I2C

- ## Introduction to i2c communication

  I^2C (i-square-c) is an acronym for "Inter-Integrated-Circuit" which was originally created by Philips Semiconductors (now NXP) back in 1982. I^2C^{TM} is a registered trademark for its respective owner and maybe it was the reason they call it "Two Wire Interface (TWI)" in some microcontrollers like Atmel AVR. The I^2C is a multi-master, multi-slave, synchronous, bidirectional, half-duplex serial communication bus. It's widely used for attaching lower-speed peripheral ICs to processors and microcontrollers in short-distance, intra-board communication.

- ## I2C Modes & Bus Speeds

  Originally, the I2C-bus was limited to 100 kbit/s operations. Over time there have been several additions to the specification so that there are now five operating speed categories. Standard-mode, Fast-mode (Fm), Fast-mode Plus (Fm+), and High-speed mode (Hs-mode) devices are downward-compatible. This means any device may be operated at a lower bus speed. Ultra Fast-mode devices are not compatible with previous versions since the bus is unidirectional.

  Bidirectional bus:

  - **Standard-Mode (Sm)**, with a bit rate up to 100 kbit/s
  - **Fast-Mode (Fm)**, with a bit rate up to 400 kbit/s
  - **Fast-Mode Plus (Fm+)**, with a bit rate up to 1 Mbit/s
  - **High-speed Mode (Hs-mode)**, with a bit rate up to 3.4 Mbit/s.

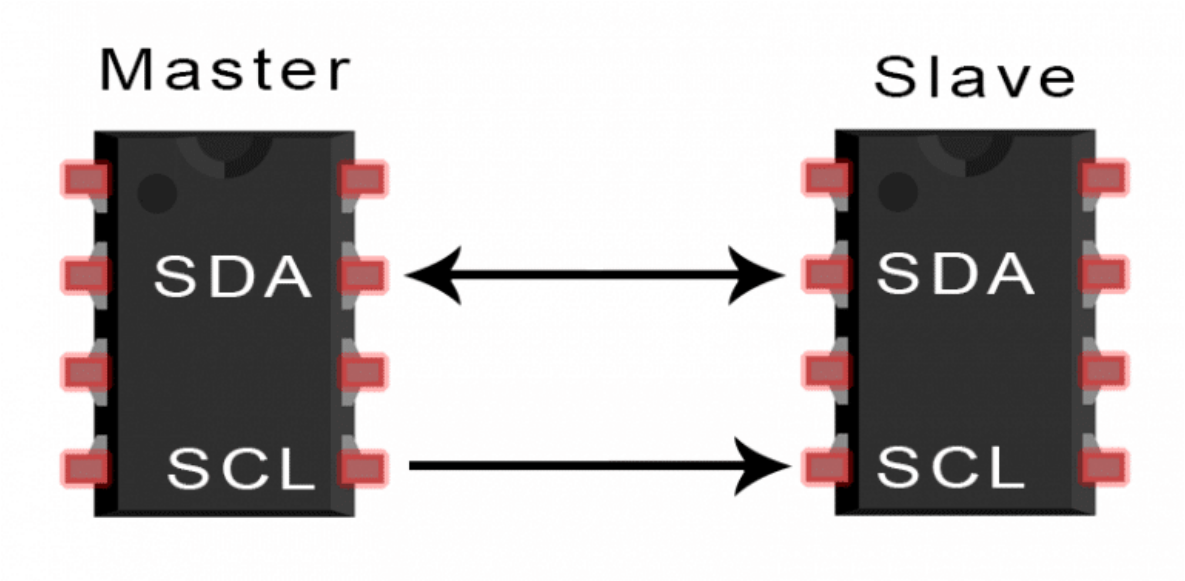
  Unidirectional bus:

  - **Ultra Fast-Mode (UFm)**, with a bit rate up to 5 Mbit/s

  Both SDA and SCL are bidirectional lines, connected to a positive supply voltage via a current-source or pull-up resistor. When the bus is free, both lines are HIGH. The output stages of devices connected to the bus must have an open-drain or open-collector to perform the wired-AND function.

