Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics

Department of Automation and Applied Informatics
Embedded and control systems specialization

Embedded and control systems laboratory

documentation

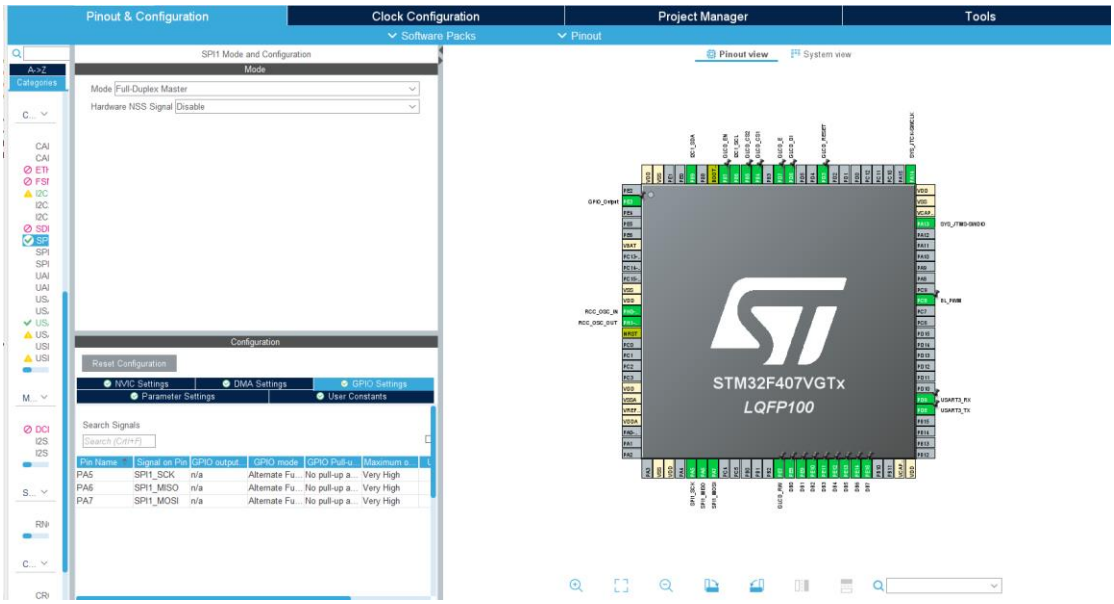
[Using the STM32 communication peripherals 2]

Student names: [Das Anjan Kumar] [D42DQA]
[Yousha Mohammed Shamin Yeasher] [WBDDO4]
Measurement group: [03]
Time of measurement: [2022.04.26]
Place of measurement: BME AUT, lab: QB127
Measurement Instructors: [Kiss Domokos]

Laboratory tasks

Task 4: SPI part 1-1:

In this task we will be able to configure the SPI interface from the MARCOM file according to the lab guide. After that we will build the marcom file and generate the spi.c file.



After that we will go tot he main file and add the spi.h as header file and initialize the spi interface by the following command:

```
HAL_StatusTypeDef SPI_Init()
{
	MX_SPI1_Init();

	return HAL_OK;
}
```

Task 4: SPI part 1-2:

In this task we will set the PE3 GPIO pin by using the function given in the instruction. We will use a condition for when enable is 1 we will reset the pin and when enable is 0 we wil SET the pin. The necessary piece of code is given below.

CODE:

```
void SPI_EnableChip(int enable)
{

	if(enable == 1)
	    HAL_GPIO_WritePin(GPIOE, GPIO_PIN_3 , GPIO_PIN_RESET);
	else
		HAL_GPIO_WritePin(GPIOE, GPIO_PIN_3 , GPIO_PIN_SET);
}
```

Task 4: SPI part 1-3:

In this task we will be sending data over spi by using the given functions in the intruction.

