University of Würzburg

Aerospace Project Work

PROJECT WORK REPORT

Made by:
Pedro Sousa
João Sousa

Supervisors:
Sergio Montenegro
Erik Dilger
Thomas Walter

Table of Contents

Introduction

This reported presents the work of two Portuguese Erasmus embedded systems students in which consists of the development of an SPI module, to provide to a RODOS developer an abstraction layer to communicate with a SiLabs EFR32 Mighty Gecko boards.
The report is structured in stages to follow the knowledge iteratively procedure that was adopted.

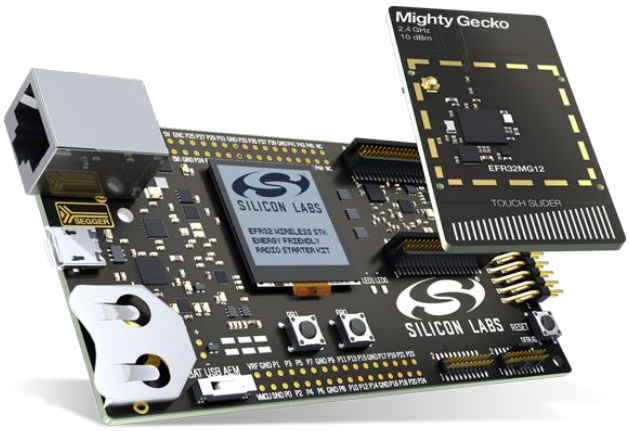


Figure 1- SiLabs EFR32 Mighty Gecko Wireless Starter Kit

Stage 1 - Rodos Introduction

At this stage, a superficial study of RODOS (Real time Onboard Dependable Operating System) provided an overview of how it was structured.
RODOS was implemented as a software framework in C++ with an object-oriented application program interface (API).
It is organized in 4 layers:

- HAL - Hardware abstraction layer.
- Kernel - Management the local resources, threads and time.
- Communication with building blocks.
- User application.

The project presented in this report acts on a service-oriented interface that provides services to the SPI hardware abstraction layer which allows developers to communicate with EFR32FG1P boards without worrying about detailed implementations.

Compile and Flash

In order to implement and test the first codes assembled, “10-first-steps” was followed. This document provides several information’s like:

- How to compile?
- How to design a basic structure of .cpp file, using RODOS API?
- How to use time functions?
- How to use Thread Communication?
- How to use semaphore?
- How to react to interrupts from timers and signals from devices?
- And other RODOS familiarization functions.

After several compile and flash procedures, was concluded that a lot of time was waste. To decrease the spending time between these procedures an execution of a script was provided. The created script is named “CandF.sh”.

```
#!/bin/bash
cd ../../
source setenvs.sh
cd ~/git-rodos/rodos-geckoport2/<DIR_WHERE_CPP>
rodos-executable.sh gecko1M <CPP_FILE_NAME>.cpp
arm-none-eabi-objcopy -O binary tst tst.bin
openocd -f board/efm32.cfg -c " init;halt;flash write_image erase tst.bin 0 bin;reset run; shutdown"
```

Figure 2- Script File

In the beginning, some shell variables must be defined, as they are necessary for the build scripts. The next step is to compile RODOS if it wasn´t compiled before or whether some alterations were done on API source files.

With RODOS prepared, the application shall be compiled with that specific format and name. To conclude, for flashing a specific board, the last command mentioned on the script should be executed.

Stage 2 - Assembly of PCB board

In this stage of project work, three PCB boards were assembled following a specific schematic. This procedure showed the disposition/constitution of each device on the boards. The assembly of these boards was important to achieve the main goal of this project because without doing that, the tests of SPI module communications between several devices on the board couldn’t be done.

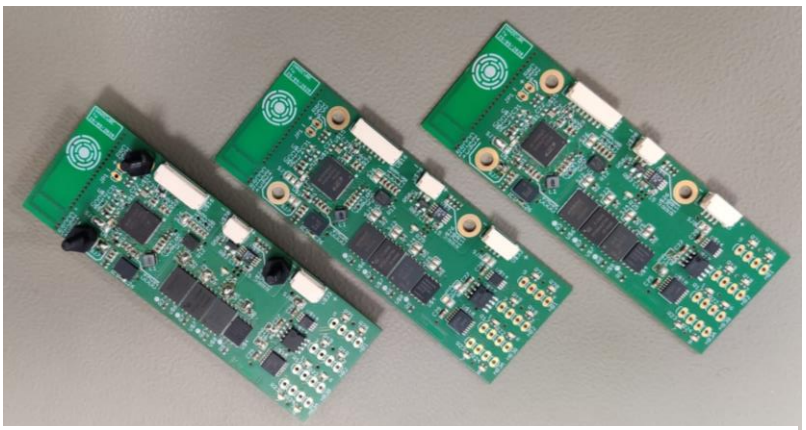


Figure 3- Three assembled boards

Board constitution

Have known of the hardware features related to a specific board, is a crucial thing that all developers should know. A hardware verification ensures the good software functionalities on the related board.

In the following pictures, a description of the board is made.

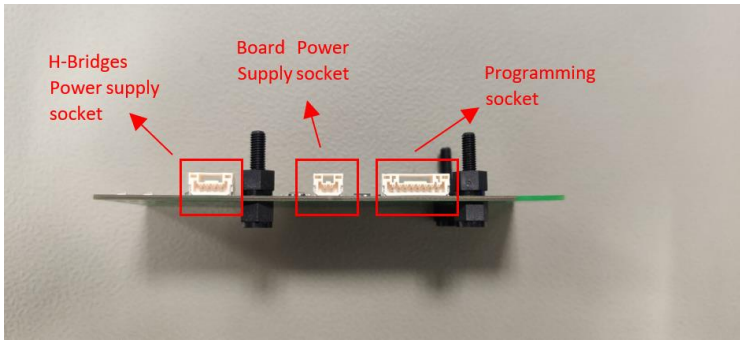


Figure 4- Sockets description

To power board, three sockets are provided, each of them with different purposes. H-Bridge power supply socket which can be used if the user wants to take advantage of their functionalities, for example, to connect several motors to the board. The board power-supply socket to supply the different devices on the board, such as sensors, MRAM, flashes and so ever and to conclude, the programming socket, which deserves to flash the board with a specific application.

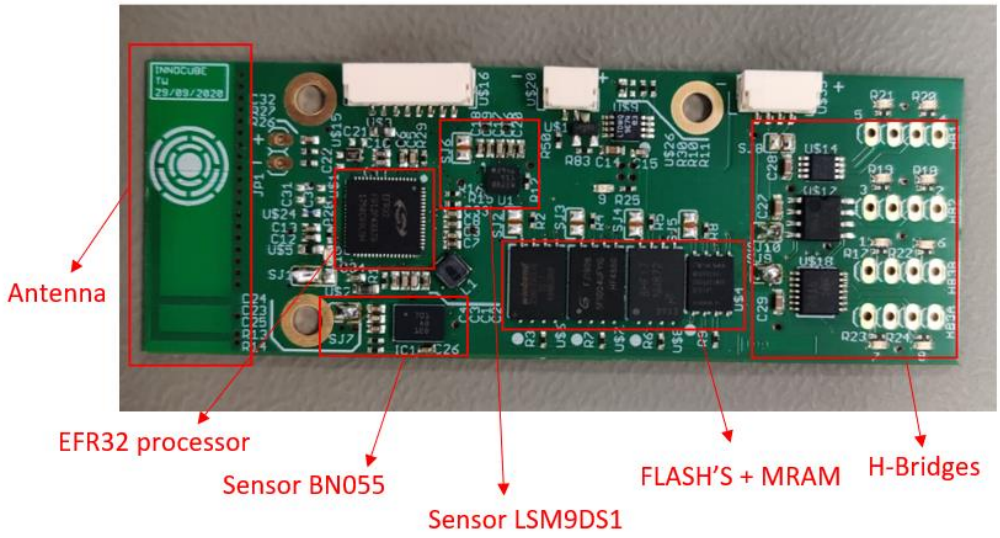


Figure 5 – Devices disposition

The constitution of the gecko board includes several devices that communicate with each other. There are all present in the figure above *(figure 5)*. The test of the SPI module was done with corresponding devices that use SPI communication (sensor LSM9DS1, FLASH's and MRAM).

