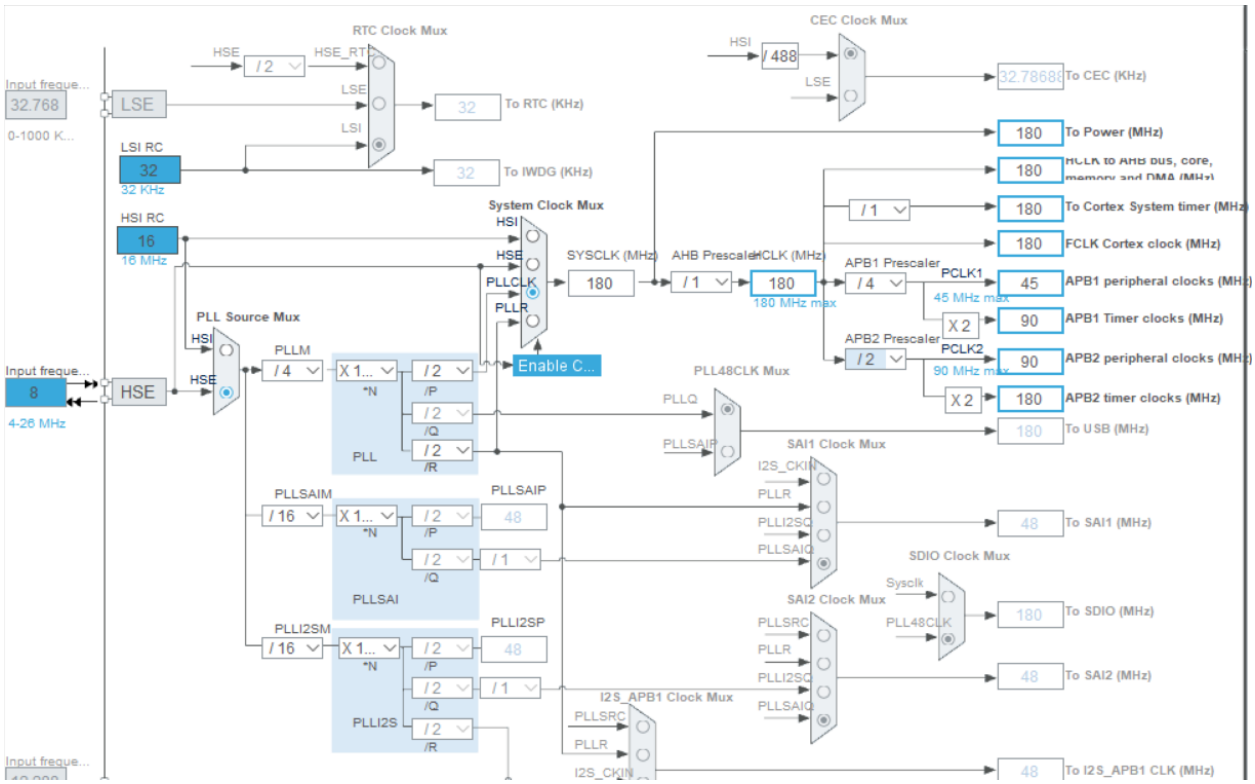
$$H_{Electrolyzer}(t) = 1.43 \times 10^{-3} + 2.39 \times 10^{-2} E_{Electrolyzer}(t) - 4.32 \times 10^{-5} E^{2}_{Electr}$$

RCC Configuration:

This is done to set the peripherals to work under maximum frequency.
The HSE (High Speed External) is used to create accurate clock pulses.



Input is 8Mhz
Input goes to HSE
HSE goes to be selected by multiplexer.
PLLM = ¼
PLLN = 180
PLLP = 1/2

So input x ¼ x 180 x 1/2 = input x 22.5= 180 again

So the system clock is running at 180 Mhz
AHB prescalar = 1

APB1 prescalar = ¼
APB2 prescalar = ½

This is because APB1 has max frequency of 45
APB2 has max frequency of 90

Steps to do this

1. ENABLE HSE and wait for the HSE to become Ready

This needs RCC_CR register

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→CR |= (1<<16) //turn it on
while (!(RCC→CR & (1<<17)); wait until HSERDY is set

2. Set the POWER ENABLE CLOCK and VOLTAGE REGULATOR
Power enabling clock resides in 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

RCC→APB1ENR |= (1<<28)

Voltage regulation needs the PWR→CR register

Table 20. PWR - register map and reset values

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x000	PWR_CR	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	FISSR	FMSSR	UDEN[1:0]		ODSWEN	ODEN	VOS[1:0]		ADCDC1	Res.	MRUDS	LPUDS	FPDS	DBP	PLS[2:0]			PVDE	CSBF	CWUF	PDDS	LPDS

PWR→CR |= (3<<14)

11 means

[Mishtek, scale mode should be 1, so value written must be 11]

1: Over-drive enabled

Bits 15:14 **VOS[1:0]**: Regulator voltage scaling output selection

These bits control the main internal voltage regulator output voltage to achieve a trade-off between performance and power consumption when the device does not operate at the maximum frequency (refer to the STM32F446xx datasheet for more details).

These bits can be modified only when the PLL is OFF. The new value programmed is active only when the PLL is ON. When the PLL is OFF, the voltage scale 3 is automatically selected.

00: Reserved (Scale 3 mode selected)

01: Scale 3 mode

10: Scale 2 mode

11: Scale 1 mode (reset value)

3. Configure the FLASH PREFETCH and the LATENCY Related Settings

First we have to enable Data cache, Instruction cache, and prefetch. It's in the FLASH→ACR (Access control register)

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.	DCRST	ICRST	DCEN	ICEN	PRFTEN	Res.	Res.	Res.	Res.	LATENCY			
			rw	w	rw	rw	rw					rw	rw	rw	rw

FLASH→ACR |= (1<<8)

FLASH→ACR |= (1<<9)

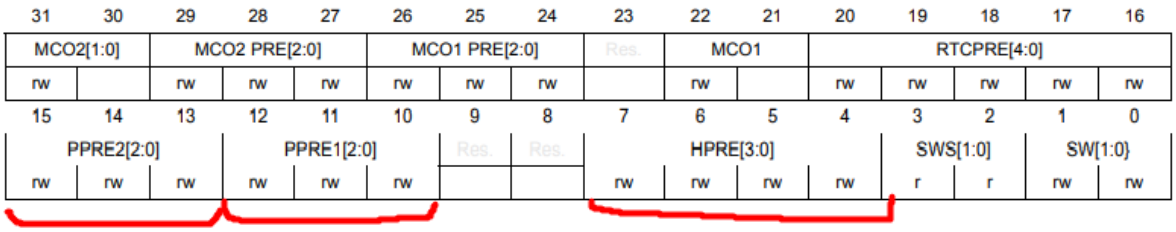
FLASH→ACR |= (1<<10)

FLASH→ACR |= (5<<0)//for latency with 5 wait states

4. Configure the PRESCALARS HCLK, PCLK1, PCLK2

To set the prescalar values, we have to look at the first picture.

For AHB, APB1, APB2
For this we need 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	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

HPRE is AHB
We divide by 1. So

Bits 7:4 **HPRE:** AHB prescaler

Set and cleared by software to control AHB clock division factor.

Caution: The clocks are divided with the new prescaler factor from 1 to 16 AHB cycles after HPRE write.

Caution: The AHB clock frequency must be at least 25 MHz when the Ethernet is used.

0xxx: system clock not divided
1000: system clock divided by 2
1001: system clock divided by 4
1010: system clock divided by 8
1011: system clock divided by 16
1100: system clock divided by 64
1101: system clock divided by 128
1110: system clock divided by 256
1111: system clock divided by 512

RCC→CFGR &= ~(15<<4)
1 works too, 15 should work too.

Bits 15:13 **PPRE2:** APB high-speed prescaler (APB2)

Set and cleared by software to control APB high-speed clock division factor.

Caution: The software has to set these bits correctly not to exceed 90 MHz on this domain. The clocks are divided with the new prescaler factor from 1 to 16 AHB cycles after PPRE2 write.

0xx: AHB clock not divided
100: AHB clock divided by 2
101: AHB clock divided by 4
110: AHB clock divided by 8
111: AHB clock divided by 16

Bits 12:10 **PPRE1:** APB Low speed prescaler (APB1)

Set and cleared by software to control APB low-speed clock division factor.

Caution: The software has to set these bits correctly not to exceed 45 MHz on this domain. The clocks are divided with the new prescaler factor from 1 to 16 AHB cycles after PPRE1 write.

0xx: AHB clock not divided
100: AHB clock divided by 2
101: AHB clock divided by 4
110: AHB clock divided by 8
111: AHB clock divided by 16

Figure the two others on your own

5. Configure the MAIN PLL

For the PLL's we need 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

PLL P has set prescalars with 00 meaning divide by 2.
Rest are direct bit map of the value we want

PLLP = RCC→PLLCFGR &= ~(3<<16)
PLLN = RCC→PLLCFGR |= (180<<6)
PLLM = RCC→PLLCFGR |= (4<<0)
Choose external crystal
RCC→PLLCFGR |= (1<<22)

6. Enable the PLL and wait for it to become ready

register descriptions.

