Use of SPI Component on Microcontroller

Targeted competences: Use of SPI in order to retrieve sensor data

Hardware: STM32F7 Nucleo board

Framework: STM32CubeIDE v1.0.1 from STMircoelectronics

The aim of this document is to show how to use an SPI bus in order to read the data from a barometric pressure sensor.
(https://cdn.sparkfun.com/assets/f/0/f/3/9/MPL115A1.pdf)

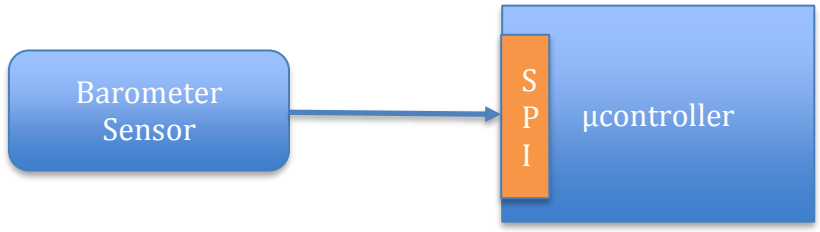


Figure 1: Acquisition Stream

In order to sample the read of the barometer data, we develop an algorithm to read and write the internal registers of the sensor.

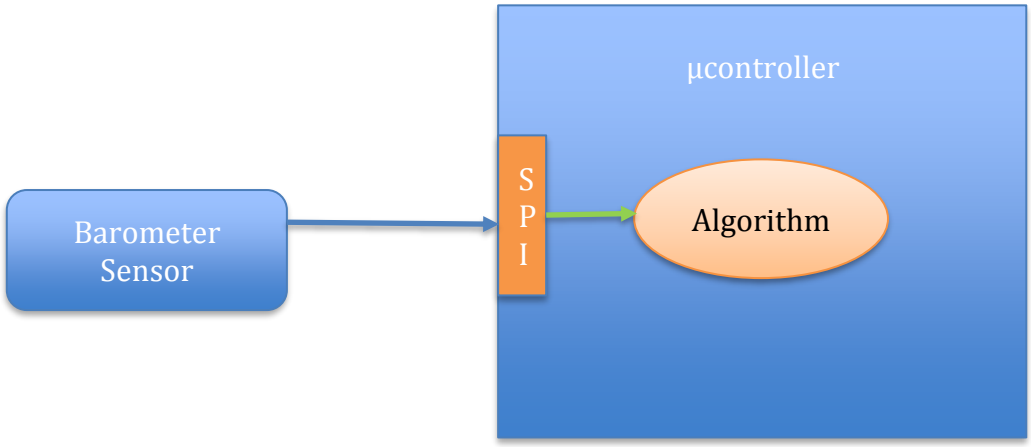


Figure 2: Global Scheme

1. Microcontroller configuration

The first step is to configure the microcontroller. In our case, we use the NUCLEO-F767ZI platform based on a STM32F7 architecture. In order to configure this board we will use the CubeMx software; this software is now integrated into the TrueStudio framework. The new framework called CubeIde. The first step after choosing the board is to name the project.

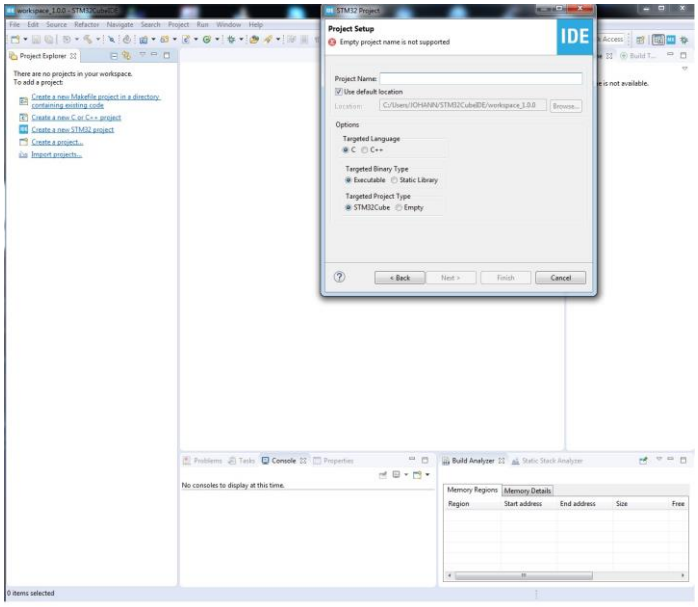


Figure 3: CubeIDE interface

After the choice of the board (attention the configuration can depend on the board used) you obtain the figure below.

