Microcontroller

A microcontroller is composed of

- A microprocessor (A microprocessor on it's own doesn't have peripherals)
- Flash memory and RAM
- Peripheral port/devices

Basically, a microcontroller is a microprocessor augmented by certain other components that allow it to control specific external portions of the system (e.g. external devices, sensors, actuators, etc.)

It's usages include

- For specific control or group of control over a system
- In industry machinery
- Vehicle and autonomous vehicle, robot, IoT, aircraft, bus, train, home management

In this course we focus mainly on the Nucleo-STM32F446re Cortex-M4 microcontroller. It's features include

- ARMv7 Architecture
- 32-bit Cortex-M4 processor with FPU, 180 MHz, DSP
- Harvard Bus Matrix
- Memory: 512 Kbytes Flash Memory
- 128 Kbytes SRAM
- 3 × 12 bit ADC up to 24 Channels
- 2 × 12 bit DAC
- 16-Channel General purpose DMA
- 17-timers
- Debug Mode SWD and JTAG
- 4-I2C,
- 6-USART (two UART)
- 4-SPI
- 2–SAI (Serial Audio Interface)
- 2-CAN
- SDIO, USB 2.0
- 8 to 14-bit camera interface up to 54 Mbytes/2
- CRC calculation unit
- RTC: SubSecond Accuracy
- 96-bit Unique ID

Peripherals vs Ports vs Pins

- **Peripherals** are parts of the device that serve a specific purpose which help the microcontroller achieve a certain task. Some common microcontroller peripherals are Timers, Analog to Digital converters (ADC), Serial Peripheral Interfaces (SPI), Pulse Width Modulation and 2 wire interfaces (I2C).
- **Ports** provide direct access to the CPU and provide direct input/output to and from the CPU

- **Pins** are physical connections on a microcontroller board and may be connected to a bus through one or more ports/peripherals

The CPU is connected to these peripherals through buses collectively called the peripheral bus matrix

Peripherals

A peripheral can be generally described as a port **(not literally a MCU port)** or terminal to communicate with the outside world. Its input and output can be both analog and digital. Speed of communication may vary for data transmission and reversion, and can be asynchronous (UART) or synchronous (SPI, I2C)

Analog input is taken using ADC, and analog output using DAC. Peripherals are an autonomous component, and are memory mapped or I/O mapped. Generally not functioning if electrically isolated from the system. In many cases, it is needed to configure the supply clock and power before enabling, as well as the mode of operation, type of input/output, and (if needed) Baud rate or speed of communication

Microprocessor can access the peripheral directly and configure using memory mapped I/O or I/O mapped I/O

Memory Mapped I/O

Peripherals and I/O devices share memory space with the RAM. Same instructions (register to/from memory) are used to transfer data to I/O.
[Meaning the memory needed for Peripherals and I/O devices is in the RAM, and the processor can just write to those locations to directly configure the peripherals]

I/O Mapped I/O

Separate memory address (not in the RAM), and one address line to enable or disable I/O or memory. Special instructions required to modify these memory (in/out in Intel), and send ‘1’ to activate I/O and disable memory

Clock Configuration (System Clock)

Necessary Registers (along with descriptions of used bits)

RCC Control Register (RCC_CR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	PLLSAI RDY	PLLSAI ON	PLLI2S RDY	PLLI2S ON	PLL RDY	PLL ON	Res.	Res.	Res.	Res.	CSS ON	HSE BYP	HSE RDY	HSE ON
		r	rw	r	rw	r	rw					rw	rw	r	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HSICAL[7:0]								HSITRIM[4:0]					Res.	HSI RDY	HSI ON
r	r	r	r	r	r	r	r	rw	rw	rw	rw	rw		r	rw

RCC Clock Configuration Register (RCC_CFGR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MCO2[1:0]		MCO2 PRE[2:0]			MCO1 PRE[2:0]			Res.	MCO1		RTCPRE[4:0]				
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
PPRE2[2:0]			PPRE1[2:0]			Res.	Res.	HPRE[3:0]				SWS[1:0]		SW[1:0}	
rw	rw	rw	rw	rw	rw			rw	rw	rw	rw	r	r	rw	rw

RCC PLL Configuration Register (RCC_PLLCFGR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	PLLR[2:0]			PLLQ[3:0]				Res.	PLLSRC	Res.	Res.	Res.	Res.	PLLP[1:0]	
	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
Res.	PLLN[8:0]									PLLM[5:0]					
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

RCC APB1 peripheral clock enable register (RCC_APB1ENR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	DAC EN	PWR EN	CEC EN	CAN2 EN	CAN1 EN	FMPI2C1 EN	I2C3 EN	I2C2 EN	I2C1 EN	UART5 EN	UART4 EN	USART3 EN	USART2 EN	SPDIFRX EN
		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
SPI3 EN	SPI2 EN	Res.	Res.	WWDG EN	Res.	Res.	TIM14 EN	TIM13 EN	TIM12 EN	TIM7 EN	TIM6 EN	TIM5 EN	TIM4 EN	TIM3 EN	TIM2 EN
rw	rw			rw			rw	rw	rw	rw	rw	rw	rw	rw	rw

PWR Power Control Register (PWR_CR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	FISSR	FMSSR	UDEN[1:0]		ODSWEN	ODEN
										rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VOS[1:0]		ADCDC1	Res.	MRUDS	LPUDS	FPDS	DBP	PLS[2:0]			PVDE	CSBF	CWUF	PDDS	LPDS
rw	rw	rw		rw	rw	rw	rw	rw	rw	rw	rw	rc_w1	rc_w1	rw	rw