6.3.1 RCC clock control register (RCC_CR)

Address offset: 0x00

Reset value: 0x0000 XX83 where X is undefined.

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
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→CR |= (1<<24)
While RCC→CR & (1<<25) not 1

7. Select the Clock Source and wait for it to be set
SW for clock select
SWS for clock select status

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	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

Bits 31:30 **MCO2[1:0]:** Microcontroller clock output 2

Bits 1:0 **SW[1:0]:** System clock switch
Set and cleared by software to select the system clock source.
Set by hardware to force the HSI selection when leaving the Stop or Standby mode or in case of failure of the HSE oscillator used directly or indirectly as the system clock.
00: HSI oscillator selected as system clock
01: HSE oscillator selected as system clock
10: PLL_P selected as system clock
11: PLL_R selected as system clock

We need PLL_P as the sys clock
RCC→CFGR |= (2<<0)

We need to check if it's up and running

1111: system clock divided by 512

Bits 3:2 **SWS[1:0]:** System clock switch status

Set and cleared by hardware to indicate which clock source is used as the system clock.

00: HSI oscillator used as the system clock

01: HSE oscillator used as the system clock

10: PLL used as the system clock

11: PLL_R used as the system clock

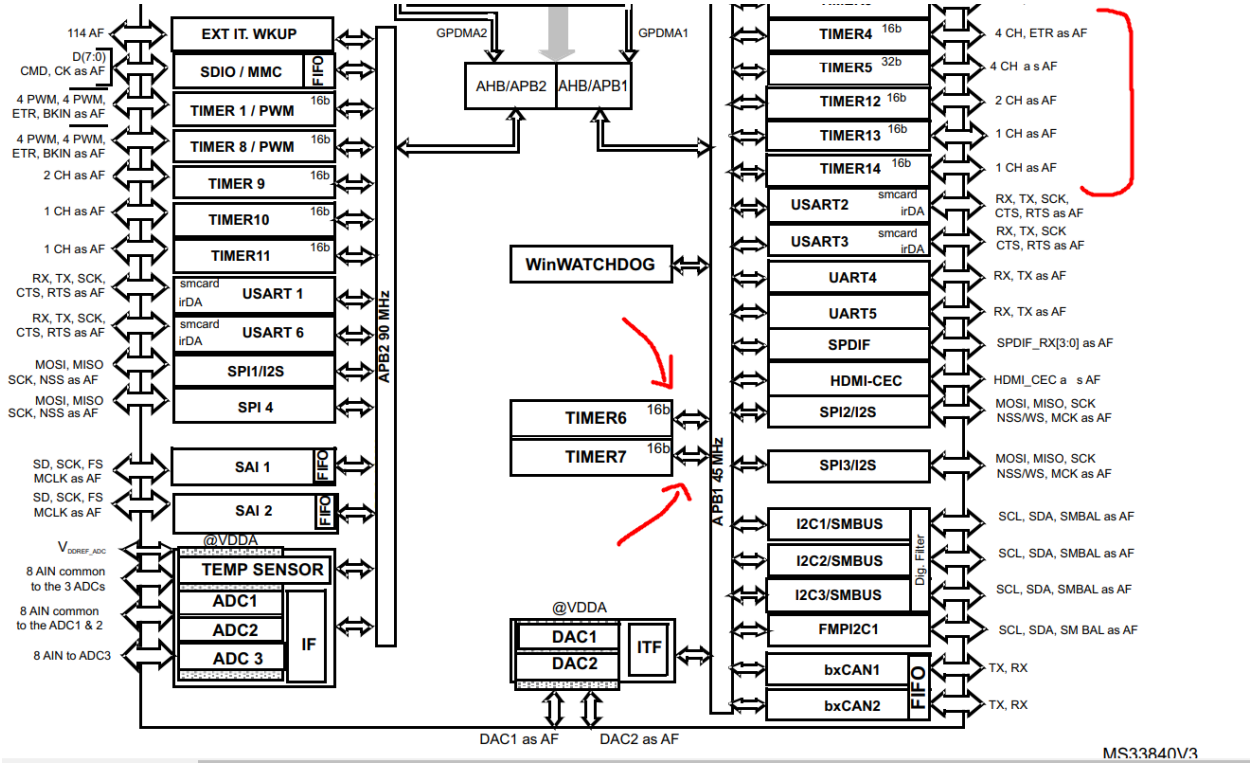
while (!(RCC→CFGR & (2<<2));

AWD.

—----------###----------

Timer configuration:

First step is to choose which Timer to choose, and find out which peripheral it is connected to.



This was taken from the Datasheet.

Timer6 and 7 are general purpose timers.

So timer6 is in APB1 peripheral and its max frequency is 45mhz

First enable the timer clock!

This is in RCC→APB1ENR register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	DAC RST	PWR RST	CECRST	CAN2 RST	CAN1 RST	FMPI2C1 RST	I2C3 RST	I2C2 RST	I2C1 RST	UART5 RST	UART4 RST	UART3 RST	UART2 RST	SPDIFRX RST
		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 RST	SPI2 RST	Res.	Res.	WWDG RST	Res.	Res.	TIM14 RST	TIM13 RST	TIM12 RST	TIM7 RST	TIM6 RST	TIM5 RST	TIM4 RST	TIM3 RST	TIM2 RST
rw	rw			rw			rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:30 Reserved, must be kept at reset value.

RCC→APB1ENR |= (1<<4)

Set the prescalar and the ARR

Go to the timer6's PSC register

19.4.7 TIM6&TIM7 prescaler (TIMx_PSC)

Address offset: 0x28

Reset value: 0x0000

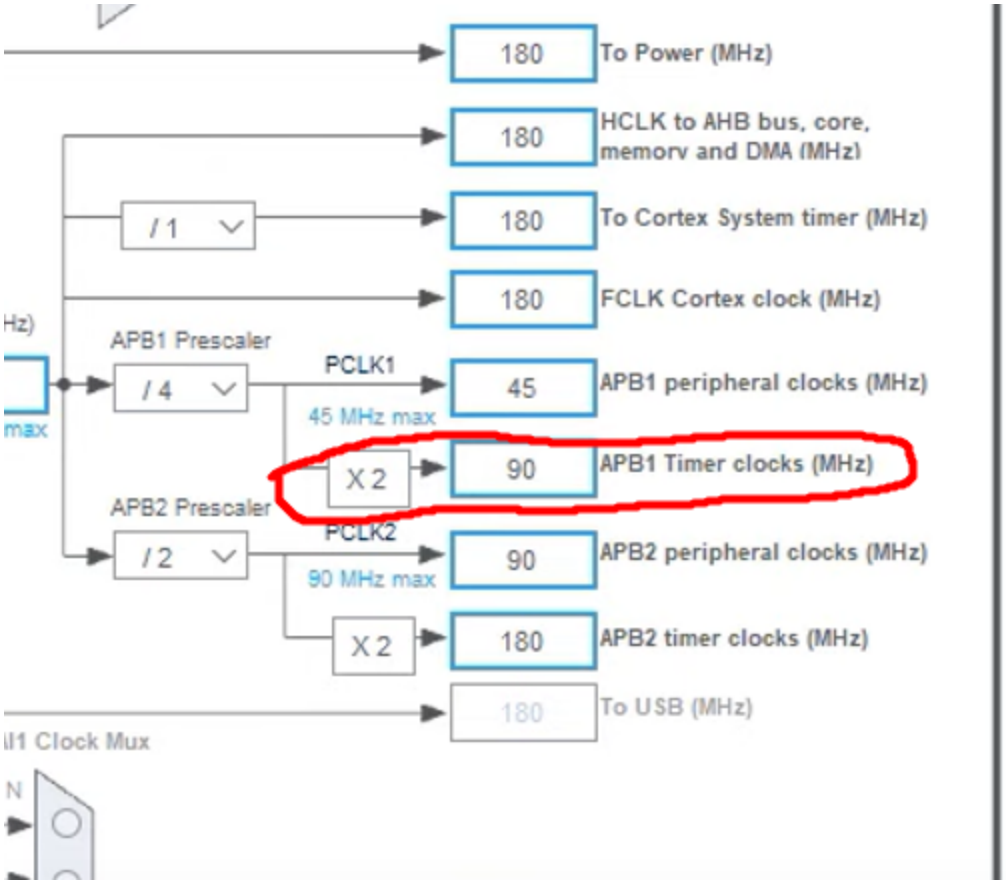
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15.0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **PSC[15:0]**: Prescaler value

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

PSC contains the value to be loaded into the active prescaler register at each update event.

Since 1 is added, whatever the frequency, we add 1 to it.
Fck_PSC == 90 mhz
Dividing by 90 gives
Keep in mind, it's because APB1 clock and APB1 timer has different frequencies



90 x 10^6 / 90 = 10^6 = 1 Micro second.

