Networked Embedded Systems

Practicum 4: Interrupts and DMA

Group number: 8

Name	Student ID	Email
Bernberger Sarah	k12112018	sarahbernberger.sb@gmail.com
Grundner Simon	k12136610	simon.grundner@gmail.com

18.06.2024

Map of Content

Theory Questions

What is DMA? How does it work?

DMA is short for “Direct Memory Access” and is a core component of a microcontroller to ensure constant data flow between a peripheral and the memory, meaning that data acquisition is not interrupted by CPU operations such as reading or processing the data. Data can be quickly moved by the DMA without any CPU action. This keeps the CPU resources free for other operations, which is especially important for time critical signals like audio real time sensor data.

Processing without a DMA-Controller

- CPU is constantly polling the peripheral
- Minimal processing time for other tasks
- Data provided by the peripheral during other processes is lost
- No consistent sampling period possible

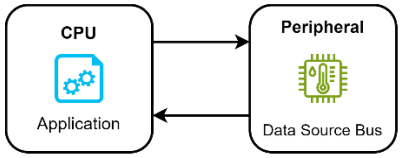

Figure 1: Interconnection

Processing with a DMA-Controller

- Stream samples directly into a queue
- Works independently from the CPU-Core
- Data loss only occurs when the FIFO-Buffer overflows

An application where the data transfer between memory and the peripheral is handled by a DMA-Controller would be setup as shown in Figure 2. Note that RX- and TX-Operation are handled by a different DMA-Stream with different queues.

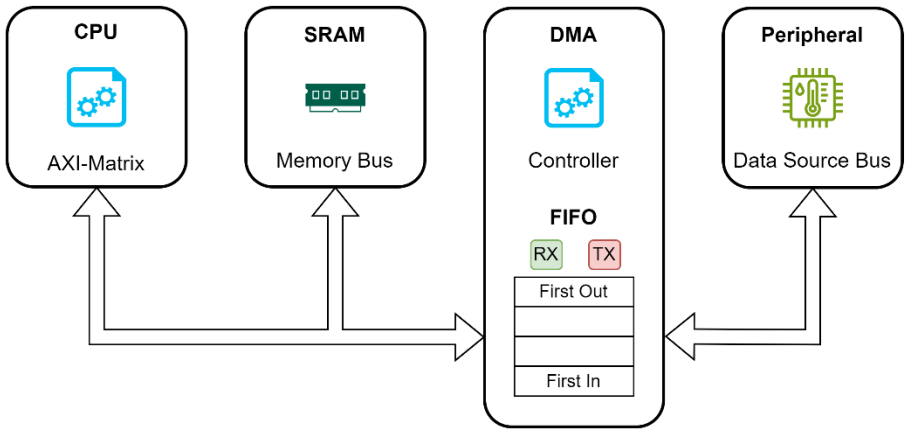

Figure 2: Simplified DMA Interconnection Diagram

Setting up DMA

A problem with the DMA controllers is that, although they can access almost all memory units, they cannot access the TCM[1]-RAM, which is dedicated to the CPU. Therefore, a memory location accessible to both the CPU and the DMA controllers must be chosen. The most convenient memory unit for this purpose is AXI SRAM in D1. The connection is made through the Advanced Extensible Interface Matrix (AXI-M). This structure links the CPU to the Inter-Domain Buses (AHB), where the DMA controller and the Peripheral Buses (APB) are connected to as well.

The most important connections are highlighted in Figure 3. These connections are established by a bus multiplexer.