  Due to the variety of different technology devices (CMOS, NMOS, bipolar), that can be connected to the I^2C-bus, the levels of the logical '0' (LOW) and '1' (HIGH) are not fixed and depend on the associated level of V_{DD}. Input reference levels are set as 30 % and 70 % of V_{DD}; V_{IL} is $0.3V_{DD}$ and V_{IH} is $0.7V_{DD}$. The data on the SDA line must be stable during the HIGH period of the clock. The HIGH or LOW state of the data line can only change when the clock signal on the SCL line is LOW. One clock pulse is generated for each data bit transferred

.

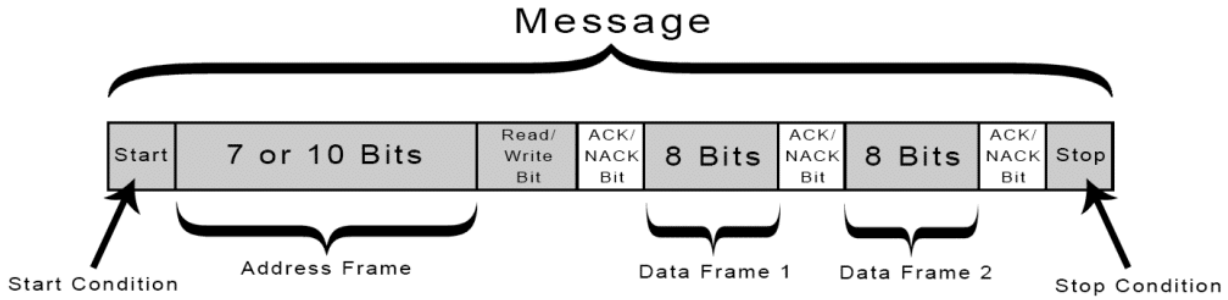


SDA (Serial Data) – The line for the master and slave to send and receive data.

SCL (Serial Clock) – The line that carries the clock signal.

- ## I2C working abstract

With I2C, data is transferrd in *messages*, Messages are broken up into *frames* of data.
Each message has an address frame that contains the binary address of the slave, and one or more data frames that contain the data being transmitted. The message also includes start and stop conditions, read/write bits, and ACK/NACK bits between each data frame



Start Condition: The SDA line switches from a high voltage level to a low voltage level *before* the SCL line switches from high to low.

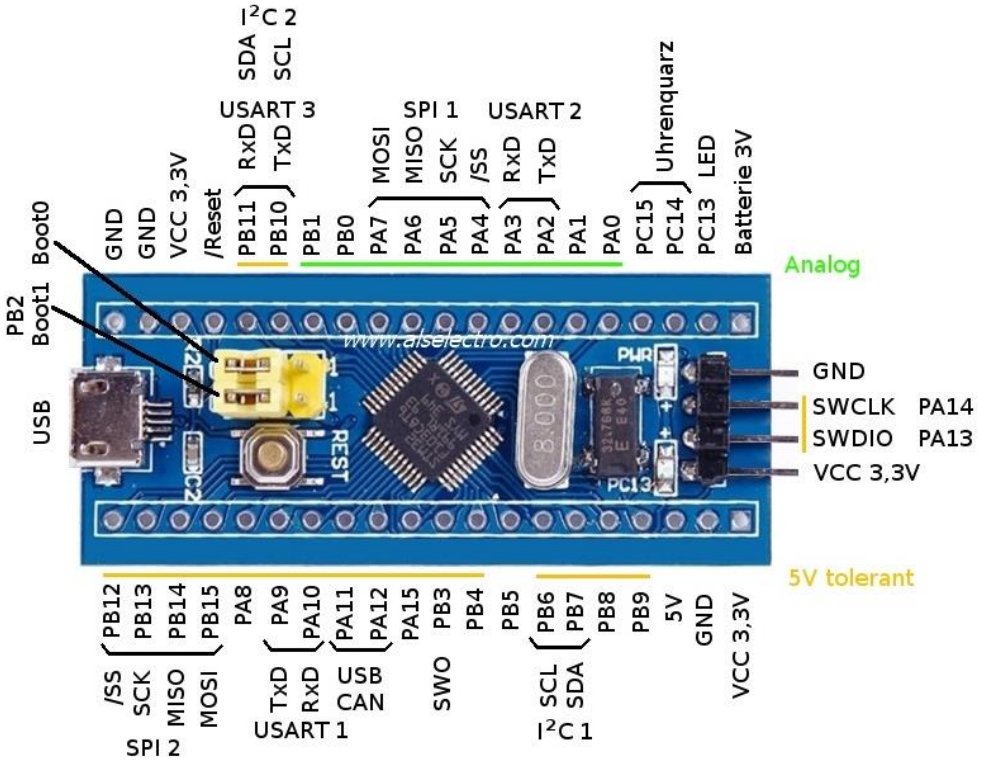
Stop Condition: The SDA line switches from a low voltage level to a high voltage level *after* the SCL line switches from low to high.