Now set maximum value here

19.4.8 TIM6&TIM7 auto-reload register (TIMx_ARR)

Address offset: 0x2C

Reset value: 0x0000

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

Bits 15:0 **ARR[15:0]**: Auto-reload value

ARR is the value to be loaded into the actual auto-reload register.

Refer to *Section 19.3.1: Time-base unit on page 629* for more details about ARR update and behavior.

The counter is blocked while the auto-reload value is null.

So,

TIM6→PSC = 90-1

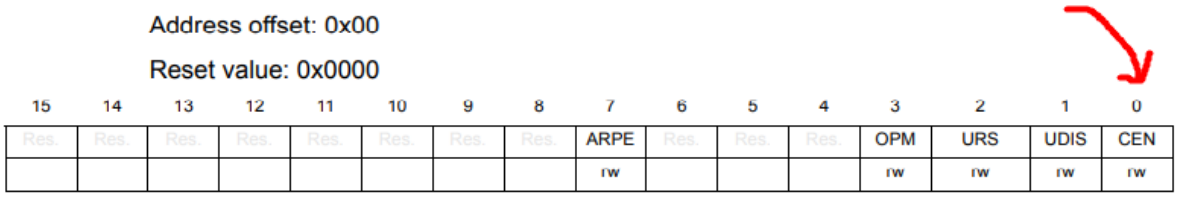
TIM6→ARR = 0xffff

Enable the Timer, and wait for the update Flag to set
First enable this TIM6 timer
It needs TIM6_CR1
So ,
TIM6→CR1 |= (1<<0);

19.4.1 TIM6&TIM7 control register 1 (TIMx_CR1)

Address offset: 0x00

Reset value: 0x0000



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

Bits 15:8 Reserved, must be kept at reset value.

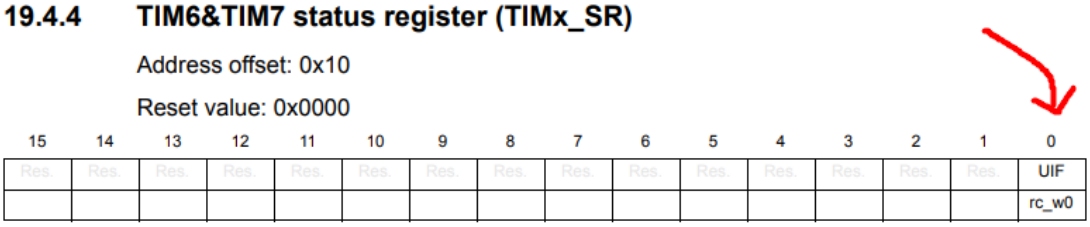
Now we wait for the bits to update

while (!(TIM6→SR & (1<<0)));
Update interrupt flag

19.4.4 TIM6&TIM7 status register (TIMx_SR)

Address offset: 0x10

Reset value: 0x0000



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

Bits 15:1 Reserved, must be kept at reset value.

Now with this, we can use for loops to set

AWD.

—----------#----------

GPIO

GPIO's all registers are important to understand.

Firstly, there is

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

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

To set, exactly how one wants to use the GPIO pin

GPIOx→OTYPER Output Type Register

7.4.2 GPIO port output type register (GPIOx_OTYPER) (x = A..H)

Address offset: 0x04

Reset value: 0x0000 0000

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

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

Bits 15:0 **OTy**: Port x configuration bits (y = 0..15)
These bits are written by software to configure the output type of the I/O port.
0: Output push-pull (reset state)
1: Output open-drain

Sets either to push-pull or open drain mode.

GPIOx→OSPEEDR

7.4.3 GPIO port output speed register (GPIOx_OSPEEDR) (x = A..H)

Address offset: 0x08

Reset values:

- 0x0000 00C0 for port B
- 0x0000 0000 for other ports

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

Bits 2y:2y+1 **OSPEEDRy[1:0]:** Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O output speed.

00: Low speed
01: Medium speed
10: Fast speed
11: High speed

Note: Refer to the product datasheets for the values of OSPEEDRy bits versus V_{DD} range and external load.

So,
Saying
GPIOB→OSPEEDR |= (3<<10)
Then, GPIOB5 will be set to high speed.

GPIOx→PUPDR (pull up pull down register)

7.4.4 GPIO port pull-up/pull-down register (GPIOx_PUPDR) (x = A..H)

Address offset: 0x0C

Reset values:

- 0x6400 0000 for port A
- 0x0000 0100 for port B
- 0x0000 0000 for other ports

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

Bits 2y:2y+1 **PUPDRy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O pull-up or pull-down
00: No pull-up, pull-down
01: Pull-up
10: Pull-down
11: Reserved

GPIOA→PUPDR |= (2<<4) will set GPIOA2 to pull-down mode

GPIOx→ IDR is input data register
GPIOx→ODR is output data register

GPIO→BSRR (bit set/reset register)

7.4.7 GPIO port bit set/reset register (GPIOx_BSRR) (x = A..H)

Address offset: 0x18

Reset value: 0x0000 0000

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

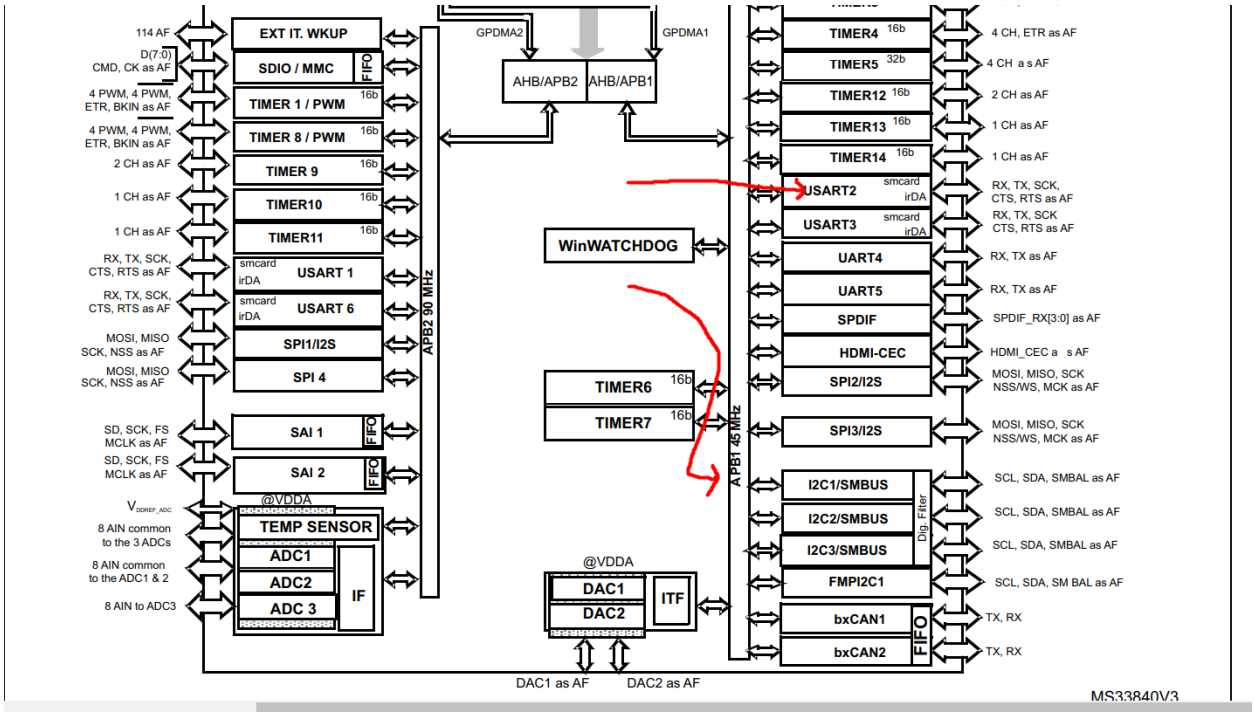
GPIOC→BSRR |= (1<<13) will set GPIOC13 and GPIOC→BSRR |= (1<<13+16) will reset it.

USARTx

Configuration

Lets assume first we will use two pins from the MCU
PA2 and PA3. One works like Rx (receive), the other works like Tx(Transmit)

First, enable the UART clock.
If we do UART2, then UART2 can be found in APB1 peripheral



From datasheet

So, to enable, we consult APB1ENR register in RCC

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

RCC→APB1ENR |= (1<<17); //enable uart2 clock

Clear the CR1 register, and then enable the UART itself. This is ofcourse, the internals of the UART2 register in this context.

25.6.4 Control register 1 (USART_CR1)

Address offset: 0x0C

Reset value: 0x0000 0000

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

So,
USART2→CR1 = 0×00; // clear all
USART2→CR1 |=(1<<13); //Enable USART2

Set the word length

M bit set to 0, means word length = 8
M bit set to 1, means word length = 9
Cannot modify this while uart is transmitting or receiving

25.6.4 Control register 1 (USART_CR1)

Address offset: 0x0C

Reset value: 0x0000 0000

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

So,
USART2→CR1 &= ~(unsigned int)(1<<12);
Setting 0, means word length is now 8

Setting the baud rate

For this, we consult UARTx→BRR (Baud rate register)

But first, take a look at how the baud rate is actually calculated

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.

