ALL ABOUT GPIOS

A general guide for microcontroller inputs/outputs.

Introduction

There is something in common among all the microcontrollers you can find in the market: they have pins. While some pins have a predefined function, like the RESET pin or serial communication RX/TX pins, some others can be used as a **General Purpose Input/Output (GPIO)**.

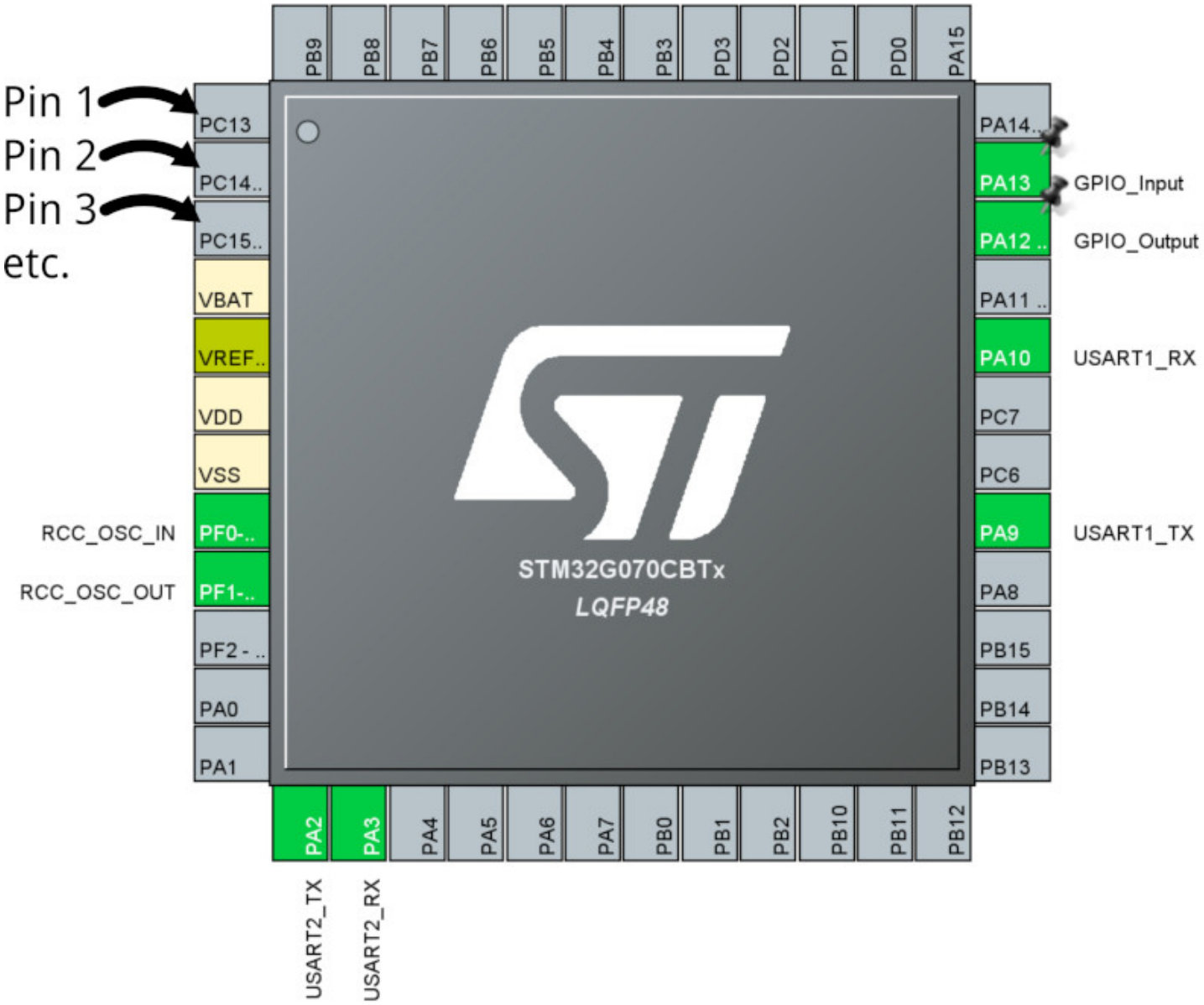
As the name indicates purpose of this kind of pin is *generic*, as they don't perform any specific function. They are there for you to configure them and use them as inputs or outputs.

In this guide we will learn what a GPIO does and the different ways you can configure and use them. The text is oriented to embedded developers and contains basic and to-the-point explanations, as a general reference.

We'll be using the STmicroelectronics STM32F401 microcontroller in many of the examples, but much of the concepts explained here are valid for many chips of different brands.

Basics and conventions

The way a microcontroller is connected to other components in the circuit is through its pins.



An STM32 microcontroller with its pins, as shown in the STM32CubeMX software.