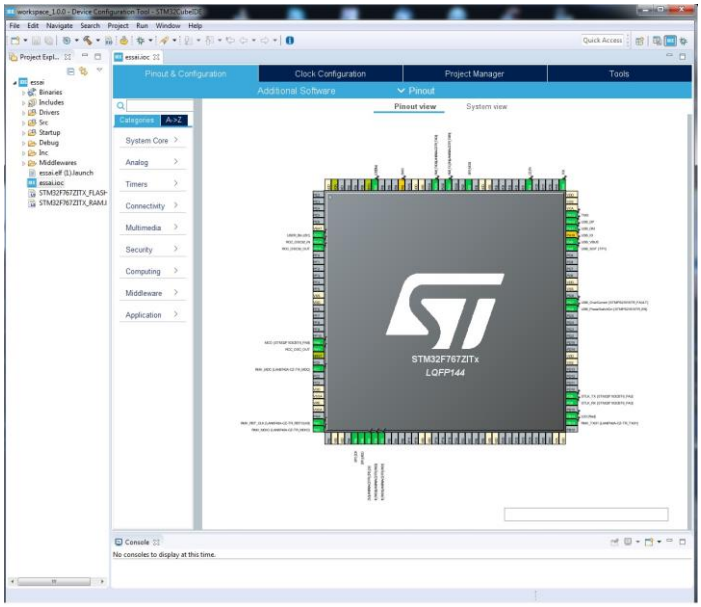


Figure 4: CubeMx interface to configure the STM32 Device

Here, I chose to use the SPI bus (here SPI1) and I configure the different parameters. The first one is the Frame format; here Motorola format, data size equal to 8 bits and first bit is equal to MSB first. For the clock parameters, I choose a prescaler of 64 in order to have a baud rate of 1.6875 MBits/s (this value depends of the clock frequency of the device; here the clock frequency is 216Mhz, the APB1 peripheral clock is 54MHz and the APB2 peripheral clock is 108MHz).

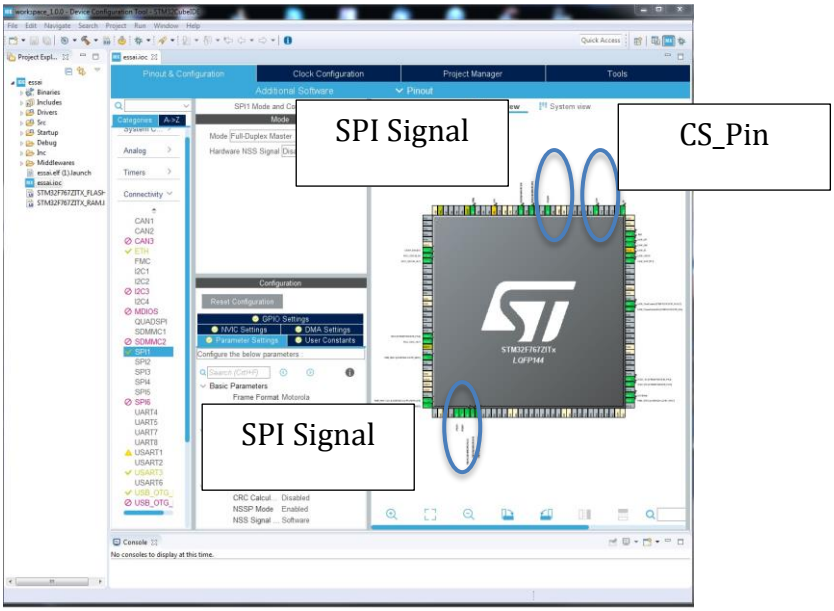


Figure 5: SPI signals and configuration

We must add a pin in order to generate a CS signal in order to enable the sensor. This pin (PD0) is a GPIO configured in GPIO output mode.

Now we have to configure the clock for the whole board; to do this click on clock configuration item. With this configuration panel, we can choose the frequency of different µcontroller components as CPU frequency, AHB, APB1 buses and so on. I decide to use the maximum frequency of the SYSCLK that is 216MHz. In this case, the frequency of the APB1 and APB2 timer clocks are respectively 108MHz and 216MHz (see Figure 5)..