Procedure

1. Choose the source of the clock, from the following sources
    a. HSI - High Speed Internal Crystal
    b. HSE - High Speed External Crystal (8 MHz for F446)
2. Enable chosen crystal and wait for it to become ready **(in RCC_CR, HSE/HSI ON, wait for HSE/HSI RDY)**
3. Next we need to provide power to the clock and peripherals, by enabling the power interface **(in RCC_APB1ENR, PWREN)** and then manipulating the main internal voltage regulator output voltage **(in PWR_CR, VOS [we set it to 11])**
4. Configure the flash memory and flash latency **(using the FLASH_ACR)**
5. Configure the prescalars for the buses (AHB1, APB1, APB2 bus) by configuring their values in the configuration register **(in RCC_CFGR, HPRE, PPRE1, PPRE2)**
6. Now choose whether to use the clock directly or use PLL or Phase Lock Loop, if using clock directly, skip these next two steps (skip to step 9)
7. Configure the main PLL, by using the 3 PLL prescalars M, N, P (divide, multiply, divide) **(in RCC_PLLCFGR, using PLL_M, PLL_N, and PLL_P)** and configuring the source of the PLL to be the initially chosen crystal (HSI or HSE) **(in RCC_PLLCFGR, using PLLSRC)**
8. Enabling the PLL and waiting for it to become ready **(in RCC_CR, PLL ON, wait for PLL RDY)**
9. Select the clock source (either HSE, HSI or PLL) and wait for it to become ready **(in RCC_CFGR, using SW, wait for SWS)**

GPIO

Input/Output pins are the entrances of the microprocessor to communicate with the outside world.

- Analog In/Out Communication
- Digital In/Out communication

GPIO pins are connected to the processor through GPIO ports (A...H) and each can be configured to one of 4 modes

- Input
- General Purpose Output
- Alternate Function Mode
- Analog Mode

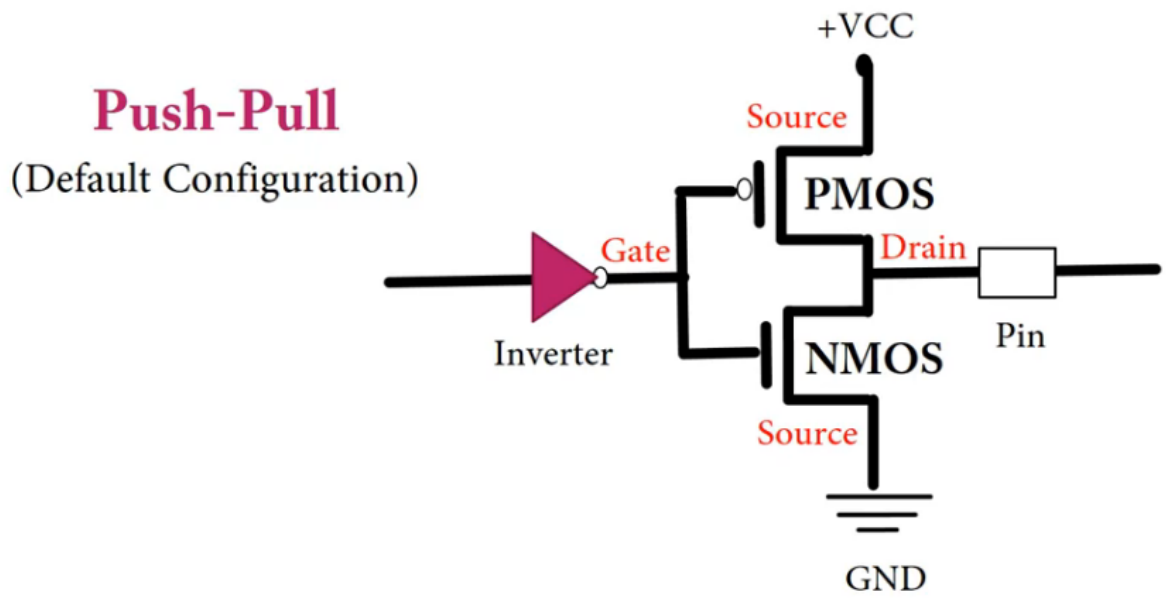
Input states:

- floating (Input + no PUPD)
- pull-up/pull-down (Input + PU/PD)
- analog (Analog)

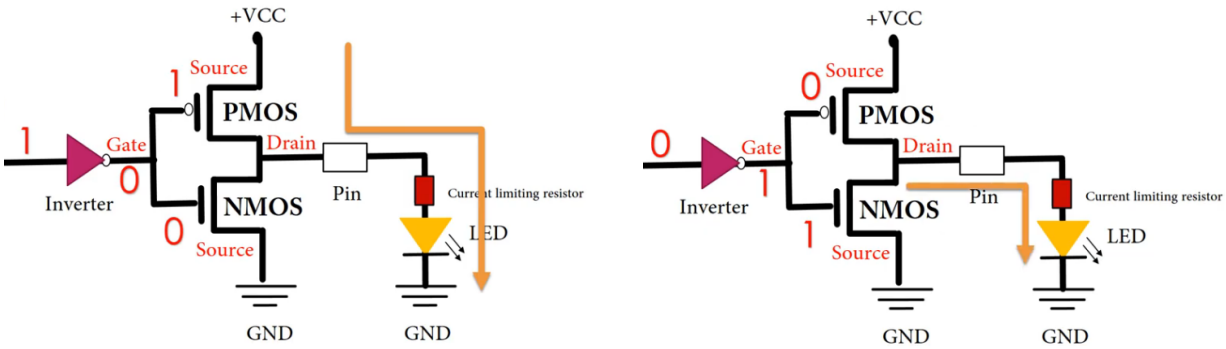
Output states:

- push-pull(Output + PushPull)
- open drain + pull-up/down (Output + Open Drain + PU/PD)