Note:
These boards provide jumpers that can be iteratively welded according to each device test. This iterative procedure ensures the good functionality of each device on the board.

Stage 3 - Led Blinking

After have the knowledge of how to programme with RODOS and familiarize with all of hardware, the interaction with both was tested. As the first interaction with the board some peripherals, as GPIO and UART, were used to put a simple led blinking.

Compile and Flash script

To optimize the process of compiling and flashing the board, the script to *"CandF.sh"* was updated as shown in the following figure.

```
#!/bin/bash

cd ../../
source setenvs.sh
rodos-lib.sh gecko1MIC
cd ~/git-rodos/rodos-geckoport2/<DIR_WHERE_CPP>
rodos-executable.sh gecko1MIC <CPP_FILE_NAME>.cpp
arm-none-eabi-objcopy -O binary tst tst.bin
openocd -f board/efm32.cfg -c " init;halt;flash write_image erase tst.bin 0 bin;reset run; shutdown"
```

Figure 6- Script to compile and flash PCB assembled

Code analysis

To start the code, the API of RODOS was included. It provides a several of modules, allowing the control of peripherals integrated on the Gecko board.

Next, an instantiation of an object "ledBlue" through the class HAL_GPIO, on the correspondent pin of the assembled board (GPIO_058), was inserted. Through this object, we initialize the GPIO pin as set and as an output pin with the *"init()"* function.

To test the peripheral of UART, a counter denominated as *"cnt"* was declared and initialized with value 1.

The main goal of this little application is, during the run function execution, the counter will be incremented and printed with *"PRINTF"* function and the led will change the state in 100 milliseconds, periodically.

```
#include "rodos.h"
static Application module01("LED_Test", 2001);

HAL_GPIO ledBlue(GPIO_058);
//HAL_GPIO ledRed(GPIO_061);
//HAL_GPIO ledOrange(GPIO_062);
//HAL_GPIO ledGreen(GPIO_063);

class LEDTest: public StaticThread<> {
public:
void init() {
        ledBlue.init(true,1,1);
}

void run(){
        int8_t cnt=1;

        while(1){
                ledBlue.setPins(cnt&0x01);
                cnt++;
                suspendCallerUntil(NOW() + 100 * MILLISECONDS);
                PRINTF("Cnt: %d\n",cnt);
        }
}
};

LEDTest ledtest;
```

Stage 4 - SPI communication

This following stage will start to abord an introduction/revision about what is and how work SPI. Will be presented the needed configurations to get a SPI communication, by **pooling** or **preemptive** implementation and the structure of **SPI module** implementation.

Serial Peripheral Interface

SPI is a synchronous, full duplex master-slave-based interface. The data from the master or the slave is synchronized on the rising or falling clock edge. Both master and slave can transmit data at the same time. The SPI interface can be either 3-wire or 4-wire. The realized implementation follows 4-wire SPI interface.

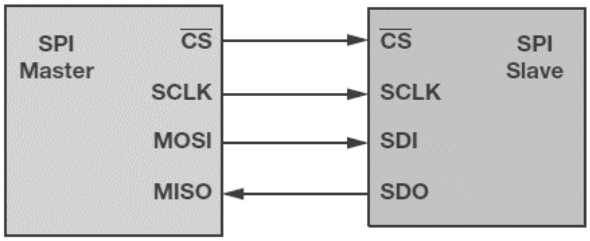


Figure 7- SPI interface between master and a slave

4-wire SPI devices have four signals:

- Clock (SPI CLK, SCLK)
- Chip select (CS)
- Master out, slave in (MOSI)
- Master in, slave out (MISO)

SPI interfaces can have only one master and can have one or multiple slaves. The device that generates the **clock signal** (CLK) is called the master. Data transmitted between the master and the slave is synchronized to the clock generated by the master.

The **chip select** (CS) signal from the master is used to select the slave. Normally an active low signal and is pulled high to disconnect the slave from the SPI bus.

MOSI and **MISO** are the data lines. MOSI transmits data from the master to the slave and MISO transmits data from the slave to the master.

SPI modes

The CPOL bit sets the polarity of the clock signal. The CPHA bit sets the clock phase and with them the SPI mode is defined.

SPI Mode	CPOL	CPHA	Clock Polarity in Idle State	Clock Phase Used to Sample and/or Shift the Data
0	0	0	Logic low	Data sampled on rising edge and shifted out on the falling edge
1	0	1	Logic low	Data sampled on the falling edge and shifted out on the rising edge
2	1	1	Logic high	Data sampled on the falling edge and shifted out on the rising edge
3	1	0	Logic high	Data sampled on the rising edge and shifted out on the falling edge

Figure 8- SPI modes

SPI implementation

In this section two SPI implementations will be shown to explain how to configure SPI for assembled gecko board. One will be done by pooling implementation and the other by a preemptive implementation.

To test an SPI communication is necessary to establish which device will be the slave. Each device is built with a specific SPI configuration, which are:

- The SPI mode
- Baudrate
- Most-significant-bit first or at the end.

Polling implementation

A pooling implementation was approached using the LSM9DS1 sensor. This sensor use SPI mode 3, can communicate with a 100kHz of baudrate and with most-significant-bit first. It has also a special particularity, that is the fact of having two chip select-pins to communicate with different sensors on the same device. One of the chip select pin controls the Accelerometer and Gyroscope sensors and the other one controls the Magnetometer sensor.

To initialize and configure the SPI an *"initUSART3()"* function was implemented. According to the datasheet of the gecko board to control SPI configurations, it's necessary to go through USART configurations.

Before configuring the SPI settings, the connection pins of the sensor must be initialized according to their functions. For instance, according to the schematic the CLK pin must be set to PC2, TX_pin to PC0, and RX_pin to PC1. Furthermore, as these are the pins, according to the gecko board datasheet *(page 145),* the USART3 should be selected to establish the SPI communication to this sensor on pin location 18.

```
/**************************************************************************//**
 * @brief Initialize USART3
 *****************************************************************************/
void initUSART3 (void)
{
  CMU_ClockEnable(cmuClock_GPIO, true);
  CMU_ClockEnable(cmuClock_USART3, true);

  // Configure GPIO mode
  GPIO_PinModeSet(gpioPortC, 2, gpioModePushPull, 0); // US3_CLK is push pull
  GPIO_PinModeSet(gpioPortC, 0, gpioModePushPull, 1); // US3_TX (MOSI) is push pull
  GPIO_PinModeSet(gpioPortC, 1, gpioModeInput, 1);    // US3_RX (MISO) is input


  // Start with default config, then modify as necessary
  USART_InitSync_TypeDef config = USART_INITSYNC_DEFAULT;
  config.master       = true;             // master mode
  config.baudrate     = 1000000;          // CLK freq is 1 MHz
  config.autoCsEnable = false;            // CS pin controlled by hardware, not firmware
  config.clockMode    = usartClockMode3;  // clock idle low, sample on rising/first edge
  config.msbf         = true;             // send MSB first
  config.enable       = usartDisable;     // Make sure to keep USART disabled until it's all set up
  USART_InitSync(USART3, &config);

  // Set USART pin locations
  USART3->ROUTELOC0 = (USART_ROUTELOC0_CLKLOC_LOC18) | // US2_CLK       on location 1 = PC2
                      /*(USART_ROUTELOC0_CSLOC_LOC18)  |*/ // US2_CS        on location 1 = PC3
                      (USART_ROUTELOC0_TXLOC_LOC18)  | // US2_TX (MOSI) on location 1 = PC0
                      (USART_ROUTELOC0_RXLOC_LOC18);   // US2_RX (MISO) on location 1 = PC1

  // Enable USART pins
  USART3->ROUTEPEN = USART_ROUTEPEN_CLKPEN /*| USART_ROUTEPEN_CSPEN*/ | USART_ROUTEPEN_TXPEN | USART_ROUTEPEN_RXPEN;

  // Enable USART3
  USART_Enable(USART3, usartEnable);
}
```

