**Networked Embedded Systems**

**Practicum 3: Sensors and Communication**

**Group number: 8**

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

**19.05.2024**

Map of Content

[Theory Questions 4](#_Toc168699516)

[Task A: Read Data from the LIS3MDL Magnetic Sensor 7](#_Toc168699517)

[A.1. Calculations 7](#_Toc168699518)

[Control Register Configuration 7](#_Toc168699519)

[Measurement Registers and Interpretation 9](#_Toc168699520)

[A.2. Implementation 10](#_Toc168699521)

[Function Description 11](#_Toc168699522)

[Main Function 12](#_Toc168699523)

[Most Relevant Sensor Functions 13](#_Toc168699524)

[A.3. Results 15](#_Toc168699525)

[A.4. Discussion 15](#_Toc168699526)

[Task B: Read Humidity Values from the HTS221 Sensor 16](#_Toc168699527)

[B.1. Calculations 16](#_Toc168699528)

[Control Register Configuration 16](#_Toc168699529)

[Humidity Calibration 17](#_Toc168699530)

[B.2. Implementation 18](#_Toc168699531)

[Function Description 18](#_Toc168699532)

[Main Function 19](#_Toc168699533)

[Most Relevant Sensor Functions 20](#_Toc168699534)

[B.3. Results 21](#_Toc168699535)

[B.4. Discussion 21](#_Toc168699536)

[Task C: Read Temperature Values from the HTS221 Sensor 22](#_Toc168699537)

[C.1. Calculations 22](#_Toc168699538)

[Temperature Calibration 22](#_Toc168699539)

[C.2. Implementation 24](#_Toc168699540)

[Function Description 24](#_Toc168699541)

[Main Function 24](#_Toc168699542)

[Most Relevant Sensor Functions 25](#_Toc168699543)

[C.3. Results 26](#_Toc168699544)

[C.4. Discussion 26](#_Toc168699545)

[Task D: Read Temperature & Pressure Values from the LPS22HH Sensor 27](#_Toc168699546)

[D.1. Calculations 27](#_Toc168699547)

[Control Register Configuration 27](#_Toc168699548)

[Calculating the Pressure Data 27](#_Toc168699549)

[Calculating the Temperature Data 28](#_Toc168699550)

[D.2. Implementation 29](#_Toc168699551)

[Function Description 29](#_Toc168699552)

[Main Function 30](#_Toc168699553)

[Most Relevant Sensor Functions 31](#_Toc168699554)

[D.3. Results 32](#_Toc168699555)

[D.4. Discussion 32](#_Toc168699556)

[Index 33](#_Toc168699557)

[Figures 33](#_Toc168699558)

[Code Segments 33](#_Toc168699559)

[Tables 34](#_Toc168699560)

# Theory Questions

1. **What is calibration and why do the sensors need that? How are calibration values used to calculate the measurement result? Visualize the relation between calibration curve and measured values (simplified, e.g., linear curves).**

Calibration values are needed, to ensure that the acquired sensor data corresponds to the actual physically present measurement (such as ambient temperature). Wrong calibration can lead to systematic errors, where the sensor makes consistent measurements, but is offset from the real-world value.

|  |  |  |
| --- | --- | --- |
| Zufällige und systematische Fehler  Figure 1: Good Sensor Good Calibration | Zufällige und systematische Fehler  Figure 2: Bad Sensor | Zufällige und systematische Fehler  Figure 3: Good Sensor Bad Calibration |

The Sensor Value is often acquired through an Analog/Digital-Converter reading a Wheatstone-Bridge as shown in Figure 4. Because the sensor only delivers small measurement currents , extra calibration is required to compensate for resistor tolerances and ADC-precision.

Ein Bild, das Diagramm, Reihe, Plan, technische Zeichnung enthält.