Push-Pull

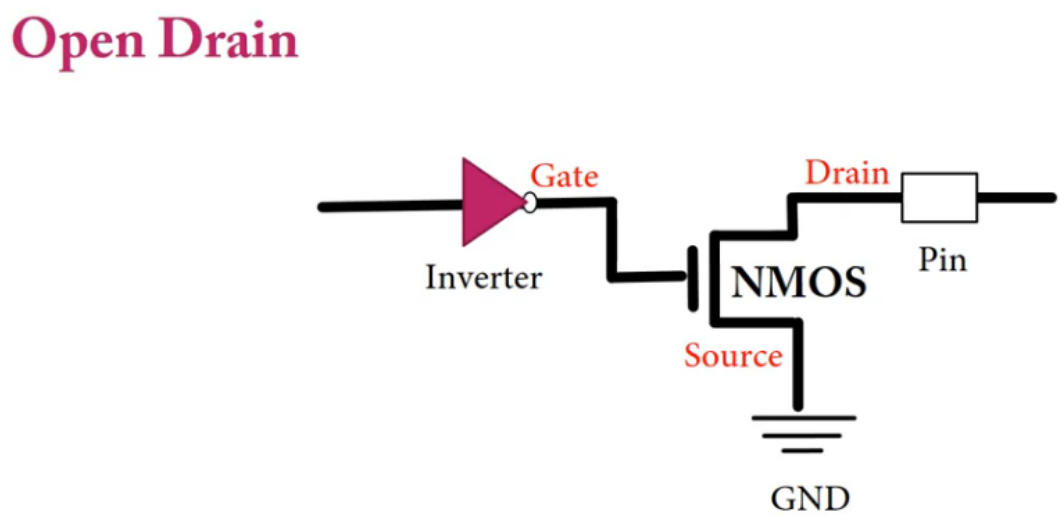


In push pull output mode, when a signal of 1 is received (e.g. when the ODR bit corresponding to this GPIO pin is set to 1), the upper PMOS transistor allows the passage of current (NMOS is inactive) and thus, there is an output of VCC. Conversely, when a signal of 0 is received, the lower NMOS transistor is active and the PMOS is inactive, thus pulling the output to ground.



Here we show, output in Push-Pull mode, and ground in Push-Pull mode

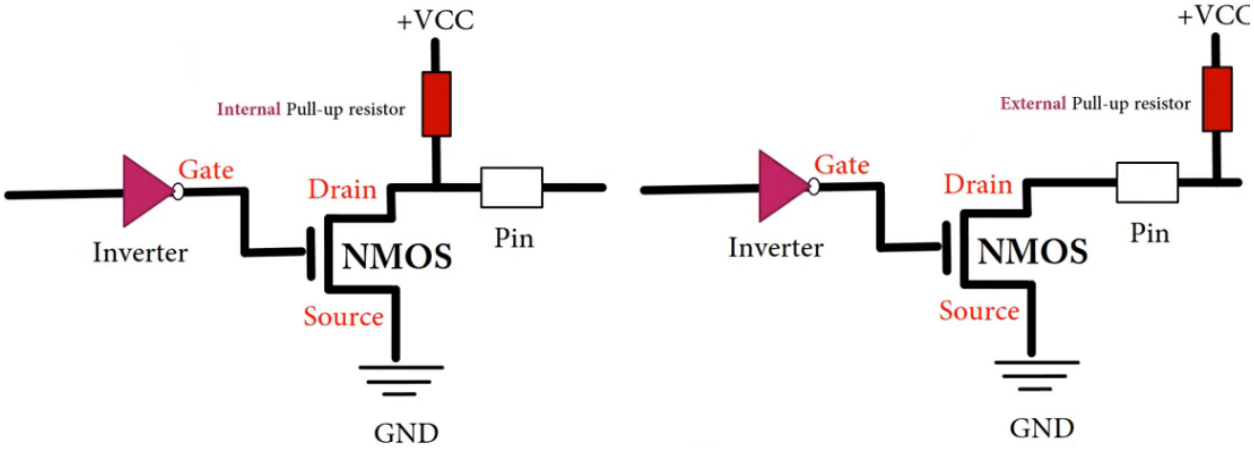
Open Drain



In open drain, when a signal of 0 is received, the output is pulled to ground, and when a signal of 1 is received, output is floating (no VCC or ground, in fact no connection at all).

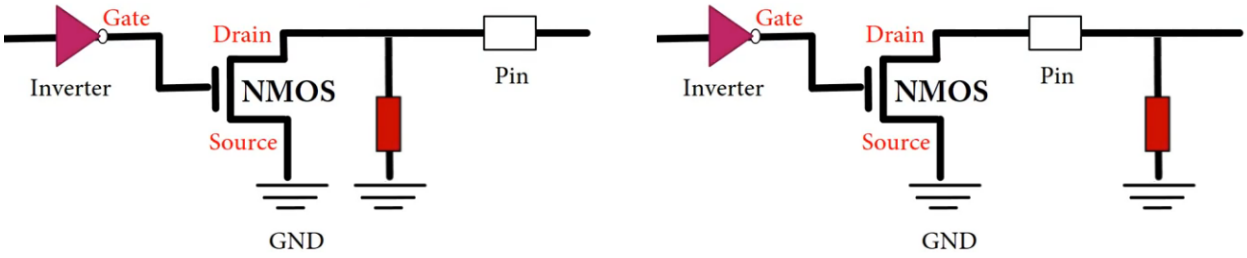
Floating state is also a possibility during input (if no connection is provided to the pin). As this is undesirable for General Purpose I/O **(might be used for other cases)**, open drain (as well as input) is used in conjunction with a pull-up or pull-down resistor (whether internal or external)

Pull-Up



Internal Pull-Up can be configured within the processor by manipulating relevant registers **(GPIOx_PUPDR).** Using this resistor, we can ensure that when a signal of 1 is received, an output of VCC is obtained.