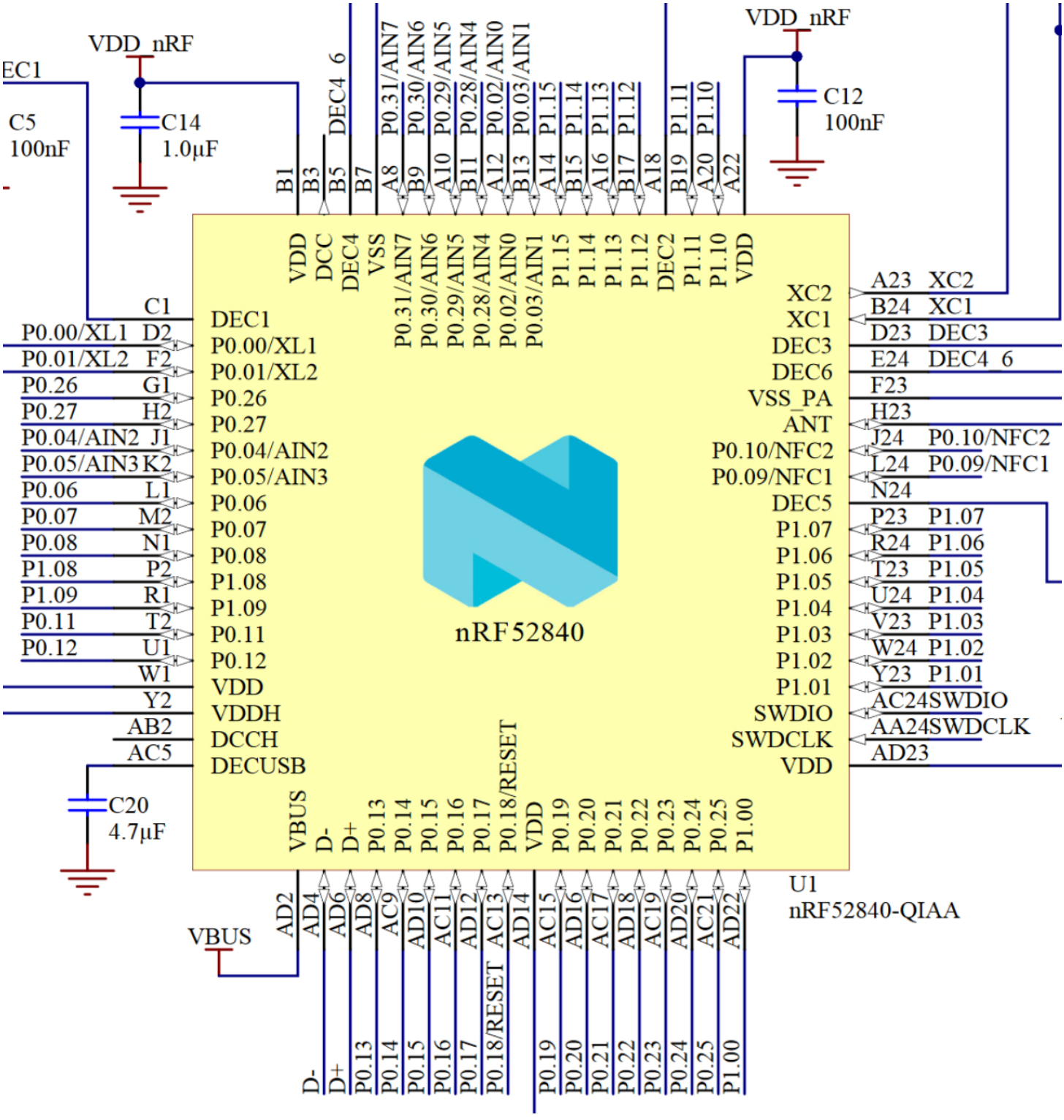
Some pins have specific functions like the above "*VDD*" (where the power supply is connected) or "*VSS*" (where the ground is connected) and cannot be changed. Other pins can be configured through software at runtime, by instructing them to perform a specific function (i.e. I2C) or as a GPIO.

Many of the microcontrollers in the market put their pins into groups called *ports*. The name of the port can vary according to the brand and model of the chip, but it's usually a letter, for example "*Port A*".

Every port can hold a fixed number of pins. These pins within a port have a number and will be referred as *P + name of the port + number of pin*. For example, the *pin number 0* of the *Port A* will be called **PA0**.

Note that this naming is be useful to you when programming the microcontroller or referring to a given pin, but it has nothing to do with the *REAL* pin number: note the little circle at the top-left of the chip in the above picture. That is is pin 1, and you can enumerate all the pins by counting them counter-clockwise. That is, *PC13* is the pin number 1, *PC14* is pin 2, *VBAT* is pin 4.

Some brands like to number the *ports* with numbers, like *Port 1*, but for this guide we will stick with the *letter* convention.



