Spirit Level - Project Report

Spirit Level detector with STM32F407 Discovery and external 8x8 Led Matrix controlled by a MAX7219.

898773 Sinico Matteo
898733 Taglia Andrea

POLITECNICO
MILANO 1863

Computer Science and Engineering
Master Degree

February 2018

Contents

Chapter 1

Introduction

1.1 Purpose

The purpose of the document is to describe the design and implementation process of a spirit level detector using the STM32F407 Discovery board and an external 8x8 Led Matrix controlled by a MAX7219.

1.2 Functionalities

The goal of the application is to realize an electronic version of the popular bulls eye level tool. This is achieved by a led matrix to give a visible level feedback, using an accelerometer to guess the device orientation. Basically our application has to

- collect data regarding the x-axis and the y-axis from the accelerometer;
- compute the data in order to translate them into an exact orientation position;
- plot this position on the led matrix;

The application should also be easily configurable in order to exploit different level of sensitivity and display update frequency (which actually depends on the data acquisition frequency).

1.3 Document Structure

1. Introduction: A brief description of the content and the purpose of the project;

2. Software Design: This section provides the design of the application. It contains UML diagrams needed to specify how the software is implemented and how it works;

3. Hardware Design: Here is described the hardware component we have used and how we have configured them;

4. Known Issue: The section describes known project issues;

5. References;

Chapter 2

Software Design

2.1 Code Structure

The project is developed entirely in C++. The code structure of the application is the following:

```
| ____ spirit-level/
    | ____ include/
        | ____ spirit_level.h
        | ____ IRQhandler.h
        | ____ lis3dsh.h
        | ____ lis3dsh_reg.h
        | ____ led_matrix_driver.h
    | ____ main.cpp
    | ____ Makefile
    | ____ miosix/
    | ____ miosix_np_2/
    | ____ README.md
    | ____ src/
        | ____ IRQhandler.cpp
        | ____ led_matrix_driver.cpp
        | ____ lis3dsh.cpp
        | ____ spi.cpp
        | ____ spirit_level.cpp
```

we followed a common path to organize the source code, that is separating source files from the header ones and leaving the other files in the main folder.

2.2 Class Diagram

In order to give a high level overview of the software involved, we show here the class diagram of our application. Below you can find more detailed description of the classes.

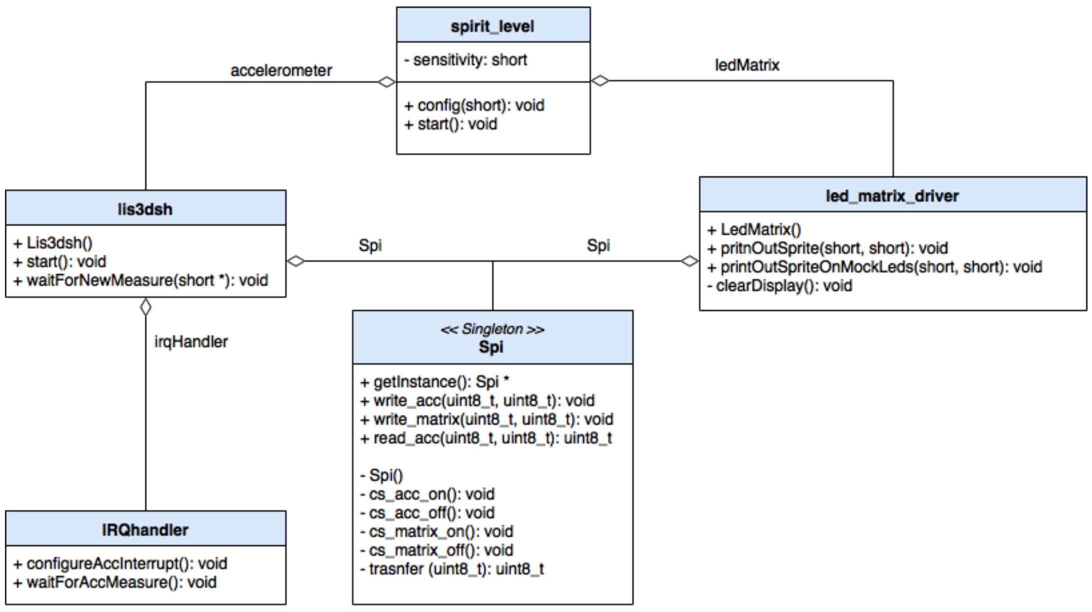

Figure 2.1: Class Diagram

2.2.1 Spirit_level

The spirit_level class is where the application logic is implemented. It has only one attribute, that is sensitivity which is set in the main.cpp file for quick configuration, and then passed to the config(short) method of the class.