Pull-Down



Pull-Down is the same but with the resistor being connected to ground. This isn't particularly useful for output, as both 1 and 0 will provide an output of ground, but it will pull floating inputs to ground.

Necessary Registers (along with descriptions of used bits)

GPIO port mode register (GPIOx_MODER)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
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
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

GPIO port output type register (GPIOx_OTYPER)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15 [1:0]		OSPEEDR14 [1:0]		OSPEEDR13 [1:0]		OSPEEDR12 [1:0]		OSPEEDR11 [1:0]		OSPEEDR10 [1:0]		OSPEEDR9 [1:0]		OSPEEDR8 [1:0]	
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
OSPEEDR7 [1:0]		OSPEEDR6 [1:0]		OSPEEDR5 [1:0]		OSPEEDR4 [1:0]		OSPEEDR3[1:0]		OSPEEDR2 [1:0]		OSPEEDR1 [1:0]		OSPEEDR0 1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

GPIO port output speed register (GPIOx_OSPEEDR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

GPIO port pull-up/pull-down register (GPIOx_PUPDR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
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
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

GPIO port input data register (GPIOx_IDR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
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

GPIO port output data register (GPIOx_ODR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

GPIO port bit set/reset register (GPIOx_BSRR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Procedure

1. Choose the pin(s) you will be using for GPIO (e.g. PA_5)
2. Enable the GPIO port to which the pin is connected from the AHB1 bus **(in RCC_AHB1ENR, GPIOxEN)**
3. Configure the mode in which the pin will be used, input, output, alternate function or analog **(in GPIOx_MODER, for the particular pin)**
4. [for output mode] Configure the output type, push-pull or open drain **(in GPIOx_OTYPER, for the particular pin)**
5. [for output mode] Configure the output speed **(in GPIOx_OSPEEDR, for the particular pin)**
6. Configure pull-up or pull-down mode, which will enable the internal pull-up or pull-down resistors **(in GPIOx_PUPDR, for the particular pin)**
7. Now use the pin in the selected mode as follows:
    a. Input Mode: Read the input data register to see if the pin is getting input **(GPIOx_IDR, for the particular pin)**
    b. Output Mode: Write 1 to the output data register to create an output in that particular pin **(GPIOx_ODR, for the particular pin)**. Alternatively, write to the binary set reset register to set the pin's output, and reset it respectively **(GPIOx_BSRR, writing to the set bit corresponding to a pin generates output, and writing to the reset bit clears it)**

Timer

Timers in microcontrollers are usually a counter of an oscillator clock, or clock counter. That is, it counts clock pulses. It uses the frequency of the internal clock and generates a delay and operates by incrementing a value in every machine cycle. After a configurable number of counts, it resets and begins counting from zero.
There are a number of different timers, with different capabilities:

- Advanced Control
- General Purpose
- Basic

Timer type	Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/ compare channels	Complementary output	Max interface clock (MHz)	Max timer clock (MHz)(1)
Advanced-control	TIM1, TIM8	16-bit	Up, Down, Up/down	Any integer between 1 and 65536	Yes	4	Yes	90	180
General purpose	TIM2, TIM5	32-bit	Up, Down, Up/down	Any integer between 1 and 65536	Yes	4	No	45	90/180
	TIM3, TIM4	16-bit	Up, Down, Up/down	Any integer between 1 and 65536	Yes	4	No	45	90/180
	TIM9	16-bit	Up	Any integer between 1 and 65536	No	2	No	90	180
	TIM10, TIM11	16-bit	Up	Any integer between 1 and 65536	No	1	No	90	180
	TIM12	16-bit	Up	Any integer between 1 and 65536	No	2	No	45	90/180
	TIM13, TIM14	16-bit	Up	Any integer between 1 and 65536	No	1	No	45	90/180
Basic	TIM6, TIM7	16-bit	Up	Any integer between 1 and 65536	Yes	0	No	45	90/180

We will be focusing on the basic timers, TIM6 and TIM7 who are capable of DMA request generation (or in other words, capable of generating an interrupt). The other timers are capable of more advanced functions like generating PWM, etc. which can be seen above..

Necessary Registers (along with descriptions of used bits)

TIMx Control Register 1 (TIMx_CR1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	ARPE	Res.	Res.	Res.	OPM	URS	UDIS	CEN
								rw				rw	rw	rw	rw

TIMx Prescalar (TIMx_PSC)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The counter clock frequency CK_CNT is equal to f_{CK_PSC} / (PSC[15:0] + 1).

TIMx Auto-reload Register (TIMx_ARR)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

TIMx Status Register (TIMx_SR)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	UIF
															rc_w0

TIMx Counter (TIMx_CNT)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

TIMx DMA/Interrupt Enable Register (TIMx_DIER)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	UDE	Res.	Res.	Res.	Res.	Res.	Res.	Res.	UIE
							rw								rw

Procedure

1. Choose the desired minimum time delay or the unit for the timer.
2. Enable the timer and provide it with clock **(using RCC_APBxENR, TIMxEN)**
3. Set the prescalar in order to get the desired delay (e.g if a delay of 1 µS is desired, and the clock of the APB bus is 90 MHz, the prescalar should be 89 as 90/(89+1) = 1 MHz and delay will be 1/1 MHz = 1 µS) **(using TIMx_PSC)**
4. Set the **auto-reload register** for the maximum value the timer should count to **(using TIMx_ARR)**
5. Enable or turn on the counter **(by using TIMx_CR1, CEN)** and wait for it to be active **(using TIMx_SR, UIF)**
6. Now in order to count to a certain value (i.e. generate a certain delay), set the count to 0 and wait for the count to reach the desired value **(using TIMx_CNT)**
7. [Optional additional steps] Have the AR be preloaded by **using TIMx_CR1, ARPRE**, and enable the interrupt/DMA by **using TIMx_DIER, UIE and UDE**

UART

N.B. We are focusing on UART here but it is simple enough to convert to USART by sending a clock from master to slave

---

Universal Asynchronous Receiver Transmitter. It is a serial, two-wire communication protocol in which data format & transmission speed are configurable. UART can be simplex, half duplex and full duplex, and can provide basic error detection using an optional parity bit.