After that, all SPI configurations can be done according to the SPI characteristics of the sensor and the USART3 can be enabled.

```
void SPIreadBytes()
{
    uint8_t var_adress      = 2;
    uint8_t var_data        = 2;
    uint8_t rAddress        = 0x8f;                    //who I am register

    CS.setPins(0);
    var_adress = USART_SpiTransfer(USART3, rAddress);
    var_data = USART_SpiTransfer(USART3, 0x00);        // Read into destination array
    CS.setPins(1);

}
```

In pooling mode, the function *“SPIreadBytes()”* was provided to be called when the user wants to read the *“WHO_I_AM”* address from the sensor, which is located on 0x8f (this is a register that the sensor has to normally check if the communication is working).
The first thing to do is the initialization of the SPI communication, putting the respective CS to low (This can be done easily as the chip select is a GPIO pin which is instantiated by the class HAL_GPIO). Next thing to do is sending an address with the respective most significant bit to 1, as the operation is classified as a read operation. After that, our slave

device will transmit the respective data, which can be received with the return of the same function used to transfer data. To finish this transmission the pin CS must be set again.

```
static Application nameNotImportantTM("spicomunication");

class SPI_com : public StaticThread<>{
    public:
    SPI_com() : StaticThread<>("SPI") {}

    void init(){
        CS.init(true, 1, 0x01);
        CSB.init(true, 1, 0x01);
        initUSART3();
    }

    void run(){
        while(1){
            AT(NOW() + 2*SECONDS);
            SPIreadBytes();
            AT(NOW() + 2*SECONDS);
            xprintf("VAR ADDR: %x\n", (uint32_t)var_adress );
            xprintf("VAR DATA: %x\n\n", (uint32_t)var_data );
        }
    }
};

SPI_com module1;
```

To conclude and put the above functions in practice we initialize those two CS pins of the sensor to high and *"initUSART3()"* function is called.

In the *"run()"* function, the *"SPIreadBytes"* is called every four seconds periodically, and the value of the *"WHO_I_AM"* address is read into the *"var_data"* variable.

Preemptive implementation

Preemptive implementation means that interrupts are used, explained by other words determines that an interruption handler is triggered when some message is sent from Tx/Rx bus through the USART.

To add the interruption functionalities, the SPI configuration code should be modified. Then, the function "*initUSART3()*" must be implemented as follow.

```
/**************************************************************************//**
 * @brief Initialize USART3
 *****************************************************************************/
void initUSART3 (void)
{
…
…
…

    // Enabling TX interrupts to transfer whenever
    // there is room in the transmit buffer

    USART_IntClear(USART3, USART_IF_TXBL);
    USART_IntEnable(USART3, USART_IEN_TXBL);
    NVIC_ClearPendingIRQ(USART3_TX_IRQn);
    NVIC_EnableIRQ(USART3_TX_IRQn);

    // Enabling RX interrupts to trigger whenever
    // new data arrives from the slave
    USART_IntClear(USART3, USART_IF_RXDATAV);
    USART_IntEnable(USART3, USART_IEN_RXDATAV);
    NVIC_ClearPendingIRQ(USART3_RX_IRQn);
    NVIC_EnableIRQ(USART3_RX_IRQn);


    // Enable USART3
    USART_Enable(USART3, usartEnable);
}
```

The handlers of interruptions are triggered by USART, they are separated in two sections, the “USART3_TX_IRQHandler” and “USART3_RX_IRQHandler”. Those handlers are responsible to transmit information when something is on Tx bus and hence receive something when something is on Rx bus.

```
extern "C"
{
/**************************************************
 * @brief USART3 TX IRQ Handler
 *
 *
 **************************************************/
void USART3_TX_IRQHandler(void)
{
    USART_Tx(USART3, TxBuffer[TxBufferIndex++] );
    // Stop once we've gone through the whole TxBuffer
    if (TxBufferIndex == TX_BUFFER_SIZE)
    {
        TxBufferIndex = 0;
    }
}
```

```
/**************************************************
 * @brief USART3 RX IRQ Handler
 *
 *
 **************************************************/
void USART3_RX_IRQHandler(void)
{
    // Send and receive incoming data
    RxBuffer[RxBufferIndex++] = USART_Rx(USART3);

    // Stop once we've filled up RxBuffer
    if (RxBufferIndex == RX_BUFFER_SIZE)
    {
        RxBufferIndex = 0;
        xprintf("byte1: %x \n", RxBuffer[0]);
        xprintf("byte2: %x \n", RxBuffer[1]);
        USART_Enable(USART3, usartDisable);
    }
}// end extern "C"
```

As the code is illustrating, *“USART3_TX_IRQHandler()”,* send what is in TxBuffer periodically. Hence, when some address is sent through SPI communication for this sensor the second byte on bus is sent by slave, then, data is saved in RxBuffer.

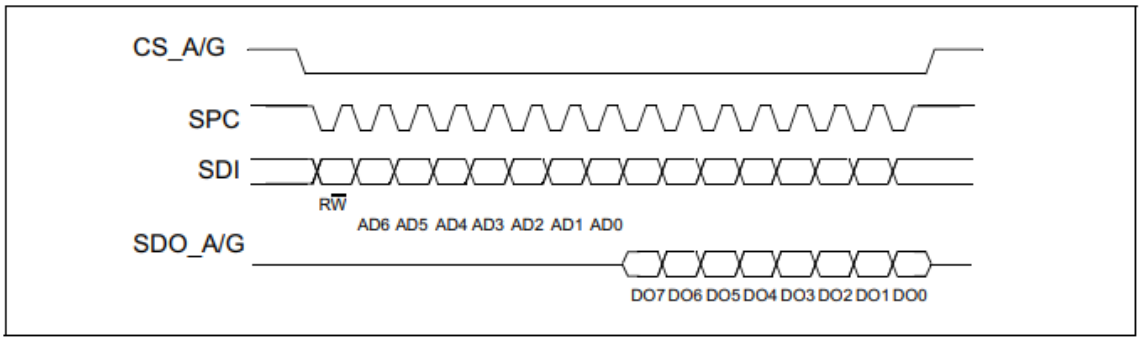


Figure 9 - Accelerometer and gyroscope SPI read protocol

On *"init()"* function in main system thread, the CSs pins are set and the USART3 is initialized with preemtive implementation. On other hand, the *"run()"* function don't need to make anything, since the communication is implemented to use interruptions.

```
static Application nameNotImportantTM("spicomunication");

class SPI_com : public StaticThread<>{
    public:

    SPI_com() : StaticThread<>("SPI") {}

    void init() {
        CS.init(true, 1, 0x01);
        CSB.init(true, 1, 0x01);
        initUSART3();
    }

    void run(){
        while(1);
    }
};
SPI_com module1;
```

SPI module implementation

After doing these procedures all pieces, to assembly the SPI module, were provided.

The API of RODOS already stores an "*hal_spi.h"*. This header file was used as a template to choose the function names, the function parameters, and some variable names.

The implemented module will be located at directory: "~/location_of_rodos/rodos-geckoport2/src/bare-metal/efr32fg1p/hal/*hal_spi.cpp*", as is a module assembled for EFR32FG1P boards.

At beginning of the module, a several APIs were included. They connect the SPI module to header files and provide some important functions which were used on this module implementation.

```
#include <new>
#include "rodos.h"
#include "hal/hal_spi.h"
#include "hw_hal_gpio.h"


#include "em_device.h"
#include "em_chip.h"
#include "em_cmu.h"
#include "em_gpio.h"
#include "em_usart.h"
```

The *"HW_HAL_SPI"* class was implemented to serve all instantiated objects of SPI. Where, on it, is declared all necessary variables which were used in module implementation.

```
class HW_HAL_SPI {
public:
SPI_IDX idx;
bool initialized;
USART_TypeDef *SPIx;
uint32_t baudrate;
USART_InitSync_TypeDef config;

uint32_t GPIO_Pin_MISO;
uint32_t GPIO_Pin_MOSI;
uint32_t GPIO_Pin_SCK;
uint32_t GPIO_Pin_NSS;
GPIO_Port_TypeDef GPIO_Port_MISO;
GPIO_Port_TypeDef GPIO_Port_MOSI;
GPIO_Port_TypeDef GPIO_Port_SCK;
GPIO_Port_TypeDef GPIO_Port_NSS;


public:
        HW_HAL_SPI(SPI_IDX idx, GPIO_PIN sckPin, GPIO_PIN misoPin, GPIO_PIN mosiPin, GPIO_PIN nssPin);
HW_HAL_SPI(SPI_IDX idx);

~HW_HAL_SPI(){}

void waitOnTransferReady();

uint32_t setBaudrate(uint32_t baudrate);

        void initMembers(SPI_IDX idx, GPIO_PIN sckPin, GPIO_PIN misoPin, GPIO_PIN mosiPin, GPIO_PIN nssPin =
GPIO_INVALID);

};
```

