_MODE15_0;

  while(1) {
      /* set HIGH value on pin PD15 */
      GPIOD->BSRR |= GPIO_BSRR_BS_15;
      for(uint32_t i=DELAY_MAX; i--;);

      /* set LOW value on pin PD15 */
      GPIOD->BSRR |= GPIO_BSRR_BR_15;
      for(uint32_t i=DELAY_MAX; i--;);
    }
}
```

GPIO Configuration Example

- **Low-Level Generated Code Example**

1. **Configure the GPIO using the GUI tool provided by CubeIDE**
2. **Make sure the Driver Selector is set to LL**
3. **Write the program using LL APIs**

```c
int main(void)
{
  /* Initialization begin  */
        /*.*/
        /*.*/
  /* Initialization end  */


  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */


    /* USER CODE BEGIN 3 */
      LL_GPIO_SetOutputPin( LED_GPIO_Port, LED_Pin );
      LL_mDelay(DELAY_MAX);
      LL_GPIO_ResetOutputPin( LED_GPIO_Port, LED_Pin);
      LL_mDelay(DELAY_MAX);
  }
  /* USER CODE END 3 */
}
```

GPIO Configuration Example

- **HAL Generated Code Example**

1. **Configure the GPIO using the GUI tool provided by CubeIDE**
2. **Make sure the Driver Selector is set to HAL**
3. **Write the program using HAL APIs**

```
int main(void)
{
  /* Initialization begin  */
        /*.*/
        /*.*/
  /* Initialization end  */


  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */


    /* USER CODE BEGIN 3 */
    HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin,SET);
    HAL_Delay(50);
    HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin,RESET);
    HAL_Delay(50);

  }
  /* USER CODE END 3 */
}
```

Questions?