By serial communication we mean that a single bit is transmitted at a time, and thus sending data through a single channel bit by bit sequentially. It is character oriented, that is data is sent one word at a time.

By asynchronous we mean that no clock signal is sent by the master to the slave for synchronization. Data transmission and reception between devices is done without synchronization, by using the following two mechanisms

- Both the devices must agree on timing parameters, by setting the baud rate
- By using two special bits at the start and end of each byte, namely the start and stop bits

In UART, the TX of the sender is connected to the RX of the receiver and vice versa, and both are connected to common ground.

Disadvantages of UART

- UART doesn't support multiple slave or multiple master systems.
- Communication is limited by baud rate of the device with the lesser baud rate, and this limited speed is a bottleneck for data transmission.
- The maximum size of a data frame is limited.

Baud Rate

Baud rate is the measure of the speed of data transfer, expressed in bits per second (bps). Both devices participating in UART communication need to have exactly the same baud rate.

Configuration of Baud Rate

Equation 1: Baud rate for standard USART (SPI mode included)

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

Equation 2: Baud rate in Smartcard, LIN and IrDA modes

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

Here, USARTDIV is calculated from the DIV_Mantissa and DIV_Fraction in USART_BRR. DIV_Mantissa is simply converted to decimal, while DIV_Fraction is **divided** by the oversampling method (by 8 or by 16) and then converted to decimal.

Example:
If DIV_Mantissa = 0d27 and DIV_Fraction = 0d12 (USART_BRR = 0x1BC), then
Mantissa (USARTDIV) = 0d27
Fraction (USARTDIV) = 12/16 = 0d0.75
Therefore USARTDIV = 0d27.75

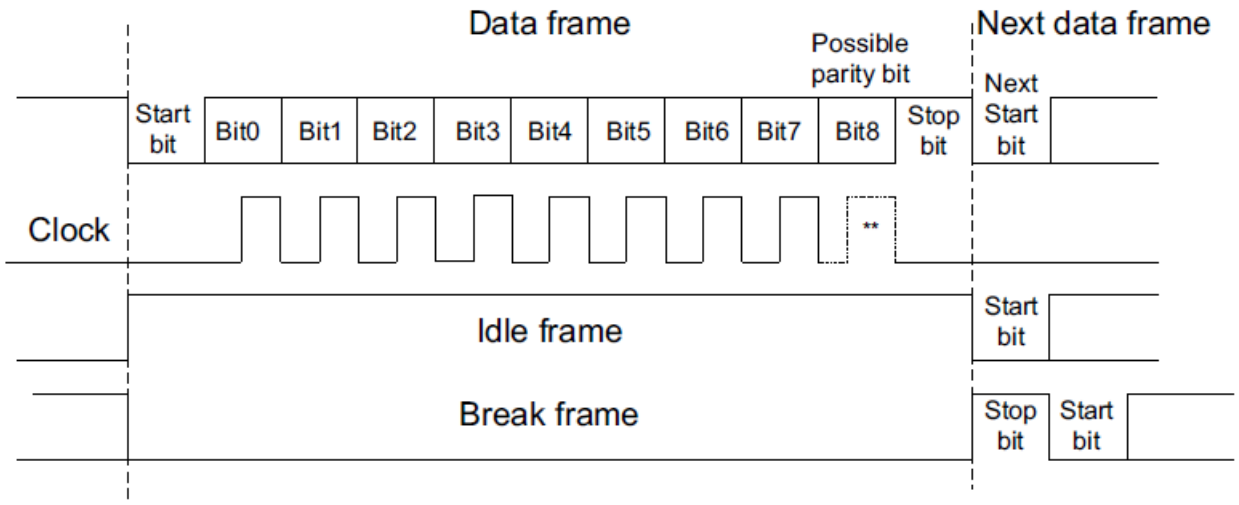
To be noted, when converting a decimal USARTDIV value to DIV_Mantissa and DIV_Fraction, the mantissa is simply converted to binary and stored in DIV_Mantissa, while the fraction is **multiplied** by the oversampling method (by 8 or by 16), rounded to nearest whole number and then converted to binary and stored in DIV_Fraction.

Example:
To program USARTDIV = 0d50.99
This leads to:
DIV_Fraction = 16*0d0.99 = 0d15.84
The nearest real number is 0d16 = 0x10 => overflow of DIV_frac[3:0] => carry must be added up to the mantissa
DIV_Mantissa = mantissa (0d50 + carry) = 0d51 = 0x33
Then, USART_BRR = 0x330 hence USARTDIV = 0d51.000

