25th Batch MCU Solve at the END. [question 7(c) updated]

WILL BE REFERRING TO THESE AS [1] and [2]

[1] Miko Bhais Note: Note_1.pdf

[2] Radi Bhais Note: Note_2.pdf

Clock Configuration

- Read through [1] for theory, [2] for code
- Additionally Sirs Slide (maybe)
  Final Code Snippet for Clock Configuration:

```
void initClock(void)
{
RCC->CR |=RCC_CR_HSEON; // set CR bit 16
/** Check if the clock is ready RCC CR register 17th-bit set*/
while(!(RCC->CR & RCC_CR_HSERDY)); //wait for the clock is
enabled See RCC CR bit-17; HSE crystal is On
/* Set the POWER to enable CLOCK and VOLTAGE REGULATOR */
RCC->APB1ENR |= RCC_APB1ENR_PWREN; //power enable for APB1
PWR->CR |= PWR_CR_VOS; //VOS always correspond to reset value

/*3. Configure the FLASH PREFETCH and the LATENCY Related
Settings */
FLASH->ACR |= FLASH_ACR_ICEN | FLASH_ACR_DCEN | FLASH_ACR_PRFTEN
| FLASH_ACR_LATENCY_5WS; //ICEN -- instruction cache, DCEN --
Data Cache, PRFTEN -- prefetch and LAtency;

/* 4. Configure the PRESCALERS HCLK, PCLK1, PCLK2 */
//AHB prescaler
RCC->CFGR |= RCC_CFGR_HPRE_DIV1;
//APB1 prescaler
RCC->CFGR |= RCC_CFGR_PPRE1_DIV4;
//APB2 prescaler
RCC->CFGR |= RCC_CFGR_PPRE2_DIV2;
//5. Configure the main PLL
RCC->PLLCFGR = (PLL_M<<0) | (PLL_N<<6) | (PLL_P<<16) |
(RCC_PLLCFGR_PLLSRC_HSE);
//6. Enable PLL and wait for it to become ready
RCC->CR |= RCC_CR_PLLON;
//Check if PLL clock is ready
while(!(RCC->CR & RCC_CR_PLLRDY)); //wait for PLL ready

//7. Select clock source and wait for it to be set
```

```
      RCC->CFGR |= RCC_CFGR_SW_PLL;
while((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_PLL);
}
```

GPIO

- **Important:** DIfferent Circuits for output states from note [1]. (page 5,6,7)
    - Open Drain
    - Push-Pull
    - Pull up, pull down

- **GPIO Configuration**
    - Procedure: Note [1], [2]
    - Code Snippet:
        - Just use your code from Assignment 1

- **Additional Topics Not available in [1], [2]**
    - Configure In Input Mode

7.4.1 GPIO port mode register (GPIOx_MODER) (x = A..H)

Address offset: 0x00

Reset values:

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

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

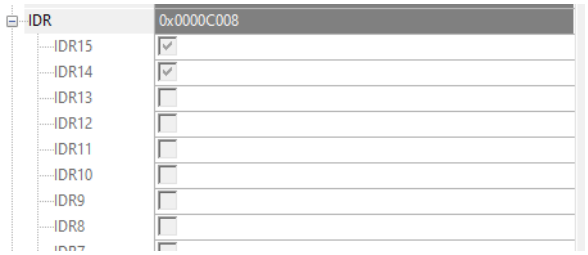
Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

For configuring a pin in input mode simply set ‘00’ in ‘MODER’ register.
For example, if you want to configure PIN, PA8 in input mode, [bit 11:10 is used for MODER8]

```
GPIOA->MODER &= ~((1 << 11) | (1 << 10));
```

- But why tho?
    - If a pin is configured in input mode it can be used to check if some other pin has an output state of HIGH or

LOW. This information is stored in the GPIOx->IDR register.



- For example you want to check if the Pin PA5 is on or off using the IDR register. Connect a pin from PA5 to PA8 (which we will use in input mode). Now if the 8th bit of GPIOA->IDR is on it indicates that PA5 is on otherwise off.
  You can check this with a simple bitwise operation.

```
if (GPIOA->IDR & (1 << 8)){
	//PA5 is on
}else{
	//PA5 is off
}
```

USART/UART

- Again, For theory and configuration refer to [1] and [2]
  Example Config Code for UART4

```
void UART4_Config(void){

	//1. Enable UART clock and GPIO clock
	RCC->APB1ENR |= RCC_APB1ENR_UART4EN;
	RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; //enable GPIOA clock


	//2. Configure UART pin for Alternate function
	GPIOA->MODER |= (2<<0);  //PA0 -> Tx
	GPIOA->MODER |= (2<<2);  //PA1 -> Rx

	GPIOA->OSPEEDR |= (GPIO_SPEED_FREQ_HIGH<<0) |
	                  (GPIO_SPEED_FREQ_HIGH<<2);

	GPIOA->AFR[0] |= (8<<0); // bits(3,2,1,0) = (1000)
	GPIOA->AFR[0] |= (8<<4); // bits(7,6,5,4) = (1000)

	//3. Enable UART on UART4_CR1 register
	UART4->CR1 = 0x00; //clear USART
```

```
    UART4->CR1 |= (1<<13);  // UE-bit enable USART

    //4. Program M bit in USART CR1 to define the word length
    UART4->CR1 &= ~(1U<<12); // set M bit  = 0 for 8-bit word length

    //5. Select the baud rate using the USART_BRR register.
    UART4->BRR |= (7<<0) | (24<<4); //115200

    //  6. Enable transmission TE and reception bits in USART_CR1 register
    UART4->CR1 |= (1<<2); // enable RE for receiver
    UART4->CR1 |= (1<<3); //enable TE for transmitter

   UART4->CR1 |= USART_CR1_RXNEIE; //enable RXNEIE for receiver
}
```

- Make sure to enable Receive interrupt (RXNEIE) for all the UARTs if you want to use IRQ_Handler for the input. Also, enable the interrupt in the NVIC.

```
UART4->CR1 |= USART_CR1_RXNEIE;
NVIC_EnableIRQ(USART4_IRQn);
```

Taking input from Hercules(or any other debugger) using USART2

```
char input_buff[100];

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

void getString(void){
    uint8_t ch,idx = 0;
    ch = UART_GetChar(USART2);
    while(ch != '!'){
        input_buff[idx++] = ch;
        ch = UART_GetChar(USART2);
        if(ch == '!')break;
    }
    input_buff[idx] = '\0';

}
```

Explanation: Whenever you send some string or data from the debugger the

‘USART2_IRQHandler(void)’ function will automatically be called in the background. You don’t have to invoke it manually.