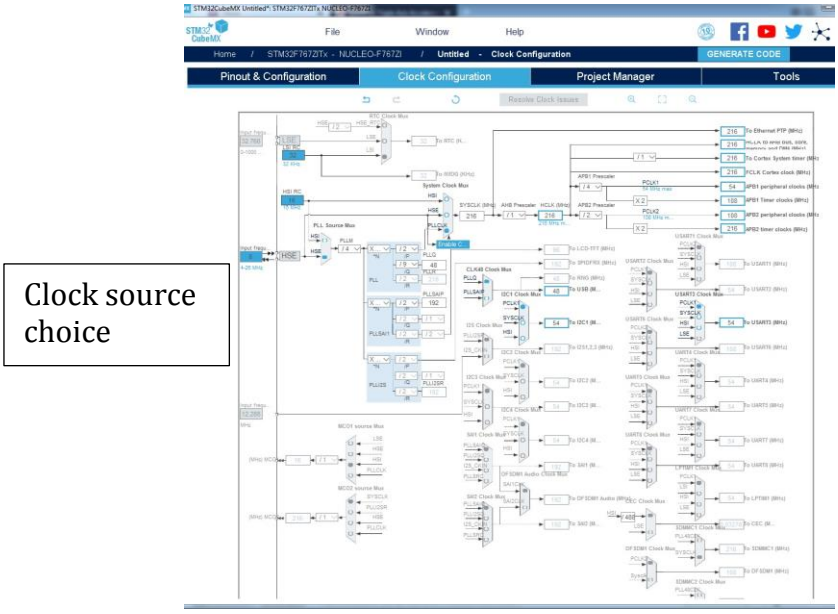


Figure 6: Clock configuration for the board

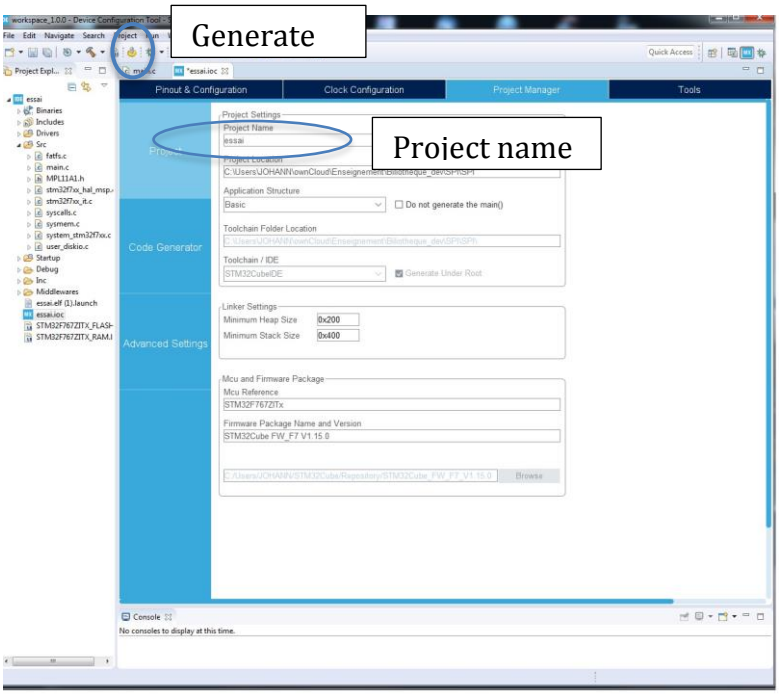


Figure 7: Project manager configuration

You must give a name for your project and modify if you need the stack and heap parameters. Be careful if you use dynamic memory allocation indeed you should try to estimate the maximum memory size you need in order to choose the best heap size. If the heap size is too small, you will have some bugs during the program execution.

After clicking on Generate Code, CubeMx generates the application code and creates the project as shown in the figure below.