Automatisch generierte Beschreibung

Figure 4: LIS3MDL Block Diagram (Symbolic Wheatstone Bridges and A/D-Converter)

By linear interpolation, the acquired sensor data is shifted to the correct systematic offset. In this case (HTS221) the calibration values are stored in non-volatile memory and set from factory, so no additional calibration is required by the user.

Ein Bild, das Dunkelheit, Schwarz, Astronomisches Objekt, Mondlicht enthält.

Automatisch generierte Beschreibung

Figure 5: Linear Interpolation Diagram for HTS221 Humidity Sensor

Interpolation can be achieved with following formular (derived from Figure 5):

1. **Describe I2C. Where do we use it? How does communication via I2C work?**

I2C is short for Inter-Integrated Circuit, which literally means the interconnection between multiple ICs. I2C is a simple Communication protocol which works on a serial data bus consisting of two wires[[1]](#footnote-1), the Serial Data Line (SDA) and the Serial Clock Line (SCL). The I2C-bus has following aspects:

* **Serial:** The data is transmitted on one line , where the data word is sent out bit by bit.
* **Synchronous:** The bus has a clock line, controlling when a data word begins, and a bit is read.
* **Simplex:** Transmitting and receiving data takes place on the same line, meaning only one peripheral can talk at a time.

Ein Bild, das Screenshot, Diagramm, Symbol, Schrift enthält.

Automatisch generierte BeschreibungThe I2C bus requires Pull-Up resistors on each data line because the output stage only consists of a FET Pulling the bus line low, leaving the line floating during the highperiod and the idle state. This topology is called an *Open Collector* output.

I2C is usually used where the master is in short proximity to its I2C peripherals, for example on printed circuit boards. Due to its bus topology, longer data lines limit the data rate due to slower rise and fall times caused by a higher bus capacitance. The edge rise time can be somewhat controlled by the pullup resistors. Lower pull ups can compensate for an increased bus capacitance () but affect the current consumption.

Figure 6: I2C Open Collector Output

Communication via I2C works as Follows: (In this case writing one byte to a register)

* **ST**: Start-Bit - Pulling the Data line Low initiates a transmission
* **SAD + W**: 7-Bit Slave Address + Read (1) / Write (0) bit
* **SAK**: Slave Acknowledge
* **SUB**: Sub address (Register)
* **DATA**: 8-Bit Data-Word
* **SP**: Stop-Condition

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Master | ST | SAD + W |  | SUB |  | DATA |  | SP |
| Slave |  |  | SAK |  | SAK |  | SAK |  |

Table 1: I2C Transmission Example

1. **Describe UART. What does the abbreviation mean? How does communication via UART work?**

Ein Bild, das Screenshot, Diagramm, Rechteck, Design enthält.

Automatisch generierte BeschreibungUART stand for **U**niversal **A**synchronous **R**eceive and **T**ransmit and is, as the name suggests, a **serial**, **asynchronous** transmission protocol. UART can be used to interconnect **two** Peripherals, in this case the STM32H7 and some sort of USB-Bridge to communicate with the PC via USB. The UART bus has two signals RX and TX, while RX from one peripheral must connect to TX from the other. The bus participants are equal meaning that there is **no master controller**, which enables each peripheral to send and receive at the same time, making the data transmission **full duplex.**

Figure 7: UART Interconnection

A UART Transmission has following Format:

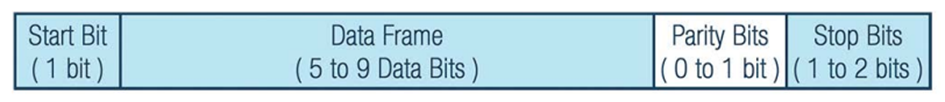


Figure 8: UART Transmission Frame

A Start Bit signalizes the receiver that a new transmission is beginning. The voltage on the transmission line is *Normally High* and is pulled down by the Start Bit. This is followed by the data frame, which is five to nine bits long depending on the configuration. A parity bit is used to validate the transmission (can be disabled). At the end of the transmission, a stop bit sets the bus back to the idle state.