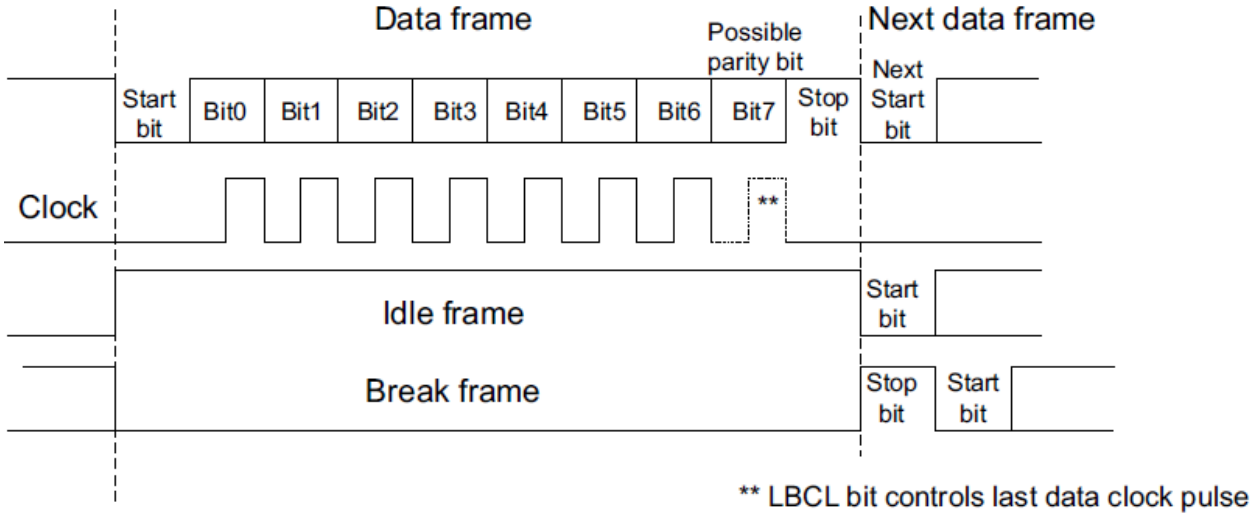
OVER8 is the oversampling mode and is set in USART_CR1. It is 0 when oversampling by 16, and 1 when oversampling by 8. It should be noted however that

- When OVER8 = 0, the fractional part is coded on 4 bits and programmed by the DIV_fraction[3:0] bits in the USART_BRR register
- When OVER8 = 1, the fractional part is coded on 3 bits and programmed by the DIV_fraction[2:0] bits in the USART_BRR register, and bit DIV_fraction[3] must be kept cleared.

UART Framing



9-bit word length (M bit is reset), 1 stop bit



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

An UART transmitter sends out these words of data one at a time, and the receiver realizes that it has received a frame when the channel goes from logical high to logical low (since the start bit is generally a logical low, while the line is at a logical high at idle).
There are two ways for determining that an entire frame has been sent or received

- By waiting for the stop bit to be read and checking each bit if it is the stop bit
- By generating an interrupt when the whole frame has been sent or received and determining completion using it.

Since the stop bit is essential to determining the entire frame, it can be configured and both devices have to agree on the size of the stop bit.

Similarly, the size of the data frame can be configured as well, and both devices need to agree on this configuration.

UART in STM32

STM32 has a total of 6-USART peripherals.

- USART 1,2,3 and 6 are full USART, while 4 and 5 can only be used for UART.
- It uses NRZ standard digital signaling.
- USART 1 and 6 – Maximum Baud Rate: 11.25 Mbits/s
- USART 2, 3, 4, 5 – Maximum Baud Rate: 5.62 Mbits/s

Necessary Registers (along with descriptions of used bits)

Control register 1 (USART_CR1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8	Res.	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Baud rate register (USART_BRR)

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

Status register (USART_SR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

Data register (USART_DR)

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

Procedure

1. Enable the clock for the USART peripheral **(from the RCC_APBxENR, USARTxEN)** and the clock for the GPIO pins for that peripheral (consult reference manual table 10) **(from RCC_AHBxENR, GPIOxEN)**
2. Configure the pins for alternate functions **(in GPIOx_MODER, set the particular pins to 10 for alternate function mode)**, then set them to high speed **(in GPIOx_OSPEEDR)**, and lastly select their alternate function to be that of USART **(in GPIOx_AFR)**
3. Then enable the USART by writing the **UE bit in USARTx_CR1** register to 1
4. Program the **M bit in USARTx_CR1** to define the word length
5. Configure the desired baud rate **using the USARTx_BRR** register (common baud rates include 9600, 115200)
6. Enable the Transmitter/Receiver by Setting the **TE and RE bits in USARTx_CR1** Register
7. Enable interrupt for simpler completion of transmission/receiver **(in USARTx_CR1, RXNEIE)**
8. Transmit data in the following manner
    a. Write the data to send in the **USARTx_DR** register (this clears the **TXE bit in USARTx_SR**).
    b. After writing the last data into the **USARTx_DR** register, wait until **TC=1 (in USARTx_SR)**. This indicates that the transmission of the last frame is complete.
9. Receive data in the following manner

a. Wait for the **RXNE bit (in USARTx_SR)** to set. It indicates that the data has been received and can be read.
b. Read the data from **USARTx_DR** Register. This also clears the **RXNE bit (in USARTx_SR)**

I2C

Inter Integrated Circuit (IIC or I^2C) is a half-duplex, serial, synchronous communication protocol. It is a two wire protocol, with the data line and the clock line to synchronize the clock.

Masters are the devices that generate clock signal and initiates communication, and slave are those devices that receives the clock and responds when addressed by master. It is multi-master and multi-slave bus.

It uses two bi-directional open collector or open drain lines:

- SDA : Serial Data Line
- SCL : Serial Clock Line

Advantages

- Supports multi-master and multi-slave
- Due to device addressing, no chip select pin (SPI) required. Hence, reduced number of pins
- Better error handling is available with ACK bit
- Due to clock stretching, it can work well with both slow and fast ICs

Transmission Mode

- Slave transmitter
- Slave receiver
- Master transmitter
- Master receiver

Slave Mode

By default the I2C interface operates in Slave mode. As soon as a start condition is detected, the address is received from the SDA line and sent to the shift register. Then it is compared with the address of the interface.
Header or address not matched: the interface ignores it and waits for another Start condition.
Header matched (10-bit mode only): the interface generates an acknowledge pulse if the ACK bit is set and waits for the 8-bit slave address.
Address matched: the interface generates in sequence:

- An acknowledge pulse if the ACK bit is set

- The ADDR bit is set by hardware and an interrupt is generated if the ITEVFEN bit is set.
- If ENDUAL=1, the software has to read the DUALF bit to check which slave address has been acknowledged.