For Transmission

```
1 HAL_SPI_Transmit(SPI_HandleTypeDef * hspi, uint8_t * pData, uint16_t Size, uint32_t Timeout);
```

Tod o that we will use the following part of the code:

```
HAL_StatusTypeDef SPI_Send(uint8_t* pData, uint16_t dataSize)
{

	HAL_SPI_Transmit( &hspi1, pData, dataSize, spi_timeout);

	return HAL_OK;
}
```

Task 4: SPI part 1-4:

In this task we will be receiveing data over spi by using the given functions in the intruction.

For Reception

```
1 HAL_SPI_Receive(SPI_HandleTypeDef * hspi, uint8_t * pData, uint16_t Size, uint32_t Timeout);
```

The used code for doing this part is as below:

```
HAL_StatusTypeDef SPI_Receive(uint8_t* pData, uint16_t dataSize)
{

	HAL_SPI_Receive(&hspi1, pData, dataSize, spi_timeout);

	return HAL_OK;
}
```

Task 4: SPI part 1-5:

In this part we will be sending and receiving the data at the same time using the given function in the lab guide:

For Transmit-Receive

```
1 HAL_SPI_TransmitReceive(SPI_HandleTypeDef * hspi, uint8_t * pTxData, uint8_t * pRxData, uint16_t
```

Code to complete this part :

```
HAL_StatusTypeDef SPI_SendReceive(uint8_t* pDataIn, uint8_t *pDataOut,
uint16_t dataSize)
{


	HAL_SPI_TransmitReceive &hspi1, pDataIn, pDataOut, dataSize,
spi_timeout);
```

```
	return HAL_OK;
}
```

Task 4: SPI part 2-1:

In this part we will be working ont he bsp_accemerometer.c file. In this part we will send the write address and the actual dsata. First we will enable the chipset and then send the writeaddress and then the actual data and after that we will send the disable command.

The code to accomplish that :

```
void LIS302DL_Write(uint8_t* pData, uint8_t WriteAddr, uint16_t dataSize)
{
	if(dataSize > 0x01)
	{
		// In case of writing multiple bytes, the first byte has to be
modified
		// in order to signalize multibyte command
		WriteAddr |= (uint8_t)MULTIPLEBYTE_CMD;
	}


    SPI_EnableChip(1);

    SPI_Send(&WriteAddr, 1);

    SPI_Send(pData, dataSize);

    SPI_EnableChip(0);


}
```

Task 4: SPI part 2-2:

In this task we are again doing the same task but receiving the data we have sent previously.

The Code:

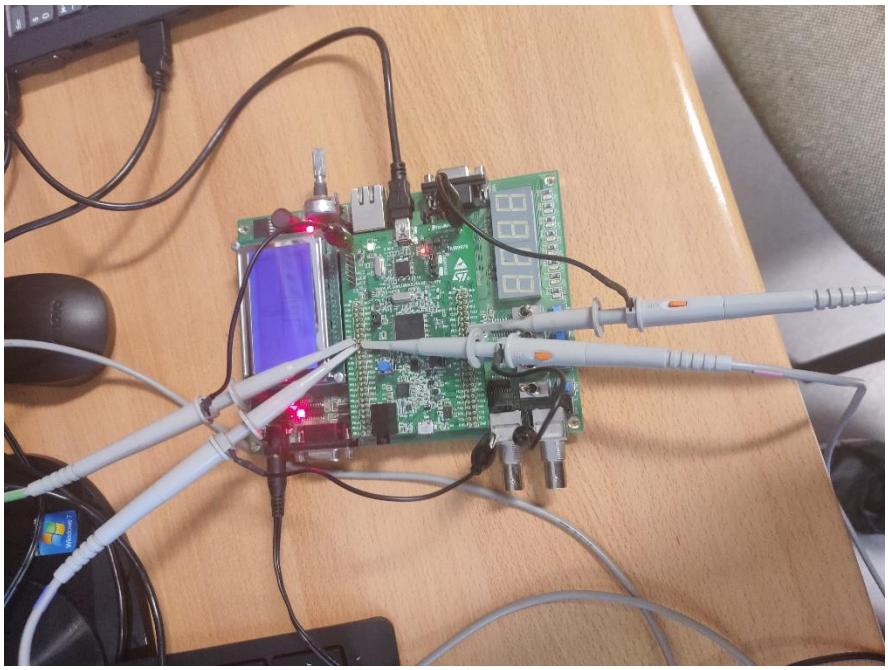
```
void LIS302DL_Read(uint8_t* pData, uint8_t ReadAddr, uint16_t dataSize)
```

```
{
	// In the first byte the reading operation and optionally multibyte
transfer are signalized
	if(dataSize > 0x01)
	{
		ReadAddr |= (uint8_t)(READWRITE_CMD | MULTIPLEBYTE_CMD);
	}
	else
	{
		ReadAddr |= (uint8_t)READWRITE_CMD;
	}


    SPI_EnableChip(1);

    SPI_Send(&ReadAddr, 1);

    SPI_Receive(pData, dataSize);

    SPI_EnableChip(0);


}
```