- Turn off the RXNEIE so that the function doesn’t get called again when the current string is being received
- Take input until the terminating character is received using the getString function. I used ‘!’ as the last character you can use anything you like.
- After the input string is received turn on the RXNEIE again.

- I’ve used the UART_GetChar() function provided by sir.

```
uint8_t UART_GetChar(USART_TypeDef *usart){
      uint8_t tmp;
      while(!(usart->SR & (USART_SR_RXNE)));
      tmp=(uint8_t)usart->DR;
      return tmp;
}
```

Explanation: Whenever a character arrives at the data register of the usart (USART2->DR) the RXNE bit is automatically set. So we wait until this bit is set. Then we simply read whatever is in the USART2->DR and return the result.

Transmitting Data from one interrupt to another

We will see how we can transfer data from UART4 to UART5 and vice-versa

```
char input_buff[100],output_buff[100];
int out_idx = 0, in_idx = 0;

void UART4_IRQHandler(void)
{
    if (UART4->SR & USART_SR_RXNE){
        output_buff[out_idx] = (uint8_t) UART4->DR;
        UART4->SR &= ~(USART_SR_RXNE);
    }

    if (UART4->SR & USART_SR_TXE){
        //handle queue here
        UART4->DR = input_buff[in_idx];
        while(!(UART4->SR & USART_SR_TXE));
        UART4->CR1 &= ~(USART_CR1_TXEIE);
    }

}
```

** Write the exact same function for UART5. Just replace all the ‘UART4’ with ‘UART5’

```
void transmit_data(uint32_t direction)
{
    //transmit data from UART4 to UART5
    uint32_t i = 0;
    in_idx = 0;
      out_idx = 0;

    for (i = 0;i < strlen(input_buff);i++){
        //Enable Interrupt
        UART4->CR1 |= USART_CR1_TXEIE;
        while((UART4->CR1 & USART_CR1_TXEIE));
        ms_delay(1);
        in_idx++;
        out_idx++;
    }
    output_buff[out_idx++] = '\0';
}
```

Explanation:

- Whenever we have data in the input_data[] string we will transfer this data from input_data[] to output_data[] string through UART4->UART5.
- Call the transmit_data() function when we have a string in the input_data[].
- The transmit_data() function will enable the TXEIE (transmit interrupt) for transmitting each character one by one. When TXEIE is enabled the USARTx_IRQHandler() will get called whenever the TXE bit is enabled.
- The TXE bit is enabled whenever the UARTx->DR register is empty and ready for data transfer. Here. “Data transferred to the shift register” means any previous data in the DR register has been transferred over to the shift register and the DR register is currently empty.

Bit 7 **TXE**: Transmit data register empty

This bit is set by hardware when the content of the TDR register has been transferred into the shift register. An interrupt is generated if the TXEIE bit =1 in the USART_CR1 register. It is cleared by a write to the USART_DR register.

0: Data is not transferred to the shift register

1: Data is transferred to the shift register)

Note: This bit is used during single buffer transmission.

Bit 5 **RXNE**: Read data register not empty

This bit is set by hardware when the content of the RDR shift register has been transferred to the USART_DR register. An interrupt is generated if RXNEIE=1 in the USART_CR1 register. It is cleared by a read to the USART_DR register. The RXNE flag can also be cleared by writing a zero to it. This clearing sequence is recommended only for multibuffer communication.

0: Data is not received

1: Received data is ready to be read.

- After the UART4_IRQHandler is invoked it will check if this UART4 has received any data using the RXNE bit. If it has it will read the data and put it in the output_buff[] string. It also checks if the TXE bit is enabled or in other words if we need to transmit a character.
- If yes we simply put the character we want to transfer into the DR register and it will be transmitted to the data register of UART5.

I2C

Configuration

```
void I2C1_Config(uint8_t mode){
    //Enable I2C and GPIO
    RCC->APB1ENR |= RCC_APB1ENR_I2C1EN;
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;

    //Configure Pin 8,9 (GPIOB)
    GPIOB->MODER |= (GPIO_MODER_MODER8_1 | GPIO_MODER_MODER9_1);
    GPIOB->OTYPER |= (GPIO_OTYPER_OT8 | GPIO_OTYPER_OT9);
    GPIOB->OSPEEDR |= (GPIO_OSPEEDR_OSPEED8 | GPIO_OSPEEDR_OSPEED9);
    GPIOB->PUPDR |= (GPIO_PUPDR_PUPD8_0 | GPIO_PUPDR_PUPD9_0);
    GPIOB->AFR[1] |= (GPIO_AFRH_AFSEL8_2 | GPIO_AFRH_AFSEL9_2);

    //Configure I2C1

    I2C1->CR1 |= I2C_CR1_SWRST;
    I2C1->CR1 &= ~I2C_CR1_SWRST;

    I2C1->CR2 |= (45 << I2C_CR2_FREQ_Pos);
    I2C1->CCR = 225 << I2C_CCR_CCR_Pos;
    I2C1->TRISE = 46 << I2C_TRISE_TRISE_Pos;

    I2C1->CR1 |= I2C_CR1_PE;
    I2C1->CR1 &= ~I2C_CR1_POS;
```

```
    //additional slave config if mode is set to 0
    if(mode == 0){
        /*Set Address and Enable Interrupts for Slave Receieve*/
        I2C1_SetAddress(0);
        I2C1->CR2 |= I2C_CR2_ITEVTEN; /*Enable Event Interrupt*/
        NVIC_EnableIRQ(I2C1_EV_IRQn);
    }
}
```

Steps:

- For the first few steps [until if(mode == 0) line] follow through with note [1] and [2].
- Since we also have to configure a slave device we need to set an address for the slave in the OAR1 register.

24.6.3 I²C own address register 1 (I2C_OAR1)

Address offset: 0x08
Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD MODE	Res.	Res.	Res.	Res.	Res.	ADD[9:8]		ADD[7:1]							ADD0
rw						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 15 **ADDMODE** Addressing mode (slave mode)
0: 7-bit slave address (10-bit address not acknowledged)
1: 10-bit slave address (7-bit address not acknowledged)

Bit 14 Should always be kept at 1 by software.

Bits 13:10 Reserved, must be kept at reset value

Bits 9:8 **ADD[9:8]**: Interface address
7-bit addressing mode: don't care
10-bit addressing mode: bits9:8 of address

Bits 7:1 **ADD[7:1]**: Interface address
bits 7:1 of address

Bit 0 **ADD0**: Interface address
7-bit addressing mode: don't care
10-bit addressing mode: bit 0 of address

```
void I2C1_SetAddress(uint8_t address){
    I2C1->OAR1 |= (uint32_t)(address << 1U);
    I2C1->OAR1 &= ~I2C_OAR1_ADDMODE;
}
```

- We will be using 7-bit addressing
- Also, enable the I2C event interrupt which we will use for receiving data.