# Task A: Read Data from the LIS3MDL Magnetic Sensor

## A.1. Calculations

There are several control registers (CTRL\_REGx, x: 1-5) to configure, how the sensor operates. The configuration for this Task is:

* Enabled Temperature Sensor
* Ultra-High Performance (UHP) on all axes (X, Y and Z)
* 5Hz output data rate. Slower reading to de-clutter the Terminal
* Plus/minus 12 gauss sensitivity
* Continuous data conversion

Each Control Register has 8 Bits which can be set or reset specific to this configuration. The Notation (1<<x) means, that the x-th Bit is set and corresponds to the value of . This is often preferred over writing a hexadecimal value because it can be read more easily. Setting multiple bits in a word can be achieved by bitwise *or-ing* (|).

|  |  |  |  |
| --- | --- | --- | --- |
| **Shift Left Operation** | **Decimal value** | **Binary Value** | **Hexadecimal Value** |
| (1<<0) |  | 0b 0000 0001 | 0x01 |
| (1<<7) |  | 0b 1000 0000 | 0x80 |
| (1<<6) | (1<<3) |  | 0b 0100 1000 | 0x48 |

Table 2: Left-Shift Operation Example

The Actual Implementation will hide expressions such as (1<<6)|(1<<3) and #defines it with the register name, the operation name or the name of the bit. Every time the value is needed, it can for example be called with TEMP\_EN instead of (1<<7). The following tables will describe the setup steps for each control register. Bits that are not mentioned will be overwritten with zero (which is usually their reset value anyway) because they have no influence on the configuration.

### Control Register Configuration

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **CTRL\_REG1 (**0x20**)** | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | TEMP\_EN | OM1 | OM0 | DO2 | DO1 | DO0 | FAST\_ODR | ST | | | |
| **Operation Description** | | **Bits** | **Value** |
| Enable the temperature sensor | | **TEMP\_EN = 1** | (1<<7) |
| Set operating mode to UHP on X and Y axes  Datasheet Table 21 [[DocID024204 Rev 6](https://www.mouser.com/datasheet/2/389/lis3mdl-1849592.pdf), p. 25] | | **OM0 = 1**  **OM1 = 1** | (1<<6) | (1<<5) |
| Set data output rate to 5Hz  Datasheet Table 22 [[DocID024204 Rev 6](https://www.mouser.com/datasheet/2/389/lis3mdl-1849592.pdf), p. 25] | | **DO0 = 1**  **DO1 = 1**  **DO2 = 0** | (1<<3) | (1<<2) |

Table 3: LIS3MDL - CTRL\_REG1

**Final Value for Address 0x20:** (1<<7)|(1<<6)|(1<<5)|(1<<3)|(1<<2) = 0b 1110 1100 = 0xEC

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **CTRL\_REG2 (**0x21**)** | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | FS1 | FS0 | 0 | REBOOT | SOFT\_RST | 0 | 0 | | | |
| **Operation Description** | | **Bits** | **Value** |
| Set the Scale for a measurement register to ±12 gauss.  A full register (High and Low concatenated) will have a value of -12 Gauss, where the MSB determines the sign. Having a smaller scale will increase the precision, but the measurement cap out sooner.  Datasheet Table 24 [[DocID024204 Rev 6](https://www.mouser.com/datasheet/2/389/lis3mdl-1849592.pdf), p. 25] | | **FS1 = 1**  **FS0 = 0** | (1<<6) |

Table 4: LIS3MDL - CTRL\_REG2

**Final Value for Address 0x21:** (1<<6) = 0b 0100 0000 = 0x40

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **CTRL\_REG3 (**0x22**)** | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | LP | 0 | 0 | SIM | MD1 | MD0 | | | |
| **Operation Description** | | **Bits** | **Value** |
| Set the system operating mode to **continuous**. These Bits must be explicitly set to **0,** because their reset value is **1**, which corresponds to the **Power-Down** operating mode.  Datasheet Table 28 [[DocID024204 Rev 6](https://www.mouser.com/datasheet/2/389/lis3mdl-1849592.pdf), p. 26] | | **MD0 = 0**  **MD1 = 0** | 0 |