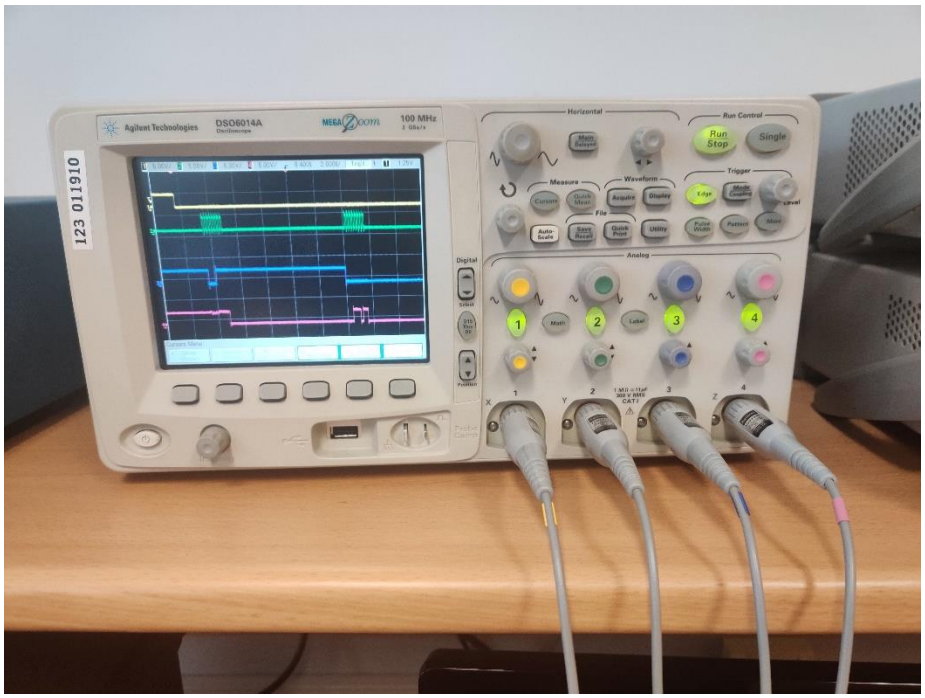
After we are done with this process we can read various sort of readings with the microcontroller i.e. tilt, temp etc.
A screen shot is given below for clarification:

```
Repeat mode ON
>>> LocalZoneID: 0 | Temp: 26 | Tilt:  2 | Zones: (0) 0 0 0 <<<
>>> LocalZoneID: 0 | Temp: 26 | Tilt:  2 | Zones: (0) 0 0 0 <<<
>>> LocalZoneID: 0 | Temp: 26 | Tilt:  2 | Zones: (0) 0 0 0 <<<
Local zone (0) state: 2
>>> LocalZoneID: 0 | Temp: 26 | Tilt:  6 | Zones: (2) 0 0 0 <<<
Local zone (0) state: 0
>>> LocalZoneID: 0 | Temp: 26 | Tilt:  2 | Zones: (0) 0 0 0 <<<
Local zone (0) state: 4
>>> LocalZoneID: 0 | Temp: 26 | Tilt:  1 | Zones: (4) 0 0 0 <<<
Local zone (0) s 0
>>> LocalZoneID: 0 | Temp: 26 | Tilt:  2 | Zones: (0) 0 0 0 <<<
Local zone (0) s 4
>>> LocalZoneID: 0 | Temp: 26 | Tilt:  2 | Zones: (4) 0 0 0 <<<
Local zone (0) state: 0
>>> LocalZoneID: 0 | Temp: 26 | Tilt:  3 | Zones: (0) 0 0 0 <<<
GRH: Mode has changed.
GRH: Mode has changed.
Local zone (0) s 4
>>> LocalZoneID: 0 | Temp: 26 | Tilt:  3 | Zones: (4) 0 0 0 <<<
Local zone (0) state: 0
>>> LocalZoneID: 0 | Temp: 26 | Tilt:  2 | Zones: (0) 0 0 0 <<<
Local zone (0) s 4
>>> LocalZoneID: 0 | Temp: 26 | Tilt:  4 | Zones: (4) 0 0 0 <<<
Local zone (0) s 0
>>> LocalZoneID: 0 | Temp: 26 | Tilt:  1 | Zones: (0) 0 0 0 <<<
>>> LocalZoneID: 0 | Temp: 26 | Tilt:  2 | Zones: (0) 0 0 0 <<<
>>> LocalZoneID: 0 | Temp: 26 | Tilt:  2 | Zones: (0) 0 0 0 <<<
>>> LocalZoneID:0 | Temp: 26 | Tilt:  2 | Zones: (0) 0 0 0 <<<
>>> LocalZoneID: 0 | Temp: 26 | Tilt:  2 | Zones: (0) 0 0 0 <<<
```