In 10-bit mode, after receiving the address sequence the slave is always in Receiver mode.

It will enter Transmitter mode on receiving a repeated Start condition followed by the header sequence with matching address bits and the least significant bit set (11110xx1).

The TRA bit indicates whether the slave is in Receiver or Transmitter mode.

To switch from default Slave mode to Master mode a Start condition generation is needed.

Master Mode

In Master mode, the I2C interface initiates a data transfer and generates the clock signal. A serial data transfer always begins with a Start condition and ends with a Stop condition. Master mode is selected as soon as the Start condition is generated on the bus with a START bit.
The following is the required sequence in master mode.

- Program the peripheral input clock in I2C_CR2 Register in order to generate correct timings
- Configure the clock control registers
- Configure the rise time register
- Program the I2C_CR1 register to enable the peripheral
- Set the START bit in the I2C_CR1 register to generate a Start condition

Slave address transmission

Then the slave address is sent to the SDA line via the internal shift register.

- In 10-bit addressing mode, sending the header sequence causes the following event:
    - The ADD10 bit is set by hardware and an interrupt is generated if the ITEVFEN bit is set.

  Then the master waits for a read of the SR1 register followed by a write in the DR register with the second address byte
    - The ADDR bit is set by hardware and an interrupt is generated if the ITEVFEN bit is set.

  Then the master waits for a read of the SR1 register followed by a read of the SR2 register
- In 7-bit addressing mode, one address byte is sent. As soon as the address byte is sent,

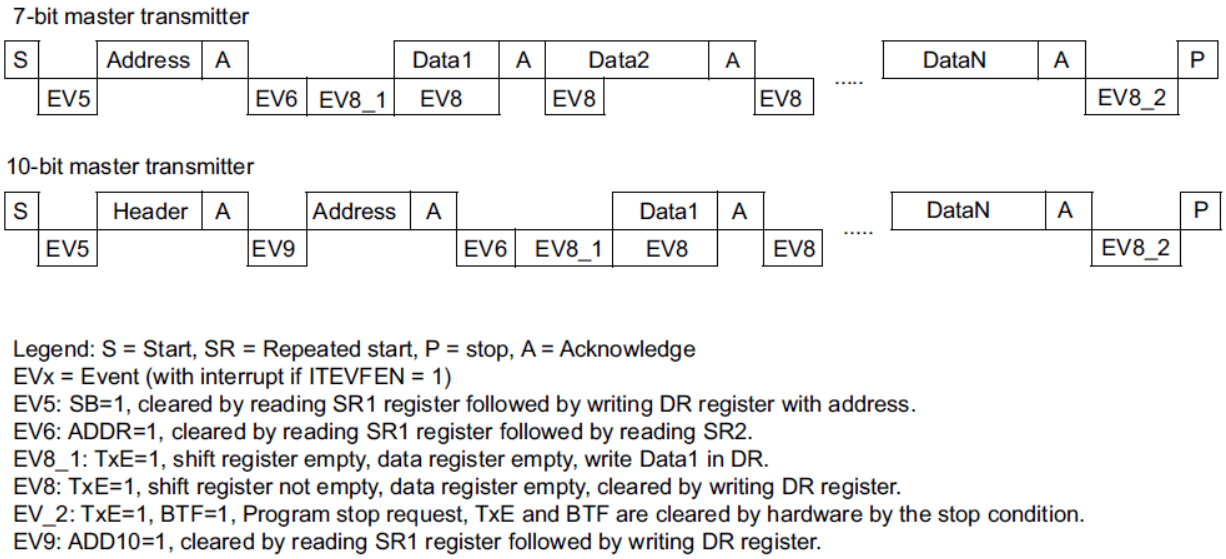
- The ADDR bit is set by hardware and an interrupt is generated if the ITEVFEN bit is set.

Then the master waits for a read of the SR1 register followed by a read of the SR2 register

I2C Transfer Sequence

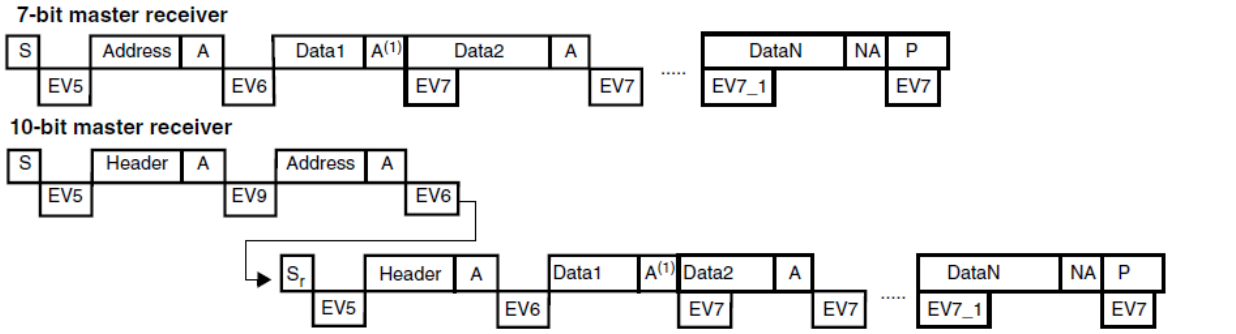
First, a start bit or start condition is generated by a master, and sends its clock. There is bus arbitration on this line, and only one master can use the bus at a time.
Along the I2C bus, each node (master and slave) has a address (7 or 10 bits), and the transmitter sends the address on the bus along with a bit for read or write and waits for an acknowledgement. Once an acknowledgement is received, i.e. the addressed device exists on the bus, data is sent and acknowledgement is waited for.