Table 5: LIS3MDL - CTRL\_REG3

**Final Value for Address 0x21:** 0x00

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **CTRL\_REG4 (**0x23**)** | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | 0 | 0 | OMZ1 | OMZ0 | BLE | 0 | | | |
| **Operation Description** | | **Bits** | **Value** |
| Set the Z-axis operating mode to UHP.  Datasheet Table 31 [[DocID024204 Rev 6](https://www.mouser.com/datasheet/2/389/lis3mdl-1849592.pdf), p. 27] | | **OMZ0 = 1**  **OMZ1 = 1** | (1<<3) | (1<<2) |

Table 6: LIS3MDL - CTRL\_REG4

**Final Value for Address 0x23:** (1<<3)|(1<<2) = 0b 0000 1100 = 0xC0

**Notes:**

* CTRL\_REG5 can be left at reset state, because no further configuration is necessary.
* Bits marked with 0 are for internal use and not to be overwritten.

### Measurement Registers and Interpretation

|  |  |  |
| --- | --- | --- |
| **Register Name** | **Register Description** | **Register Address** |
| LIS3MDL\_OUT\_X\_L | Lower X-Axis Sensor Data Byte | 0x28 |
| LIS3MDL\_OUT\_X\_H | Upper X-Axis Sensor Data Byte  MSB is the Sign | 0x29 |
| LIS3MDL\_OUT\_Y\_L | Lower Y-Axis Sensor Data Byte | 0x2A |
| LIS3MDL\_OUT\_Y\_H | Upper Y-Axis Sensor Data Byte  MSB is the Sign | 0x2B |
| LIS3MDL\_OUT\_Z\_L | Lower Y-Axis Sensor Data Byte | 0x2C |
| LIS3MDL\_OUT\_Z\_H | Upper Y-Axis Sensor Data Byte  MSB is the Sign | 0x2D |

*How is the magnetic value of each axis calculated from the involved registers? Explain!*

With each reading operation two registers must be read by the I2C-master for each axis. Each sensor reading spans over 16 bits, but the actual precision is 15 bits because the MSB is relevant for the sign.

|  |  |  |
| --- | --- | --- |
| LIS3MDL\_OUT\_x\_H | | LIS3MDL\_OUT\_x\_L |
| Sign | 15-Bit resolution value | |
| 0: +, 1: - | xxx xxxx | xxxx xxxx |

Table 7: LIS3MDL - 2's complement representation of the sensor data

This representation is called 2’s complement

Multiple steps are required to convert the read value into the physical value with unit gauss (Gs).

1. Concatenate the Upper and Lower bytes with bit math:

|  |  |  |
| --- | --- | --- |
|  | int16\_t raw\_x = (OUT\_x\_H\_value << 8) | OUT\_x\_L\_value | Formula 1 |

1. Apply the previously configured sensitivity. In this case, the sensitivity has been set to

|  |  |  |
| --- | --- | --- |
|  | int16\_t x\_val = raw\_x / sensitivity | Formula 2 |

1. The sensitivity is calculated as follows:

|  |  |  |
| --- | --- | --- |
|  |  | Formula 3 |

Because the resolution of the sensor data is 15 Bit, a full register (0b 0111 1111 1111 1111) contains times the value of the least significant bit. As configured, the full register represents which sets the sensitivity to the calculated value in . Furthermore, a little bit of accuracy is lost, as the integer cuts off all decimal places. This formula can be interpreted as: *The raw sensor data has a value of when it measures of magnetic field strength.*