Sensitivity is the minimum variation in acceleration (in milliG) which is recognized as a change in position. We use this attribute to compute the position of the board in a linear manner:

$$new_x = center_x - measure[X]/sensitivity;$$

where center_x depends on the width of the led matrix and measure[X] is the measure of the x-axis collected by the accelerometer.

config(*short sensitivity***)** simply assigns the sensitivity parameter set in the main.cpp file to the private attribute of the class, then it starts the accelerometer by calling its method start().

start() the function enters an infinite loop, waiting for new measures to be acquired from the accelerometer, computing the new position of the board in terms of x-axis and y-axis, after having eventually filtered some repeated measure, in order to implement a software anti-bounce, and then send the new positions to the led matrix.

2.2.2 Led_matrix_driver

Led_matrix_driver[1] is the driver to communicate with the led matrix, it is composed by three public methods and one private.

LedMatrix() is the constructor of the class and is the one responsible to configure the led matrix. Hardware chapter describes the necessary configuration which is then implemented in this method.

printOutSprite(*short x_position, short y_position***)** takes in input two shorts which are the positions to plot onto the led matrix, clears the display, and then prints them onto the led matrix. We have decided to print always a 2x2 square to make the position more visible.

printOutSpriteOnMockLeds(*short x_position, short y_position***)** does the same things of the previous one but, instead of printing on the led matrix, it exploits the four user controllable leds on the board to indicate the rotated axis, or the centered position which corriponds to all leds switched off. This has mainly debug purposes.

clearDisplay() clears the display of the led matrix by switching off each single led.

2.2.3 Lis3dsh

Lis3dsh is the driver to communicate with the on-board accelerometer[2], and it has three public methods.

Lis3dsh()is the constructor of the class and is the one responsible to configure the accelerometer. Again, see hardware chapter for detailed information about the configuration implemented here.

start() starts the accelerometer by configuring and enabling the correspondent interrupt.

waitForNewMeasure(*short * measure***)** each time a new measure is available reads it from the accelerometer register and return it in the array passed as input. Given the fact that data are expressed in 2s complement and

[1]Please refer to section 3.1 of the current document to have further details about the led matrix used

[2]Please refer to section 3.2 of the current document to have further details about the accelerometer used

divided into low and high part, before returning them, we have to concatenate them and convert into milliG.

2.2.4 IRQhandler

Each time a new measure is ready the accelerometer generates a new interrupt which is then managed by the irqHandler. It is composed by two public method and two in-line function.

configureAccInterrupt() configure the board to handle the accelerometer interrupt. This is done by programming the two trigger registers with the rising edge detection and by enabling the interrupt request by writing a 1 to the corresponding bit in the interrupt mask register. Finally configures the NVIC with the right channel and setting the priority to low.

waitForAccMeasure() puts the current thread in wait, until new data are available from the accelerometer.

EXTI0_IRQHandler() Is the method called by the accelerometer interrupt, performs the context switch and calls the actual implementation of the ISR, which is EXTI0HandlerImpl().

EXTI0HanldlerImpl() clean the pending bit of the interrupt controller, then if there is a waiting thread wakes it up and calls the scheduler to find the thread which has to execute next, which will be the application logic one.

2.2.5 SPI

Spi is a singleton class which configures the board to implement the SPI protocol. We have chosen to made it singleton because it is used to communicate both with the accelerometer and the led matrix, and given the fact we use the same hardware peripheral (SPI1[3]) we have to be sure that is instantiated only one time.

Spi() is the constructor of the class and is the one responsible to configure the spi.

getInstance() if already exist an instance of the spi return it, else generates a new one and return it.

cs_acc_on() drive low the cs bit connected to the accelerometer to start the communication with the accelerometer.

cs_acc_off() drive high the cs bit connected to the accelerometer to end the communication with the accelerometer.