In the BRR we see, we have control over div_mantissa and div_fraction

Reset value: 0x0000 0000

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

This over8 = 16 if we ignored it in the CR1.
It's called the oversampling rate
Lets say we want 115200 bps

115200 = fck/(8 x (2-OVER8) x USARTDIV)
Fck is 45Mhz = 45 x 10^6

115200 = (45×10^6)/(8 x (2-0) x USARTDIV)
USARTDIV = 24.4140625

Mantissa = 24
Fraction = 0.4140625 * 16 = 6.625 ~ 7
So,
USART2→BRR |= (24<<4) | (7<<0);

Now, enable the transmitter and receiver
We consult CR1 register

25.6.4 Control register 1 (USART_CR1)

Address offset: 0x0C

Reset value: 0x0000 0000

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

So, USART2→CR1 |= (1<<2) | (1<<3);

Let's see how to code and enable interrupts.

25.6.4 Control register 1 (USART_CR1)

Address offset: 0x0C

Reset value: 0x0000 0000

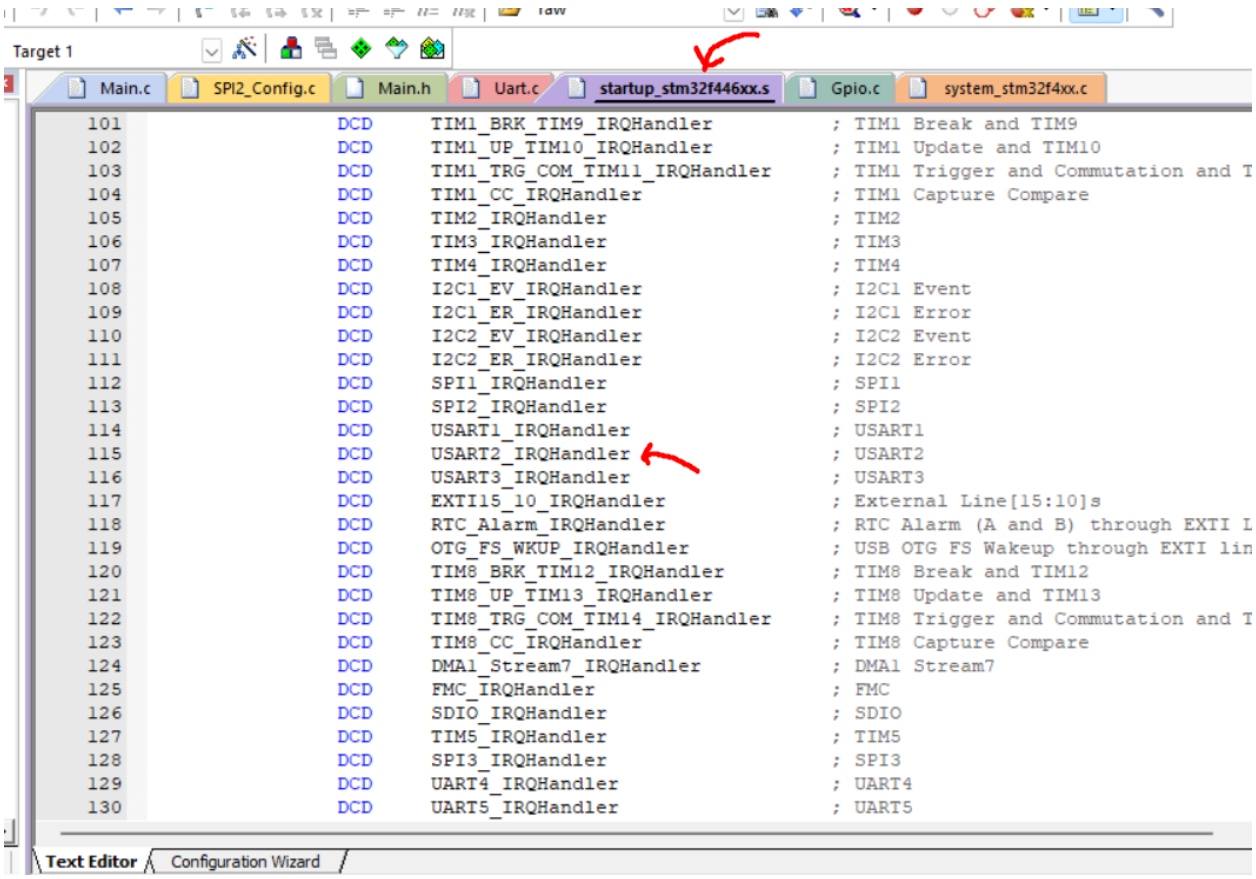
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8	Res.	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Same register, but RXNEIE means Read data register Not Empty Interrupt Enable.

So,
USART2→CR1 |= (1<<5);

How to overwrite interrupt handlers.

Go to the startup code in Keil IDE



To set priority to this interrupt we write
NVIC_SetPriority(USART2_IRQn, 1); // replace IRQHandler with IRQn
//lower the number, higher the priority
NVIC_EnableIRQ(USART2_IRQn);
// this enables the handler

Now make a function that overrides the handler.

```
void USART2_IRQHandler(void) {
  USART2->CR1 &= ~USART_CR1_RXNEIE;
  getInputString(currentMessage);
  USART2->CR1 |= USART_CR1_RXNEIE;
}
```

First, we clear the RXNEIE bit to disable interrupt termporarily. Handle the message. And then reset the RXNEIE bit so interrupt works again.

AWD.

Some interrupts need more configurations though.

Any form of data transmission and reception has to do with
Status register, SR and data register DR

Sending data

25.6.2 Data register (USART_DR)

Address offset: 0x04

Reset value: 0x0000 0000

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

This is where we put the data to send

25.6.1 Status register (USART_SR)

Address offset: 0x00

Reset value: 0x00C0 0000

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

This transmission complete bit is set when the transmission is over.

So, to send a data
Let a character be X = 47; // 8 bit data
USART2→DR = X; // load the data into DR register

while (!(USART2→SR & (1<<6))); //wait for TC to set

Receiving data
We can actually do this two ways. One could be with interrupts (non blocking) , the other is blocking. When we sit on our ass and wait.
First lets see the **blocking one**

25.6.1 Status register (USART_SR)

Address offset: 0x00

Reset value: 0x00C0 0000

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

RXNE set means “you got mail”

while(!(USART2→SR & (1<<5))); // wait till i got data
uint8_t character = USART2→DR; // read data from DR

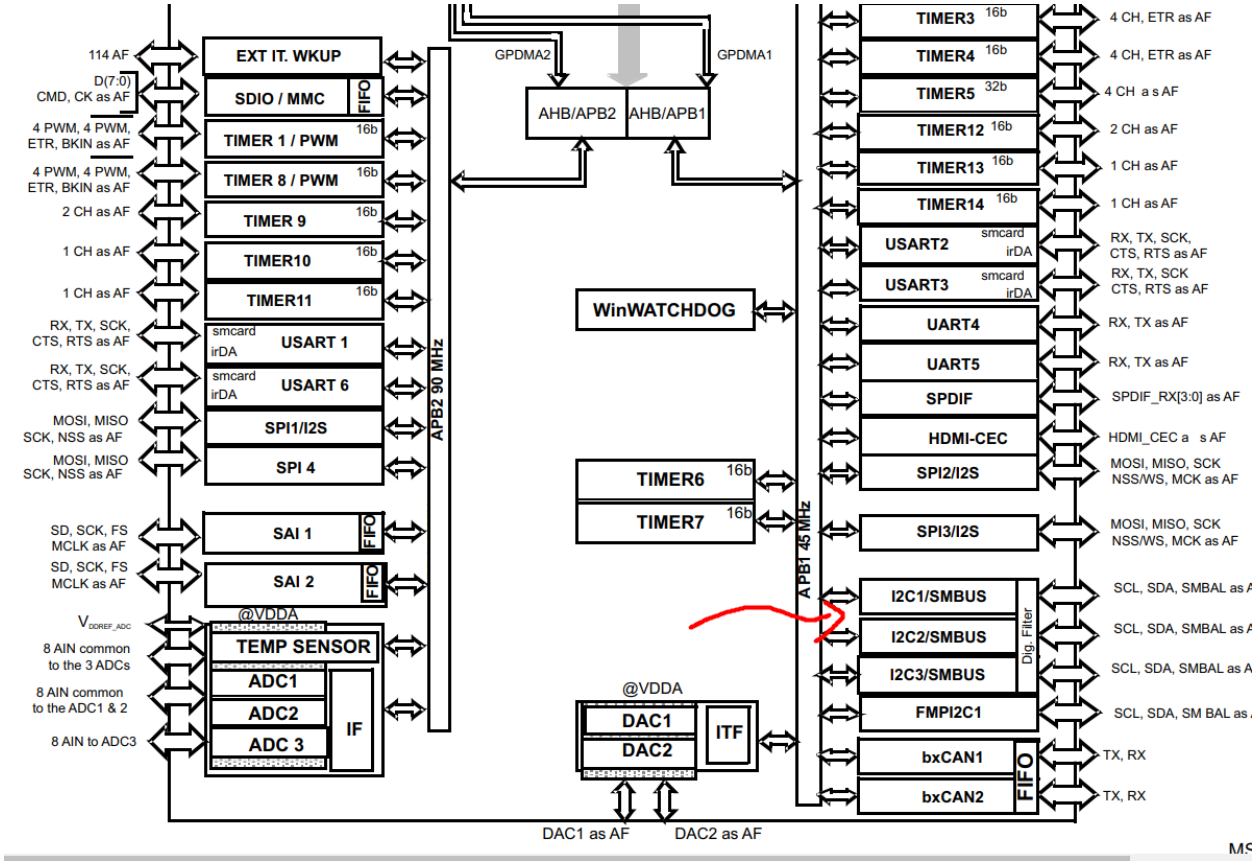
Non blocking one needs interrupts.
So,
Interrupt handler will call the function that has the blocking mechanisms.
Hence allowing parallel works to be done.