Interrupt event	Event flag	Enable control bit
Start bit sent (Master)	SB	ITEVFEN
Address sent (Master) or Address matched (Slave)	ADDR	
10-bit header sent (Master)	ADD10	
Stop received (Slave)	STOPF	
Data byte transfer finished	BTF	

- Finally, enable the interrupt in the NVIC

Some additional functions needed for data transmission

* I've added a timeout for each of these but it can be excluded for theory exams (maybe)

```
bool I2C1_Start(void){
    I2C1->CR1 |= I2C_CR1_START;
    TIM3->CNT = 0;
    while(!(I2C1->SR1 & I2C_SR1_SB)){
        if(TIM3->CNT > timeout_ms){
            return false;
        }
    }

      return true;
}

void I2C1_Stop(void){
    I2C1->CR1 |= I2C_CR1_STOP;
}


bool I2C1_Address(uint8_t address){
    I2C1->DR = (uint8_t)(address << 1);
    TIM3->CNT = 0;

    while(!(I2C1->SR1 & I2C_SR1_ADDR)){
        if(TIM3->CNT > timeout_ms){
            return false;
        }
    }
    //Clear ADDR (read SR1 and SR2)
    address = (uint8_t)(I2C1->SR1 | I2C1->SR2);

      return true;
}
bool I2C1_Write(uint8_t data){
    TIM3->CNT = 0;

    while(!(I2C1->SR1 & I2C_SR1_TXE)){
        if(TIM3->CNT > timeout_ms){
```

```
            return false;
        }
    }

    I2C1->DR = data;

    TIM3->CNT = 0;
    while(!(I2C1->SR1 & I2C_SR1_BTF)){
        if(TIM3->CNT > timeout_ms){
            return false;
        }
    }
      return true;
}
```

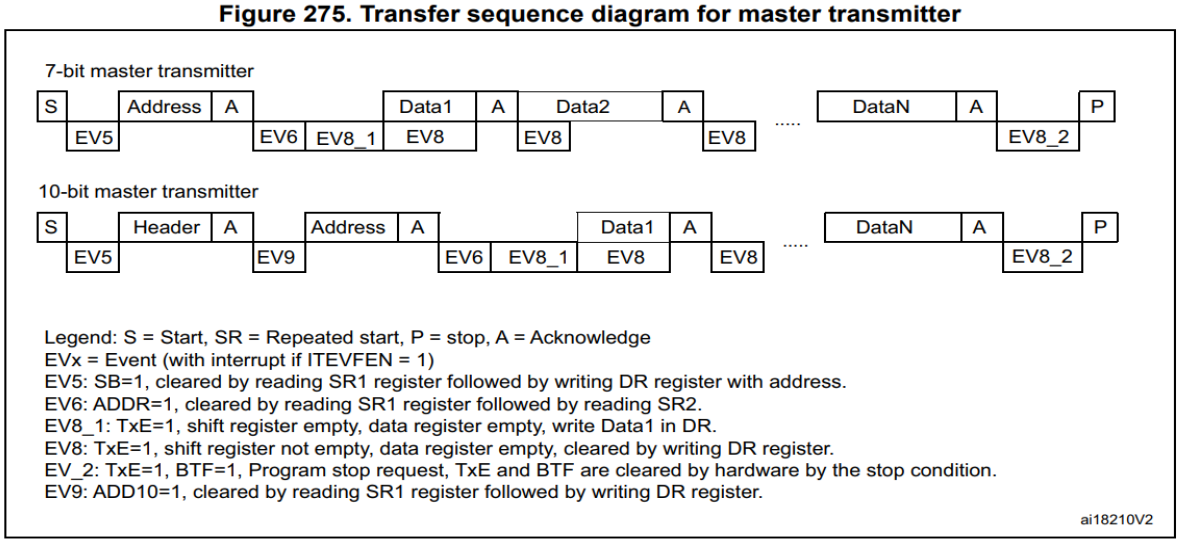
Explanation:

- I2C1_Start(): In i2c communication protocol the master always sends the start bit. In order to do that we just set the START bit in I2C1->CR1 register.
- I2C1_Stop(): The master also sends the stop bit. For that, we simply turn on the STOP bit in I2C1->CR1 register
- I2C1_Adress(): This function is used to send the address from master to slave.
    - First, simply put the address to the DR register
    - Wait for the ADDR bit to set. In master mode, this bit indicates if the address has been sent or not.
    - Finally to clear the ADDR bit we need to do a sequence of tasks. We cannot just simply set this bit 0. If you read the I2C1->SR1 and I2C1->SR2 register in order the ADDR will automatically be cleared. ** I read these registers in the ‘address’ variable but you can read it using any variable you choose.
- I2C1_Write(): Similar to what we did in USART we first check if the I2C1->DR register is empty using the TXE bit in I2C1->SR1 register. If yes, we simply put the character in the I2C1->DR register. After that we use the BTF (by transfer finished) bit in the I2C1->SR1 register to check if the bit has been properly transferred.

Data Transfer

Master Transmit

Figure 275. Transfer sequence diagram for master transmitter



1. The EV5, EV6, EV9, EV8_1 and EV8_2 events stretch SCL low until the end of the corresponding software sequence.
2. The EV8 event stretches SCL low if the software sequence is not complete before the end of the next byte transmission.

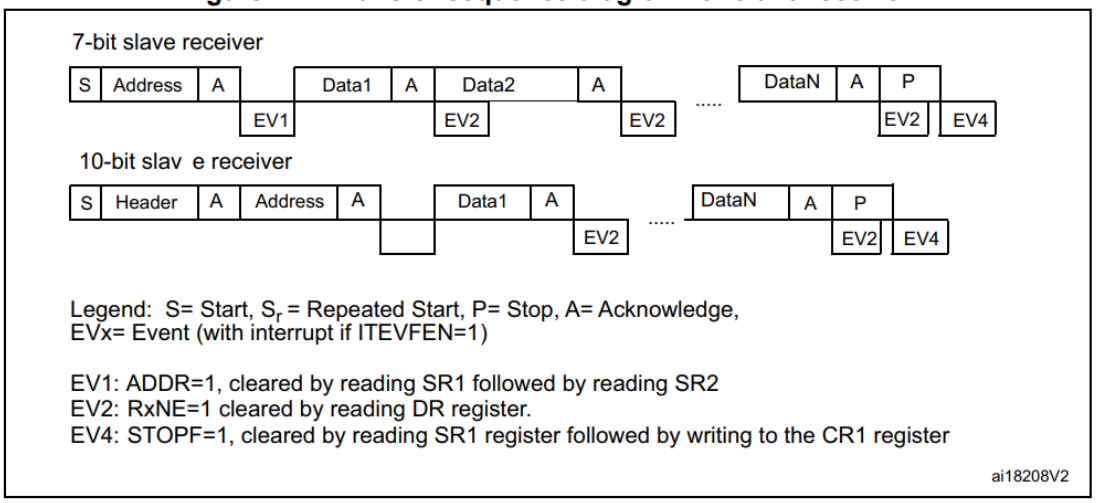
```
/*read and write functions*/
bool I2C1_TransmitMaster(char *buffer, uint32_t size){
    if(!I2C1_Start())return false;    /* generate start */
    if(!I2C1_Address(0))return false; /* send address */
    for(int idx = 0;idx < (int)size;idx++){
            if(!I2C1_Write(buffer[idx])){
                  return false;
            }/* send data */
      }
    I2C1_Stop();  /* generate stop */
    return true;
}
```