[3] Please refer to section 3.4 of the current document to have further details about the hardware configuration of the spi protocol

cs_matrix_on() drive low the cs bit connected to the led matrix to start the communication with the led matrix.

cs_matrix_off() drive high the cs bit connected to the led matrix to start the communication with the led matrix.

transfer(*uint8_t data***)** waits for the transmitter buffer to be empty, writes the data received in input to the transmission buffer, wait for the reply and then return the reply.

write_acc(*uint8_t address, uint8_t data***)** transfer to the accelerometer the address where to write the data, transfer to the accelerometer the actual data

read_acc(*uint8_t address***)** transfers to the accelerometer the address where to read the data, performs a dummy write, waits for the reply with the data that he wants to read, return that data.

write_matrix(*uint8_t address, uint8_t data***)** transfers to the led matrix the address where to write the data, transfers to the led matrix the actual data

2.3 Run-time view

Below is a run-time view of the application logic, please note that to keep the diagram more readable we have omitted the configuration of both the accelerometer and of the led matrix, which takes place in the constructor of the two classes. They simply are a bunch of write performed as explained in chapter 3.

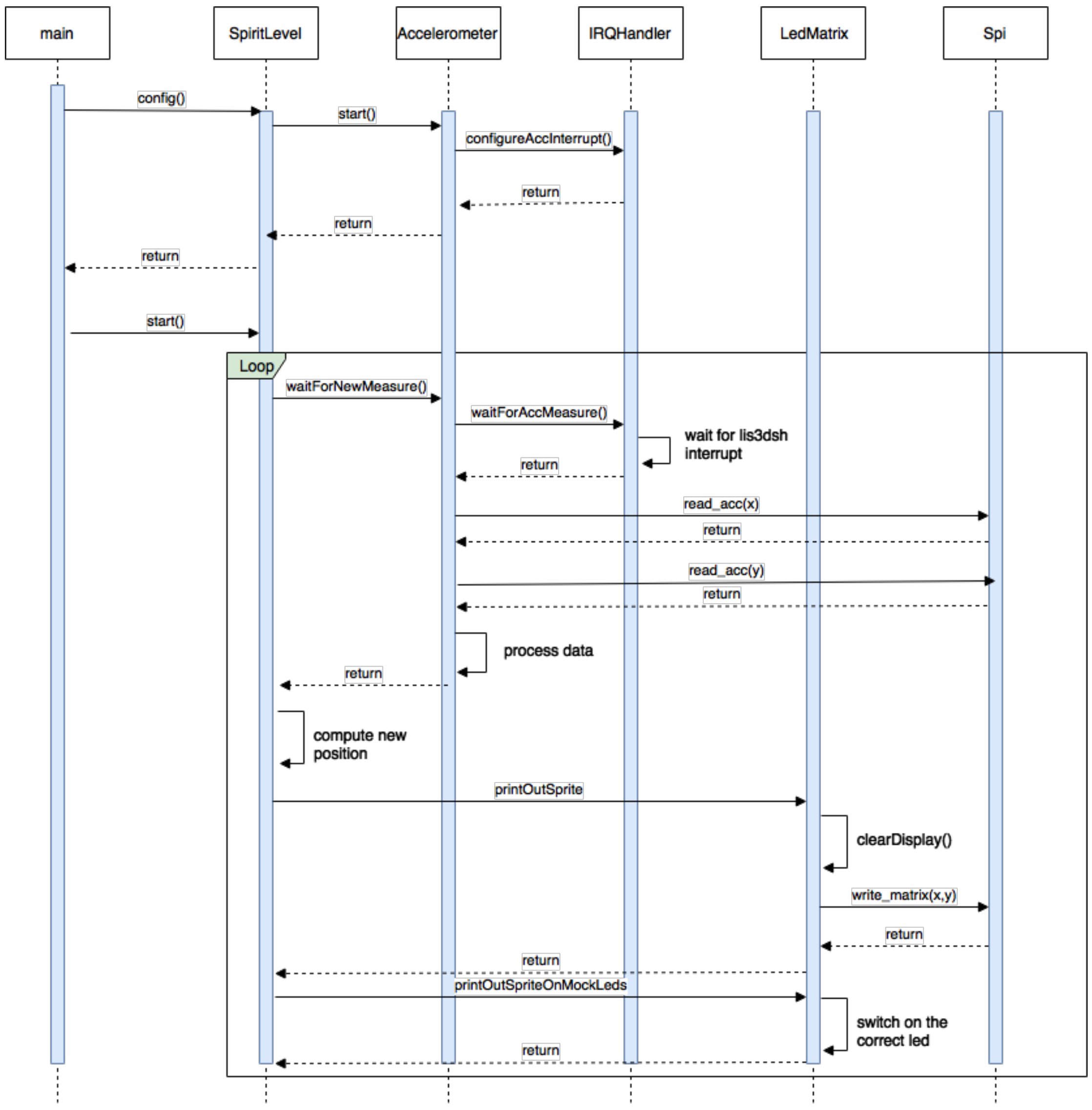

Figure 2.2: Run-time view

Chapter 3

Hardware Design

3.1 Overview

The section aims at giving an overview of the hardware components being used for the project, the way they have been configured and the way they talk to each other. Hardware components worth mentioning for the Spirit Level project are the following:

- Led Matrix
- MAX7219 display driver
- STM32F407 microcontroller
- LIS3DSH accelerometer

3.2 Led Matrix / MAX7219

We first give a brief description of the component MAX7219. To do this we use the General Description section from the datasheet[5] of the component itself, cutting out what not relevant to our purpose, and commenting what deserves more details. The MAX7219/MAX7221 are compact, serial input/output common-cathode display drivers that interface microprocessors (Ps) to 7-segment numeric LED displays of up to 8 digits, bar-graph displays, or 64 individual LEDs. Included on-chip are a segment and digit drivers, and an 8x8 static RAM that stores each digit. Only one external resistor is required to set the segment current for all LEDs. A convenient 3-wire serial interface connects to all common Ps. Individual digits may be addressed and updated without rewriting the entire display. The MAX7219/MAX7221 also allow the user to select code B decoding or no-decode for each digit. The devices include a 150A low-power shutdown mode, analog and digital brightness control, a scan- limit register that allows the user to display from 1 to 8 digits, and a test mode that forces all LEDs on.