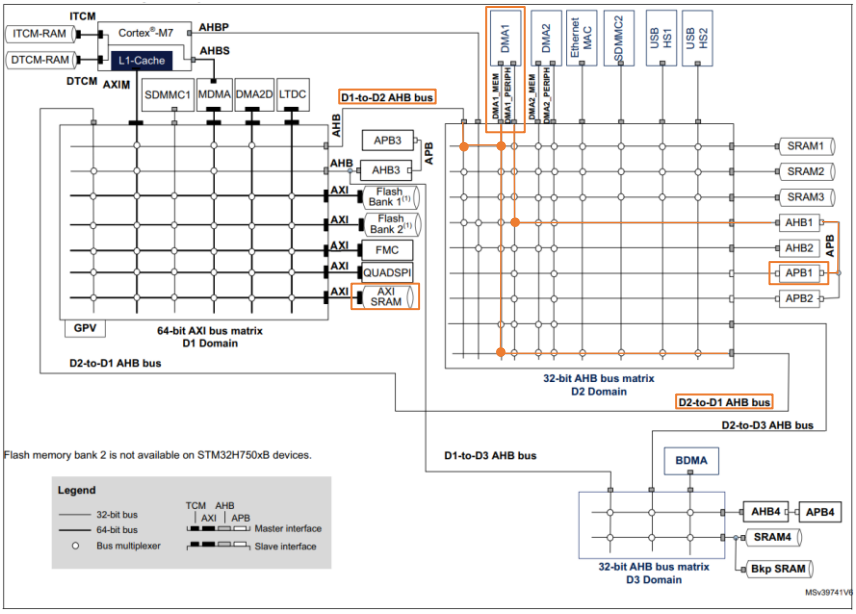

Figure 3: Bus Interconnection Matrices

A simplified diagram shows how the CPU is connected to the internal SRAM.

Furthermore, the CPU contains two internal caches, I-Cache for loading instructions and D-Cache for data. The D-Cache can affect the functionality of DMA transfers, since it will hold the new data in the internal cache and do not write them to the SRAM memory. However, the DMA controller loads the data from SRAM memory and not D-Cache. Therefore, the D-Cache is globally disabled in Cube MX under:
System Core > CORTEX_M7 > Cortex Interface Settings.

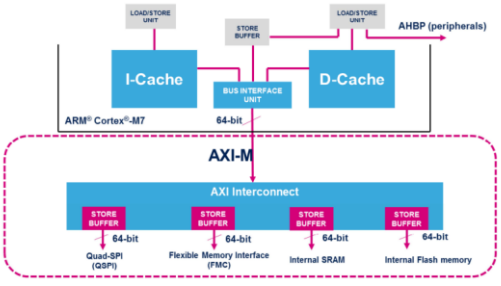

Figure 4: Simplified AXI Interconnection description

[1] *Tightly Coupled Memories (TCM) are dedicated memories directly connected to the processor, but not through a bus, so avoiding arbitration and latencies for frequently executed code.*

The change of the memory unit requires changes made to the linker script to configure the memory location to be in the SRAM in D1. `DTCMRAM` must be replaced with `RAM_D1` for the following sections in *STM32H743XIHx_FLASH.ld* file:

- *.data*
- *.bss*
- *._user_heap_stack*

```
.data :
{
  . = ALIGN(4);
  _sdata = .;        /* create a global symbol at data start */
  *(.data)           /* .data sections */
  *(.data*)          /* .data* sections */
  . = ALIGN(4);
  _edata = .;        /* define a global symbol at data end */
} >DTCMRAM AT> FLASH
```

Code Segment 1: .data field in linker script before change

```
.data :
{
  . = ALIGN(4);
  _sdata = .;        /* create a global symbol at data start */
  *(.data)           /* .data sections */
  *(.data*)          /* .data* sections */
  . = ALIGN(4);
  _edata = .;        /* define a global symbol at data end */
} >RAM_D1 AT> FLASH
```

Code Segment 2: .data field in linker script after change

The *_estack* variable in the linker script contains the address for the end of the stack. The default memory address, which is 0x20020000, is composed of the origin plus the length of the DTCMRAM. Since the TCM is no longer used, this field must be changed as well.

Update the section `.user_heap_stack` as follows:

```
._user_heap_stack :
{
  . = ALIGN(8);
  PROVIDE ( end = . );
  PROVIDE ( _end = . );
  . = . + _Min_Heap_Size;
  . = . + _Min_Stack_Size;
  _estack = .; /* <<<< line added */
  . = ALIGN(8);
} >RAM_D1
```

Code Segment 3: ._user_heap_stack in linker script

The end-of-stack variable is now set in this section to the dot-variable value at the shown location. The special linker dot-variable '.' always contains the current output location counter.

NOTE: The linker script will be reset when the code is regenerated by Cube MX. Make sure to keep a copy before editing the IO-Configuration.

What are the differences between Interrupt and DMA mode?