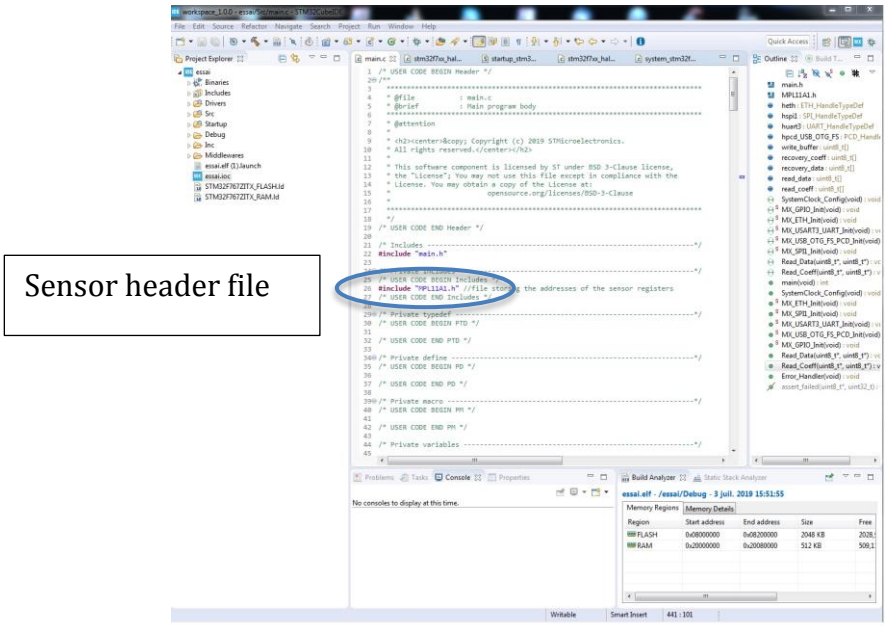


Figure 8: Code generation

As the sensor used has different register to store its calibration coefficients and its data (pressure & temperature), I have created a header file to define all the register addresses (MPL11A.h file).

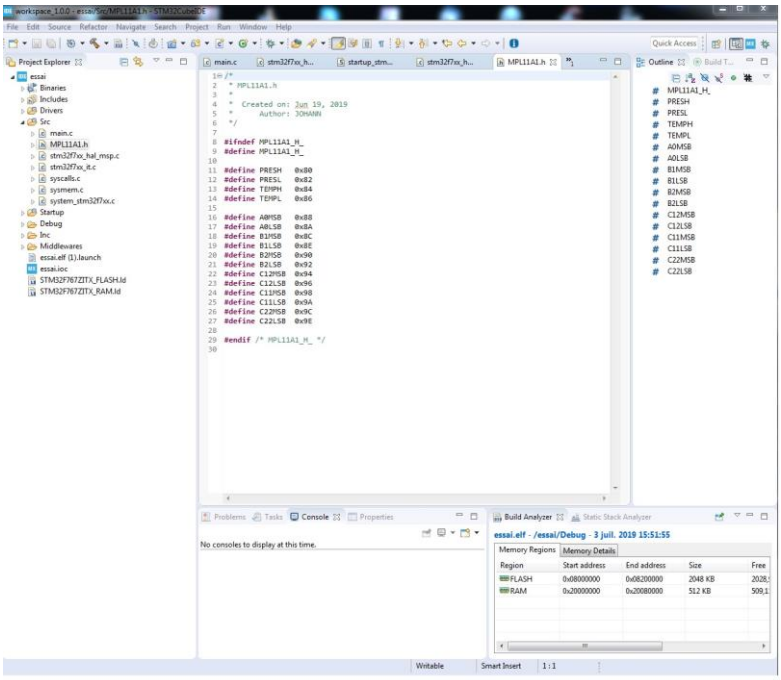


Figure 9: Sensor header file

In order to start the sensor, to read the calibration coefficients and to read the data (pressure & temp), we need to transmit some commands to the sensor. To do this, I created some data structures so I just have to transmit these data structures to realize the desired commands. I also created data structures to store the data and coefficients read (see the figure below).