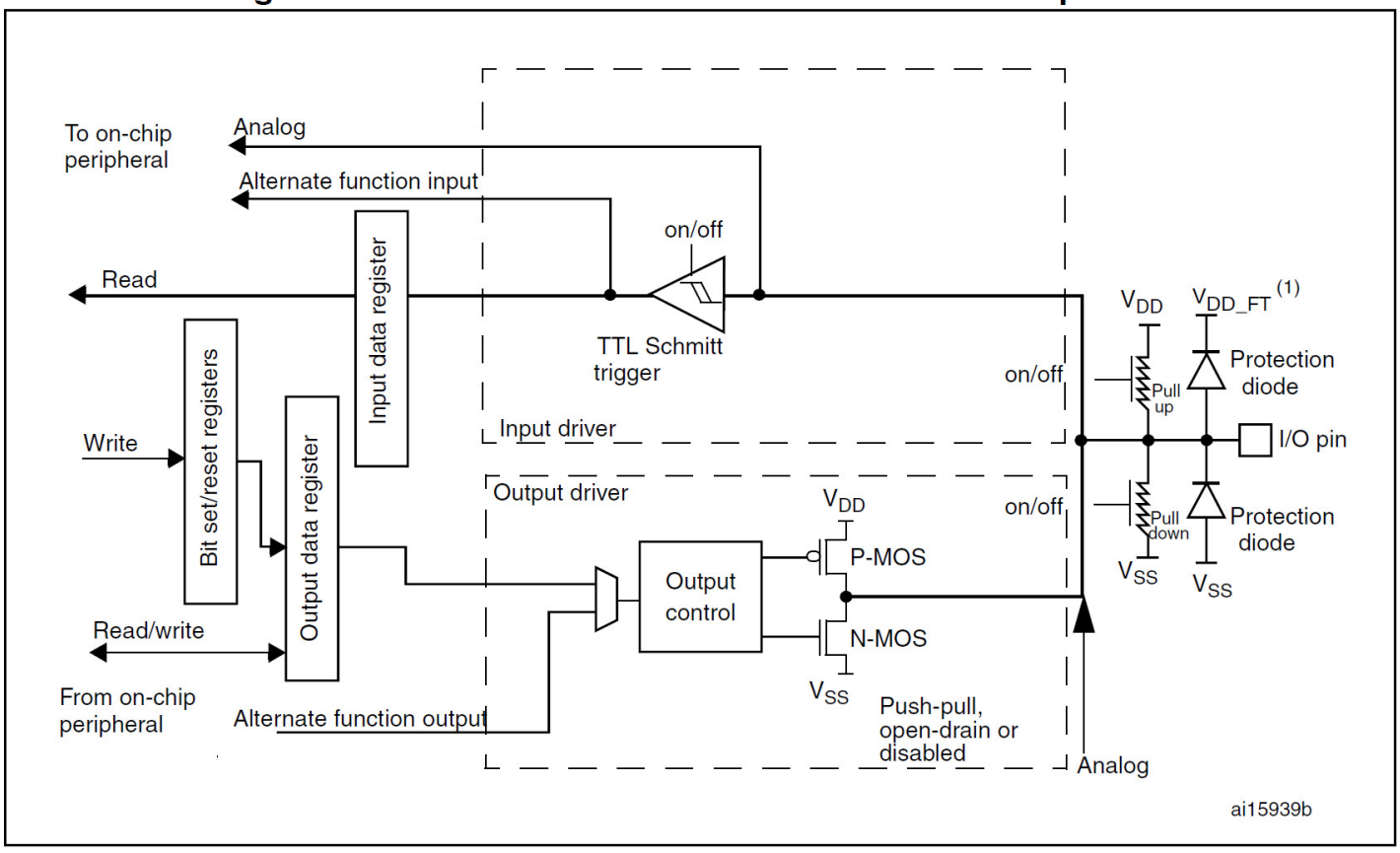
Nordic NRF52840 pinout. This chip uses numbers to name the Ports.

Modern microcontrollers can share different functions in many of their pins. In STM32 world this is called “Alternate function" (more on this later). This means that a pin can be configured to either have a specific peripheral function (like I2C) or to act as a GPIO.

Function and structure

Let's have a look at what a GPIO pin might look like internally.

Figure 16. Basic structure of a five-volt tolerant I/O port bit



Typical structure of a GPIO pin. This is from an STM32F401 microcontroller.

You can see in the above picture, in the right side, there is a square with the *“I/O pin”* label. That square represents the physical pin that goes outside the microcontroller package.

The pin is then connected internally to an *input driver* and an *output driver* (enclosed in the dotted squares). Which driver the pin is going to use is decided by configuring the pin through code. This means that once a pin is configured to act as a GPIO, it can have one of the following functions: **input** or **output**.