Interrupt Mode

Setting up an interrupt eliminates the need to repeatedly check if a peripheral is ready. Instead, whenever the corresponding interrupt flag is set, a **callback function** is executed to handle an operation when the peripheral sends or receive data.

In contrast to Practicum 3, where the status register content of the sensor device was requested in each while-loop cycle, causing unnecessary traffic on the I2C bus, interrupts ensure that only necessary transmissions are made. This is particularly beneficial when multiple devices are connected to the bus.

With an interrupt-based implementation the CPU is signaled every time a byte is sent and must provide new data. This means, that other processes must wait until the transmission is executed.

DMA Mode

The implementation with DMA looks similar to the interrupt driven method with the difference, that the DMA-Controller fully handles the data transfer. This makes the transmission a non-blocking operation and the CPU is only signaled if a transfer is complete instead of for every Byte sent. The CPU can simply provide data and send it onto a buffer without needing to wait until a word is transmitted. Individual DMA-Transfers do not involve the Processor and are generally faster.

Task A: Blink an LED with External Interrupts

This task is similar to the task D of practicum 1 but now you are required to use external interrupts (EXTI) on PJ12 and PJ13 to detect user inputs. You should also use the timer for the blinking cycle (as introduced in practicum 2). The EXTI line (connected to PJ12) should be used to increase the cycle time while the EXTI line (connected to PJ13) should be used to decrease the cycle time. The LED located near the 7-Segment display should be used for digital output. The initial value of the blinking 2 cycle time should be 100 ms. The range of the blinking cycle time should be between 100 and 1000 ms. The respective interrupts either increase or decrease the current blinking cycle time by 100 ms. Pressing the buttons should update the blinking frequency with immediate effect.

Requirements:

- Minimum cycle time = **100 ms**
- Maximum cycle time = **1000 ms**
- Use **PJ7 (SEGDP)** as digital output
- Use **PJ12 (BTN1)** and **PJ13 (BTN2)** to generate external interrupts
- One step = **100 ms**

A.1. Implementation

Helpful constants are defined in the private define section. The main function only starts the timer in interrupt mode.

```
#define MIN_STEP_MS 100
#define MAX_STEP_MS 1000
#define STEPSIZE_MS 100

int main(void)
{
  HAL_TIM_Base_Start_IT(&htim2);
  while (1) {}
}
```

The Timer 2 callback function executed by an overflow interrupt is implemented, so that if the ARR of Timer 2 overflows, the decimal point LED of the segment display is toggled.

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  if (htim->Instance == TIM2)
  {
    HAL_GPIO_TogglePin(SEGDP_GPIO_Port, SEGDP_Pin);
  }
}
```

Code Segment 4: Task A – Timer Callback

The callback function of the external interrupt handles the recalculation of the timer period. A so called local-static variable `steps_ms` is declared to preserve the current timestep. Using the `static` keyword has the effect of keeping the variable alive even after the function returns. The variable is only initialized **once** at the first call of the function. A switch-case statement checks, which button caused the interrupt and in- / decrements the current step. In any case the ARR of TIM2 is updated using Code Segment 6.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  // use local static variable to preserve the current step_ms value
  static uint16_t step_ms = MIN_STEP_MS;

  switch (GPIO_Pin)
  {
  case BTN1_Pin: // increase the step_ms value
    step_ms += STEPSIZE_MS;
    if (step_ms > MAX_STEP_MS)
      step_ms = MAX_STEP_MS;
    break;

  case BTN2_Pin: // decrease the step_ms value
    step_ms -= STEPSIZE_MS;
    if (step_ms < MIN_STEP_MS)
      step_ms = MIN_STEP_MS;
    break;

  default:
    break;
  }

  // update the TIM2 ARR register with the new step_ms value
  TIM2->ARR = MS_TO_COUNTERPERIOD(step_ms);
}
```

Code Segment 5: Task A - EXTI Callback

Timer Setup

For this Task Timer 2 is chosen, since its 32-Bit counter register width comes in handy for calculations. The value for the Auto Reload Register is calculated as follows:

$$ARR = T_{OFL} \cdot \frac{\mathrm{f_{TIM}}}{\mathrm{PSC}}$$

To achieve convenient ARR values for a counter period of $T_{OFL} = 0{,}1s$, a prescaler value of $PSC = 75$ has been chosen. With a timer count frequency of $f_{TIM} = 75MHz$, this prescaler ensures a simple conversion from a counter period in seconds to a counter period in steps, since $0{,}1s \mathrel{\widehat{=}} 100000steps$.

