I2C communication on STM32 Microcontrollers using HAL

You can use the STM32CubeMX tool to create the necessary config. files to enable the I2C-Drivers.
The HAL library provides the necessary functions to communicate to with the I2C protocol.
I'm going to show you how to output I2C with the HAL library using a PCA-9685 16-channel 12-Bit LED driver.

Note ! For this tutorial, it is advised to have at least a basic understanding of the I2C protocol and the necessary components.

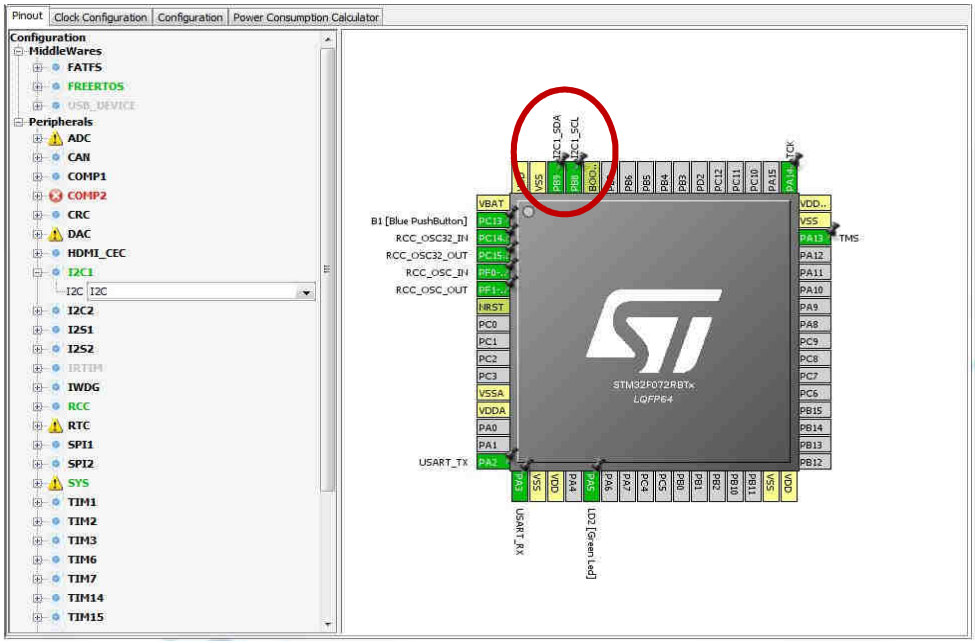
This tutorial uses the following equipment:

- NUCLEO-F072RB Board
- Keil uVision 5 with the necessary packages for Nucleo boards installed
- STLink USB Driver
- STM32CubeMX
- PCA 9865 chip or module
- (Oscilloscope to analyse the signal)

STM32CubeMX

Generating the config. files from STM32CubeMX.

1. Open STM32CubeMX and open a new project.
2. Select the Nucleo-F072RB from the Borards tab
3. Enable FreeRTOS
4. Set the RCC (HSE & LSE) to Crystal/Ceramic Resonator
5. Enable the USART2 port in Asynchronous mode
6. Enable the I2C1 module and remap the SDA/SCL pins to PB8/PB9
7. Go to Project > Generate code
8. Enter a project name and select MDK-ARM V5
9. Generate the code and open the project in Keil uVision



Now let's see what the code generator did

```
174 /* I2C1 init function */
175 void MX_I2C1_Init(void)
176 {
177
178   hi2c1.Instance = I2C1;
179   hi2c1.Init.Timing = 0x2000090E;
180   hi2c1.Init.OwnAddress1 = 0;
181   hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
182   hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLED;
183   hi2c1.Init.OwnAddress2 = 0;
184   hi2c1.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
185   hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLED;
186   hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLED;
187   HAL_I2C_Init(&hi2c1);
188
189     /**Configure Analogue filter
190     */
191   HAL_I2CEx_AnalogFilter_Config(&hi2c1, I2C_ANALOGFILTER_ENABLED);
192
193 }
194
```

Nothing special here. Just note, that you have the option to change the addressing mode to `I2C_ADDRESSING_MODE_10BIT`.

I2C Basics

Note ! This info is essential when working with I2C protocol. Possible sources of error are marked with red blocks.

The I2C always needs to be terminated with a pull-up resistor on each SDA and SCL. Typical values can range from 4k7 to 22k.

error Bus not terminated. This is the case, when SDA and SCL are in a low state (GND) instead of pulled high.

The I2C protocol works as follows:

The bus is activated with a start condition. (SCL pulled low after SDA was pulled low)
Followed by the slave address.
Slave address:

- Bits 7 – 4 are a unique address, each family of chips has a unique combination
- Bits 3 – 1 determines the id of a chip. They are usually set with pull-up / pull-down resistors
- Bit 0 determines if a read or a write access is being performed on the slave

The address is followed by the first data byte.
The slave acknowledges the reception of the byte with an ACK bit.
After the first data byte and its acknowledgment, a series of any number of bytes can be transmitted.

error	The slave does not ACK the reception. This case occurs, if the address was incorrect. Check your datasheets to find the correct address.

The end of transmission is then marked with the stop condition. (SDA is pulled high after SCL was pulled high)

How to send data over I2C

Sending bytes of data using the function provided within the HAL library.

```
1  HAL_I2C_Master_Transmit(&hi2c1, 0x80, 0x0F, 1, 1);
```

The first argument is a pointer to the instance of an **I2C module**.
The second argument is the **slave address**. In this case, 0x80 is the default address of the pca 9685 family.
The third argument is the **byte to be sent**. 0x0f is easy to check on an oscilloscope.
The fourth argument is the **size** (**in bytes**) of the outputted data. In our case it is only 1 byte great.
The last argument is the timeout value. If sending only one byte, you should be good with a value of 1.