In this part of the experiment we will be seeing the signals created by the master in the oscilloscope. Oscilloscope configuration for the micrcontroller:

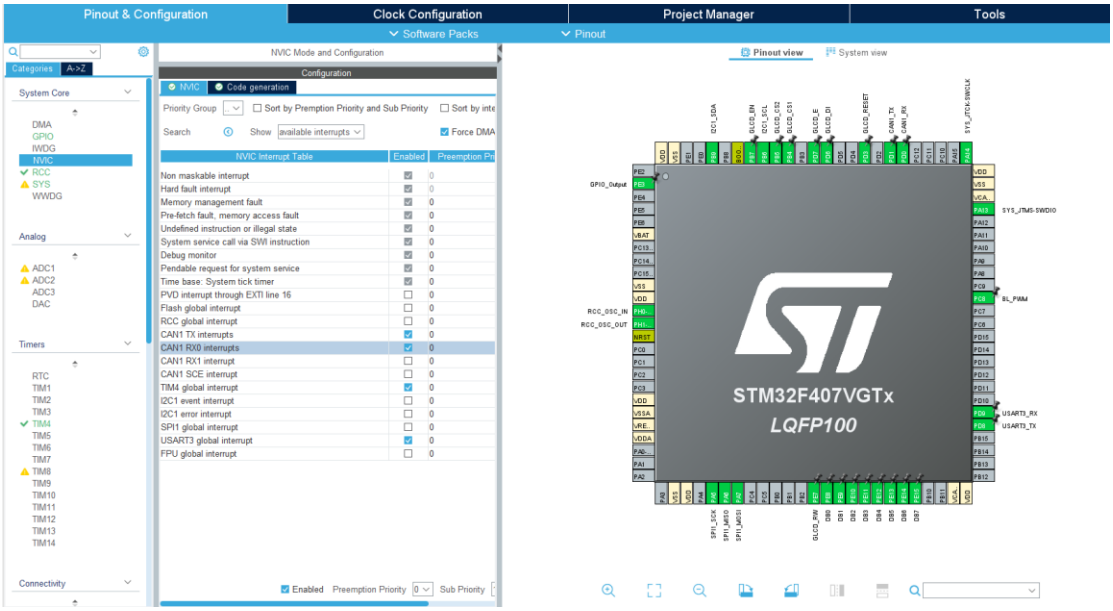


In the oscilloscope we can see the signals from the master and the sent data and also the signals for the slave signals as well.