$$ARR = 0{,}1s \cdot \frac{75\,000\,000Hz}{75} = 100\,000$$

During runtime, this formula is simplified and applied by a macro:

```
// PS 75, f_tim = 75 MHz -> 1 ms ^= 1000 steps
#define MS_TO_COUNTERPERIOD(x) ((x) * 1000)
```

Code Segment 6: Convert Milliseconds to a Counter Period Value

For the callback to be executed by the IRQ handler, the Timer 2 interrupt must be enabled beforehand in Cube MX.

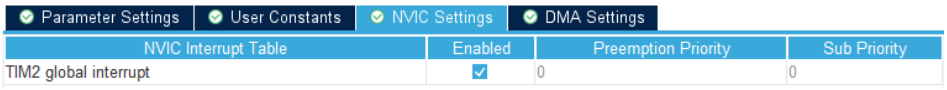

Figure 5: Enable TIM2 Interrupt

Interrupt Setup

To enable interrupts for button inputs the `GPIO_EXTIxx` option must be selected for each input pin. Furthermore, the EXTI interrupts must also be checked in the NVIC interrupt table.

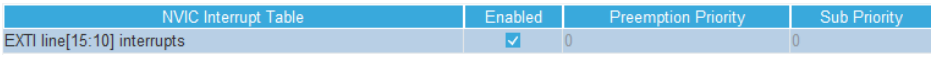

Figure 7: Enable EXTI interrupts for GPIOs

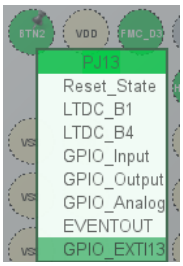

Figure 6: Set GPIO as external inter-

A.2. Discussion

This implementation has two advantages over the solution in task D of practicum 1. Both operations are performed in a non-blocking manner. Firstly, the LED blinking is handled by a timer, which interrupts to perform the toggling operation. Secondly, the button input is handled by interrupts as well, which eliminates the need to read the GPIO state in the while loop.

Task B: Blink an LED using Non-Blocking DMA and Timer Interrupts

In this task, we will again work on our blinking LED. However, in this task, a user interface should be implemented. You are required to control the Timer 4 with user input via UART4. The user should be allowed to start and stop the timer. At each interrupt of Timer 4, the LED of the 7-segment display must be toggled.

For user input, the following command structure must be used: #X*.

- '#' stands for the start of a command
- 'X' stands for the command ID number:
  - 1 = Start timer
  - 2 = Stop timer
- '*' marks the end of a command

(Start command example: #1*)

The UART4 interface must be configured for a non-blocking DMA receiving mode. The Timer 4 must be configured to generate an interrupt every 100 ms. Don't forget to implement some error handling. Wrong commands should be handled gracefully. Give helpful feedback to the user in case of incorrect commands.

Requirements:

- Use **PJ7 (SEGDP)** for blinking
- Timer 4 generates periodic interrupts at **100ms**
- UART configured for non-blocking **DMA** receiving
- Use **#Command ID*** as command message format
- Start Command ID = **1**
- Stop Command ID = **2**

B.1. Implementation

After defining a few constants and the receive buffer `rx_data` for UART, the main function starts the timer and initiates the first UART transmission in DMA mode. Note that the while loop is completely empty in this application, meaning that no unnecessary UART polling or delay is performed.

```
#define CMD_LENGTH (3)
#define CMD_START ('*')
#define CMD_END ('#')

uint8_t rx_data[CMD_LENGTH];

int main(void)
{
  HAL_TIM_Base_Start_IT(&htim4);
  HAL_UART_Receive_DMA(&huart4, rx_data, CMD_LENGTH);
  while (1) {}
}
```

Code Segment 7: Task B - Relevant code in main.c

When the first UART reception is complete, the UART RX Complete Callback is executed. Firstly, it checks, if the UART-handle instance matches UART4 and after that, the `rx_buffer` is checked if it matches the command format. Depending on whether a `1` or a `0` is received, Timer 4 will be enabled or disabled. In any case, the transmission is restarted to allow further UART interrupts to occur.