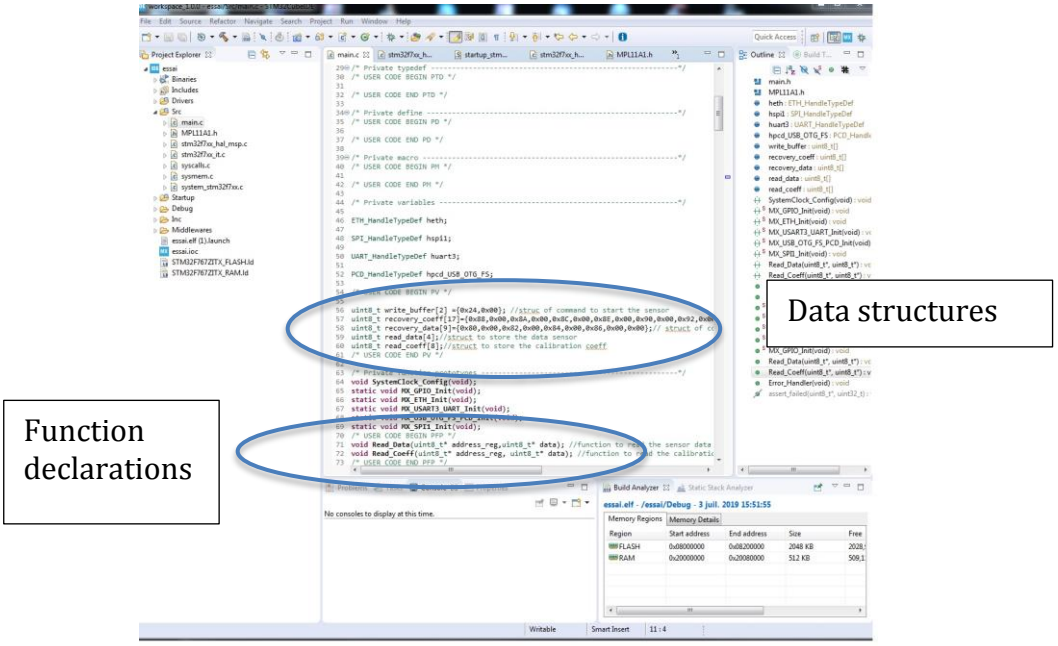


Figure 10: Data structure declarations to realize commands and store the data

Then I declare two functions, respectively Read_Data and Read_Coeff, in order to read the data from the sensor and the other one to read the calibration coefficients. These 2 functions have two parameters: the first one to send the command to do with the address_reg parameter; the second one to store the data come from the sensor with the data parameter.

After the initialization functions, I called the Read_Data function in order to retrieve the calibration coefficients. After that in the while(1) structure I called the Read_data function in order to retrieve the pressure and temperature data from the sensor.

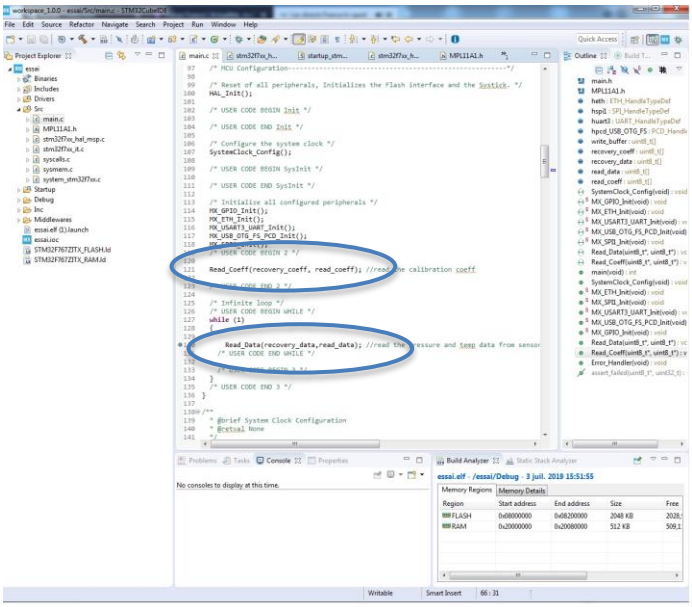


Figure 11: Read_Coeff and Read_Data function calls

In the "USER CODE BEGIN 4" section, I wrote the code of these two functions. In the Read_Data function, the first step is to enable the SPI communication by resetting the CS_Pin. Then, I used the SPI_Transmit function (provided in the HAL API) in order to send a command to the sensor to start it (see the datasheet of the sensor). The next step is to disable the SPI communication by setting the CS_Pin following by a delay of 3ms to allow the sensor to start correctly. After this delay the data are enable so we enable again the SPI communication and then I used the SPI_TransmitReceive function (HAL API) in order to read the data sensor. The SPI API functions work all in the same way: we need to specify the SPI port used, the pointer to the input data structure, the pointer to the output data structure (if we use the TransmitReceive function), the number of byte to read/transmit and finally a timeout value.

In the Read_Coeff function, I enabled the SPI communication then I used the SPI TransmitReceive function (HAL API) in order to read the calibration coefficients.