—----------#----------

I2C

I2C configuration

1. **Enable I2C’s clock peripheral**



Using datasheet to figure out where I2C is at
We gotta go to 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

RCC→APB1ENR |= (1<<22) //whichever I2c you want;

GPIOx configuration that goes along with it

We need two wires of I2C,
One is SDA (Data line)
The other is SCL (Clock line)

The assignment wanted
PB7 be SDA and PB8 be SCL

For them to work like this, we need them in

1. Alternative function mode
2. Output type = open drain (refere to Miko's sheet to know what open drain is Microcontroller)
3. Highspeed mode
4. Set the proper alternate function on those pins

So, firstly, alternative function set with moder (GpioB peripheral clock too)

```
RCC→AHB1ENR |= (1<<1);  // Enable GPIOB CLOCK
```

Configure the I2C PINs for Alternate Functions

```
GPIOB→MODER |= (2<<16) | (2<<14); // No need to explain anymore hopefully
```

Output type drain. Look at the register map to confirm

```
GPIOB→OTYPER |= (1<<8) | (1<<7);
```

Set speed to high.

```
GPIOB→OSPEEDR |= (3<<16) | (3<<14);
```

Pull up mode. Refer to miko's sheet to know why so.

```
GPIOB→PUPDR |= (1<<16) | (1<<14);
```

Looking at alternative functions of PB7 and PB8, we see
AF4 works for both of these pins. How to write this?

Table 11. Alternate function (continued)

Port		AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15
		SYS	TIM1/2	TIM3/4/5	TIM8/9/10/11 CEC	I2C1/2/3 /4/CEC	SPI1/2/3/4	SPI2/3/4/ SAI1	SPI2/3/ USART1/2/3 /UART5/ SPDIFRX	SAI/ USART6/ UART4/5/ SPDIFRX	CAN1/2 TIM12/13/ 14/ QUADSPI	SAI2/ QUADSPI/ OTG2_HS/ OTG1_FS	OTG1_FS	FMC/ SDIO/ OTG2_FS	DCMI	-	SYS
	PB0	-	TIM1_CH2N	TIM3_CH3	TIM8_ CH2N	-	-	-	SPI3_MOSI/ I2S3_SD	UART4_ CTS	-	OTG_HS_ ULPI_D1	-	SDIO_D1	-	-	EVENT OUT
	PB1	-	TIM1_CH3N	TIM3_CH4	TIM8_ CH3N	-	-	-	-	-	-	OTG_HS_ ULPI_D2	-	SDIO_D2	-	-	EVENT OUT
	PB2	-	TIM2_CH4	-	-	-	-	SAI1_ SD_A	SPI3_MOSI/ I2S3_SD	-	QUADSPI_ CLK	OTG_HS_ ULPI_D4	-	SDIO_CK	-	-	EVENT OUT
	PB3	JTDO/ TRACE SWO	TIM2_CH2	-	-	I2C2_ SDA	SPI1_SCK /I2S1_CK	SPI3_SCK / I2S3_CK	-	-	-	-	-	-	-	-	EVENT OUT
	PB4	NJTRS T	-	TIM3_CH1	-	I2C3_ SDA	SPI1_MISO	SPI3_ MISO	SPI2_NSS/ I2S2_WS	-	-	-	-	-	-	-	EVENT OUT
	PB5	-	-	TIM3_CH2	-	I2C1_ SMBA	SPI1_MOSI /I2S1_SD	SPI3_ MOSI/ I2S3_SD	-	-	CAN2_RX	OTG_HS_ ULPI_D7	-	FMC_ SDCKE1	DCMI_ D10	-	EVENT OUT
	PB6	-	-	TIM4_CH1	HDMI_ CEC	I2C1_ SCL	-	-	USART1_ TX	-	CAN2_TX	QUADSPI_ BK1_NCS	-	FMC_ SDNE1	DCMI_D5	-	EVENT OUT
B	PB7	-	-	TIM4_CH2	-	I2C1_ SDA	-	-	USART1_ RX	SPDIF_ RX0	-	-	-	FMC_NL	DCMI_ VSYNC	-	EVENT OUT
B	PB8	-	TIM2_CH1/ TIM2_ETR	TIM4_CH3	TIM10_ CH1	I2C1_ SCL	-	-	-	-	CAN1_RX	-	-	SDIO_D4	DCMI_D6	-	EVENT OUT
	PB9	-	TIM2_CH2	TIM4_CH4	TIM11_ CH1	I2C1_ SDA	SPI2_NSS/ I2S2_WS	SAI1_ FS_B	-	-	CAN1_TX	-	-	SDIO_D5	DCMI_D7	-	EVENT OUT
	PB10	-	TIM2_CH3	-	-	I2C2_ SCL	SPI2_SCK/ I2S2_CK	SAI1_ SCK_A	USART3_ TX	-	-	OTG_HS_ ULPI_D3	-	-	-	-	EVENT OUT
	PB11	-	TIM2_CH4	-	-	I2C2_ SDA	-	-	USART3_ RX	SAI2_ SD_A	-	-	-	-	-	-	EVENT OUT
	PB12	-	TIM1_BKIN	-	-	I2C2_ SMBA	SPI2_NSS/ I2S2_WS	SAI1_ SCK_B	USART3_ CK	-	CAN2_RX	OTG_HS_ ULPI_D5	-	OTG_ HS_ID	-	-	EVENT OUT
	PB13	-	TIM1_CH1N	-	-	-	SPI2_SCK/ I2S2_CK	-	USART3_ CTS	-	CAN2_TX	OTG_HS_ ULPI_D6	-	-	-	-	EVENT OUT
	PB14	-	TIM1_CH2N	-	TIM8_ CH2N	-	SPI2_MISO	-	USART3_ RTS	-	TIM12_CH1	-	-	OTG_ HS_DM	-	-	EVENT OUT
	PB15	RTC_ REFIN	TIM1_CH3N	-	TIM8_ CH3N	-	SPI2_MOSI /I2S2_SD	-	-	-	TIM12_CH2	-	-	OTG_ HS_DP	-	-	EVENT OUT

To find alternative functions of the pins, we refer to the data sheet.

7.4.9 GPIO alternate function low register (GPIOx_AFRL) (x = A..H)

Address offset: 0x20

Reset value: 0x0000 0000

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

Bits 31:0 **AFRLy:** Alternate function selection for port x bit y (y = 0..7)

These bits are written by software to configure alternate function I/Os

AFRLy selection:

0000: AF0
0001: AF1
0010: AF2
0011: AF3
0100: AF4
0101: AF5
0110: AF6
0111: AF7
1000: AF8
1001: AF9
1010: AF10
1011: AF11
1100: AF12
1101: AF13
1110: AF14
1111: AF15

GPIOB 7,8

7.4.10 GPIO alternate function high register (GPIOx_AFRH) (x = A..H)

Address offset: 0x24

Reset value: 0x0000 0000

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

```
GPIOB→AFR[1] |= (4<<0);
GPIOB→AFR[0] |= (4<<28);
```

0 refers to low, 1 to high

Lets worry about the I2C related registers now

First we have to reset the I2C, this is in I2C→CR1 register

24.6.1 I²C control register 1 (I2C_CR1)

Address offset: 0x00
Reset value: 0x0000

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

Bit 15 **SWRST**: Software reset

When set, the I2C is under reset state. Before resetting this bit, make sure the I2C lines are released and the bus is free.

0: I²C Peripheral not under reset

1: I²C Peripheral under reset state

Note: This bit can be used to reinitialize the peripheral after an error or a locked state. As an example, if the BUSY bit is set and remains locked due to a glitch on the bus, the SWRST bit can be used to exit from this state.

Set this bit, to reset the I2c

Clear this bit to not be stuck in reset state

I2C1→CR1 |= (1<<15);

I2C1→CR1 &= ~(1<<15);

Set the frequency of the clock in CR2

second STOP, START or PEC request.