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
  if (huart->Instance == UART4)
  {
    if ((rx_data[0] == CMD_START) && (rx_data[CMD_LENGTH - 1] == CMD_END))
    {
      if (rx_data[1] == '1') // *1#
      {
        HAL_TIM_Base_Start_IT(&htim4);
        HAL_UART_Transmit(&huart4, (uint8_t *)"Timer Enabled\n", 15, 1000);
      }
      else if (rx_data[1] == '0') // *0#
      {
        HAL_TIM_Base_Stop_IT(&htim4);
        HAL_UART_Transmit(&huart4, (uint8_t *)"Timer Disabled\n", 16, 1000);
      }
    }
    HAL_UART_Receive_DMA(&huart4, rx_data, CMD_LENGTH);
  }
}
```

Code Segment 8: Task B - UART RX Complete Callback

The Timer manages the blinking. For that, the decimal point LED is toggled, every time the ARR of Timer 4 overflows and generate an Interrupt.

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  if (htim->Instance == TIM4)
  {
    HAL_GPIO_TogglePin(SEGDP_GPIO_Port, SEGDP_Pin);
  }
}
```

Code Segment 9: Task B - Timer Callback

Timer Setup

This time Timer 2 is used and configured as follows to achieve a period of 100ms:

$$ARR = 0{,}1s \cdot \frac{75\,000\,000Hz}{7500} = 1000$$

Again, the TIM4 global interrupt must be enabled in the NVIC Settings for an overflow interrupt to be triggered.

UART Setup

To Enable UART4 in non-blocking DMA Mode, the RX and TX DMA-Request must be enabled:

Parameter Settings | User Constants | NVIC Settings | DMA Settings | GPIO Settings

DMA Request	Stream	Direction	Priority
UART4_RX	DMA1 Stream 0	Peripheral To Memory	Low
UART4_TX	DMA1 Stream 1	Memory To Peripheral	Low

Figure 8: Enable UART4 DMA Mode

UART4 global interrupt must be enabled in the NVIC-Settings as well. Configurations from *Setting up DMA* must be considered to enable the functionality of the DMA mode.

B.2. Results

The serial monitor used to test the application is *HTerm*. The Board shows, that the blinking stops when the command '**0#*' is sent and continues when the command '**1#*' is sent. Furthermore, a message is received on the serial monitor, to confirm that the command was executed.

An issue is that the command must match the defined command length exactly, otherwise the application does not take further commands into account.

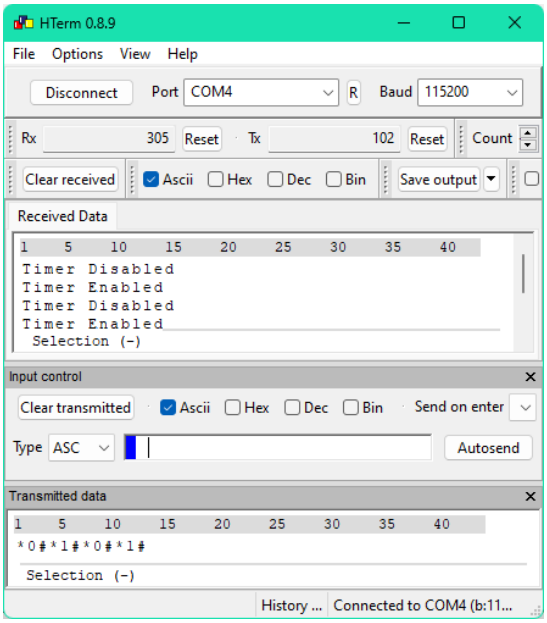

Figure 9: Task B - Result

Task C: Read Magnetic Data from LIS3MDL Sensor in Non-Blocking Mode using Interrupts

This task is an extension to task A of practicum 3. Previously, a polling method was implemented to read the X, Y, and Z-axis magnetic data from the LIS3MDL sensor. In this task, you are required to **implement interrupts** for I2C read and write. The respective addresses can be found in the datasheet and application note. After reading the magnetic data from the three axes, they must be displayed on a serial monitor of your choice. To do this, you are required to use UART4 of the STM32 MCU with the configuration specified in the tutorial named "**UART Interrupt and DMA Mode**". The read values must be displayed in the following format:

X: xxxx Gauss, Y: yyyy Gauss, Z: zzzz Gauss

Requirements:

- Use the **I2C2 interface** to read data from the digital sensor
- Use the **UART4** interface to send data to the serial monitor
- Use the **interrupt mode** for reading/writing data
- Follow the display format mentioned above

C.1. Implementation

Thanks to the previously developed LIS3MDL-Library the code for the sensor configuration could be entirely taken over from the previous practicum. Therefore, not everything will be elaborated in detail. Only changes will be documented.

Main file

The main function is responsible for the sensor initialization, as well as performing the first sensor readout. After that the data acquisition is fully handled by interrupts.

```
#define LIS3MDL_SAD (0b0011110 << 1) // LIS3MDL Slave Address

LIS3MDL_HandleTypeDef hlis3mdl;
const char *sensor_msg = "X: %d Gs, Y: %d Gs, Z: %d Gs\r\n";

int main(void)
{
  LIS3MDL_Init(&hlis3mdl, &hi2c2, LIS3MDL_SAD); // Initialize LIS3MDL Magnetometer
  LIS3MDL_ReadXYZ_IT(&hlis3mdl);                // Start non-blocking I2C Interrupt reception
  while (1) {}
}
```

Code Segment 10: Task C – Relevant code in main.c

When a Receive is completed the sensor data is transmitted onto the UART bus. An advantage of using DMA mode on the UART transmission, is that the callback routine finishes as quickly as possible. This is important to minimize data loss, as the sensor overwrites its registers when the Interrupt service routine is slower than the sample rate. This is still an issue which will be addressed in the final Task, where the I2C operations will also be handled by DMA.

```
void HAL_I2C_MemRxCpltCallback(I2C_HandleTypeDef *hi2c)
{
  if (hi2c->Instance == hlis3mdl.hi2c->Instance)
  {
    char tx_buf[100];
    memset(tx_buf, 0, sizeof(tx_buf));
    sprintf(tx_buf, sensor_msg, hlis3mdl.x, hlis3mdl.y, hlis3mdl.z);
    HAL_UART_Transmit_DMA(&huart4, (uint8_t *)tx_buf, sizeof(tx_buf));

    LIS3MDL_ReadXYZ_IT(&hlis3mdl); // Restart non-blocking I2C Interrupt reception
  }
}
```

Code Segment 11: Task C - I2C Reception Complete Callback

```c
void HAL_I2C_MemTxCpltCallback(I2C_HandleTypeDef *hi2c)
{
  if (hi2c->Instance == hlis3mdl.hi2c->Instance)
  {
    LIS3MDL_ReadXYZ_IT(&hlis3mdl); // Restart non-blocking I2C Interrupt reception
  }
}
```

Code Segment 12: Task C - I2C Transmission Complete Callback

Library Changes

The implementation required minimal changes in the LIS3MDL Library. Firstly, the function signature of the `LIS3MDL_ReadXYZ()` was changed to `LIS3MDL_ReadXYZ_IT()` to clarify that this will trigger an Interrupt once the transmission is finished. The same was done to the `LIS3MDL_ReadRegisters()` function, which now executes the Interrupt variant of the `HAL_I2C_Mem_Read()` function.

One more thing which has been changed in the Read-XYZ function is the `static` prefix for the I2C data buffer `data[6]`. This is required because the data buffer needs to be preserved for it to be loaded from the callback function. Otherwise, the data buffer would always be empty.