Here, I've simply followed the transfer sequence for the 7-bit addressing given in the figure above.
Steps:

- Send the start bit. Use the I2C1_Start() function.
- Send the address. Use the I2C1_Adress() function.
- Send all the data one character at a time. Use the I2C1_Write() function
- Finally, send the stop bit. Use the I2C1_Stop() function.

Slave Receive

Figure 274. Transfer sequence diagram for slave receiver



1. The EV1 event stretches SCL low until the end of the corresponding software sequence.
2. The EV2 event stretches SCL low if the software sequence is not completed before the end of the next byte reception.
3. After checking the SR1 register content, the user should perform the complete clearing sequence for each flag found set.
   Thus, for ADDR and STOPF flags, the following sequence is required inside the I2C interrupt routine:
   READ SR1
   if (ADDR == 1) {READ SR1; READ SR2}
   if (STOPF == 1) {READ SR1; WRITE CR1}
   The purpose is to make sure that both ADDR and STOPF flags are cleared if both are found set.

```
char* I2C1_ReceiveSlave(uint8_t *buffer){
    uint32_t idx = 0,size = 0;
    uint8_t ch = 0;
    char ret[100];

    while(!(I2C1->SR1 & I2C_SR1_ADDR)); /*wait for address to set*/
    idx = I2C1->SR1 | I2C1->SR2; /*clear address flag*/

    idx = 0;

    while(ch != '@'){
        while(!(I2C1->SR1 & I2C_SR1_RXNE)); /*wait for rxne to set*/
        ch = (uint8_t)I2C1->DR;/*read data from DR register*/
        if(ch == '@')break;
        buffer[idx++] = ch;
        size++;
    }
    buffer[idx] = '\0';

    while(!(I2C1->SR1 & I2C_SR1_STOPF)); /*wait for stopf to set*/
    /*clear stop flag*/
    idx = I2C1->SR1;
    I2C1->CR1 |= I2C_CR1_PE;
```

```
    I2C1->CR1 &= ~I2C_CR1_ACK; /*disable ack*/
    for(idx = 0;idx < size;idx++){
        ret[idx] = buffer[idx];
    }ret[idx] = '\0';

    return ret;
}
```

Again, Follow the slave receive diagram from the reference manual.
Steps:

- After the address is received the slave device will automatically check if the address is matched. If it is matched the ADDR bit in I2C1->SR1 will be set. Similar to what we did in the I2C1_Address() function wait for the ADDR bit to set and then clear it by reading SR1 and SR1.
- Receive each character one by one. Similar to what we did in USART, check if the RXNE bit in the I2C1->SR1 register is enabled. If yes, read the character (or byte) from the I2C1->DR register.
- Finally, we wait for the STOP bit to be set by the master at the end of data transfer. In order to clear the STOP bit set in I2C1->SR1 register we also have to follow some sequence of operation like we did for the ADDR bit.
    - Read I2C1->SR1
    - Write I2C1->CR1 (Just enable the PE bit which is already enabled)
- Finally, disable ACK and return the string.

Master Receive

Disclaimer: The code for master receive has not been tested. Probably won't work :3. Use at your own risk.

Master receiver

Following the address transmission and after clearing ADDR, the I^2C interface enters Master Receiver mode. In this mode the interface receives bytes from the SDA line into the DR register via the internal shift register. After each byte the interface generates in sequence:

1. An acknowledge pulse if the ACK bit is set
2. The RxNE bit is set and an interrupt is generated if the ITEVFEN and ITBUFEN bits are set (see *Figure 276* Transfer sequencing EV7).

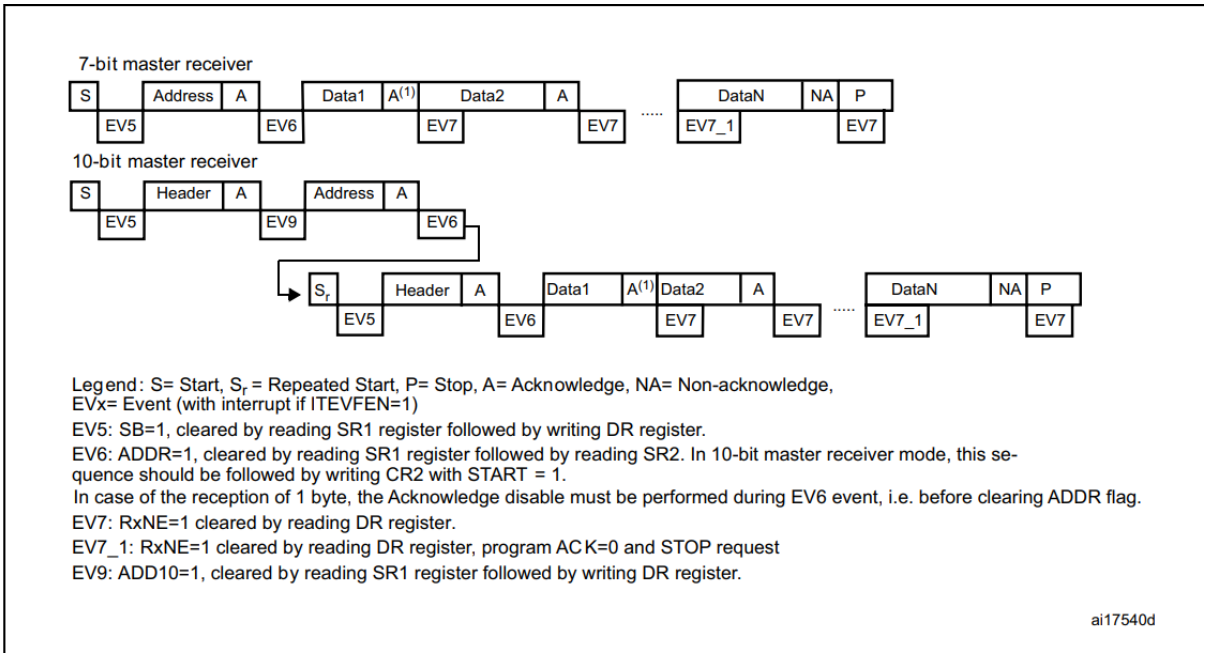
If the RxNE bit is set and the data in the DR register is not read before the end of the last data reception, the BTF bit is set by hardware and the interface waits until BTF is cleared by a read in the DR register, stretching SCL low.