Note !	If more than one byte of data is being outputted, and the slave does not acknowledge the first byte, the timeout comes in handy. It prevents the function from unsuccessfully waiting for an acknowledgement (infinite loop).

You can also enter a pointer to an 8bit value like this.

```
1  uint8_t outbuffer = 0xFF;
2
3  HAL_I2C_Master_Transmit(&hi2c1, 0x80, &outbuffer, 1, 1);
4
```

```
1  uint8_t outbuffer[3] = {0xFF, 0xF0, 0x0F};
2
3  HAL_I2C_Master_Transmit(&hi2c1, 0x80, &outbuffer[0], 1, 1);
4
```

Sending multiple bytes at once

```
1  uint8_t outbuffer[5] = {0x01, 0x02, 0x04, 0x08, 0x10};
2
3  // send the first 3 bytes
4  HAL_I2C_Master_Transmit(&hi2c1, 0x80, outbuffer, 3, 1);
5
6  // send the entire array
7  HAL_I2C_Master_Transmit(&hi2c1, 0x80, outbuffer, sizeof(outbuffer), 1);
8
```

To send multiple bytes of data, you can create an array as a buffer and give it to the HAL function.
Also change the **size** argument to the number of bytes you want to read.

Example Code to control LEDs on the PCA9685

```c
void pca9685_init(I2C_HandleTypeDef *hi2c, uint8_t address)
{

  #define PCA9685_MODE1 0x00
  uint8_t initStruct[2];
  uint8_t prescale = 3;       // hardcoded
  HAL_I2C_Master_Transmit(hi2c, address, PCA9685_MODE1, 1, 1);
  uint8_t oldmode = 0;        // hardcoded
  // HAL_I2C_Master_Receive(hi2c, address, &oldmode, 1, 1);
  uint8_t newmode = ((oldmode & 0x7F) | 0x10);
  initStruct[0] = PCA9685_MODE1;
  initStruct[1] = newmode;
  HAL_I2C_Master_Transmit(hi2c, address, initStruct, 2, 1);
  initStruct[1] = prescale;
  HAL_I2C_Master_Transmit(hi2c, address, initStruct, 2, 1);
  initStruct[1] = oldmode;
  HAL_I2C_Master_Transmit(hi2c, address, initStruct, 2, 1);
  osDelay(5);
  initStruct[1] = (oldmode | 0xA1);
  HAL_I2C_Master_Transmit(hi2c, address, initStruct, 2, 1);

}

```

```c
void pca9685_pwm(I2C_HandleTypeDef *hi2c, uint8_t address, uint8_t num, uint16_t on, uint16_t off)
{

  uint8_t outputBuffer[5] = {0x06 + 4*num, on, (on >> 8), off, (off >> 8)};
  HAL_I2C_Master_Transmit(&hi2c1, address, outputBuffer, 5, 1);

}

```

Note ! These functions are taken and modified from the Adafruit-PWM-Servo-Driver-Library

Fading an LED on and off.
(Vcc -> resistor -> anode of LED -> cathode of LED -> pwm channel 0 of pca9685)

```c
void StartDefaultTask(void const * argument)
{
  /* USER CODE BEGIN 5 */
       pca9685_init(&hi2c1, 0x80);
       pca9685_pwm(&hi2c1, 0x80, 0, 0, 4095);
  /* Infinite loop */
  for(;;){
    for(int i=0; i<255; i++){
      pca9685_pwm(&hi2c1, 0x80, 0, 0, 4095-(16*i));
      osDelay(5);
    }
    for(int i=0; i<255; i++){
      pca9685_pwm(&hi2c1, 0x80, 0, 0, (16*i));
      osDelay(5);
    }
  }
  /* USER CODE END 5 */
}

```

```
 1  void StartDefaultTask(void const * argument)
 2  {
 3    /* USER CODE BEGIN 5 */
 4         pca9685_init(&hi2c1, 0x80);
 5         pca9685_pwm(&hi2c1, 0x80, 0, 0, 4095);
 6    /* Infinite loop */
 7    for(;;){
 8      for(int i=0; i<255; i++){
 9        pca9685_pwm(&hi2c1, 0x80, 0, 0, 4095-(16*i));
 1  void StartDefaultTask(void const * argument)
 2  {
 3
 4    /* USER CODE BEGIN 5 */
 5    for(;;)
 6    {
 7      debugPrint(&huart2,  "oi, mate!");       // print
 8      debugPrint(&huart2, "\r\n");            // manual new line
 9      debugPrintln(&huart2, "how are you?");  // print full line
10      osDelay(1000);
11    }
12    /* USER CODE END 5 */
13  }
14
```

Note:

You might also want to look at the following functions when working with I2C peripherals.

```
HAL_I2C_Mem_Read()
HAL_I2C_Mem_Write()
```

They provide a cleaner and more direct way to access the registers on a peripheral chip.

Document Created by Simon Burkhardt

This tutorial is very basic and might not show the best way to use the STM32 environment.
It still might help you get into the whole HAL philosophy of STM if you are coming from another platform.

Simon Burkhardt
electronical engineering
simonmartin.ch

History:

V1.0	tested the code, created this document
V1.0.1	added contact info and cc0 notice