The internal function `__LIS2MDL_ConvertToMGs()` now handles the conversion from the raw data buffer to milli Gauss, because this operation will be required as well for the DMA mode implementation.

```c
HAL_StatusTypeDef LIS3MDL_ReadRegisters_IT(LIS3MDL_HandleTypeDef *hlis3mdl, uint8_t reg, uint8_t *data, uint16_t size)
{
  return HAL_I2C_Mem_Read_IT(hlis3mdl->hi2c, hlis3mdl->address, reg, I2C_MEMADD_SIZE_8BIT, data, size);
}

// Read / Write Specific Registers

HAL_StatusTypeDef LIS3MDL_ReadXYZ_IT(LIS3MDL_HandleTypeDef *hlis3mdl)
{
  // Burst read 6 bytes starting from OUT_X_L
  static uint8_t data[6]; // needs to be static to be accessed in the callback
  if (LIS3MDL_ReadRegisters_IT(hlis3mdl, LIS3MDL_OUT_X_L, data, 6) != HAL_OK)
    return HAL_ERROR;

  __LIS2MDL_ConvertToMGs(data, hlis3mdl); //

  return HAL_OK;
}
```

Code Segment 13: Task C - Updated Interrupt-Mode Variant in LIS3MDL Library

Setup

Enable I2C2 event Interrupts:

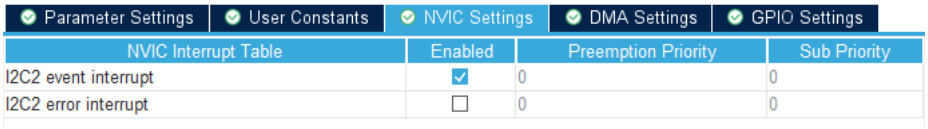

Figure 10: Enable I2C Interrupts

Also consider the UART configuration shown in *UART Setup* for non-blocking UART operations.

C.2. Results

For this Taks, the Serial Monitor extension for VSCode was used, because HTerm could not keep up with the transmission speed. Following Figures display the output on the serial monitor. The idle state (Figure 11) was recorded with the board lying flat on the table. A second test was performed with a magnet in proximity.

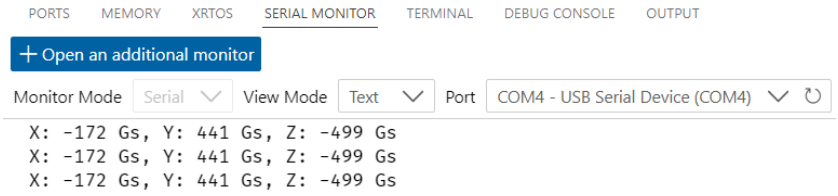

Figure 11: Task C - Result in Idle State

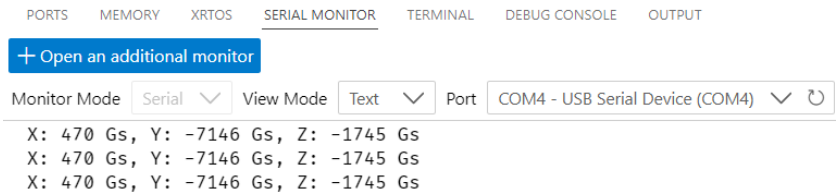

Figure 12: Task C - Result with Magnet in Proximity

C.3. Discussion

Describe your experiences (e.g., design decisions, problems, lesson learned). Which part of the code will be reusable?

The already reused code to initialize the sensor as well as internal functions like calculating the gauss value from the raw data can be reused in applications which require to use extend the sensor interface.

What are the differences to your implementation in Practicum 3? Compare the implementations!

Visually, only a function call was edited. However, the process behind the scenes is much more involved. The main difference is where and how the code is executed. One thing which is ignored in this implementation is that the status register is no longer queried for new sensor data, resulting in a very fast data stream. Since this stream is controlled by interrupts, it is unlikely for timing issues to occur, and the output data registers are read even if they are not updated. A consideration would be to change the hardware and wire up the Data Ready (DRDY) line from the sensor to an external interrupt of the STM32. This would eliminate the need to query the status register.

What are the advantages and disadvantages?

Using Interrupts for the implementation, has the advantage of being non-blocking[2] and is generally the better option in contrast to the brute force polling method from practicum 3, as the I2C transactions are more controlled, and the CPU does not waste as much time on waiting operations. However, setting up the callback function requires more global variables and is much more complex.

[2] no wait time until the transaction is finished

Task D: Read Magnetic Data from LIS3MDL Sensor in Non-Blocking Mode using DMA

This task is a modification of the previous task. In this task, you need to use the **DMA mode** of the I2C2 interface for implementing read and write operation.