## A.2. Implementation

The LIS3MDL Magnetometer Interface consists of two header files and one source file, where one header *(lis2mdl.h)* declares the function prototypes for the application interface and the other *(lis3mdl\_regsiters.h)* defines values for the registers to reduce magic numbers in code and make it more readable. The Source file will implement the actual functionality of predefined Functions. The usage of a header – source structure provides a better overview of the LIS3MDL-API, which functions are available and what each function does. Separating the sensor implementation from the main file ensures a shorter and more decluttered main application. Implementation can therefore be treated as a library and can copied to other projects as well. This will be especially important for [Task B](#_Task_B:_Read) and [C](#_Task_C:_Read), where the same peripheral is used.

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Figure 9: LIS3MDL Interface Structure

|  |  |
| --- | --- |
| The lis3mdl\_registers.h header provides an extensive list of values from the datasheet. This includes register addresses, bitmasks and configuration masks for certain settings. An important note is that the \*.h Files must be added to the include path (In our case, the *Inc*-Folder is already added) and the \*.c Files to the C\_SOURCES variable in the Makefile. | Code Segment 1: Adding C-Sources to the Makefile |
| A simple struct LIS3MDL\_HandleTypeDef is defined to pass I2C relevant parameters, such as the I2C-handle and the slave address, into the different functions. Additionally, data read from the sensor is stored in this struct as well. The stored data will already be the processed milli-gauss value and can be read without further calculations. The sensitivity field is the LSB per Gauss value as described in Formula 3 and is stored as it is needed to calculate the gauss value for the x, y and z fields. | |



Code Segment 2: LIS2MDL\_HandleTypeDef

### Function Description

|  |
| --- |
| LIS3MDL\_Init(LIS3MDL\_HandleTypeDef\* hlis3mdl, I2C\_HandleTypeDef\* hi2c, uint8\_t address) |
| Initializes the sensor. Assigns the I2C-Handle and the address to the LIS3MDL-Struct. Performs all initial control register configurations.  HAL\_OK: success  HAL\_ERROR: device not connected, address invalid or I2C communication failed |
| LIS3MDL\_ReadRegister(LIS3MDL\_HandleTypeDef\* hlis3mdl, uint8\_t reg, uint8\_t\* data); |
| Reads the provided register and stores the result in a data pointer. Acts as a wrapper function for the HAL\_I2C\_MemRead() function to reduce the number of redundant parameters.  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |
| LIS3MDL\_ReadRegisters(LIS3MDL\_HandleTypeDef\* hlis3mdl, uint8\_t reg, uint8\_t\* data, uint16\_t size) |
| Reads multiple registers starting at the one provided and auto increments the register address by one for each reading operation. Stores the acquired data in a provided data-pointer.  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |
| LIS3MDL\_WriteRegister(LIS3MDL\_HandleTypeDef\* hlis3mdl, uint8\_t reg, uint8\_t data) |
| Writes provided data to a specified register. Acts as a wrapper function for the HAL\_I2C\_MemWrite() function to reduce the number of redundant parameters.  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |
| LIS3MDL\_WriteRegisters(LIS3MDL\_HandleTypeDef\* hlis3mdl, uint8\_t reg, uint8\_t\* data, uint16\_t size) |
| Writes multiple registers starting at the one provided and auto increments the register address by one for each writing operation. Uses the data in the provided pointer, which is also incremented with each operation.  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |
| LIS3MDL\_ReadXYZ(LIS3MDL\_HandleTypeDef\* hlis3mdl) |
| Reads the Data on all six sensor registers and converts it into a Gauss value. The data is stored in the x, y and z field of the LIS3MDL-Struct.  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |
| LIS3MDL\_ReadStatus(LIS3MDL\_HandleTypeDef\* hlis3mdl) |
| Reads the status register and stores it in the status field of the LIS3MDL-Struct  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |

Table 8: LIS3MDL - Function Description

### Main Function

To handle errors and debug code, an error count variable err\_cnt is introduced, which increments by 1 each time an operation fails. The Code optimization option must be set to -O0 to ensure the variable won’t be optimized out by the compiler, as it isn’t used in any operations. Another option to prevent the variable being removed, is to use the **volatile** prefix. The **volatile** keyword tells the compiler, that the value of the variable may change at any time without any action being taken by the code. This is not the case here but serves to maintain the variable at runtime.