Address Frame: A 7 or 10 bit sequence unique to each slave that identifies the slave when the master wants to talk to it.

Read/Write Bit: A single bit specifying whether the master is sending data to the slave (low voltage level) or requesting data from it (high voltage level).

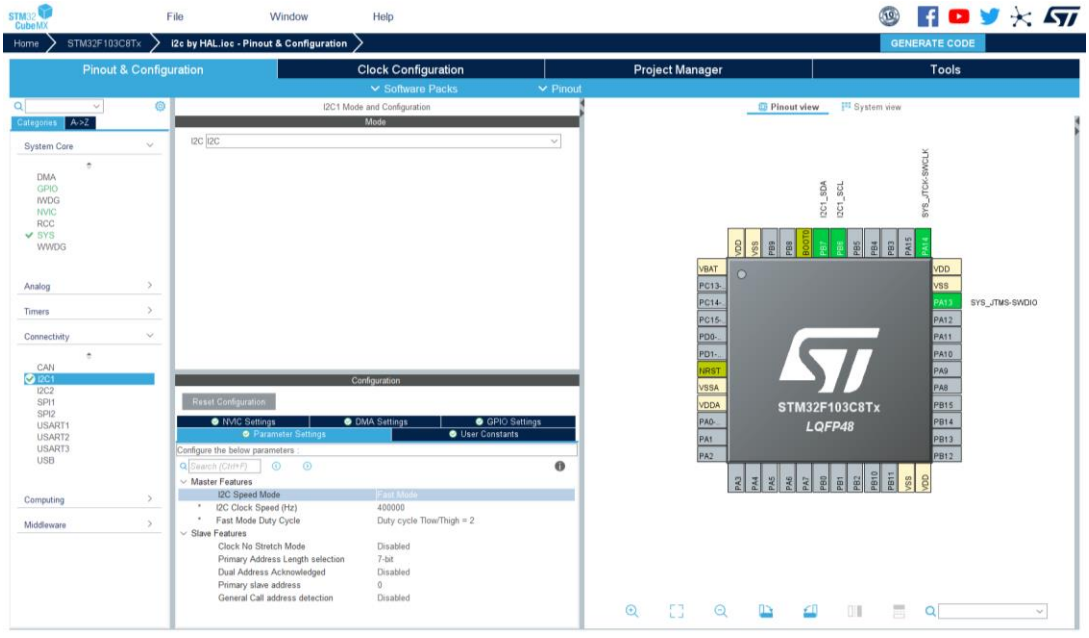
ACK/NACK Bit: Each frame in a message is followed by an acknowledge/no-acknowledge bit. If an address frame or data frame was successfully received, an ACK bit is returned to the sender from the receiving device.

I2C interface the stm32F103 (BP)



AS shown in the previous figure pins PB11 & PB10 represent the I2C connection in the blue pill (stm32f103) with the SDA & SCL respectively.