Task 5: CAN part 1-1:

In this task we will be working with a CAN interface communication.
First we will work with the marcom file and the specification as in the guide and set the configuration and generate the code.



After that we will work on the initialization of the can interface. we can easily do it by the following code:

```
MX_CAN1_Init();
```

After that we will write a code for use message filter. Tod o that we will define a variable of type can_filtertypedef and after hat we will set the variable values and after that we will retunr a value if the operation was unsuccessful.

Code:

```
CAN_FilterTypeDef can_filter;

can_filter.FilterBank = 0;
can_filter.SlaveStartFilterBank = 0;
can_filter.FilterActivation = ENABLE;
can_filter.FilterFIFOAssignment = CAN_FILTER_FIFO0;
can_filter.FilterMode = CAN_FILTERMODE_IDMASK;
can_filter.FilterScale = CAN_FILTERSCALE_32BIT;
can_filter.FilterIdLow = 0;
can_filter.FilterIdHigh = 0;
can_filter.FilterMaskIdLow = 0;
can_filter.FilterMaskIdHigh = 0;
```

```
	HAL_StatusTypeDef check_error;

	check_error = HAL_CAN_ConfigFilter(&hcan1, &can_filter);



	if(check_error != HAL_OK)
	{
		Log_LogStringAndHalStatus("Unsuccessful\n\r", LOGLEVEL_NORMAL,
check_error);
		return check_error;

	}


	// After successful configuration, start the CAN periphery using
function HAL_CAN_Start()
	// Allow the "message pending" interrupt for FIFO0 using the
function HAL_CAN_ActivateNotification()


	HAL_CAN_Start(&hcan1);

	HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO0_MSG_PENDING);



	// Finally print to the Log whether CAN_Init was successful.
	return check_error;
```

Task 5: CAN part 1-2:

In this task we will send message using CAN interface. We can write the following code to send the message accross the CAN interface:

```
HAL_StatusTypeDef CAN_SendMessage(uint8_t zoneID, uint8_t newStatus)
{



	if (hcan1.State != HAL_CAN_STATE_READY &&
HAL_CAN_GetTxMailboxesFreeLevel(&hcan1) == 0 )

		return HAL_BUSY;
```

```
            txHeader.DLC = 2;
            txHeader.IDE = CAN_ID_STD;
            txHeader.RTR = CAN_RTR_DATA;
            txHeader.StdId = 0x321;
            txBuffer[0] = zoneID;
            txBuffer[1] = newStatus;
```

```
      uint32_t pTxMailbox = 0;
      HAL_StatusTypeDef MessageAddress = HAL_CAN_AddTxMessage(&hcan1,
&txHeader, txBuffer, &pTxMailbox);
```

```
      // In case of success, log the successful message sending.

      if(Status == HAL_OK)
      {
            Log_LogStringAndHalStatus("Successful\n\r", LOGLEVEL_NORMAL,
Status);

      }


      return HAL_OK;
}
```

Task 5: CAN part 1-3:

In this task we will write the necessary code for `HAL_CAN_RxFifo0MsgPendingCallback.`

The function implimentation:

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
{


      HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO0 , &rxHeader, rxBuffer);

      if(rxHeader.StdId == 0x321 && rxHeader.DLC == 2)
ProcessReceivedCanMessage(rxBuffer[0], rxBuffer[1];)
}
```

In the end after the successful implimentation of CAN tasks, we can see that when we change in one STM board, the change can be seen across all the other connected microcontrollers.