The next step is to initialize the sensor. The initialization function takes the sensor handle as well as the I2C handle and the slave address (LIS3MDL\_SAD), which in this case is 0b 0011 1100 = 0x3C[[2]](#footnote-2) (defined above the main function).

After defining the sensor message, the while loop reads the status indefinitely, and is stored in the handle struct. As soon as the XYZ-data available bit is set, the sensor data is read and stored in the sensor handle struct. The data is immediately transmitted via UART by formatting the sensor message into the transmission buffer which is subsequently transmitted onto the UART-Bus and cleared afterward.



Code Segment 3: Task A - Relevant Code in main.c

### Most Relevant Sensor Functions

Before any measurements can be taken, the sensor must be initialized first. Firstly, a connection test is performed by reading the who am I register from the sensor. This checks, if the correct device is connected and the address is valid. Because each control register is only offset by one to the next (0x20, 0x21, 0x22…) the whole configuration for all control registers can be written in only one I2C transmission. Conveniently, the I2C slave auto increments the register when multiple bytes are written, until the master throws a stop (SP) condition. This will further be referred to as burst-write or burst-read.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Master | ST | SAD + W |  | SUB |  | DATA |  | DATA |  | SP |
| Slave |  |  | SAK |  | SAK |  | SAK |  | SAK |  |

Table 9: LIS3MDL - Transfer when master is writing multiple bytes to slave (2 bytes)

A data array cfg\_data with a size of four bytes – one for each register – is prepared with the values determined in the Control Register Configuration from the calculations, which is subsequently transmitted onto the bus. A private helper function \_\_LIS3MDL\_Set\_Sensitivity() dynamically applies Formula 3 onto the sensitivity field of the sensor handle struct, depending on the previously configured **FS**-Bits in **CTRL\_REG2** (not shown here but available in the appended code [Src/lis3mdl.c, line 7]).



Code Segment 4: LIS3MDL\_Init()

In the same fashion, as the burst-write from the initialization, a burst-read can be performed similarly. In the LIS3MDL\_ReadXYZ() function, all six registers are read and stored in a 6-byte long data array.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Master | ST | SAD + W |  | SUB |  | SR | SAD + R |  |  | MAK |  | NMAK | SP |
| Slave |  |  | SAK |  | SAK |  |  | SAK | DATA |  | DATA |  |  |

Table 10: LIS3MDL - Transfer when master is reading multiple bytes of data from slave (2 bytes)

The data is then being concatenated as shown in Formula 1 to obtain the raw sensor data. This value is converted to milli-gauss (Formula 2 with a multiplier of 1000) afterwards and stored in their respective fields of the sensor handle struct.



Code Segment 5: LIS3MDL\_ReadXYZ()

## A.3. Results

The Terminal HTerm (ver 0.8.1beta) has been used to connect to the COM Port of the Elite-Board and read the transmitted data.

|  |  |
| --- | --- |
| Ein Bild, das Text, Screenshot, Zahl, Schrift enthält.  Automatisch generierte Beschreibung  Figure 10: Task A - Sensor Results in Idle State | In this measurement example, the board is laid flat on the table. Here, the magnetic field of the Earth is measured. |
| Ein Bild, das Text, Screenshot, Zahl, Schrift enthält.  Automatisch generierte Beschreibung  Figure 11: Task A - Sensor Results with Magnet (Medium Distance) | In this measurement, a magnet is held about 5cm above the sensor demonstrating the increase in magnetic field strength. |
| Ein Bild, das Text, Screenshot, Zahl, Schrift enthält.  Automatisch generierte Beschreibung  Figure 12: Task A - Sensor Results with Magnet (Close Distance) | Here, the magnet is held in immediate proximity () to the sensor. As configured, the measurement saturates at . |

## A.4. Discussion

Using an error count variable proved to be very useful, as errors were quickly spotted during debugging. This method also does not slow down the code when left in, as the compiler optimizes the variable out when the debugging phase is over.