On the other side, the 8x8 common-cathode LED display (referred as Led Matrix in this document) is controlled through the MAX7219 driver. The communication between the two is something out of the scope of this document, as they will be seen as a single component to the microcontroller. Indeed, from now on we will just refer to the MAX chip.

The MAX7219 has 14 addressable registers, 5 of which are control registers, and the remaining ones are the digits which will used to set the leds values. We now go through all the relevant control registers which need to be set in order to make it behaves the way we intend (described in previous sections):

- Display Test register (address 0xXF) needs to be set to 0x00 in order to turn off the "all-leds-on" debug feature;
- Scan Limit register (address 0xXB) needs to be set to 0x07 in order to enable all the digit registers;
- Led Intensity register (address 0xXA) indicates the intensity of the leds on the matrix. The value indicates a certain duty cycle. We opted for a mid-level intensity of 0x07;
- Decode Mode register (address 0xX9) needs to be cleared in order to allow free selection of the single 64 leds with no decoding in the middle;
- At last the Shutdown Mode register (address 0xXC) needs to be set to 0x01 to make the device fully operative;

From this point on a write into the digit registers (from 0xX1 to 0xX8) will set the relative display segment line to have the leds turned on or off based on the value of the registry.

The interface exposed by the chip is made of five pins:

1. 5V Vdd
2. Gnd
3. Data In
4. Chip Select
5. Serial Clock

Vdd and Gnd will be trivially linked to the relative board pins. The other three pins will be used for the SPI communication which will be tackled further in chapter 3.3.

3.3 LIS3DSH Accelerometer

The LIS3DSH[3] is an ultra-low-power high-performance 3-axis linear accelerometer belonging to the nano family. It has dynamically user-selectable full scales of 2g/4g/6g/8g/16g and is capable of measuring accelerations with output data rates from 3.125 Hz to 1.6 kHz. We find this piece of hardware directly mounted on the discovery board. The use we make of the accelerometer is pretty simple and it doesnt take advantage of the state machines which could be configured for certain motion recognition. The accelerometer acquires measures about the X and Y axis and it sends back to the microcontroller at a certain (software configurable) frequency. Once new data is ready an interrupt tells the microcontroller that can now read the new values, which will be placed in the registers OUT_X_L, OUT_X_H, OUT_Y_L, OUT_Y_H (from 0x28 to 0x2B). In order to configure the accelerometer to work this way the following are needed:

- Setting XEN and YEN from CTRL_REG4 (address 0x20) we enable the X and Y axis;
- Still from the CTRL_REG4 the frequency of the measures is set through the ODRx[7:4];
- INT1_EN from CTRL_REG3 (address 0x23) needs to be set in order to enable Interrupt 1 on DRDY pin;
- Data-ready interrupt must be enabled and routed to INT1, and this is done by setting the DR_EN;
- Then IEA also is set to have interrupt signal active high.