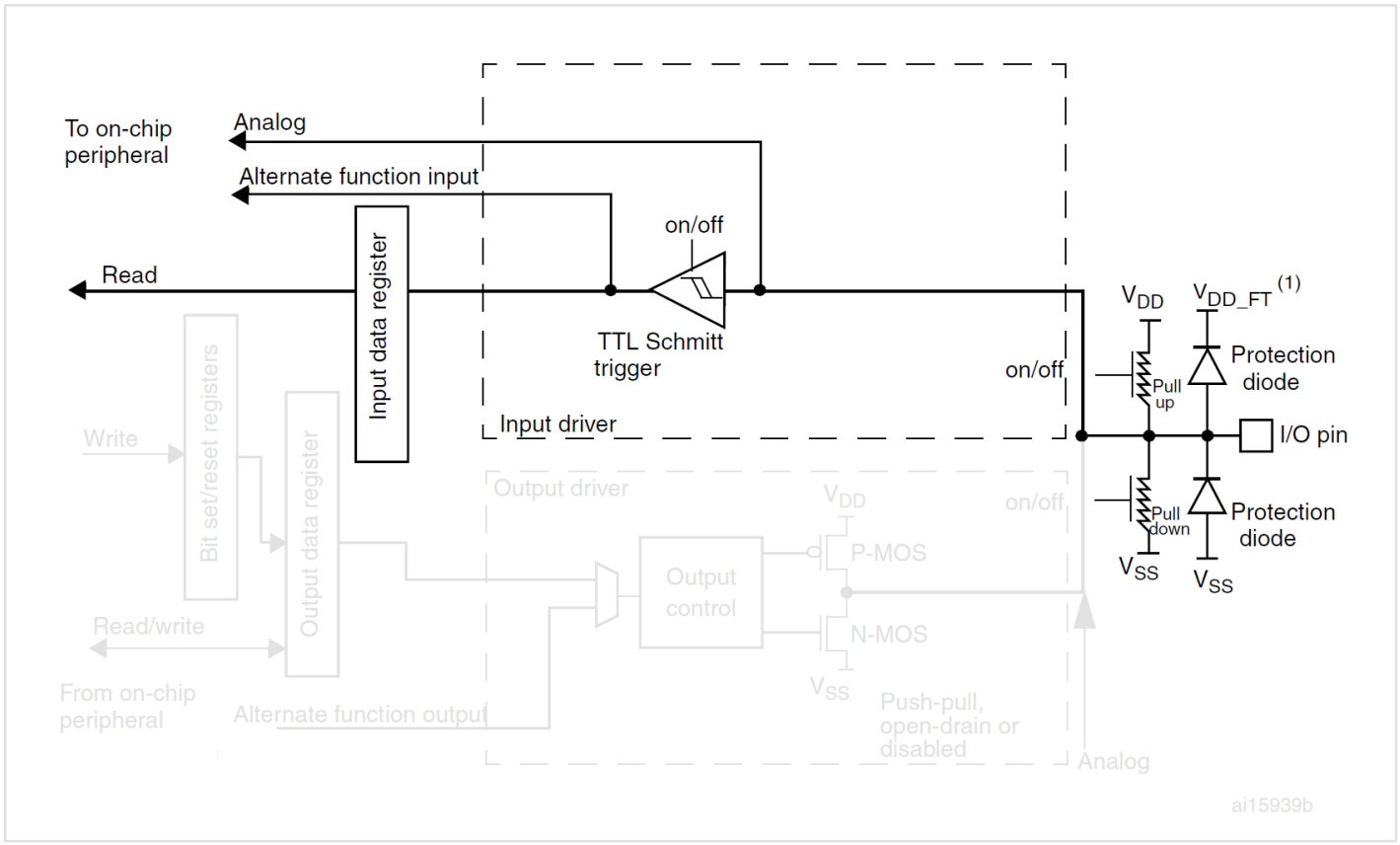
Just before the output drivers there are the “Input Data Register” and “Output Data Register". These registers are in a specific memory-mapped area from where you can read the state of an input or set the state of an output. That is, by reading or writing to this specific memory-mapped area you can interact with a physical pin of the microcontroller. Along with the Input/Output Data Registers, there are other registers that allows you to configure some characteristics of the GPIO.

Usually these registers are port-wide, for example there is the Input Data Register for Port A, the Input Data Register for Port B, and so on. This means that these registers can be used to read, write and configure all the pins for a given port. For example, you can read the logical value of the input pin PC3 by checking the bit 3 of the Input Data Register for Port C.

GPIO as an input

When the GPIO is configured as an input, it is possible to *read* the state of an *“incoming”* signal into the pin. This signal can be either a digital state (high or low) or an analog signal.

Figure 16. Basic structure of a five-volt tolerant I/O port bit



You can know the state of a pin by reading the corresponding bit from the Input Data Register in your microcontroller, and typically you will read a value of *‘1’* when the pin has a logical *high* state (3.3V for example), and *‘0’* when the pin has a logical *low* state (0V or

ground).

It is common to find the following options when configuring a GPIO as an input:

Pull-up, pull-down resistors

Optionally you can enable either a pull-up or pull-down resistor. Typically this resistor has a value above 40kΩ (what's is called a *weak* pull-up/pull-down).

Some old microcontrollers allow you to configure the resistors only across a port (you cannot enable the resistors individually by pin). This limitation is not present anymore in modern mainstream microcontrollers.

Floating

This mode is simply a pin configured as an input that has no pull-up/pull-down resistor. It's usually called a *floating* or *high-impedance* input.