24.6.2 I²C control register 2 (I2C_CR2)

Address offset: 0x04
Reset value: 0x0000

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

Bits 5:0 **FREQ[5:0]**: Peripheral clock frequency

The FREQ bits must be configured with the APB clock frequency value (I2C peripheral connected to APB). The FREQ field is used by the peripheral to generate data setup and hold times compliant with the I2C specifications. The minimum allowed frequency is 2 MHz, the maximum frequency is limited by the maximum APB frequency (45 MHz) and cannot exceed 50 MHz (peripheral intrinsic maximum limit).

0b000000: Not allowed

0b000001: Not allowed

0b000010: 2 MHz

...

0b110010: 50 MHz

Higher than 0b101010: Not allowed

So,

I2C1→CR2 |= (45<<0);
Direct binary to decimal.

Next configure the clock control registers

It's the CCR register for I2C

24.6.8 I²C clock control register (I2C_CCR)

Address offset: 0x1C
Reset value: 0x0000

Note: *f_{PCLK1} must be at least 2 MHz to achieve Sm mode I²C frequencies. It must be at least 4 MHz to achieve Fm mode I²C frequencies*

The CCR register must be configured only when the I2C is disabled (PE = 0).

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

Bit 15 **F/S:** I2C master mode selection
0: Sm mode I2C
1: Fm mode I2C

The value calculation is given in reference of this register

Bits 11:0 **CCR[11:0]:** Clock control register in Fm/Sm mode (Master mode)
Controls the SCL clock in master mode.
Sm mode or SMBus:
$T_{high} = CCR * T_{PCLK1}$
$T_{low} = CCR * T_{PCLK1}$
Fm mode:
If DUTY = 0:
$T_{high} = CCR * T_{PCLK1}$
$T_{low} = 2 * CCR * T_{PCLK1}$
If DUTY = 1:
$T_{high} = 9 * CCR * T_{PCLK1}$
$T_{low} = 16 * CCR * T_{PCLK1}$
For instance: in Sm mode, to generate a 100 kHz SCL frequency:
If FREQ = 08, T_{PCLK1} = 125 ns so CCR must be programmed with 0x28
(0x28 <=> 40d x 125 ns = 5000 ns.)

Note: *The minimum allowed value is 0x04, except in FAST DUTY mode where the minimum allowed value is 0x01*

$t_{high} = t_{r(SCL)} + t_{w(SCLH)}$. See device datasheet for the definitions of parameters.

$t_{low} = t_{f(SCL)} + t_{w(SCLL)}$. See device datasheet for the definitions of parameters.

I2C communication speed, fSCL ~ 1/(thigh + tlow). The real frequency may differ due to the analog noise filter input delay.

The CCR register must be configured only when the I²C is disabled (PE = 0).

Let's see the values in data sheet for I2C Tr(SCL) Tw(SCLH) and and Tw(SCLL)

Table 61. I²C characteristics

Symbol	Parameter	Standard mode I²C(1)(2)		Fast mode I²C(1)(2)		Unit
		Min	Max	Min	Max	
$t_{w(SCLL)}$	SCL clock low time	4.7	-	1.3	-	µs
$t_{w(SCLH)}$	SCL clock high time	4.0	-	0.6	-	
$t_{su(SDA)}$	SDA setup time	250	-	100	-	ns
$t_{h(SDA)}$	SDA data hold time	-	3450(3)	-	900(4)	
$t_{v(SDA, ACK)}$	Data, ACK valid time	-	3.45	-	0.9	
$t_{r(SDA)}$ $t_{r(SCL)}$	SDA and SCL rise time	-	1000	-	300	
$t_{f(SDA)}$ $t_{f(SCL)}$	SDA and SCL fall time	-	300	-	300	
$t_{h(STA)}$	Start condition hold time	4.0	-	0.6	-	µs
$t_{su(STA)}$	Repeated Start condition setup time	4.7	-	0.6	-	
$t_{su(STO)}$	Stop condition setup time	4.0	-	0.6	-	µs
$t_{w(STO:STA)}$	Stop to Start condition time (bus free)	4.7	-	1.3	-	µs
	Pulse width of the spikes that are suppressed by the					

So, Thigh = Tr(SCL) + Tw(SCLH)

Tw(SCLH) = 4000 ns

Tr(SCL) = 1000 ns

Given formula is Thigh = CCR*TPCLK

TPCLK is 1/Fpclk = 1/45mhz = 22.22222 ns

CCR = Thigh / 22.2222 ns

Thigh = 5000 ns

So CCR = 5000/22.2222

= 225

So,

I2C1→CCR = 225<<0;

LOL

Configure the rise time register

For this we consult the TRise register

The CCR register must be configured only when the I²C is disabled (PE = 0).

24.6.9 I²C TRISE register (I2C_TRISE)

Address offset: 0x20
Reset value: 0x0002

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

TRISE = (Tr(SCL) / Tpclk) + 1
So, TRISE = 46
I2C1→TRISE = 46;

Bits 5:0 **TRISE[5:0]**: Maximum rise time in Fm/Sm mode (Master mode)
These bits should provide the maximum duration of the SCL feedback loop in master mode.
The purpose is to keep a stable SCL frequency whatever the SCL rising edge duration.
These bits must be programmed with the maximum SCL rise time given in the I²C bus specification, incremented by 1.
For instance: in Sm mode, the maximum allowed SCL rise time is 1000 ns.
If, in the I2C_CR2 register, the value of FREQ[5:0] bits is equal to 0x08 and T_{PCLK1} = 125 ns therefore the TRISE[5:0] bits must be programmed with 09h.
(1000 ns / 125 ns = 8 + 1)
The filter value can also be added to TRISE[5:0].
If the result is not an integer, TRISE[5:0] must be programmed with the integer part, in order to respect the t_{HIGH} parameter.
Note: TRISE[5:0] must be configured only when the I2C is disabled (PE = 0).

Enable the I2c

I2C1→CR1 |= (1<<0);

Bit 0 **PE**: Peripheral enable
0: Peripheral disable
1: Peripheral enable
Note: If this bit is reset while a communication is on going, the peripheral is disabled at the end of the current communication, when back to IDLE state.
All bit resets due to PE=0 occur at the end of the communication.
In master mode, this bit must not be reset before the end of the communication.

AWD.

How to send data to your slave so you can read its register values.
To know the protocol and inner workings, refer to Miko's doc

1. **Start the I2C**

```
I2C1→CR1 |= (1<<10);  // Enable the ACK
I2C1→CR1 |= (1<<8);  // Generate START
while (!(I2C1→SR1 & (1<<0)));
```

24.6.1 I²C control register 1 (I2C_CR1)

Address offset: 0x00
Reset value: 0x0000

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

24.6.6 I²C status register 1 (I2C_SR1)

Address offset: 0x14
Reset value: 0x0000

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

Note: ADDR is not set after a NACK reception

Bit 0 **SB**: Start bit (Master mode)

0: No Start condition
1: Start condition generated.

– Set when a Start condition generated.
– Cleared by software by reading the SR1 register followed by writing the DR register, or by hardware when PE=0

2. **Send slave address in Write mode**
   First thing to consider is, when we want to read from the slave and we're writing its address, the LSB of the address needs to be set to 0.
   Lets assume our address for slave is 0×76, then we send Address = 0×76*2 (left shifted once to have 0 as lsb)

We keep any data we wanna send into the DR, and it sends it (too ez)

I2C1→DR = Address; // send the address

while (!(I2C1→SR1 & (1<<1))) // THIS time we wait for a different bit to set. As we are sending address, addr bit gets set

register and PEC=1 in I2C_CR1 register)

Bit 1 **ADDR**: Address sent (master mode)/matched (slave mode)

This bit is cleared by software reading SR1 register followed reading SR2, or by har
when PE=0.

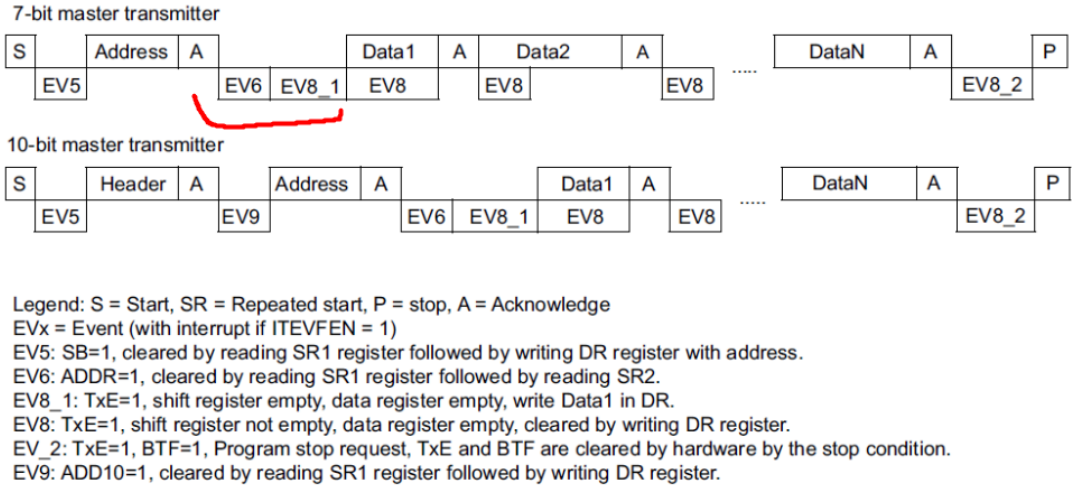
Address matched (Slave)

0: Address mismatched or not received.