Closing the communication

The master sends a NACK for the last byte received from the slave. After receiving this NACK, the slave releases the control of the SCL and SDA lines. Then the master can send a Stop/Restart condition.

1. To generate the nonacknowledge 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).
2. 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).
3. In case a single byte has to be received, the Acknowledge disable is made during EV6 (before ADDR flag is cleared) and the STOP condition generation is made after EV6.

After the Stop condition generation, the interface goes automatically back to slave mode (MSL bit cleared).



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.

ai17540d

The procedures described below are recommended if the EV7-1 software sequence is not completed before the ACK pulse of the current byte transfer.

These procedures must be followed to make sure:

- The ACK bit is set low on time before the end of the last data reception
- The STOP bit is set high after the last data reception without reception of supplementary data.

For 2-byte reception:

- Wait until ADDR = 1 (SCL stretched low until the ADDR flag is cleared)
- Set ACK low, set POS high
- Clear ADDR flag
- Wait until BTF = 1 (Data 1 in DR, Data2 in shift register, SCL stretched low until a data 1 is read)
- Set STOP high
- Read data 1 and 2

For N >2 -byte reception, from N-2 data reception

- Wait until BTF = 1 (data N-2 in DR, data N-1 in shift register, SCL stretched low until data N-2 is read)
- Set ACK low
- Read data N-2
- Wait until BTF = 1 (data N-1 in DR, data N in shift register, SCL stretched low until a data N-1 is read)
- Set STOP high
- Read data N-1 and N

We will again follow through with the procedure given in detail in the reference manual.

```
char *I2C1_MasterReceive(uint8_t *buffer,int size){
	uint32_t idx = 0;
	if(size == 2){
		I2C1_Address(0); //send address

		I2C1->CR1 &= ~I2C_CR1_ACK; //set ACK to low
		I2C1->CR1 |= I2C_CR1_POS; // set POS to high

		while(!(I2C1->SR1 & I2C_SR1_BTF)); //wait for BTF to set
		I2C1_Stop(); // generate stop

		//read the 2 bytes
		while(!(I2C1->SR1 & I2C_SR1_RXNE));
		buffer[idx++] = I2C1->DR;
		while(!(I2C1->SR1 & I2C_SR1_RXNE));
		buffer[idx++] = I2C1->DR;
	}
	else{
```

```
            I2C1_Address(0); //send address
    for(int i = 0;i < size-3;i++){
                while(!(I2C1->SR1 & I2C_SR1_RXNE));
                buffer[idx++] = I2C1->DR;
            }

            while(!(I2C1->SR1 & I2C_SR1_BTF)); //wait for the BTF to set
            I2C1->CR1 &= ~I2C_CR1_ACK; //disable ack

    //read the second to last byte
    while(!(I2C1->SR1 & I2C_SR1_RXNE));
    buffer[idx++] = I2C1->DR;


            while(!(I2C1->SR1 & I2C_SR1_BTF));
            I2C1_Stop();

            //read the last two bytes.
            while(!(I2C1->SR1 & I2C_SR1_RXNE));
            buffer[idx++] = I2C1->DR;
            while(!(I2C1->SR1 & I2C_SR1_RXNE));
            buffer[idx++] = I2C1->DR;
      }
}
```

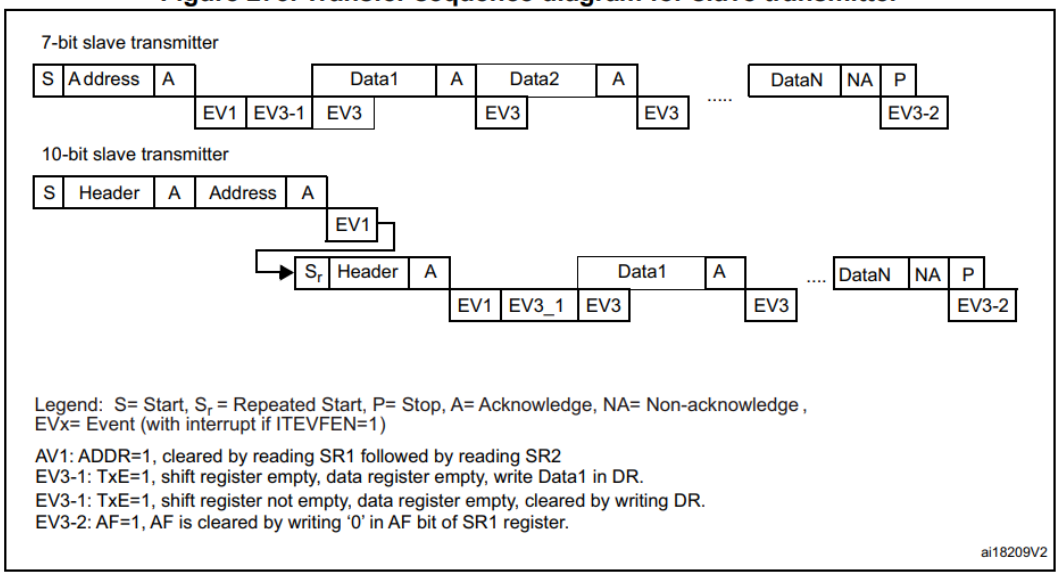
Steps:

- If size is 2 we only have to receive two bytes of data. For this,
    - First, send the address. Use I2C1_Address() function.
    - Disable ACK bit in I2C1->CR1
    - Disable POS bit in I2C1->POS
    - Wait for the BTF bit to set
    - Generate a STOP. Use I2C1_Stop() function
    - Read the two bytes of data.
- If size > 2,[assume N = number of bytes]
    - Send address
    - Read all the bytes except the last 3 bytes. [from 1 to N-3]
    - Wait until the BTF bit is set
    - Set ACK bit to low.
    - read the (N-2)th data
    - Generate a stop.
    - Read the last two bytes of data. [N-1,N]

Slave Transmit

Disclaimer: The code for slave transmit has not been tested. Probably won't work :3. Use at your own risk.