Also, most of the pins of the microcontroller (if not all) will be in this *floating* configuration when the microcontroller is freshly out of reset.

Analog

You can configure the input as an *analog* input. This mode connects the pin to an internal ADC (analog-to-digital converter) and allows you to read a value that represents a given voltage in a pin. The value depends on the resolution of the ADC, for example a 12-bit ADC can have values that go from 0 to 4095. This value is mapped to a voltage that is between 0V and (usually) the voltage the microcontroller is running (3.3V for example).

When a GPIO is configured in analog mode the input pull-up/pull-down resistors are disconnected (floating).

Note that not all the GPIOs in a microcontroller can be configured as analog input. Refer to your microcontroller datasheet to know which ones can be configured as such.

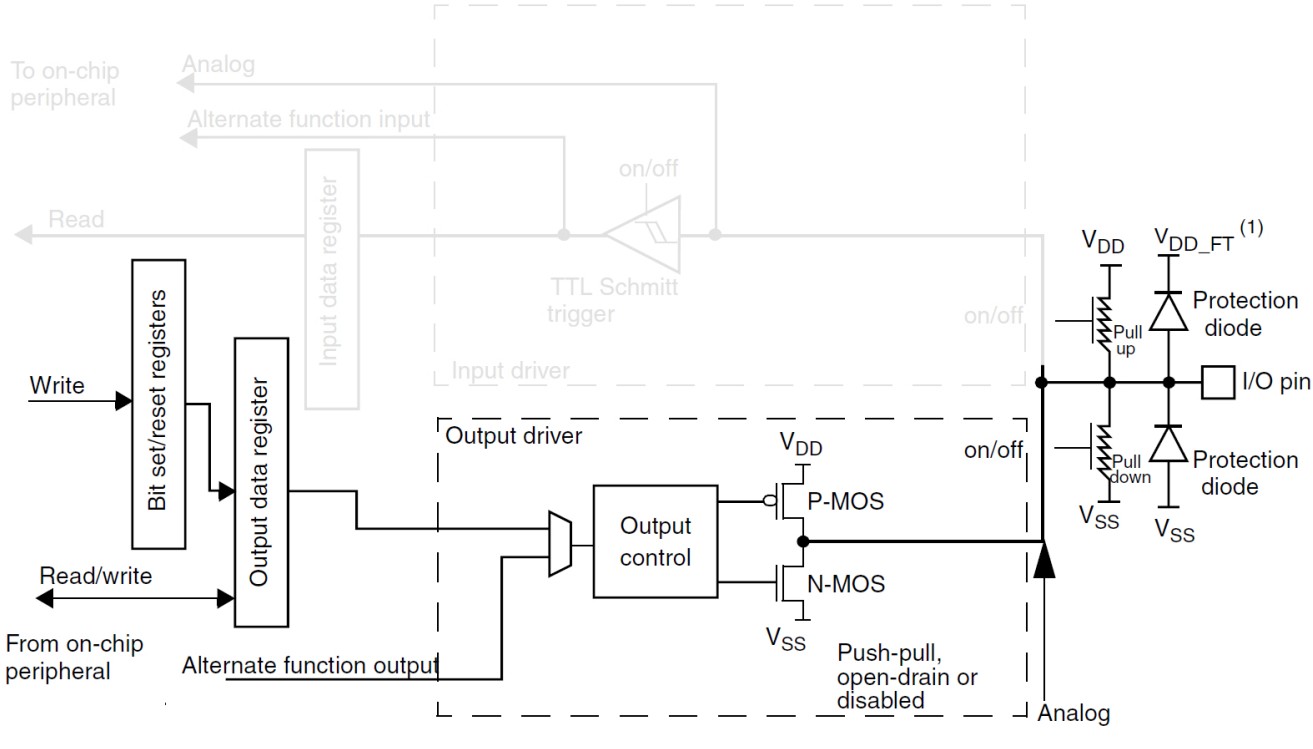
Interrupt

Optionally some pins can be *attached* to an interrupt. This way, instead of continously polling the signal state of a pin, the microcontroller will *notify* your program about a change in the signal for a given pin. For example, you can intruct the microcontroller to interrupt your program and call a special function (that you have to write) when PA1 goes from *low* to *high*. This is called an *edge-triggered* interrupt, and most microcontrollers allows you to configure an interupt for *rising*, *falling* or *both* flanks. Other microcontrollers allows you to configure *level-triggered* interrupts. That is, the interrupt is triggered when the microcontroller detects a *high* or *low* value on a signal (instead of detecting a transition/edge).

GPIO as an output

When a GPIO is configured as an output, it allows you to generate a logical signal (voltage) through a pin (high or low, for example 3.3V or 0V) by writing to the Output Data Register in your microcontroller.

Figure 16. Basic structure of a five-volt tolerant I/O port bit



Concurrency

Since the output data register may be written by your main program and perhaps an interrupt (or between threads in multithreaded environments), you should be aware that a writing to the Output Data Register can be interrupted *half-way*. Typically a write to the Output Data Register is made in 3 steps: read the register, set/reset the bit for the output and then write the resulting value to the Output data Register. To avoid concurrency issues, all the mentioned steps should be performed *atomically*.