When an object is instantiated, some parameter must be accomplished to establish a SPI communication. The first parameter is the *"IDX",* where identity which device will be the slave and the other ones are optional, however, they have the purpose of identifying specific pins for MISO, MOSI, SCLK, and CS. As normal implementation the user just should worry about which device he wants to communicate, only defining the index when the SPI object is instantiated.

```
HW_HAL_SPI::HW_HAL_SPI(SPI_IDX idx) {

switch(idx){
    case SPI_IDX1:
         //SCK = PC2, MISO = PC1, MOSI = PC0, CS = PC3    - TO COMUNICATE WITH LSM9DS1  -   accelerometer and gyroscope
//SCK = PC2, MISO = PC1, MOSI = PC0, CS = PC4    - TO COMUNICATE WITH LSM9DS1  -     magnetometer
    initMembers(idx, GPIO_034, GPIO_033, GPIO_032 );
         break;
    case SPI_IDX2:
//SCK = PF5(GPIO_085), MISO = PF4(GPIO_084), MOSI = PF3(GPIO_083), CS = PC10(GPIO_042)  -TO COMUNICATE WITH RAM
//SCK = PF5(GPIO_085), MISO = PF4(GPIO_084), MOSI = PF3(GPIO_083), CS = PC11(GPIO_043)  -TO COMUNICATE WITH FLASH1
//SCK = PF5(GPIO_085), MISO = PF4(GPIO_084), MOSI = PF3(GPIO_083), CS = PB9 (GPIO_025)  -TO COMUNICATE WITH FLASH2
//SCK = PF5(GPIO_085), MISO = PF4(GPIO_084), MOSI = PF3(GPIO_083), CS = PB10(GPIO_026)  -TO COMUNICATE WITH FLASH3
initMembers(idx,GPIO_085 ,GPIO_084 ,GPIO_083 );
         break;
default:
            RODOS_ERROR("SPI index out of range");
}
}
```

The constructer implementation was done consonant the input index previously defined by the instantiation. The index defines which pin will be used to establish the communication between master and slave device as the USART to being used. This module already provides GPIO pins consonant with the desired communication device, where the access of GPIO pins connection is explicit on the schematic.

```
void HW_HAL_SPI::initMembers(SPI_IDX idx, GPIO_PIN sckPin, GPIO_PIN misoPin, GPIO_PIN mosiPin, GPIO_PIN nssPin) {

this->idx = idx;

switch(idx){
case SPI_IDX1:
         CMU_ClockEnable(cmuClock_GPIO, true);
                  CMU_ClockEnable(cmuClock_USART3, true);
         break;
case SPI_IDX2:
         CMU_ClockEnable(cmuClock_GPIO, true);
                  CMU_ClockEnable(cmuClock_USART2, true);
         break;
default:
         RODOS_ERROR("SPI index out of range");
}



         GPIO_Pin_SCK    = HW_HAL_GPIO::getEFR32Pin(sckPin);
GPIO_Port_SCK  = HW_HAL_GPIO::getEFR32Port(sckPin);

GPIO_Pin_MISO  = HW_HAL_GPIO::getEFR32Pin(misoPin);
GPIO_Port_MISO = HW_HAL_GPIO::getEFR32Port(misoPin);

GPIO_Pin_NSS  = HW_HAL_GPIO::getEFR32Pin(nssPin);
GPIO_Port_NSS = HW_HAL_GPIO::getEFR32Port(nssPin);

GPIO_Pin_MOSI  = HW_HAL_GPIO::getEFR32Pin(mosiPin);
GPIO_Port_MOSI = HW_HAL_GPIO::getEFR32Port(mosiPin);
```

```
//SPI_IDX1
//GPIO_PinModeSet(gpioPortC, 2, gpioModePushPull, 0); // US3_CLK is push pull
        //GPIO_PinModeSet(gpioPortC, 0, gpioModePushPull, 1); // US3_TX (MOSI) is push pull
        //GPIO_PinModeSet(gpioPortC, 1, gpioModeInput, 1);    // US3_RX (MISO) is input

//SPI_IDX2
//GPIO_PinModeSet(gpioPortF, 5, gpioModePushPull, 0); // US2_CLK is push pull
        //GPIO_PinModeSet(gpioPortF, 3, gpioModePushPull, 1); // US2_TX (MOSI) is push pull
        //GPIO_PinModeSet(gpioPortF, 4, gpioModeInput, 1);    // US2_RX (MISO) is input

switch(idx){
case SPI_IDX1:
        SPIx = USART3;
        config.enable  = usartDisable;
        break;
case SPI_IDX2:
        SPIx = USART2;
        config.enable  = usartDisable;
        break;
default:
        RODOS_ERROR("SPI index out of range");
}

GPIO_PinModeSet(GPIO_Port_SCK, GPIO_Pin_SCK, gpioModePushPull, 0);
GPIO_PinModeSet(GPIO_Port_MOSI, GPIO_Pin_MOSI, gpioModePushPull, 1);
GPIO_PinModeSet(GPIO_Port_MISO, GPIO_Pin_MISO, gpioModeInput, 1);

}
```

The Class "*HW_HAL_SPI*" also provide a *"initMembers"* function which is presented into above code. It has as parameters, the respective device index and the pins which are presented on schematic. This function has to main goal prepare all USART ports/pins to allow the SPI configuration in *"init()"* function.

```
/** init SPI interface
 */
int32_t HAL_SPI::init(uint32_t baudrate, bool slave, bool tiMode) {

        CMU_ClockEnable(cmuClock_GPIO, true);

switch(context->idx){
case SPI_IDX1:
                CMU_ClockEnable(cmuClock_USART3, true);

        context->config = USART_INITSYNC_DEFAULT;
        context->config.master       = true;
                context->config.baudrate     = baudrate;
                context->config.autoCsEnable = false;
        context->config.clockMode = usartClockMode3;
        context->config.msbf      = true;
        break;

case SPI_IDX2:
        CMU_ClockEnable(cmuClock_USART2, true);
        context->config = USART_INITSYNC_DEFAULT;
        context->config.master       = true;
                context->config.baudrate     = baudrate;
                context->config.autoCsEnable = false;
        context->config.clockMode = usartClockMode3;
        context->config.msbf      = true;
        break;
default:
        RODOS_ERROR("SPI index out of range");
}

        context->config.enable = usartDisable;
        USART_InitSync(context->SPIx, &(context->config));
```

```
switch(context->idx){
case SPI_IDX1:
        USART3->ROUTELOC0 = (USART_ROUTELOC0_CLKLOC_LOC18)      |        // US3_CLK       on location 1 = PC2
                            (USART_ROUTELOC0_TXLOC_LOC18)       | // US3_TX (MOSI) on location 1 = PC0
                            (USART_ROUTELOC0_RXLOC_LOC18);      //// US3_RX (MISO) on location 1 = PC1

                USART3->ROUTEPEN = USART_ROUTEPEN_CLKPEN | USART_ROUTEPEN_TXPEN | USART_ROUTEPEN_RXPEN;
        break;
case SPI_IDX2:
        USART2->ROUTELOC0 = (USART_ROUTELOC0_CLKLOC_LOC16)      |        // US2_CLK       on location 1 = PF5
                            (USART_ROUTELOC0_TXLOC_LOC16)       | // US2_TX (MOSI) on location 1 = PF3
                            (USART_ROUTELOC0_RXLOC_LOC16);      //// US2_RX (MISO) on location 1 = PF4


        USART2->ROUTEPEN = USART_ROUTEPEN_CLKPEN | USART_ROUTEPEN_TXPEN | USART_ROUTEPEN_RXPEN;
        break;
default:
        RODOS_ERROR("SPI index out of range");
}

USART_Enable(context->SPIx, usartEnable);

context->initialized = true;
return 0;
}
```

Note: Consonant IDX1 and IDX2 the USART3 and USART2 respectively, will be used to configure the sensors (USART3), RAM (USART2) and Flash (USART2).