Using CUBEMX (HAL DRIVERS) with I2C



Like the UART, HAL gives 3 methods for I2C

I2C With Polling

The first and the easiest way to do anything in embedded software is just to poll for the hardware resource until it's ready to move on to the next step in your program instructions. However, it's the least efficient way to do things and the CPU will end up wasting so much time in a "busy-waiting" state.

It's the same thing for both transmission and reception. You just wait until the current byte of data to be transmitted so you can start the next one and so on.

I2C With Interrupts

We can, however, enable the I2C interrupts and have a signal when it's done and ready for servicing by CPU. Either for data that has been sent or received. Which saves a lot of time and has been always the best way to handle events like that.

However, in some "Time Critical" applications we need everything to be as deterministic, in time, as possible. And a major problem with interrupts is that we can't expect when it'd arrive or during which task. That can potentially screw up the timing behavior of the system.

I2C With DMA

To operate at its maximum speed, the I2C needs to be fed with the data for transmission and the data received on the Rx buffer should be read to avoid overrun. To facilitate the transfers, the I2C features a DMA capability implementing a simple request/acknowledge protocol.

We will go through the i2c with interrupts firstly .

I2C With Interrupts

Interrupt flags

I^2C Interrupt requests

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	
Receive buffer not empty	RxNE	ITEVFEN and ITBUFEN
Transmit buffer empty	TxE	

Generating HAL driver code

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2022 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "fonts.h"
#include "ssd1306.h"
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
 I2C_HandleTypeDef hi2c1;

/* USER CODE BEGIN PV */

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
```

```c
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_I2C1_Init(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_I2C1_Init();
  /* USER CODE BEGIN 2 */
  SSD1306_Init();
   char snum[5];

   SSD1306_GotoXY (0,0);
   SSD1306_Puts ("moaaz", &Font_11x18, 1);
   SSD1306_GotoXY (0, 30);
   SSD1306_Puts ("I2C TEST", &Font_11x18, 1);
   SSD1306_UpdateScreen();
   HAL_Delay (1000);

   SSD1306_ScrollRight(0,7);
   HAL_Delay(3000);
   SSD1306_ScrollLeft(0,7);
   HAL_Delay(3000);
   SSD1306_Stopscroll();
   SSD1306_Clear();
   SSD1306_GotoXY (35,0);
```

```
   SSD1306_Puts ("TEAM 7", &Font_11x18, 1);
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
   while (1)
  {
      for ( int x = 1; x <= 10000 ; x++ )
      {
            itoa(x, snum, 10);
            SSD1306_GotoXY (0, 30);
            SSD1306_Puts ("             ", &Font_16x26, 1);
            SSD1306_UpdateScreen();
            if(x < 10) {
                  SSD1306_GotoXY (53, 30);  // 1 DIGIT
            }
            else if (x < 100 ) {
                  SSD1306_GotoXY (45, 30);  // 2 DIGITS
            }
            else if (x < 1000 ) {
                  SSD1306_GotoXY (37, 30);  // 3 DIGITS
            }
            else {
                  SSD1306_GotoXY (30, 30);  // 4 DIGIS
            }
            SSD1306_Puts (snum, &Font_16x26, 1);
            SSD1306_UpdateScreen();
            HAL_Delay (500);
          }
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
```

```
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_0) != HAL_OK)
  {
    Error_Handler();
  }
}

/**
  * @brief I2C1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_I2C1_Init(void)
{

  /* USER CODE BEGIN I2C1_Init 0 */

  /* USER CODE END I2C1_Init 0 */

  /* USER CODE BEGIN I2C1_Init 1 */

  /* USER CODE END I2C1_Init 1 */
  hi2c1.Instance = I2C1;
  hi2c1.Init.ClockSpeed = 400000;
  hi2c1.Init.DutyCycle = I2C_DUTYCYCLE_2;
  hi2c1.Init.OwnAddress1 = 0;
  hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
  hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
  hi2c1.Init.OwnAddress2 = 0;
  hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
  hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
  if (HAL_I2C_Init(&hi2c1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN I2C1_Init 2 */

  /* USER CODE END I2C1_Init 2 */

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();
```

```
}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```