LIS3DSH is now fully working and it keeps acquiring new data. For the system to be fully working it just takes the interrupt handler to be properly configured on the microcontroller to listen for the interrupt signal generated from the accelerometer.

3.4 STM32F407 microcontroller

The microcontroller at the heart of the board will be the man in the middle between the accelerometer and the led matrix. The way the three parts talk to each other is through the common Serial Peripheral Interface. What we care about right now is to have a look at the way it must be set up in order to have everything ready to handle the work.

We first take a look at the interfaces exposed by the lis3dsh and the max7219 in order to set up the GPIO that will be used for communication. For the majority of them there is no choice which one to choose as they are hard wired to the accelerometer. On the other side we can freely chose the ones for the led matrix as it is an external components. However, since the same SPI1 peripheral is used to communicate to both we are forced to keep the common pins for both the devices, that is all of them except for the Chip Select on the led matrix.

We thus have a look at the electrical schematics on the board user manual[1] to see where the MEMS are hard wired:

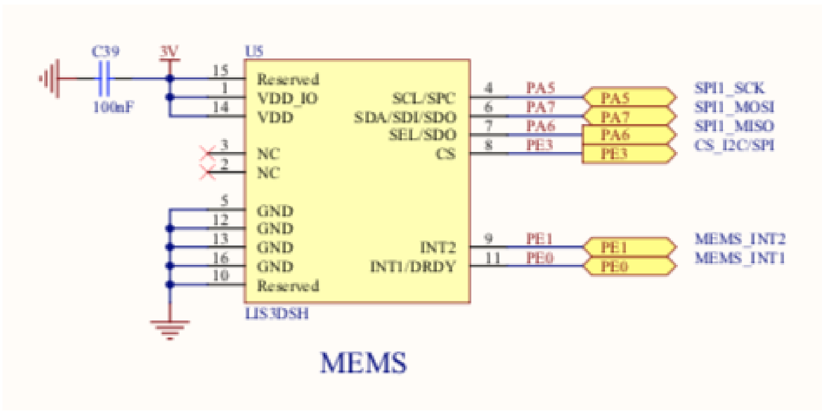

Figure 3.1: MEMS

Indeed, it takes the following to be properly configured on the microcontroller:

- Serial Clock on PA5 set to alternate function 5 (SPI1 peripheral) in order for the hardware to take its control.This is done setting GPIOA_MODER5 (address offset 0x00) to 10 and the GPIOA_AFRL5[3:0] (address offset 0x20) to 0101. This will also go to Serial Clock pin on the led matrix (pin 5);

- Master In Slave Out on PA6;

- Master Out Slave In on PA7. This also goes to Data In on the led matrix (pin 3);

- Chip Select for the accelerometer on PE3 must be set to output in order to be software controlled;

- Instead, Chip Select for the led matrix will be up to us as it is just going to be a software controlled GPIO which will then be hard wired to the CS pin on the MAX7219. We choose to set on PE4. This will be wired to the CS pin on the led matrix (pin 4);
- Since Port A and Port E will be used, they need to be clocked. This is done by setting bits GPIOAEN and GPIOEEN in register RCC→AHB1ENR.

SPI1 is the peripheral which will be used to talk to both the Led Matrix and the Accelerometer. To configure the peripheral as a master we go through the following:

- First of all, clock must be enabled on the peripheral as usual for the STM32. This is achieved by setting the bit SPI1EN on the RCC→APB2ENR register;
- Then we reset the RCC→APB2RSTR register by setting the SPI1RST[12] bit;
- In the SPI_CR1, set the BR[2:0] bits to define the serial clock baud rate. Note that the following will refer to SPI_CR1 register unless explicitly stated;
- Then clear CPOL and CPHA bits to define sampling on clock rising edge and idle clock low;
- Clear the DFF bit to define 8-bit data frame format;
- Clear the LSBFIRST bit to have Most Significant Bit sent first;
- Since NSS will be controlled in software mode, we set the SSM[9] and SSI[8] bits in the SPI_CR1 register;
- Clear the FRF bit in SPI_CR2 to select the standard Motorola protocol over TI protocol for serial communications;
- Finally, the MSTR and SPE bits must be set in the SPI_CR1 register in order to set master mode and to enable the peripheral.