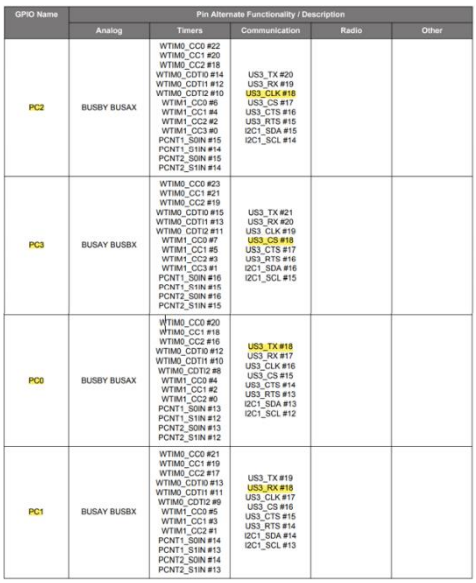
To initialize the SPI*, the "init()"* function has as parameters:

- *"baudrate",* which defines the number of bits per second;
- "slave", type of initialization, whether slave or master mode;
- *"tiMode",* which refers to the sampling on falling or rising edge of the clock.

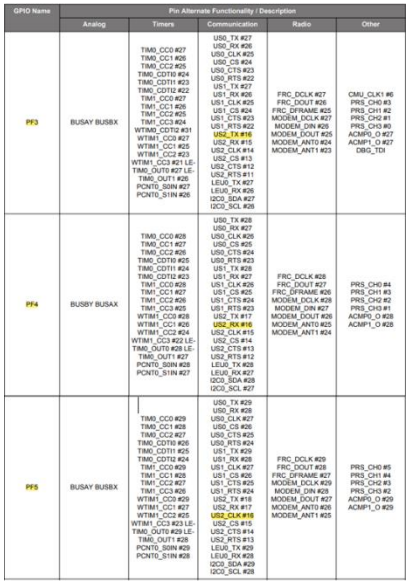
The initialization of SPI communication with sensors Magnometer/Accelerometer/Gyroscope, as mentioned before, can be achieved with the definition of SPI_IDX1 where the baudrate is declared by the user and the SPI has mode 3 configured, in other words, the data is sampled on the rising edge and shifted out on the falling edge. Talking about the FLASHs and MRAM, they have the same configuration however are used by a different USART (USART2).

Next, to ensure that the USART is not enabled yet, previous initialization a USART disable configuration must be done.

After looking into Gecko board datasheet some locations of pins has to be done consonant the respective ports.



GPIO Name	Pin Alternate Functionality / Description				
	Analog	Timers	Communication	Radio	Other
PC2	BUSBY BUSAX	WTIM0_CC0 #22 WTIM0_CC1 #20 WTIM0_CC2 #18 WTIM0_CDTI0 #14 WTIM0_CDTI1 #12 WTIM0_CDTI2 #10 WTIM1_CC0 #6 WTIM1_CC1 #4 WTIM1_CC2 #2 WTIM1_CC3 #0 PCNT1_S0IN #15 PCNT1_S1IN #14 PCNT2_S0IN #15 PCNT2_S1IN #14	US3_TX #20 US3_RX #19 US3_CLK #18 US3_CS #17 US3_CTS #16 US3_RTS #15 I2C1_SDA #15 I2C1_SCL #14		
PC3	BUSAY BUSBX	WTIM0_CC0 #23 WTIM0_CC1 #21 WTIM0_CC2 #19 WTIM0_CDTI0 #15 WTIM0_CDTI1 #13 WTIM0_CDTI2 #11 WTIM1_CC0 #7 WTIM1_CC1 #5 WTIM1_CC2 #3 WTIM1_CC3 #1 PCNT1_S0IN #16 PCNT1_S1IN #15 PCNT2_S0IN #16 PCNT2_S1IN #15	US3_TX #21 US3_RX #20 US3_CLK #19 US3_CS #18 US3_CTS #17 US3_RTS #16 I2C1_SDA #16 I2C1_SCL #15		
PC0	BUSBY BUSAX	WTIM0_CC0 #20 WTIM0_CC1 #18 WTIM0_CC2 #16 WTIM0_CDTI0 #12 WTIM0_CDTI1 #10 WTIM0_CDTI2 #8 WTIM1_CC0 #4 WTIM1_CC1 #2 WTIM1_CC2 #0 PCNT1_S0IN #13 PCNT1_S1IN #12 PCNT2_S0IN #13 PCNT2_S1IN #12	US3_TX #18 US3_RX #17 US3_CLK #16 US3_CS #15 US3_CTS #14 US3_RTS #13 I2C1_SDA #13 I2C1_SCL #12		
PC1	BUSAY BUSBX	WTIM0_CC0 #21 WTIM0_CC1 #19 WTIM0_CC2 #17 WTIM0_CDTI0 #13 WTIM0_CDTI1 #11 WTIM0_CDTI2 #9 WTIM1_CC0 #5 WTIM1_CC1 #3 WTIM1_CC2 #1 PCNT1_S0IN #14 PCNT1_S1IN #13 PCNT2_S0IN #14 PCNT2_S1IN #13	US3_TX #19 US3_RX #18 US3_CLK #17 US3_CS #16 US3_CTS #15 US3_RTS #14 I2C1_SDA #14 I2C1_SCL #13		



GPIO Name	Pin Alternate Functionality / Description				
	Analog	Timers	Communication	Radio	Other
PF3	BUSAY BUSBX	TIM0_CC0 #27 TIM0_CC1 #26 TIM0_CC2 #25 TIM0_CDTI0 #24 TIM0_CDTI1 #23 TIM0_CDTI2 #22 TIM1_CC0 #27 TIM1_CC1 #26 TIM1_CC2 #25 TIM1_CC3 #24 WTIM0_CDTI2 #31 WTIM1_CC0 #27 WTIM1_CC1 #25 WTIM1_CC2 #23 WTIM1_CC3 #21 LE- TIM0_OUT0 #27 LE- TIM0_OUT1 #26 PCNT0_S0IN #27 PCNT0_S1IN #26	US0_TX #27 US0_RX #26 US0_CLK #25 US0_CS #24 US0_CTS #23 US0_RTS #22 US1_TX #27 US1_RX #26 US1_CLK #25 US1_CS #24 US1_CTS #23 US1_RTS #22 US2_TX #16 US2_RX #15 US2_CLK #14 US2_CS #13 US2_CTS #12 US2_RTS #11 LEU0_TX #27 LEU0_RX #26 I2C0_SDA #27 I2C0_SCL #26	FRC_DCLK #27 FRC_DOUT #26 FRC_DFRAME #25 MODEM_DCLK #27 MODEM_DIN #25 MODEM_DOUT #25 MODEM_ANT0 #24 MODEM_ANT1 #23	CMU_CLK1 #6 PRS_CH0 #3 PRS_CH1 #2 PRS_CH2 #1 PRS_CH3 #0 ACMP0_O #27 ACMP1_O #27 DBG_TDI
PF4	BUSBY BUSAX	TIM0_CC0 #28 TIM0_CC1 #27 TIM0_CC2 #26 TIM0_CDTI0 #25 TIM0_CDTI1 #24 TIM0_CDTI2 #23 TIM1_CC0 #28 TIM1_CC1 #27 TIM1_CC2 #26 TIM1_CC3 #25 WTIM1_CC0 #28 WTIM1_CC1 #26 WTIM1_CC2 #24 WTIM1_CC3 #22 LE- TIM0_OUT0 #28 LE- TIM0_OUT1 #27 PCNT0_S0IN #28 PCNT0_S1IN #27	US0_TX #28 US0_RX #27 US0_CLK #26 US0_CS #25 US0_CTS #24 US0_RTS #23 US1_TX #28 US1_RX #27 US1_CLK #26 US1_CS #25 US1_CTS #24 US1_RTS #23 US2_TX #17 US2_RX #16 US2_CLK #15 US2_CS #14 US2_CTS #13 US2_RTS #12 LEU0_TX #26 LEU0_RX #27 I2C0_SDA #28 I2C0_SCL #27	FRC_DCLK #26 FRC_DOUT #27 FRC_DFRAME #26 MODEM_DCLK #28 MODEM_DIN #27 MODEM_DOUT #26 MODEM_ANT0 #25 MODEM_ANT1 #24	PRS_CH0 #4 PRS_CH1 #3 PRS_CH2 #2 PRS_CH3 #1 ACMP0_O #28 ACMP1_O #28
PF5	BUSAY BUSBX	TIM0_CC0 #29 TIM0_CC1 #28 TIM0_CC2 #27 TIM0_CDTI0 #26 TIM0_CDTI1 #25 TIM0_CDTI2 #24 TIM1_CC0 #29 TIM1_CC1 #28 TIM1_CC2 #27 TIM1_CC3 #26 WTIM1_CC0 #29 WTIM1_CC1 #27 WTIM1_CC2 #25 WTIM1_CC3 #23 LE- TIM0_OUT0 #29 LE- TIM0_OUT1 #28 PCNT0_S0IN #29 PCNT0_S1IN #28	US0_TX #29 US0_RX #28 US0_CLK #27 US0_CS #26 US0_CTS #25 US0_RTS #24 US1_TX #29 US1_RX #28 US1_CLK #27 US1_CS #26 US1_CTS #25 US1_RTS #24 US2_TX #18 US2_RX #17 US2_CLK #16 US2_CS #15 US2_CTS #14 US2_RTS #13 LEU0_TX #28 LEU0_RX #28 I2C0_SDA #29 I2C0_SCL #28	FRC_DCLK #29 FRC_DOUT #28 FRC_DFRAME #27 MODEM_DCLK #29 MODEM_DIN #28 MODEM_DOUT #27 MODEM_ANT0 #26 MODEM_ANT1 #25	PRS_CH0 #5 PRS_CH1 #4 PRS_CH2 #3 PRS_CH3 #2 ACMP0_O #29 ACMP1_O #29