Requirements:

- Use the **I2C2 interface** to read data from the digital sensor
- Use the **UART4** interface to send data to the serial monitor
- Use the **interrupt mode** for reading/writing data
- Use the display format mentioned in the previous task

D.1. Implementation

Code wise, only some function suffixes are changed from *IT* to *DMA* in this Task.

```
#define LIS3MDL_SAD (0b0011110 << 1) // LIS3MDL Slave Address

LIS3MDL_HandleTypeDef hlis3mdl;
const char *sensor_msg = "X: %d Gs, Y: %d Gs, Z: %d Gs\r\n";

int main(void)
{
  LIS3MDL_Init(&hlis3mdl, &hi2c2, LIS3MDL_SAD); // Initialize LIS3MDL Magnetometer
  LIS3MDL_ReadXYZ_DMA(&hlis3mdl);               // Start non-blocking I2C Interrupt reception
  while (1) {}
}
```

Code Segment 14: Task D – Relevant code in main.c

```
void HAL_I2C_MemRxCpltCallback(I2C_HandleTypeDef *hi2c)
{
  if (hi2c->Instance == hlis3mdl.hi2c->Instance)
  {
    char tx_buf[100];
    memset(tx_buf, 0, sizeof(tx_buf));
    sprintf(tx_buf, sensor_msg, hlis3mdl.x, hlis3mdl.y, hlis3mdl.z);
    HAL_UART_Transmit_DMA(&huart4, (uint8_t *)tx_buf, sizeof(tx_buf));

    LIS3MDL_ReadXYZ_DMA(&hlis3mdl); // Restart non-blocking I2C Interrupt reception
  }
}
```

Code Segment 15: Task D - I2C Reception Complete Callback

```
void HAL_I2C_MemTxCpltCallback(I2C_HandleTypeDef *hi2c)
{
  if (hi2c->Instance == hlis3mdl.hi2c->Instance)
  {
    LIS3MDL_ReadXYZ_DMA(&hlis3mdl); // Restart non-blocking I2C Interrupt reception
  }
}
```

Code Segment 16: Task D - I2C Transmission Complete Callback

Setup

DMA-Requests for the I2C as well as the global interrupt must be enabled. Settings explained in Setting up DMA must also be considered for the DMA-Controller to function properly.

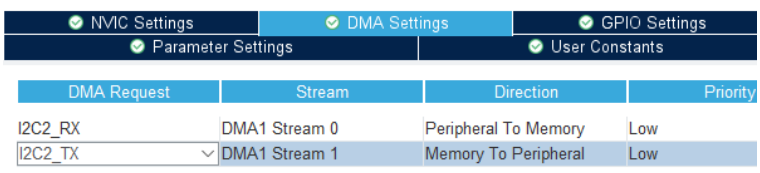

Figure 13: Enable DMA-Requests for I2C

D.2. Discussion

Describe your experiences. Does your solution have any limitations?

This is the ideal solution for having a consistent sample rate and a maximum data yield, meaning that sensor data is unlikely to be lost due to processing times.

For digital signal processing this solution is still not quite optimal. Having only the Callback functions for the data processing is also somewhat limiting, as the data can only be used safely within the scope of the callback function. For example, using the gyroscope data in the main while loop would lead to problems, when an interrupt occurs at the same time the while loop accesses the field.

How would an ideal solution behave in your opinion?

Mutex locks or semaphores would be required to control the access to the data. This again would be very an additional effort for the programmer. A solution for a project where this is required, would be to setup a Real Time Operating System, which is capable of managing tasks and controls access to critical fields.

How is this task different from Task C?

Using the DMA-Controller for I2C transactions enables the bus to operate at a fast and consistent data throughput, as all transactions are now outsourced. As the CPU arranges a I2C transmission, the transport from peripheral to memory are fully handled by the DMA-Controller and the CPU is only Interrupted once the whole transmission is finished. In Task C the CPU needs to be interrupted after each byte sent to provide new data until the transmission is completed. Likewise, the CPU takes byte by byte when receiving data.

Index

ARR ..... *Auto-Reload Register*
CPU ..... *Central Processing Unit*
DMA ..... *Direct Memory Access*
EXTI ..... *External Interrupt*
IRQ ..... *Interrupt Request*
NVIC ..... *Nested Vectored Interrupt Controller*

Figures

Code Segments