Figure 273. Transfer sequence diagram for slave transmitter



1. The EV1 and EV3_1 events stretch SCL low until the end of the corresponding software sequence.
2. The EV3 event stretches SCL low if the software sequence is not completed before the end of the next byte transmission

```
void I2C1_SlaveTransmit(char *buffer, int size){
      uint32_t idx = 0;
      while(!(I2C1->SR1 & I2C_SR1_ADDR)); /*wait for address to set*/
      idx = I2C1->SR1 | I2C1->SR2; //clear the ADDR flag
      for(int i = 0; i < size;i++){
             while(!(I2C1->SR1 & I2C_SR1_TXE));
            I2C1->DR = buffer[i];
      }

      while(!(I2C1->SR1 & I2C_SR1_STOPF)); /*wait for STOP bit to set*/
    /*clear stop flag*/
    idx = I2C1->SR1;
    I2C1->CR1 |= I2C_CR1_PE;

   //if AF (acknowledgment failure) bit is set, clear it
   I2C1->AF &= ~I2C_SR1_AF;

}
```

Steps:

- Wait for the address bit to set and then clear it as we did in the I2C1_Adress() function.
- Send the data one character at a time and check if the TXE bit is enabled each time.
- Finally, wait for the STOP flag to set and Clear it.

- If no acknowledgment is sent the AF flag is set in the I2C1->SR1 register. Simply set it to 0.

SPI

* Lekhar tel shesh hoye gese.

Here are some quick pointers to look at.

1. **Configuration. Follow through with the code + 26.3.7 of the reference manual**
2. **26.3.6 from Reference Manual [CPOL, CPHA]**
3. **Any register that has been used for configuration or data transmission**

25th MCU

6(a)

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

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

Bits 15:4 **DIV_Mantissa[11:0]**: mantissa of USARTDIV
These 12 bits define the mantissa of the USART Divider (USARTDIV)

Bits 3:0 **DIV_Fraction[3:0]**: fraction of USARTDIV
These 4 bits define the fraction of the USART Divider (USARTDIV). When OVER8=1, the DIV_Fraction3 bit is not considered and must be kept cleared.

We need to set some values to DIV_Mantissa and DIV_Fraction so that the Baud rate is 5 MBps or 5*10^6 bits/sec. [I assumed Sir meant 5 mega-bits p/s. naile answer sundor ashe na]
We will use the formula,

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

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

Here, Tx/Rx baud = 5*10^6 bits/sec
Consider OVER8 = 0 [We will set the OVER8 bit(15th bit) in the USART->CR1 register to 0]

Fck = 90mhz. [USART1, USART6 is connected to the APB2 bus with a frequency of 90mhz]

Using Equation 1, USARTDIV = 1.125
=> DIV_Mantissa = 1 or 000000000001
=>DIV_Fraction = 0.125 * 16 = 2 or 0010

6(b)

Code for Configuring USART6 with BAUD rate of 5 Mbps (Calculated in 6(a))

```
void USART6_Config(void){

	//1. Enable UART clock and GPIO clock
	RCC->APB2ENR |= (1 << 5);
	RCC->AHB1ENR |= (1 << 2); //enable GPIOA clock


	//2. Configure UART pin for Alternate function
	GPIOC->MODER |= (2<<12);  //PC6 -> Tx
	GPIOC->MODER |= (2<<14);  //PC7 -> Rx

	GPIOA->OSPEEDR |= (GPIO_SPEED_FREQ_HIGH<<0) |
	                  (GPIO_SPEED_FREQ_HIGH<<2);

	GPIOA->AFR[0] |= (8<<24); // bits(24,25,26,27) = (1000)
	GPIOA->AFR[0] |= (8<<28); // bits(28,29,30,31) = (1000)

	//3. Enable UART on USART6_CR1 rgister
	USART6->CR1 = 0x00; //clear USART

	USART6->CR1 |= (1<<13);  // UE-bit enable USART

	//4. Program M bit in USART CR1 to define the word length
	USART6->CR1 &= ~(1U<<12); // set M bit  = 0 for 8-bit word length

	//5. Select the baud rate using the USART_BRR register.
	USART6->BRR |= (2<<0) | (1<<4); //5 * 10^6 (calculated from problem 6(a)

	//  6. Enable transmission TE and reception bits in the USART_CR1 register
	USART6->CR1 |= (1<<2); // enable RE for receiver
	USART6->CR1 |= (1<<3); //enable TE for transmitter

}
```

6(c)

In order to enable Interrupts for data reception in USART we need to enable the RXNEIE bit in the USART_CR1 register and enable the interrupt using the NVIC_EnableIRQ(...) function.

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

Bit 5 **RXNEIE**: RXNE interrupt enable

This bit is set and cleared by software.

0: Interrupt is inhibited

1: An USART interrupt is generated whenever ORE=1 or RXNE=1 in the USART_SR register

```
USART6->CR1 |= (1 << 5);
NVIC_EnableIRQ(USART6_IRQn);
```

We can define the interrupt service routine as such,

```
void USART2_IRQHandler(void){
      //Handle Interrupt
}
```

The interrupt service routine will be invoked if,

- RXNE bit in the USART_SR register is enabled. This bit is set whenever the content of the RDR shift register is transferred to the USART_DR or the data register and data is ready to be read.
- ORE bit in the USART_SR register is enabled. [details below]

Bit 5 **RXNE**: Read data register not empty

This bit is set by hardware when the content of the RDR shift register has been transferred to the USART_DR register. An interrupt is generated if RXNEIE=1 in the USART_CR1 register. It is cleared by a read to the USART_DR register. The RXNE flag can also be cleared by writing a zero to it. This clearing sequence is recommended only for multibuffer communication.

0: Data is not received

- 1: Received data is ready to be read.

Bit 3 **ORE**: Overrun error

This bit is set by hardware when the word currently being received in the shift register is ready to be transferred into the RDR register while RXNE=1. An interrupt is generated if RXNEIE=1 in the USART_CR1 register. It is cleared by a software sequence (an read to the USART_SR register followed by a read to the USART_DR register).

0: No Overrun error

1: Overrun error is detected

Note: When this bit is set, the RDR register content will not be lost but the shift register will be overwritten. An interrupt is generated on ORE flag in case of Multi Buffer communication if the EIE bit is set.

7(a)

For configuring both the (i) unique slave address and (ii)Broadcast address we can use the OAR1 and OAR2 register.

24.6.3 I²C own address register 1 (I2C_OAR1)

Address offset: 0x08
Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD MODE	Res.	Res.	Res.	Res.	Res.	ADD[9:8]		ADD[7:1]							ADD0
rw						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 15 **ADDMODE** Addressing mode (slave mode)
0: 7-bit slave address (10-bit address not acknowledged)
1: 10-bit slave address (7-bit address not acknowledged)

Bit 14 Should always be kept at 1 by software.