These previous pictures of the datasheet show how the implementation values of pin locations were chosen. Location 18 and location 16 consonant IDX_1 and IDX_2 respectively. With these few lines of code pins are enable for clock, receive, and transmit functionalities.

Then the SPI is initialized, and user can take advantage of write, read and writeread functions of SPI module to communicate with EFR32 board.

```
int32_t HAL_SPI::write(const void* sendBuf, size_t len) {

if(!context->initialized) return -1;

uint8_t TxBufferIndex;
uint8_t *pointer;
pointer = (uint8_t*)sendBuf;
uint8_t val;
uint8_t pp;

//SEND DATA
for(TxBufferIndex = 0 ; TxBufferIndex < len; TxBufferIndex++){
         val = *(pointer + TxBufferIndex);
         pp = USART_SpiTransfer(context->SPIx, val);
}

return static_cast<int32_t>(len);
}
```

The write function receives a pointer to a buffer and a length as parameters. The pointer has the responsibility to pass the transmission data into the function, and hence the length is the amount of data that should be transmitted.

At the beginning is verified if the SPI was initialized or not, whether affirmative, the value that the pointer is pointing will be passed to a “val” variable, and then it will be transmitted in SPI tx bus.

```
int32_t HAL_SPI::read(void* recBuf, size_t maxLen) {

if(!context->initialized) return -1;

uint8_t RxBufferIndex;
uint8_t *pointer;
pointer = (uint8_t*)recBuf;

//RECIEVE DATA
    for(RxBufferIndex=0; RxBufferIndex < maxLen; RxBufferIndex++){
          *(pointer + RxBufferIndex) = USART_SpiTransfer(context->SPIx, 0x00);
}

return static_cast<int32_t>(maxLen);
}
```

The read function receives a pointer to a buffer and a length as parameters. The pointer has the responsibility to pass the received data out of the function, and hence the length is the amount of data that should be received.

At the beginning is verified if the SPI was initialized or not, whether affirmative, a null value will be passed, as a parameter, to “USART_SpiTranfer()” function, and then some data will be received from a slave device. This procedure only can be done if an address was already sent.

```
int32_t HAL_SPI::writeRead(const void* sendBuf, size_t len, void* recBuf, size_t maxLen) {
if(!context->initialized) return -1;

uint8_t TxBufferIndex;
uint8_t *send_pointer;
send_pointer = (uint8_t*)sendBuf;
uint8_t val;
uint8_t pp;

uint8_t RxBufferIndex;
uint8_t *rec_pointer;
rec_pointer = (uint8_t*)recBuf;

//SEND DATA
for(TxBufferIndex = 0 ; TxBufferIndex < len; TxBufferIndex++){
          val = *(send_pointer + TxBufferIndex);
          pp = USART_SpiTransfer(context->SPIx, val);
}

//RECIEVE DATA
          for(RxBufferIndex=0; RxBufferIndex < maxLen; RxBufferIndex++){
          *(rec_pointer + RxBufferIndex) = USART_SpiTransfer(context->SPIx, 0x00);
}

return static_cast<int32_t>(len);
}
```

The *“WriteRead()”* function follow the same mechanism of the two implementations above to transmit some address/comands through a transmit buffer and receive some data through a receive buffer. Is an alternative to the use of just one function instead of two of

them. Perhaps in some cases only write function is necessary, so this three functions were implemented.

Stage 5 - Tests Hardware Devices

In this chapter, is conceived through the SPI, UART and GPIO libraries an operational test of the following hardware devices:

- Flash1
- Flash2
- Flash3
- RAM
- LSM9DS1 Sensor
- H-Bridges
- BN055 Sensor

Flash1 – Reading ID Register

To test the Flash1, we will read the ID register. Based on the Flash1 datasheet, to read the ID register must be sent the command *0x9Fh* and except to receive *0xEFh 0xAAh 0x21h.*

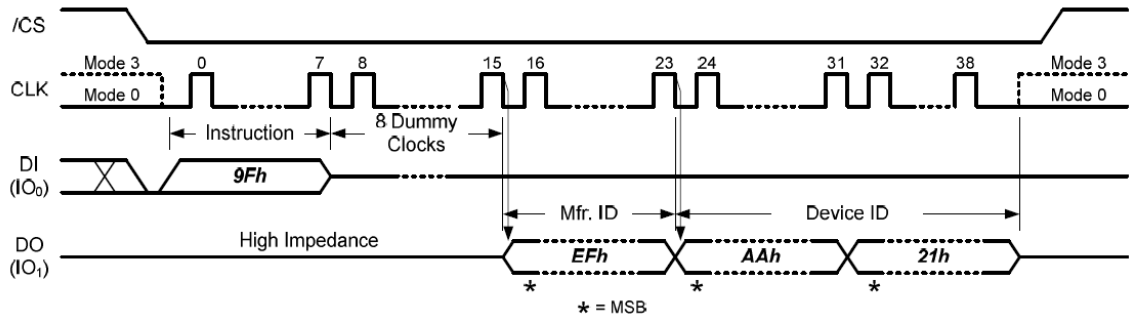


Figure 11 - Flash 1 SPI read ID-Register

However, in the Initialization of the Flash device, it must be reset by sending the command *0xFFh.*