All is left now is the configuration of the Interrupt handler. We expect the signal to come from the INT1 of the accelerometer, which is hard wired to the PE0. In the previous paragraph 2.2.4 configureAccInterrupt function briefly describes what is needed here, so we wont dig any deeper in the topic.
The whole hardware is ready to work the intended way.

Chapter 4

Known Issues

4.1 Issues Description

While we were implementing our application we found that if we change the order in which we instantiate the two class ledMatrix and Lis3dsh, hence the order in which the two hardware components are configured, the application doesnt work properly. In particular if we first configure the accelerometer instead of the led matrix, then only the accelerometer works. To dig down into the issue we went on with the logic analyzer to find out what we were sending out and receiving from the SPI pins. Here are the screenshots of the first bytes of communication while configuring the two devices:

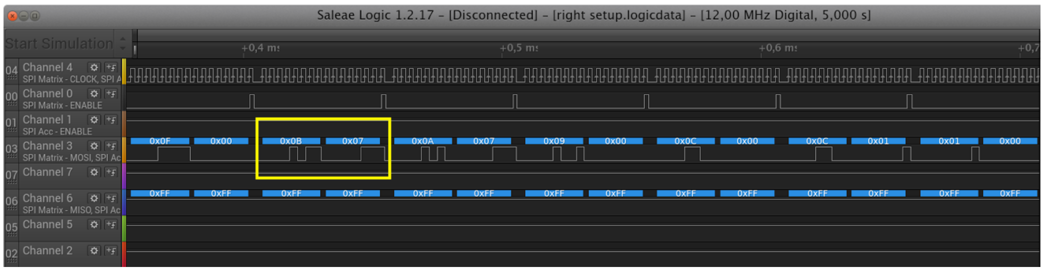
Saleae Logic 1.2.17 – [Disconnected] – [right setup.logicdata] – [12,00 MHz Digital, 5,000 s]
Start Simulation
+0,4 ms
+0,5 ms
+0,6 ms
+0,7
04 Channel 4
SPI Matrix - CLOCK, SPI A
00 Channel 0
SPI Matrix - ENABLE
01 Channel 1
SPI Acc - ENABLE
03 Channel 3
SPI Matrix - MOSI, SPI Ac
07 Channel 7
06 Channel 6
SPI Matrix - MISO, SPI Ac
05 Channel 5
02 Channel 2
0x0F 0x00 0x0B 0x07 0x0A 0x07 0x09 0x00 0x0C 0x00 0x0C 0x01 0x01 0x00
0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

Figure 4.1: right setup (led-matrix first)

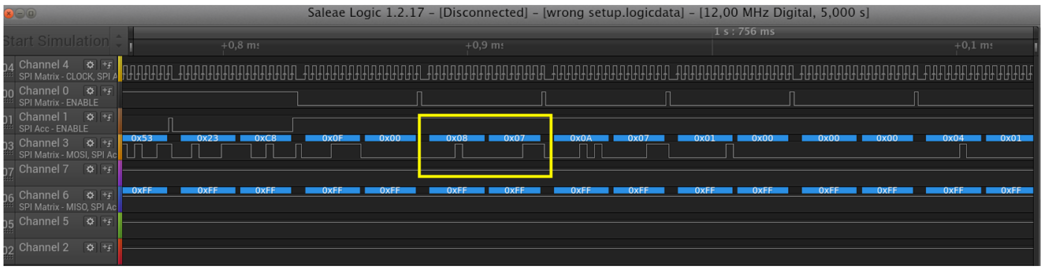

Figure 4.2: wrong setup (accelerometer first)

If the accelerometer is configured before the led matrix, we lose some bits during the configuration of the led matrix as it can be seen in the yellow boxes. Those missed bits make the led matrix not configuring properly, making the led matrix not well set for use. Only the onboard leds would work. This issue has been found empirically and no clear explanation could be pointed out.

References

[1] UM1472 discovery board user manual, link

[2] RM0090 microcontroller reference manual, link

[3] LIS3DSH accelerometer datasheet, link

[4] AN3393 accelerometer application note, link

[5] MAX7219 datasheet, link

[6] TN0897 Techincal Note ST Spi Protocol, link

[7] Advanced Operating Systems course slides, link