Some microcontrollers have special registers that serve two purposes: avoiding concurrency problems and reduce the number of instructions required to set/reset an output pin. In the case of the STM32F401 this register is called `BSRR` (port bit set/reset register) and allows you to set or reset the Output Data Registers bits atomically.

Now, among the common functions of an output, you can typically find the following:

Push-pull output

This kind of output configuration uses the P-MOS and N-MOS mosfets of the output stage (as shown in the picture of the structure of the GPIO) to either push the voltage *low* or pull the voltage *high*, thus generating 0V or 3.3V signal through the pin.

Open drain output

In this mode, the P-MOS of the output driver is disconnected, and the signal is only allowed to be *sunk* (driven to ground) through the N-MOS mosfet (refer to the above picture). This kind of output mode is commonly used with an internal or external pull-up resistor. That is, when you write a *'1'* in the Output Data Register, the output does nothing: it's disconnected, and the only way to generate a logical high signal is through the pull-up resistor. When you write a *'0'* in the Output Data Register, the N-MOS mosfet is activated and the signal is driven low to 0V.

This mode is the one you want to use when configuring outputs for the I2C protocol.

> To learn more about the I2C protocol click **here**.

Pull-up, pull-down resistors

Just like the input pull-up/pull-down resistors, some microcontrollers allow you to enable pull-up/pull-down resistors for the output pins.

DAC

Some microcontrollers allows you to connect some pins to a DAC (digital-to-analog converter). You can generate an analog signal through a pin, much as the inverse of a GPIO configured as an ADC input. This is a rare feature and most of the times is limited in terms of update frequency, resolution and quantity of pins enabled with this feature.

Drive-strength

This is an advanced feature some microcontrollers have, and is the possibility of selecting the *strength* an output signal can have. This means you can configure the maximum amount of current a pin can *supply* (*source* or *sink*, logical high or low). This is typically configured to adjust the output current of the pin to meet a specific load connected to it. If the *strength* is low and the load is high then the signal may not reach a desired logical state in time. If the *strength* is greater than needed then it may produce noise as ringing when toggled.

Slew rate

This is another advanced feature. Slew rate can be defined as the maximum rate of change of a signal (i.e. the time it takes to a signal to go from high to low and vice versa). Some microcontrollers have the option to set (typically) fast, normal or slow slew rate. This option usually couples with the *drive-strength* configuration so you can trim your output pin to meet the timing/noise/consumption/load requirements of your circuit.

Electrical characteristics

You will usually refer to your microcontroller datasheet to know about the electrical characteristics of the GPIO. Here is a list of the most common characteristics you will be looking at. We'll use as an example the datasheet of the STM32F401.

V_{IL} and V_{IH}

V_{IL} is the maximum voltage of an input signal to be considered a logical *low* signal. V_{IH} is the minimum voltage of an input signal be considered a logical *high* signal. These are important values, especially in situations where you are scratching your head while you measure a signal that is clearly greater than 0V but still the microcontroller reads it as a *'0'*.

It is typically shown in a table like the following:

Table 54. I/O static characteristics

Symbol	Parameter	Conditions	Min	Typ	Max	Unit
V_{IL}	FT, and NRST I/O input low level voltage	1.7 V≤ V_{DD}≤ 3.6 V	-	-	$0.35V_{DD}-0.04^{(1)}$	V
					$0.3V_{DD}^{(2)}$	
	BOOT0 I/O input low level voltage	1.75 V≤ V_{DD} ≤3.6 V, -40 °C≤ T_A ≤ 105 °C	-	-	$0.1V_{DD}$+0.1	
		1.7 V≤ V_{DD} ≤3.6 V, 0 °C≤ T_A ≤ 105 °C	-	-		
V_{IH}	FT and NRST I/O input high level voltage[5]	1.7 V≤ V_{DD}≤ 3.6 V	$0.45V_{DD}+0.3^{(1)}$	-	-	V
			$0.4V_{DD}^{(2)}$	-	-	
	BOOT0 I/O input high level voltage	1.75 V≤ V_{DD} ≤3.6 V, -40 °C≤ T_A ≤ 105 °C	$0.17V_{DD}+0.7^{(1)}$	-	-	
		1.7 V≤ V_{DD} ≤3.6 V, 0 °C≤ T_A ≤ 105 °C				

V_{IL} and V_{IH} for the STM32F401 GPIOs.

In the above case V_{IH} is 0.45 * VDD + 0.3, being VDD the voltage you are powering your microcontroller with. For a 3.3V VDD:

V_{IH} = 0.45 * 3.3V + 0.3 = 1.785V

That is, a signal that is going from *low* to *high* it is considered logically *high* when it goes above 1.785V.