Commands	OpCode
Device RESET	**FFh**

Figure 12- Flash 1 RESET command

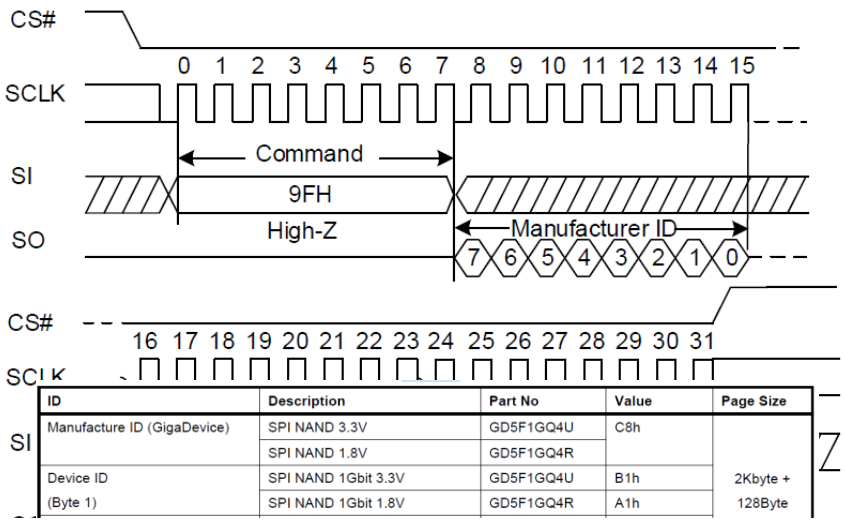
In the following code implementation, for the communication, must be defined Tx and Rx Buffer, Tx and Rx Index, the Buffers Size and data to transmit. Instantiate *SPI_IDX* to *SPI_IDX1* in this case, instantiate the *HAL_SPI* Object and the *HAL_GPIO* object for CS pin (*GPIO_043*). Then, the CS pin is initialized with value 1 (*CS.init(true, 1, 0x01))*. After, we initialize the SPI object and its baudrate (*sensor.init(1000000, false, false)).* After everything initialized, first we reset the flash by: clearing the CS pin to 0 (*CS.setPins(0))*; transmiting the 0xFFh byte *(sensor.write(TxBuffer, 1))*; and setting the CS pin back to 1 *(CS.setPins(1))*. After the reset, to read the ID register, is the same process: passing the 0x9F command to the Tx buffer (*TxBuffer[0] = 0x9F*); clearing the CS pin to 0 (*CS.setPins(0))*; transmiting the 0xFFh byte and dummy byte *(sensor.write(TxBuffer, 2))*; reading the response from the Flash (*sensor.read(RxBuffer, 3)*); and setting the CS pin back to 1 *(CS.setPins(1))*.

```
#include "rodos.h"

#define TX_BUFFER_SIZE   4
#define RX_BUFFER_SIZE   TX_BUFFER_SIZE

uint8_t TxBuffer[TX_BUFFER_SIZE] = {0xFF, 0x00, 0x00, 0x00};
uint32_t TxBufferIndex = 0;
uint8_t RxBuffer[RX_BUFFER_SIZE] = {0x00, 0x00, 0x00, 0x00};
uint32_t RxBufferIndex = 0;

SPI_IDX spi = SPI_IDX2;
HAL_SPI sensor(spi);
HAL_GPIO CS(GPIO_043);

static Application module01("SPI_test", 2001);
class SPI_READ_SENSOR: public StaticThread<> {
public:
    void init() {
        CS.init(true, 1, 0x01);
        sensor.init(1000000, false, false);
        CS.setPins(0);
        sensor.write(TxBuffer, 1);
        CS.setPins(1);
    }
    void run(){
        while(1){
            TxBuffer[0] = 0x9F;
            CS.setPins(0);
            sensor.write(TxBuffer, 2);
            sensor.read(RxBuffer, 3);
            CS.setPins(1);
        }
        }
};
SPI_READ_SENSOR read_sensor;
```

```
/dev/ttyUSB0 - PuTTY
vaules recieved: EF AA 21 44
vaules recieved: EF AA 21 44
vaules recieved: EF AA 21 44
vaules recieved: EF AA 210440
vaules recieved:0EF0AA0210440
vaules recieved:0EF0AA0210440
vaules recieved:0EF0AA021044
vaules recieved: EF AA 21 44
vaules recieved: EF AA 21 44
vaules recieved: EF AA 21 44
vaules recieved: EF AA 21 44
vaules recieved: EF AA 21 44
```

Flash2 – Reading ID Register

To test the Flash2, the ID register was read. The process is very identical to the Flash1 as the command is the same, *0x9Fh*. It changes only on terms of the CS pin (*GPIO_025*), the Flash response (*0xC8h 0xB1h 0x48h*), no need for the dummy Byte.



ID	Description	Part No	Value	Page Size
Manufacture ID (GigaDevice)	SPI NAND 3.3V	GD5F1GQ4U	C8h	
	SPI NAND 1.8V	GD5F1GQ4R		
Device ID	SPI NAND 1Gbit 3.3V	GD5F1GQ4U	B1h	2Kbyte +
(Byte 1)	SPI NAND 1Gbit 1.8V	GD5F1GQ4R	A1h	128Byte

Figure 12 - Manufacture ID read with FLASH 2 information from datasheet

```
#include "rodos.h"
#include "hal/hal_spi.h"

#define TX_BUFFER_SIZE   4
#define RX_BUFFER_SIZE   TX_BUFFER_SIZE

uint8_t TxBuffer[TX_BUFFER_SIZE] = {0xFF, 0x00, 0x00, 0x00};
uint32_t TxBufferIndex = 0;
uint8_t RxBuffer[RX_BUFFER_SIZE] = {0x00, 0x00, 0x00, 0x00};
uint32_t RxBufferIndex = 0;

SPI_IDX spi = SPI_IDX2;
HAL_SPI sensor(spi);
HAL_GPIO CS(GPIO_025);

static Application module01("SPI_test", 2001);

class SPI_READ_SENSOR: public StaticThread<> {
public:
    void init() {
        CS.init(true, 1, 0x01);
        sensor.init(1000000, false, false);

        CS.setPins(0);
        sensor.write(TxBuffer, 1);
        CS.setPins(1);
    }
    void run(){
        while(1){
            TxBuffer[0] = 0x9F;
            CS.setPins(0);
            sensor.write(TxBuffer, 2);
            sensor.read(RxBuffer, 3);
            CS.setPins(1);
        }
        }
```

```
--------------- Application running ------------
vaules recieved: B1 48 C8 44
vaules recieved: B1 48 C8 44
vaules recieved: B1 48 C8 44
vaules recieved: B1 48 C8 44
vaules recieved: B1 48 C8 44
vaules recieved: B1 48 C8 44
vaules recieved: B1 48 C8 44
```

Flash3 – Reading ID Register

To test the Flash3, the ID register was read. The process is very identical to the Flash1 as the command is the same *0x9Fh*. It changes only on terms of the CS pin (*GPIO_026*), the Flash response (*0x2Ch 0x14h*).

Table 3: READ ID Table

Byte	Description	7	6	5	4	3	2	1	0	Value
Byte 0	Manufacturer ID (Micron)	0	0	1	0	1	1	0	0	2Ch
Byte 1	1Gb 3.3V Device ID	0	0	0	1	0	1	0	0	14h

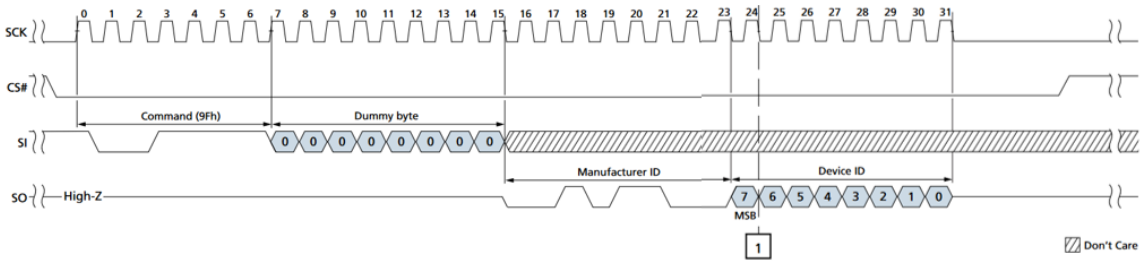


Figure 13 - Manufacture ID read with FLASH 3 information from datasheet

```
#include "rodos.h"
#include "hal/hal_spi.h"

#define TX_BUFFER_SIZE   4
#define RX_BUFFER_SIZE   TX_BUFFER_SIZE

uint8_t TxBuffer[TX_BUFFER_SIZE] = {0xFF, 0x00, 0x00, 0x00};
uint32_t TxBufferIndex = 0;

uint8_t RxBuffer[RX_BUFFER_SIZE] = {0x00, 0x00, 0x00, 0x00};
uint32_t RxBufferIndex = 0;

SPI_IDX spi = SPI_IDX2;
HAL_SPI sensor(spi);
HAL_GPIO CS(GPIO_026);

static Application module01("SPI_test", 2001);

class SPI_READ_SENSOR: public StaticThread<> {
public:
    void init() {
        CS.init(true, 1, 0x01);
        sensor.init(1000000, false, false);
        CS.setPins(0);
        sensor.write(TxBuffer, 1);
        CS.setPins(1);
    }
    void run(){
        while(1){
            TxBuffer[0] = 0x9F;
            CS.setPins(0);
            sensor.write(TxBuffer, 2);
            sensor.read(RxBuffer, 3);
            CS.setPins(1);
        }
        }
```

```
----------0Application running ----------
vaules recieved: 2C 14
vaules(recieved:02C014
vaules recieved:02C014
vaules recieved: 2C014
vaules recieved: 2C 14
vaules recieved: 2C 14
```

RAM – Reading ID Register

To test the RAM, the ID register was read. The process is very identical to the Flash1 as the command is the same *0x9Fh*. It changes only on terms of the Reset Command (*0x99h*), the CS pin (*GPIO_042*), the RAM response (*0xE6h 0x01h 0x03h 0x02h*).