The Task could have been completed with way less additional functions. However, the additional abstractness of implementing wrapper functions for the HAL\_I2C module such as LIS3MDL\_ReadRegister(), benefits the readability of the code because of the removal of redundant parameters. This also made documenting the code in the protocol way easier, because the lines weren’t as long causing less line breaks. For example:

LIS3MDL\_ReadRegister(hlis3mdl, LIS3MDL\_WHO\_AM\_I, &whoami)

Instead of:

HAL\_I2C\_MemRead(hlis3mdl->hi2c, hlis3mdl->address, LIS3MDL\_WHO\_AM\_I,  
 I2C\_MEMADD\_SIZE\_8BIT, &whoami, 1, I2C\_TIMEOUT)

In terms of reusability, we think the implemented LIS3MDL-Library would do a good job for a simple plug and play solution for STM32 applications implementing this sensor.

# Task B: Read Humidity Values from the HTS221 Sensor

## B.1. Calculations

### Control Register Configuration

The Initial value of the CTRL\_REG1 is 0x00. This would result in a configuration, where the Power Down (PD) bit is 0. The sensor would take no measurements.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **CTRL\_REG1 (**0x20**)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | PD | Reserved | BDU | ODR1 | ODR0 | | | |
| **Operation Description** | | **Bits** | **Value** |
| Enable the sensor | | **PD = 1** | (1<<7) |
| Set the output data rate of temperature and humidity sensor to 1Hz. | | **ODR1 = 0**  **ODR0 = 1** | (1<<0) |

Table 11: HTS221 - CTRL\_REG1

**Final Value for Address 0x20:** (1<<7)|(1<<0) = 0b 1000 0001 = 0x81

Note: The configuration in the other control registers is irrelevant for this Taks.

### Humidity Calibration

Ein Bild, das Text, Screenshot, Zahl, Schrift enthält.

Automatisch generierte Beschreibung

Table 12: Register Map for humidity calibration registers (Taken from the AppNote)

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter** | **Description** | **Register address** | **State** |
| H\_OUT | The sensor data acquisition register | 0x28 – 0x29 | Changing |
| H0\_rH\_x2 | Lower Output calibration value (times 2) | 0x30 | Constant |
| H1\_rH\_x2 | Upper Output calibration value (times 2) | 0x31 | Constant |
| H0\_T0\_OUT | Lower input calibration value | 0x36-0x37 | Constant |
| H1\_T0\_OUT | Upper input calibration value | 0x3A-0x3B | Constant |

Table 13: Humidity Calibration Value Description

The sensor is calibrated from factory for reasons stated in the theory questions. The calibration values are stored in non-volatile memory and set from factory, so no additional calibration is required by the user.

Ein Bild, das Dunkelheit, Schwarz, Astronomisches Objekt, Mondlicht enthält.

Automatisch generierte Beschreibung

Figure 13: Linear Interpolation Diagram for HTS221 Humidity Sensor

The humidity is interpolated from input and output calibration values. Interpolation can be achieved with following formular (derived from Figure 13):