1: Received address matched.

uint8_t temp = I2C1→SR1 | I2C1→SR2;

This piece of code is to read SR1 and SR2 to reset their values. Check miko's doc



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.

If you check the EV conditions carefully, you will see the need for these.

3. **Send the register address you want to read**

while (!(I2C1→SR1 & (1<<7))); // wait for TXE bit to set

TXE bit is transmission buffer empty. Set means we're good to send new data

I2C1→DR = data;

while (!(I2C1→SR1 & (1<<2)));//Second bit is actually BTF or

Bit transfer Complete

Important distinction for these:

A. When we start I2C and send the first ACK bit, We wait for the SB to set

B. When we send an address we wait for addr to set

C. When we send a data, we wait for BTF to set

4. **Start I2C again**
5. **Then I2C Read**

Reading is a bit complex. Now that the peripheral device knows which register values to send, you keep sending it its address, it keeps giving you data. You read the data by knowing when RXNE bit is set. That's the buffer. You send an ack to the slave and keep doing it, peripheral will send next byte and so on. If you want to end reading, send a NACK. and Stop I2C. This will let the slave send its final byte.

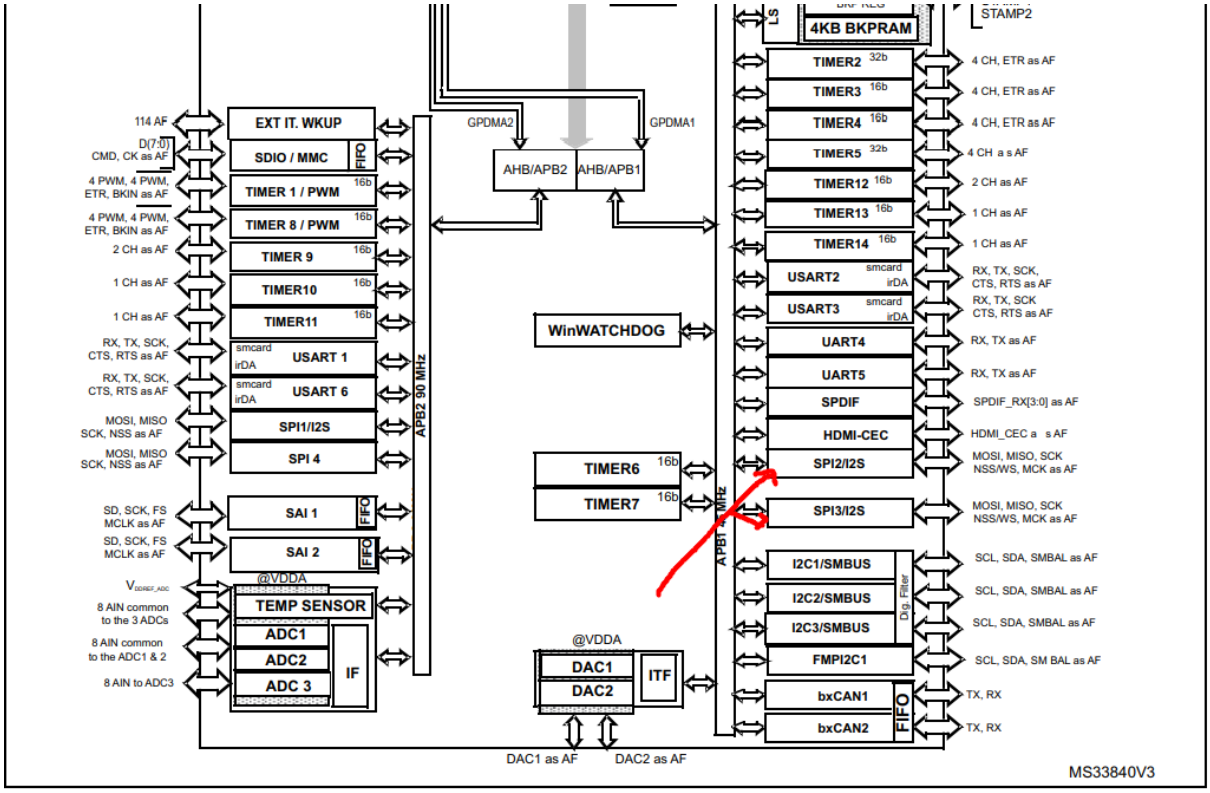
```
I2C1->DR = Address;  //  send the address
while (!(I2C1->SR1 & (1<<1)));  // wait for ADDR bit to set

/**** STEP 1-b ****/
I2C1->CR1 &= ~(1<<10);  // clear the ACK bit
uint8_t temp = I2C1->SR1 | I2C1->SR2;  // read SR1 and SR2 to clear the ADDR bit.... EV6 co
I2C1->CR1 |= (1<<9);  // Stop I2C

/**** STEP 1-c ****/
while (!(I2C1->SR1 & (1<<6)));  // wait for RxNE to set

/**** STEP 1-d ****/
buffer[size-remaining] = I2C1->DR;  // Read the data from the DATA REGISTER
```

—----------#----------

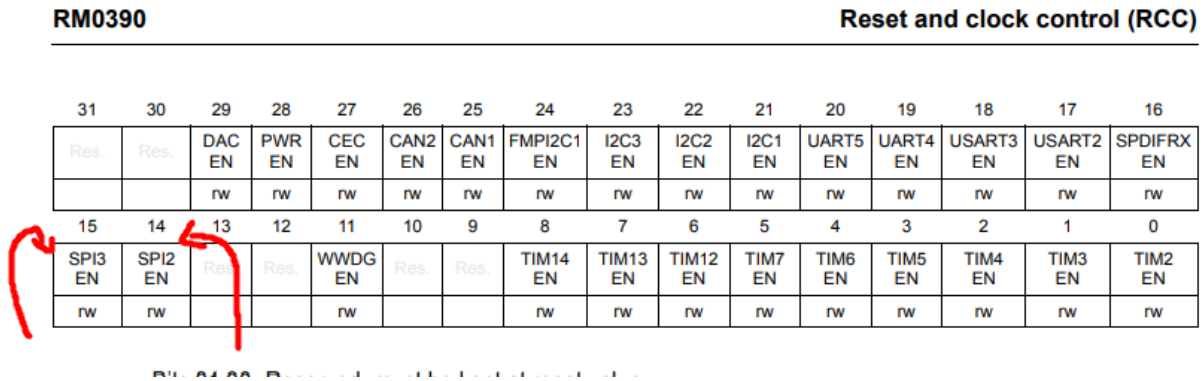
SPI configuration

SPI is way easier than I2C to configure

1. **Enable the Clock SPI is connected to**



It's in APB1 again, so



RCC→APB1ENR |= (1<<14);

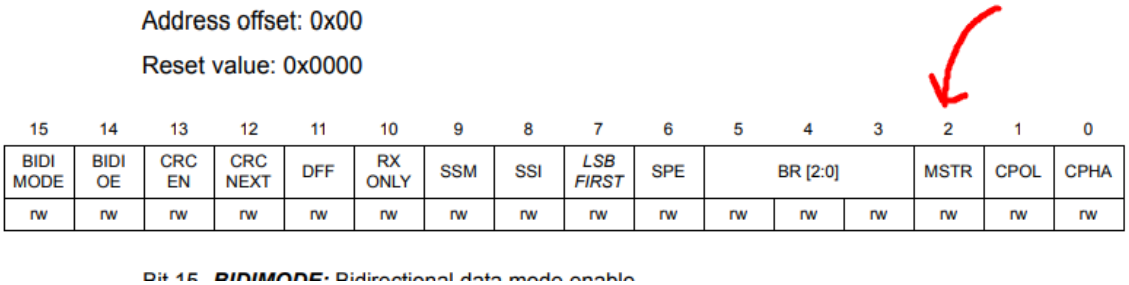
Let's look at the SPI registers

Set the SPI as master mode

26.7.1 SPI control register 1 (SPI_CR1) (not used in I²S mode)

Address offset: 0x00

Reset value: 0x0000



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRC EN	CRC NEXT	DFF	RX ONLY	SSM	SSI	*LSB FIRST*	SPE	BR [2:0]			MSTR	CPOL	CPHA
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 15 ***BIDIMODE:*** Bidirectional data mode enable

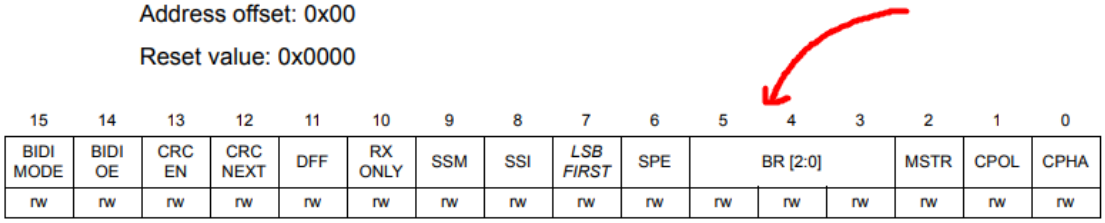
So,
SPI2→CR1 |= (1<<2);

Setting the baud Rate

26.7.1 SPI control register 1 (SPI_CR1) (not used in I²S mode)

Address offset: 0x00

Reset value: 0x0000



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRC EN	CRC NEXT	DFF	RX ONLY	SSM	SSI	*LSB FIRST*	SPE	BR [2:0]			MSTR	CPOL	CPHA
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

This bit is not used in I²S mode.