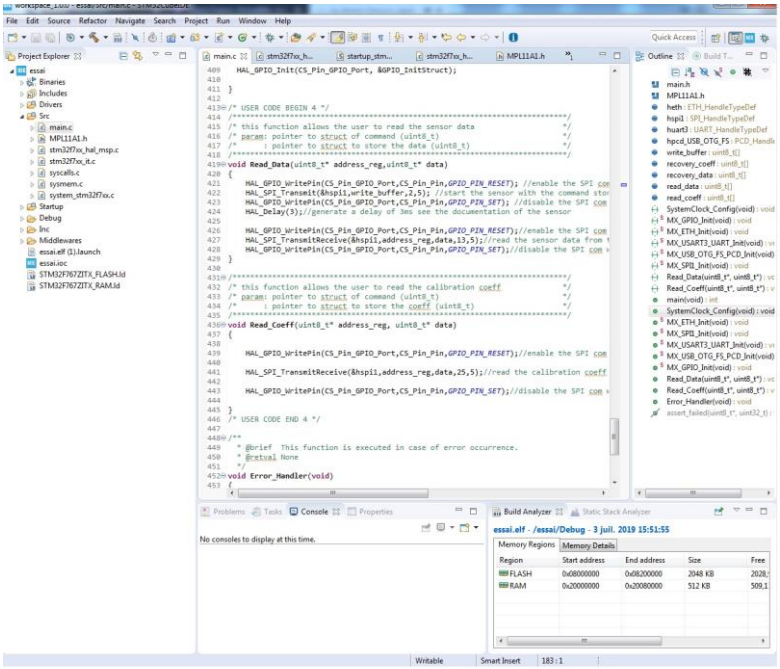
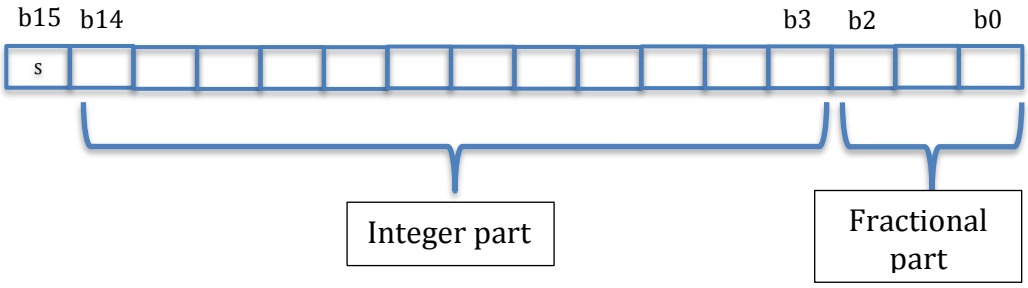


Figure 12: Read_Data and Read_Coeff function code

When you have retrieved the different calibration coefficients, you have to concatenate the two 8-bit value to obtain the coefficients. Below, I present how to concatenate the coefficient.



The bit b15 represents the sign bit, the value from b14 to b3 represents the integer part of the coefficient and finally the value from bit from b2 to b0 represents the fractional part of the coefficient. For example if A_{LSB} is equal to 0xDF and A_{MSB} is equal to 0x41 then the sign bit is equal to 0 (the value is positive), the integer part is equal to 2107 (or 0x83A) and the fractional part is equal to 7 (or 0x07) so the value is equal to 0.875. At the end the overall value of the A coefficient is equal to 2107,875.

In order to obtain this value, we have to code an algorithm. I present below an example of algorithm to determine the calibration coefficient A0. For the other one, we can proceed in the same way. Be careful if you want to generalize the algorithm below, you will have to redefine the type of the variables used thus the fractional and entire parts of the coefficients are not always the same.

```
short A0,temp, entire ;
char signe, fractional;
float fract[14]={0.5,……,0.00006103515625}; // pre computed fractional values
float temp0, CoeffA0;
A0=A0MSB <<8 ;
A0=A0 || A0LSB;
temp=A0;

Signe=temp >>15; //determine the sign of the value
temp=A0;
entire=temp<<1; //determine the entire part of the value
entire=entire>>4; //determine the entire part of the value
fractional=A0<<12 ;//determine the fractional part of the value
fractional=fractional>>12 ; // determine the fractional part of the value

if (fractional & 0x01 >0)
        temp0=fract[2];
if (fractional & 0x10 >0)
        temp0 = temp0 + fract[1];
if (fractional & 0x100 >0)
        temp0 = temp0 +fract[0];

CoeffA0 = entire + temp0
If (signe ==0)
        CoeffA0=CoeffA0 ;
Else
        CoeffA0=-CoeffA0 ;
```

At the end we have to compile the code in order to obtain the executable file for the STM32F7. When the compilation is done without error, we can transfer the code into the board.