When the master transmits, the following transfer sequence is sent (is the slave received frame)



Legend: S = Start, SR = Repeated start, P = stop, A = Acknowledge
EVx = Event (with interrupt if ITEVFEN = 1)
EV5: SB=1, cleared by reading SR1 register followed by writing DR register with address.
EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2.
EV8_1: TxE=1, shift register empty, data register empty, write Data1 in DR.
EV8: TxE=1, shift register not empty, data register empty, cleared by writing DR register.
EV_2: TxE=1, BTF=1, Program stop request, TxE and BTF are cleared by hardware by the stop condition.
EV9: ADD10=1, cleared by reading SR1 register followed by writing DR register.

When the master receives, the following transfer sequence is received (is the slave transmitted frame)



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

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

EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2. In 10-bit master receiver mode, this sequence should be followed by writing CR2 with START = 1.
In case of the reception of 1 byte, the Acknowledge disable must be performed during EV6 event, i.e. before clearing ADDR flag.

EV7: RxNE=1 cleared by reading DR register.

EV7_1: RxNE=1 cleared by reading DR register, program ACK=0 and STOP request

EV9: ADD10=1, cleared by reading SR1 register followed by writing DR register.

Necessary Registers (along with descriptions of used bits)

I2C Control Register 1 (I2C_CR1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SW RST	Res.	ALERT	PEC	POS	ACK	STOP	START	NO STRETCH	ENGC	ENPEC	ENARP	SMB TYPE	Res.	SM BUS	PE
rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw	rw

I2C Control Register 2 (I2C_CR2)

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

I2C Clock control register (I2C_CCR)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F/S	DUTY	Res.	Res.	CCR[11:0]											
rw	rw			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

I2C TRISE register (I2C_TRISE)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	TRISE[5:0]					
										rw	rw	rw	rw	rw	rw

I2C Status register 1 (I2C_SR1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMB ALERT	TIMEOUT	Res.	PEC ERR	OVR	AF	ARLO	BERR	TxE	RxNE	Res.	STOPF	ADD10	BTF	ADDR	SB
rc_w0	rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	r	r		r	r	r	r	r

I2C Status register 2 (I2C_SR2)

Note: Reading I2C_SR2 after reading I2C_SR1 clears the ADDR flag, even if the ADDR flag was set after reading I2C_SR1. Consequently, I2C_SR2 must be read only when ADDR is found set in I2C_SR1 or when the STOPF bit is cleared.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PEC[7:0]								DUALF	SMB HOST	SMB DEFAULT	GEN CALL	Res.	TRA	BUSY	MSL
r	r	r	r	r	r	r	r	r	r	r	r		r	r	r

Procedure

Configuration

1. Enable the I2C clock and GPIO clock (for the I2C pins) (using RCC_APB1ENR and RCC_AHB1ENR)
2. Configure the I2C GPIO pins for alternate functions (using GPIOx_MODER), setting the output type to open drain (using GPIOx_OTYPER), and speed to high (using GPIOx_OSPEEDR), and to pull-up (using GPIOx_PUPDR) and picking I2C as the alternate function (using GPIOx_AFR)
3. Reset the I2C (using I2C_CR1, the SW RST)
4. Program the peripheral input clock in order to generate correct timings (using I2C_CR2, FREQ) [should be the same as PCLK1]
5. Configure the clock control registers (using I2C_CCR) [example value is 225]
6. Configure the rise time register (using I2C_TRISE) [example value is 46]
7. Program the I2C_CR1 register (the PE bit) to enable the peripheral

Start/Initiating Communication by the Master

1. Enable the ACK bit so acknowledgement is sent (using I2C_CR1, ACK)
2. Send the START condition by generating START (using I2C_CR1, START)
3. Wait for the SB (in I2C_SR1) to set. This indicates that the start condition is generated

Sending Slave Address

1. Send the Slave Address to the I2C_DR Register
2. Wait for the ADDR (in I2C_SR1) to set. This indicates the end of address transmission
3. Clear the ADDR by reading the SR1 and SR2

Write

1. Wait for the TXE (in I2C_SR1) to set. This indicates that the DR is empty
2. Send the DATA to the I2C_DR Register
3. Wait for the BTF (in I2C_SR1) to set. This indicates the end of LAST DATA transmission

Read

1. If only 1 BYTE needs to be Read
    a. Write the slave Address, and wait for the ADDR bit (bit 1 in SR1) to be set
    b. the Acknowledge disable is made during EV6 (before ADDR flag is cleared) and the STOP condition generation is made after EV6
    c. Wait for the RXNE (Receive Buffer not Empty) bit to set
    d. Read the data from the DR
2. If Multiple BYTES needs to be read
    a. Write the slave Address, and wait for the ADDR bit (bit 1 in SR1) to be set
    b. Clear the ADDR bit by reading the SR1 and SR2 Registers
    c. Wait for the RXNE (Receive buffer not empty) bit to set
    d. Read the data from the DR
    e. Generate the acknowledgment by setting the ACK (bit 10 in SR1)
    f. To generate the non acknowledge pulse after the last received data byte, the ACK bit must be cleared just after reading the second last data byte (after second last RxNE event)
    g. In order to generate the Stop/Restart condition, software must set the STOP/START bit after reading the second last data byte (after the second last RxNE event)

Stop

1. Stop I2C (using I2C_CR1, STOP)

Viva Questions

1. Pin koita kon port e
2. Interrupt ki, polling method ki, difference ki
3. Synchronization (kibhabe, kothae ase kothae nai, keno lage)
4. Show signal diagrams (clock counter trail)