Bits 5:3 **BR[2:0]:** Baud rate control

000: $f_{PCLK}/2$
001: $f_{PCLK}/4$
010: $f_{PCLK}/8$
011: $f_{PCLK}/16$
100: $f_{PCLK}/32$
101: $f_{PCLK}/64$
110: $f_{PCLK}/128$
111: $f_{PCLK}/256$

Note: These bits should not be changed when communication is ongoing.
They are not used in I²S mode.

So baud rate is now 5.625 mhz

SPI2→CR1 |= (2<<3);

Next, we Clear the LSB First bit, so it becomes MSB first protocol,

This is for message format

26.7.1 SPI control register 1 (SPI_CR1) (not used in I²S mode)

Address offset: 0x00

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRC EN	CRC NEXT	DFF	RX ONLY	SSM	SSI	*LSB FIRST*	SPE	BR [2:0]			MSTR	CPOL	CPHA
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Note: ***This bit is not used in I²S mode and SPI TI mode***

Bit 7 ***LSBFIRST:*** Frame format

0: MSB transmitted first

1: LSB transmitted first

Note: This bit should not be changed when communication is ongoing.

It is not used in I²S mode and SPI TI mode

Bit 6 **SPE:** SPI enable

So,

SPI2→CR1 &= ~(unsigned) (1<<7);

Bit 9 **SSM:** Software slave management

When the SSM bit is set, the NSS pin input is replaced with the value from the SSI bit.

0: Software slave management disabled

1: Software slave management enabled

Note: ***This bit is not used in I²S mode and SPI TI mode***

+0 ignore NSS

Bit 8 **SSI:** Internal slave select

This bit has an effect only when the SSM bit is set. The value of this bit is forced onto the NSS pin and the IO value of the NSS pin is ignored.

Note: ***This bit is not used in I²S mode and SPI TI mode***

SPI2→CR1 |= (1u<<8) | (1u<<9);

Next, we want it to be in full-duplex mode, cz why not

Bit 10 **RXONLY:** Receive only mode enable

This bit enables simplex communication using a single unidirectional line to receive data exclusively. Keep BIDIMODE bit clear when receive only mode is active.

This bit is also useful in a multislave system in which this particular slave is not accessed, the output from the accessed slave is not corrupted.

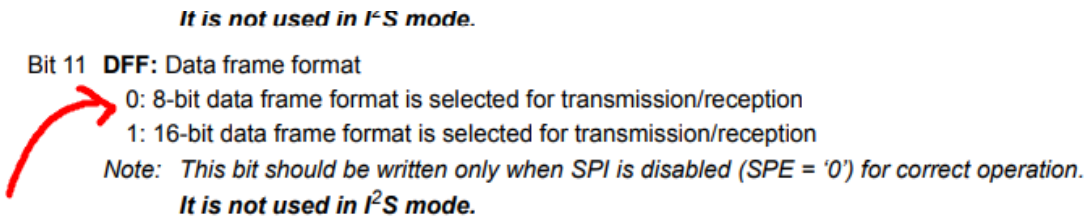
0: full-duplex (Transmit and receive)

1: Output disabled (Receive-only mode)

Note: ***This bit is not used in I²S mode***

Guess the code for this

Next, we set the data frame size



SPI2→CR1 &= ~(1<<11)

Clear the contents of the CR2
SPI2→CR2 = 0;

SPI works very simply.
The chip selector pin needs to go from 1 to 0 to start communicating, and then 0 to 1 again to stop.

So we use a GPIO pin and change its value before sending or receiving.
The GPIO configuration

The GPIO pins needed for SPI are MISO (Master in, slave out)
MOSI, and SCLK
They all need to be alternate functions, high speed
And depending on which pins we choose, we find its alternate function in data sheet. For example, if we use PC2, PC3 and PC7 as MISO, MOSI and SCK

Table 11. Alternate function (continued)

Port	AF0 SYS	AF1 TIM1/2	AF2 TIM3/4/5	AF3 TIM8/9/10/11 CEC	AF4 I2C1/2/3 /4/CEC	AF5 SPI1/2/3/4	AF6 SPI2/3/4/ SAI1	AF7 SPI2/3/ USART1/2/3 /UART5/ SPDIFRX	AF8 SAI/ USART6/ UART4/5/ SPDIFRX	AF9 CAN1/2 TIM12/13/ 14/ QUADSPI	AF10 SAI2/ QUADSPI/ OTG2_HS/ OTG1_FS	AF11 OTG1_FS	AF12 FMC/ SDIO/ OTG2_FS	AF13 DCMI	AF14 -	AF15 SYS
PC0	-	-	-	-	-	-	SAI1_ MCLK_B	-	-	-	OTG_HS_ ULPI_STP	-	FMC_ SDNWE	-	-	EVENT OUT
PC1	-	-	-	-	-	SPI3_MOSI /I2S3_SD	SAI1_ SD_A	SPI2_MOSI /I2S2_SD	-	-	-	-	-	-	-	EVENT OUT
PC2	-	-	-	-	-	SPI2_MISO	-	-	-	-	OTG_HS_ ULPI_DIR	-	FMC_ SDNE0	-	-	EVENT OUT
PC3	-	-	-	-	-	SPI2_MOSI I2S2_SD	-	-	-	-	OTG_HS_ ULPI_NXT	-	FMC_ SDCKE0	-	-	EVENT OUT
PC4	-	-	-	-	-	I2S1_MCK	-	-	SPDIF_ RX2	-	-	-	FMC_ SDNE0	-	-	EVENT OUT
PC5	-	-	-	-	-	-	-	USART3_RX	SPDIF_ RX3	-	-	-	FMC_ SDCKE0	-	-	EVENT OUT
PC6	-	-	TIM3_CH1	TIM8_CH1	FMPI2C1 _SCL	I2S2_MCK	-	-	USART6 _TX	-	-	-	SDIO_D6	DCMI_D0	-	EVENT OUT
PC7	-	-	TIM3_CH2	TIM8_CH2	FMPI2C1 _SDA	SPI2_SCK I2S2_CK	I2S3_MCK	SPDIF_RX1	USART6 _RX	-	-	-	SDIO_D7	DCMI_D1	-	EVENT OUT
PC8	TRACE D0	-	TIM3_CH3	TIM8_CH3	-	-	-	UART5_RTS	USART6 _CK	-	-	-	SDIO_D0	DCMI_D2	-	EVENT OUT
PC9	MCO2	-	TIM3_CH4	TIM8_CH4	I2C3_ SDA	I2S_CKIN	-	UART5_CTS	-	QUADSPI_ BK1_IO0	-	-	SDIO_D1	DCMI_D3	-	EVENT OUT
PC10	-	-	-	-	-	-	SPI3_SCK /I2S3_CK	USART3_TX	UART4_TX	QUADSPI_ BK1_IO1	-	-	SDIO_D2	DCMI_D8	-	EVENT OUT

How to get sum data

1. **Pull the CSB pin to 0 (simple gpio stuff)**
2. **Send the register address**

Lets look at the SR register for SPI

26.7.3 SPI status register (SPI_SR)

Address offset: 0x08

Reset value: 0x0002

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	FRE	BSY	OVR	MODF	CRC ERR	UDR	CHSIDE	TXE	RXNE
							r	r	r	r	rc_w0	r	r	r	r

while (!((SPI2→SR)&(1<<1))); // wait for transmission buffer to be empty

SPI2→DR = address

while (!((SPI2→SR)&(1<<1)));

while (((SPI2→SR)&(1<<7))); //Make sure SPI is not busy anymore

uint8_t temp = (uint8_t)SPI2→DR;

temp = (uint8_t)SPI2→SR;

Read the SR and DR to reset the flags.

3. **Get the data**

while (((SPI2→SR)&(1<<7))) {}; //wait for it to not be busy
SPI2→DR = 0; //You need to send one dummy data for transmission to start
while (!((SPI2→SR) &(1<<0))){}; //Wait for sth to come in the Receive buffer RXNE
***data++ = SPI2→DR;** //Whatever is in the DR, is the data sent by slave

4. **Pull the CSB pin to 1**
   Simple GPIO reset

Feed me coffee.