For V_{IL} we have 0.35 * VDD - 0.04. For a VDD = 3.3V, V_{IL} would be:

V_{IL} = 0.35 * 3.3V - 0.04 = 1.115V

So, for a signal going from *high* to *low* it is considered logically *low* when it goes below 1.115V.

Pull-up/pull-down resistors values

You may want to know the values of the pull-up/pull-down resistors, for example in cases where there is another resistor in series connected to the pin (and thus you'll be doing a voltage divider and you need to know if the signal will meet V_{IL} and V_{IH}) or in parallel to calculate if the signal will meet the current the pin can drive.

It is usually shown in a table like this:

Symbol	Parameter		Conditions	Min	Typ	Max	Unit
R_{PU}	Weak pull-up equivalent resistor(6)	All pins except for PA10 (OTG_FS_ID)	$V_{IN} = V_{SS}$	30	40	50	kΩ
		PA10 (OTG_FS_ID)		7	10	14	
R_{PD}	Weak pull-down equivalent resistor(7)	All pins except for PA10 (OTG_FS_ID)	$V_{IN} = V_{DD}$	30	40	50	
		PA10 (OTG_FS_ID)		7	10	14	
C_{IO}(8)	I/O pin capacitance		-	-	5	-	pF

Pull-up/pull-down resistor values for the STM32F401 GPIOs.

So now we know that both the pull-up and the pull-down resistors are typically 40kΩ.

I/O capacitance (C_{io})

This value is in the same table of the *pull-up/pull-down resistors values* here above and you can usually find it with the C_{io} symbol in datasheets. It's the *parasitic* capacitance of a pin.

The effect the capacitance will provoke is a *rounded* signal when transitioning from *low* to *high* or vice versa. This is a thing you want to avoid, especially when driving high-speed devices like an SD card. You want the digital signal to transition *in-time* and in a *square* shape (unless you are dealing with electromagnetic radiations, but that's a topic for another article). To sum up, the lower the capacitance, the better.

You can also find the maximum capacitance allowed by looking at the datasheet of the peripheral you are connecting to your pin.

For your information, preserving the shape of a signal from noise and distortions is called signal integrity.

Current drive

This value tells how much current a pin can drive. There may be a table or an unique value in your microcontroller datasheet, but in the case of the STM32F401 we have this paragraph:

Output driving current

The GPIOs (general purpose input/outputs) can sink or source up to ±8 mA, and sink or source up to ±20 mA (with a relaxed V_{OL}/V_{OH}) except PC13, PC14 and PC15 which can sink or source up to ±3mA. When using the PC13 to PC15 GPIOs in output mode, the speed should not exceed 2 MHz with a maximum load of 30 pF.

In the user application, the number of I/O pins which can drive current must be limited to respect the absolute maximum rating specified in *Section 6.2*. In particular:

- The sum of the currents sourced by all the I/Os on $V_{DD,}$ plus the maximum Run consumption of the MCU sourced on $V_{DD,}$ cannot exceed the absolute maximum rating ΣI_{VDD} (see *Table 12*).
- The sum of the currents sunk by all the I/Os on V_{SS} plus the maximum Run consumption of the MCU sunk on V_{SS} cannot exceed the absolute maximum rating ΣI_{VSS} (see *Table 12*).

Output current specs for the STM32F401 GPIOs.

What it tells basically is that the current a pin can source and sink up to 8mA, or up to 20mA with *relaxed* V_{IL} and V_{IH} (V_{IL} becomes lower and V_{IH} higher), and to not exceed what it says in Table 12, that is, the sum of the currents being driven or sunk by all the pins should not be greater or lower than 120mA and -120mA respectively.

5V tolerance

Some microcontrollers specify that some pins may be tolerant to a higher voltage (higher than the voltage you are powering your microcontroller with). This means you can use a *5V tolerant* pin as an input to read voltages (or logical levels) up to 5V even if your microcontroller is powered with 3.3V.

It's usually safe to use it as is and supply 5V to these pins. The only thing you have to have in mind is that while a pin can receive 5V, it won't generate 5V when used as an output. This generates confusion when using these kind of pins with, for example, SPI peripherals that are compatible with 5V signals only. One may think that it will work OK since our pins are 5V tolerant, but in fact it may not since our microcontroller will be sending data and clock pulses going from 0V to 3.3V, and the SPI peripheral may not be detecting them because it has a V_{IH} greater than 3.3V. In these cases it's better to put a logic level converter between your microcontroller and the SPI peripheral.

Troubleshooting tips

The following is a quick list of things to check when a GPIO is not working:

- Port not clocked: most microcontrollers can enable/disable the clock to the internal peripherals, including individual GPIO ports. Check that you have correctly enabled the clock of the port before doing any operation on it.
- Peripheral not clocked: the same applies to internal peripherals, like the ADC, you are connecting your pin to. In this case you have to enable the clock for both GPIO port and ADC.
- Operating on the wrong pin: sometimes is easy to make an error when addressing a pin, specifically if you are using libraries with macros that are easy to misinterpret. For example, the STmicroelectronics libraries contain the macro `GPIO_Pin_1` that is a bitmask and you use it with structures like `GPIO_InitTypeDef`, and macros like `GPIO_PinSource0` to `GPIO_PinSource15` that are values from 0 to 15 and are used with the `GPIO_PinAFConfig` function.
- Operating on the wrong port: yes, it can happen. Sometimes in a copy-paste spree for GPIO initialization code, you modified the code for the pins you are initializing but forgot to change the port.
- Interrupt not triggering: the interrupt might be already triggered before you enabled the interrupt (NVIC or in the interrupt controller). In this case it could be wise to check if the interrupt was already triggered and clear the interrupt pending bit right after/before enabling the interrupt.