Bits 13:10 Reserved, must be kept at reset value

Bits 9:8 **ADD[9:8]**: Interface address
7-bit addressing mode: don't care
10-bit addressing mode: bits9:8 of address

Bits 7:1 **ADD[7:1]**: Interface address
bits 7:1 of address

Bit 0 **ADD0**: Interface address
7-bit addressing mode: don't care
10-bit addressing mode: bit 0 of address

24.6.4 I²C own address register 2 (I2C_OAR2)

Address offset: 0x0C
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.	ADD2[7:1]							EN DUAL
								rw	rw	rw	rw	rw	rw	rw	rw

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

Bits 7:1 **ADD2[7:1]**: Interface address
bits 7:1 of address in dual addressing mode

Bit 0 **ENDUAL**: Dual addressing mode enable
0: Only OAR1 is recognized in 7-bit addressing mode
1: Both OAR1 and OAR2 are recognized in 7-bit addressing mode

To configure the broadcast address in the OAR2 register using 10-bit addressing since the broadcast address is 0xFF (8 bits).

```
I2C1->OAR1 |= (1 << 0); // set 10-bit addressing mode
I2C1->OAR1 |= (255 << 1); // (0xFF)hex = (255)d
```

To configure the unique slave address we will use the OAR1 register and enable EN_DUAL to enable dual addressing mode.
Let, the ‘uint8_t address’ variable defines the address we want to set in the OAR2 register,

```
I2C1->OAR2 |= (1 << 0);
I2C1->OAR2 |= (address << 1);
```

[You might wanna look up how to read and write operations are done in I2C 7/8/10 bit slave address. Link: I2C Slave Addressing - Total Phase]

In order to write the data 0x4C in the address 0x5D we need to send the 7-bit address ‘0x5D (0b1011101) + write(0)’ creating an 8-bit packet or a byte that we will send through the DR register.

Subsequently, we will send the data 0x4C using the DR register to write to the specified register address.

```
uint8_t address = (93 << 1); //0x5D = (93)decimal
I2C1->DR = address;
while (!(I2C1->SR1 & (1<<1))); // wait for ADDR bit to set
while (!(I2C1->SR1 & (1<<7)));  // wait for TXE bit to set
I2C1->DR = 0x4C;  // send data
```

7(b)

We can differentiate between a STOP, START and DATA signal using the Status Register. (I2C_SR1)

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

- The bit 0 (SB) in this register indicates a start bit generation.
  In master mode
  - SB:0 -> No start condition

    - SB:1 -> Start Condition Generated
- The bit 4 (STOPF) is used for STOP detection.
  In slave mode
  STOPF:0 -> No stop condition detected
  STOPF:1 -> Stop condition detected
- Bit 6 (RxNE) indicates that the data register (I2C_DR) is not empty
    - RxNE:0 -> Data register empty
    - RxNE: -> Data register not empty

[Disclaimer: me dumb, this could be wrong]
In order to read data from addresses 0xC0, 0xD0, 0xA1 with the slave address 0xA0 we will need 4 start signals and 1 stop signals.
Steps:

- Generate START
- Send Slave Address 0xA0
- Repeated START
- Send address '0xC0' with 1 at the end for read operation [0xC0 + read(1)]
- Receive data
- Repeated START
- Send address '0xD0' + read(1)
- Receive data
- Repeated START
- Send address '0xA1' + read(1)
- Receive data
- Generate STOP

So we will need 4 START and 1 STOP signals

7(c)

Clock stretching can be disabled for Slave mode using the NOSTRECH bit in the I2C_CR1 register.

Bit 7 **NOSTRETCH**: Clock stretching disable (Slave mode)
This bit is used to disable clock stretching in slave mode when ADDR or BTF flag is set, until it is reset by software.
0: Clock stretching enabled
1: Clock stretching disabled

if it's enabled, the Slave will stretch the clock signal,

If TxE is set and some data were not written in the I2C_DR register before the end of the next data transmission, the BTF bit is set and the interface waits until BTF is cleared by a read to I2C_SR1 followed by a write to the I2C_DR register, stretching SCL low.

If RxNE is set and the data in the DR register is not read before the end of the next data reception, the BTF bit is set and the interface waits until BTF is cleared by a read from the I2C_DR register, stretching SCL low (see *Figure 274*).

What these two lines essentially means is, the slave will stretch the clock and force the master into a wait state if the slave needs more time to process the data. For example, if data is not read/written in DR register when Txe/RxNE is set and BTF is not cleared.
The slave will also stretch the SCL low when:

- when ADDR = 1 during data transmission. The clock is stretched low until ADDR is cleared by reading SR1 followed by SR2 register.

Similarly master can also stretch the clock low when,

If TxE is set and a data byte was not written in the DR register before the end of the last data transmission, BTF is set and the interface waits until BTF is cleared by a write to I2C_DR, stretching SCL low.

If the RxNE bit is set and the data in the DR register is not read before the end of the last data reception, the BTF bit is set by hardware and the interface waits until BTF is cleared by a read in the DR register, stretching SCL low.

The Master will also stretch the SCL low when

- SB = 1 during data transmission. The clock is stretched low until SB is cleared by reading the SR1 register and writing the DR register.
- ADDR = 1 during data transmission. The clock is stretched low until ADDR is cleared by reading SR1 followed by reading SR2.

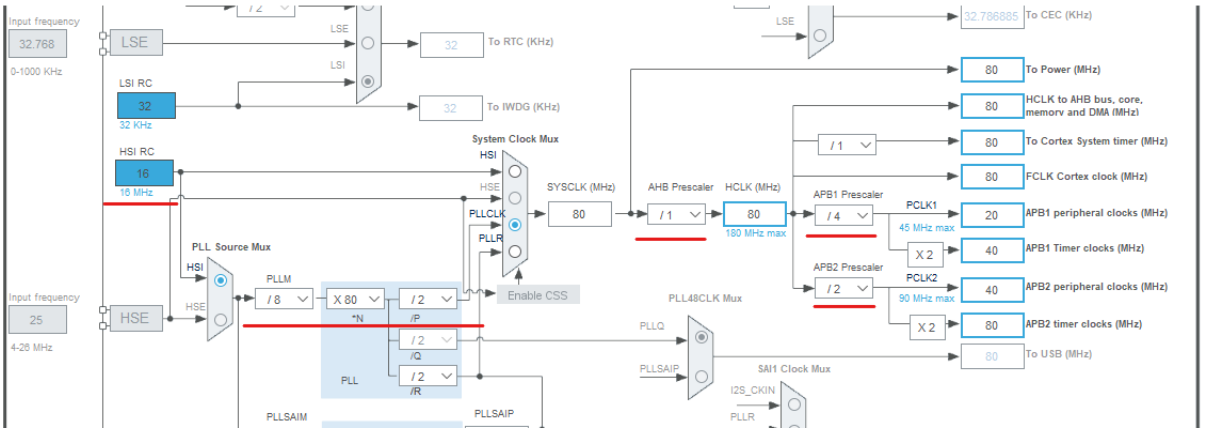
So for both cases, if STRETCHING is enabled(for slave mode) in I2C_CR1 register the clock will be stretched when:

- If clock stretching is enabled:
  - Transmitter mode: If TxE=1 and BTF=1: the interface holds the clock line low before transmission to wait for the microcontroller to write the byte in the Data Register (both buffer and shift register are empty).
  - Receiver mode: If RxNE=1 and BTF=1: the interface holds the clock line low after reception to wait for the microcontroller to read the byte in the Data Register (both buffer and shift register are full).