Manufacturer ID	Interface	Voltage	Temperature	Density	Frequency
31-24	23-20	19-16	15-12	11-8	7-0
1110 0110	0000-HP QSPI	0001 - 3V	0000 → -40ºC to +85ºC	0001 - 4Mb	0001 - 108MHz
		0010 - 1.8V	0001 → -40ºC to +105ºC	0010 - 8Mb	0010 - 54MHz
				0011 - 16Mb	0011 - Reserved
					0100 - Reserved
					0101 - Reserved

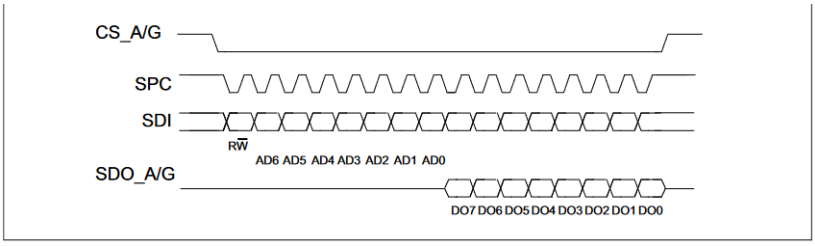
Figure 14 - Manufacture ID read with MRAM 3 information from datasheet

```
#include "rodos.h"
#include "hal/hal_spi.h"

#define TX_BUFFER_SIZE   4
#define RX_BUFFER_SIZE   TX_BUFFER_SIZE

uint8_t TxBuffer[TX_BUFFER_SIZE] = {0x99, 0x00, 0x00, 0x00};
uint32_t TxBufferIndex = 0;
uint8_t RxBuffer[RX_BUFFER_SIZE] = {0x00, 0x00, 0x00, 0x00};
uint32_t RxBufferIndex = 0;

SPI_IDX spi = SPI_IDX2;
HAL_SPI sensor(spi);
HAL_GPIO CS(GPIO_042);

static Application module01("SPI_test", 2001);

class SPI_READ_SENSOR: public StaticThread<> {
public:
    void init() {
        CS.init(true, 1, 0x01);
        sensor.init(1000000, false, false);
        CS.setPins(0);
        sensor.write(TxBuffer, 1);
        CS.setPins(1);
    }
    void run(){
        while(1){
            TxBuffer[0] = 0x9F;
            CS.setPins(0);
            sensor.write(TxBuffer, 1);
            sensor.read(RxBuffer, 4);
            CS.setPins(1);
        }
        }
};
```

```
---------- Application running ----------
vaules recieved: E6 1 3 2
vaules recieved: E6 103 2
vaules recieved: E6 1 3 2
vaules recieved: E6 1 3 2
vaules recieved: E6 1 3 2
vaules recieved: E6 1 3 2
```

LSM9DS1 Sensor - Readin *WHO_I_AM* register

To test the LSM9DS1 Sensor, the *WHO_I_AM* register was read. The process uses the command *0x8Fh* (bit read to 1 + register address). In comparison with the Flash, it doesn't need to be reset. However, for a successful communication, both CS pins of the sensor can never be set to 0 at the same time. As so, both CS pins are initialized to a value of 1 (*CS.init(true, 1, 0x01)* and *CSB.init(true, 1, 0x01)*). The write and read are executed on the same function, for this test (*sensor.writeRead(TxBuffer, 1, RxBuffer, 4)*). The CS pins are *GPIO_045* (Accelerometer and Gyroscope) and *GPIO_36*, the LSM9DS1 Sensor response (*0x2Ch 0x14h*).



Name	Type	Register address		Default
		Hex	Binary	
WHO_AM_I	r	0F	00001111	01101000

Figure 15 – WHO_AM_I read with LSM9DS1 information from datasheet

```
#include "rodos.h"
#include "hal/hal_spi.h"

#define TX_BUFFER_SIZE   4
#define RX_BUFFER_SIZE   TX_BUFFER_SIZE

uint8_t TxBuffer[TX_BUFFER_SIZE] = {0x8f, 0x00, 0x00, 0x00};
uint32_t TxBufferIndex = 0;
uint8_t RxBuffer[RX_BUFFER_SIZE] = {0x00, 0x00, 0x00, 0x00};
uint32_t RxBufferIndex = 0;

SPI_IDX spi = SPI_IDX1;
HAL_SPI sensor(spi);
HAL_GPIO CS(GPIO_035);
HAL_GPIO CSB(GPIO_036);

static Application module01("SPI_test", 2001);

class SPI_READ_SENSOR: public StaticThread<> {
public:
    void init() {
        CS.init(true, 1, 0x01);
        CSB.init(true, 1, 0x01);
        sensor.init(1000000, false, false);
    }
    void run(){
        while(1){
            CS.setPins(0);
            sensor.writeRead(TxBuffer, 1, RxBuffer, 4);
            CS.setPins(1);
        }
        }
};
```

```
----------0Application running ------------
vaules recieved: 68 0 0 0
vaules"recieved: 8 000 00
vaules recieved:06<0000000
vaules recieved:0680000000
vaules recieved: 68 000 0
```

H-Bridges – Led Blink Test

The H-Bridges were successfully tested, after some board corrections in the metal connections from assembly process. To use the H-Bridges, is required the *HAL_GPIO* module by enabling the Bridges pins (from *GPIO_000* to *GPIO_005),* turning ON and OFF the respective LEDs.

The H-Bridges combinations can be found in schematic.

IN1/IN2	IN3	OUT11/21	OUT12/22
L	L	OPEN	OPEN
H	L	OPEN	OPEN
L	H	H	L
H	H	L	H

IN1	IN2	OUT1	OUT2
L	L	OFF	OFF
H	L	H	L
L	H	L	H
H	H	H	H

FIN	RIN	OUT1	OUT2
L	L	HighZ	HighZ
H	L	H	L
L	H	L	H
H	H	L	L

Figure 16 - H-Bridges Schematic tables

BN055 Sensor

This sensor uses the HAL_UART module. It was not tested successfully due to the USART1 necessary for the communication, presented an implementation error, which we couldn't resolve it in time.

Conclusion and Final Considerations

During this project development there were some barriers encountered and some difficulties, which be had to resort to our supervisors. Such as:

1. Flashing and compiling problems on the assembled board (problem overcame;
2. H-Bridges were not well accoupled to the board, necessary to correct them;
3. Encountered implementation errors on the UART1, which didn't allowed us to use it as intended. Due to this, we were unable to test the BN055 sensor. Even though the efforts made to resolve the problem, the error was not corrected at time.

Our implementation originated some dependencies, on which:

1. Our SPI module is statically implemented to use the USART2 for the communication with the FLASH and RAM devices, and the USART3 for the communication with the LSM9DS1 sensor (does not allows to choose an USART for the SPI modules);
2. Since the USART is static, and the *data out* and *data in* lines from the FLASH and RAM devices are connected to the same shunted together (same line/pin), is not possible to communicate with different FLASHs or RAM at the same t
   However, since the the SPI for the LSM9DS1 sensor, is in another USART and the communication pins are different, it allows for concurrent communication with the Sensor and a FLASH /RAM (by creating 2 distinct SPI modules);

In Overall, this project was a great success, and very valuable for our inner development. As future steps, we indicate the construction of the FLASH driver using our module, and to solve the UART1 problem, in order to test the BN055 sensor.