As a general advice, the best way to quickly confirm that a GPIO is correctly configured is to plug a JTAG, open the GPIO port register in your IDE “watch window" and verify that the mode (input/output), pull-up and Data registers are correctly set. If you are testing an output, you can also stop the program and write into the Data Register and confirm that the pin is *moving* (with an oscilloscope or multimeter).

If a JTAG is not available, you can dump the port registers values through a serial port and check with them with a terminal/console.

Addendum: alternate function

Most modern microcontrollers allows you to configure the pins with *alternate functions* other than a GPIO. Typically a pin has a fixed number of alternate functions to enable (one at the time), for example, the pin PA5 (note the table here below) can act either as a *GPIO*, *TIM2_CH1/TIM2_ETR* or *SPI1_SCK*.

Table 9. Alternate function mapping

Port		AF00	AF01	AF02	AF03	AF04	AF05	AF06	AF07	AF08	AF09	AF10	AF11	AF12	AF13	AF14	AF15
		SYS_AF	TIM1/TIM2	TIM3/ TIM4/ TIM5	TIM9/ TIM10/ TIM11	I2C1/I2C2/ I2C3	SPI1/SPI2/ I2S2/SPI3/ I2S3/SPI4	SPI2/I2S2/ SPI3/ I2S3	SPI3/I2S3/ USART1/ USART2	USART6	I2C2/ I2C3	OTG1_FS		SDIO			
Port A	PA0	-	TIM2_CH1/ TIM2_ETR	TIM5_CH1	-	-	-	-	USART2_ CTS	-	-	-	-	-	-	-	EVENT OUT
	PA1	-	TIM2_CH2	TIM5_CH2	-	-	-	-	USART2_ RTS	-	-	-	-	-	-	-	EVENT OUT
	PA2	-	TIM2_CH3	TIM5_CH3	TIM9_CH1	-	-	-	USART2_ TX	-	-	-	-	-	-	-	EVENT OUT
	PA3	-	TIM2_CH4	TIM5_CH4	TIM9_CH2	-		-	USART2_ RX	-	-	-	-	-	-	-	EVENT OUT
	PA4	-	-	-	-	-	SPI1_NSS	SPI3_NSS/ I2S3_WS	USART2_ CK	-	-	-	-	-	-	-	EVENT OUT
	PA5	-	TIM2_CH1/ TIM2_ETR	-	-	-	SPI1_SCK	-	-	-	-	-	-	-	-	-	EVENT OUT
	PA6	-	TIM1_BKIN	TIM3_CH1	-	-	SPI1_ MISO	-	-	-	-	-	-	-	-	-	EVENT OUT
	PA7	-	TIM1_CH1N	TIM3_CH2	-	-	SPI1_ MOSI	-	-	-	-	-	-	-	-	-	EVENT OUT
	PA8	MCO_1	TIM1_CH1	-	-	I2C3_SCL	-	-	USART1_ CK	-	-	OTG_FS_ SOF	-	-	-	-	EVENT OUT
	PA9	-	TIM1_CH2	-	-	I2C3_ SMBA	-	-	USART1_ TX	-	-	OTG_FS_ VBUS	-	-	--	-	EVENT OUT
	PA10	-	TIM1_CH3	-	-	-	-	-	USART1_ RX	-	-	OTG_FS_I D	-	-	-	-	EVENT OUT
	PA11	-	TIM1_CH4	-	-	-	-	-	USART1_ CTS	USART6_ TX	-	OTG_FS_ DM	-	-	-	-	EVENT OUT
	PA12	-	TIM1_ETR	-	-	-	-	-	USART1_ RTS	USART6_ RX	-	OTG_FS_ DP	-	-	-	-	EVENT OUT
	PA13	JTMS_ SWDIO	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PA14	JTCK_ SWCLK	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENT OUT
	PA15	JTDI	TIM2_CH1/ TIM2_ETR	-	-	-	SPI1_NSS	SPI3_NSS/ I2S3_WS	-	-	-	-	-	-	-	-	EVENT OUT

Alternate function table for the STM32F401, Port A.

Is up to you to decide which pin does what, and you do that by coding. Of course, the person who really decides this beforehand is the PCB/electrical engineer(s) who designs the board you are working with. Firmware developers working for a company are often consulted by the hardware engineer to decide which pins to use for a given function, or asked if a signal can be mapped to a different pin because it's better for the PCB routing.