Part2 of the question: “Write down the steps to …….” - Sure na ami

Yes, The acknowledgment bit plays a vital role in ensuring the correct transmission of data and preventing data loss.
I2C protocol specifies that an ACK(0) or NACK(1) must be sent by the receiver after every byte sent by the sender.

5(a)



In order to set the AHB1 clock speed at 80Mhz, APB1 at 20Mhz, and APB2 at 40 MHz, we can follow the combination below,

PLLM = 8

PLLN = 80

PLLP = 2

AHB Prescaler = 1

APB1 Prescaler = 4

APB2 Prescaler = 2

Explanation: We need to generate clock pulse to run the everything on the MCU. This clock pulse can either be generated by the HSI (High-speed internal Crystal) with a frequency of 16Mhz or we can use the HSE (external crystal) which has a clock frequency of 8Mhz for stm32F446re.

Now we can multiply or divide this 16 or 8Mhz frequency at various points marked red on the figure above since not all peripherals need the same clock frequency. We can first divide the HSI/HSE frequency using PLLM, then multiply by PLLN (shown as *N in the figure) and then again divide with PLLP (shown as P in the figure).

Sequence	Prescaler	Possible values
1	PLLM. PLLN. PLLP	$2 \leq PLLM \leq 63.\ 50 \leq PLLN \leq 432.\ PLLP \in [2, 4, 6, or 8]$
2	System Clock Mask	HSI. HSE. PLLCLK or PLLR
3	AHB Prescaler (HPRE)	$HPRE \in [2, 4, 8, 16, \cdots . 512]$
4	APB1 Prescaler	$PPRE1 \in [2, 4, 8, 16]$
4	APB2 Prescaler	$PPRE2 \in [2, 4, 8, 16]$

This is the range of valid values that we can use for any of the prescalers. Next, we can divide the current output frequency using the AHB prescaler(3rd red underline). This is the frequency for the AHBx bus. Next, we can further divide this frequency using APB1 and APB2 Prescalers (last 2 underlines on the left.) The output frequency of these will be supplied to the APB1 and APB2 buses.

*** The frequency supplied to the Timer peripherals is usually multiplied

by two.

5(b)

eitar answer ektu amr aukat er baire. kalker vitor answer korar joggota hoile boshay dibo

5(c)

We will be using PA5 and PA6 in input mode for the switches. PB1, PB2, PB3, and PB4 in output mode for the LEDs.

Since we are going to enable GPIOA and GPIOB for this specific purpose the value of RCC_AHB1ENR will be,

0b00000000000000000000000000000011

6.3.10 RCC AHB1 peripheral clock enable register (RCC_AHB1ENR)

Address offset: 0x30

Reset value: 0x0000 0000

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	OTGHS ULPIEN	OTGHS EN	Res.	Res.	Res.	Res.	Res.	Res.	DMA2 EN	DMA1 EN	Res.	Res.	BKP SRAMEN	Res.	Res.
	rw	rw							rw	rw			rw		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	CRC EN	Res.	Res.	Res.	Res.	GPIOH EN	GPIOG EN	GPIOF EN	GPIOE EN	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
			rw					rw	rw	rw	rw	rw	rw	rw	rw

We will set PA5 and PA6 in input mode. Value of GPIOA_MODER,
[Set 00 for input mode]

0b00000000000000000000000000000000

We will set PB1, PB2, PB3, and PB4 in output mode. Value of GPIOB_MODER,
[set 01 for output mode]

0b00000000000000000000000001010101

7.4.1 GPIO port mode register (GPIOx_MODER) (x = A..H)

Address offset: 0x00

Reset values:

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

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

We will be using push-pull mode. So the GPIOx_OTYPER for both GPIOA and GPIOB will be,
[we only use bit 0-15 for this. Bit 16-31 is reserved]
0bXXXXXXXXXXXXXXXX0000000000000000

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

We will be using high speed for all of the pins. Thus we will be setting '11' to each of the pins in GPIOA and GPIOB that we are using.
Value of GPIOA_OSPEEDR,
0b00000000000000000011110000000000
Value of GPIOB_OSPEEDR,
0b00000000000000000000000011111111

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.

We will now write the code snippet for controlling the LEDs for each of the inputs given below. We will assume the GPIO pins for each of these are already configured.

Input (PA5 PA6)	LED Status (PB1 PB2 PB3 PB4)
00	on off off off
01	off on off off
10	off off on off
11	off off off on

```
//Input: 00 (PA5,PA6)
if(!(GPIOA->IDR & (1 << 5)) && !(GPIOA->IDR & (1 << 6)) ){
  //on off off off
  GPIOB->BSSR |= (1 << 1);
  GPIOB->BSSR |= (1 << (2 + 16));
  GPIOB->BSSR |= (1 << (3 + 16));
  GPIOB->BSSR |= (1 << (4 + 16));
}

//Input: 01 (PA5,PA6)
if(!(GPIOA->IDR & (1 << 5)) && (GPIOA->IDR & (1 << 6)) ){
  //off on off on
  GPIOB->BSSR |= (1 << (1 + 16));
  GPIOB->BSSR |= (1 << 2);
  GPIOB->BSSR |= (1 << (3 + 16));
  GPIOB->BSSR |= (1 << (4 + 16));
}

//Input: 10 (PA5,PA6)
if((GPIOA->IDR & (1 << 5)) && !(GPIOA->IDR & (1 << 6)) ){
  //off off on off
  GPIOB->BSSR |= (1 << (1 + 16));
  GPIOB->BSSR |= (1 << (2 + 16));
  GPIOB->BSSR |= (1 << 3);
```

```
  GPIOB->BSSR |= (1 << (4 + 16));
}
//Input: 11 (PA5,PA6)
if((GPIOA->IDR & (1 << 5)) && (GPIOA->IDR & (1 << 6)) ){
  //off off off on
  GPIOB->BSSR |= (1 << (1 + 16));
  GPIOB->BSSR |= (1 << (2 + 16));
  GPIOB->BSSR |= (1 << (3 + 16));
  GPIOB->BSSR |= (1 << 4);
}
```