|  |  |  |
| --- | --- | --- |
|  |  | Formula 4 |

## B.2. Implementation

### Function Description

|  |
| --- |
| HTS221\_Init(HTS221\_HandleTypeDef\* hhts221, I2C\_HandleTypeDef\* hi2c, uint8\_t address) |
| Initializes the sensor. Assigns the I2C-Handle and the address to the HTS221-Struct. Performs all initial control register configurations.  HAL\_OK: success  HAL\_ERROR: device not connected, address invalid or I2C communication failed |
| HTS221\_ReadRegister(HTS221\_HandleTypeDef\* hhts221, uint8\_t reg, uint8\_t\* data) |
| Reads the provided register and stores the result in a data pointer. Acts as a wrapper function for the HAL\_I2C\_MemRead() function to reduce the number of redundant parameters.  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |
| HTS221\_ReadRegisters(HTS221\_HandleTypeDef\* hhts221, uint8\_t reg, uint8\_t\* data, uint16\_t size) |
| Reads multiple registers starting at the one provided and auto increments the register address by one for each reading operation. Stores the acquired data in a provided data-pointer.  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |
| HTS221\_WriteRegister(HTS221\_HandleTypeDef\* hhts221, uint8\_t reg, uint8\_t data) |
| Writes provided data to a specified register. Acts as a wrapper function for the HAL\_I2C\_MemWrite() function to reduce the number of redundant parameters.  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |
| HTS221\_ReadStatus(HTS221\_HandleTypeDef\* hhts221) |
| Reads the status register. The result is stored in the status field of the HTS221-Struct  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |
| HTS221\_ReadHumidity(HTS221\_HandleTypeDef\* hhts221) |
| Reads the temperature register and converts is into relative humidity [%] with a factor of 10 for one more decimal place precision. The result is stored in the humidity field of the HTS221-Struct.  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |

Table 14: HTS211 - Function description

### Main Function

The main code works the same as in Task A. The Sensor is initialized, and the status register will be read indefinitely. If the humidity data available bit is set, the humidity is read and calculated. The result of the measurement is the relative humidity multiplied by a factor of ten. This allows to get one decimal place extra precision. This is again transmitted via UART to the serial monitor.



Code Segment 6: Task B - Relevant code in main.c

### Most Relevant Sensor Functions

The initialization function works in the same way as in Task A, by checking the who am I register and configuring the control register as determined in the calculations.



Code Segment 7: HTS221\_Init()

As the \_\_HTS221\_Get\_Calibration() function is very extensive, only relevant snippets will be explained here. The function can be viewed in the appended code to its full extent. The essence of this function is to read out the calibration values from the calibration registers tuned from factory, convert them into usable dimensions and store them in the sensor handle struct. This struct will then be passed into the  
HTS221\_ReadHumidity() function to use those values to calculate the correct humidity.



Code Segment 8: HTS221 - Get Calibration values for humidity

Since the humidity output calibration values Hx\_rH are stored with a factor of two, a right shift operation by one achieves a division by two.

The input calibration values Hx\_T0\_out are again stored in two registers. A concatenation to a 16-bit signed integer is therefore required.

For this sensor, extra precaution is required when performing burst-read/write operations. The Most significant bit of the sub address (register address) must be set to 1. This is achieved by a bitwise or operation with 0x80 (0b 1000 0000).



Code Segment 9: HTS221\_ReadRegisters()

Finally, the HTS221\_ReadHumidity() function reads the upper and lower data register, concatenates them and applies Formula 4. A temporary variable has been introduced to shorten the expression.



Code Segment 10: HTS221\_ReadHumidity()

## B.3. Results

|  |  |
| --- | --- |
| Figure 14: Task B - Result | The results show the humidity in the current ambience of the board. The sensor has been slightly blown on to show that consistent measurements are made. |

## B.4. Discussion

The Application notes helped immensely with provided formulas and code snippets.

# Task C: Read Temperature Values from the HTS221 Sensor

## C.1. Calculations

The control register configuration is reused from Task B.

### Temperature Calibration

Ein Bild, das Text, Zahl, Screenshot, parallel enthält.

Automatisch generierte Beschreibung

Table 15: Register Map for Temperature Calibration

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter** | **Description** | **Register address** | **State** |
| T\_OUT | The sensor data acquisition register | 0x2A – 0x2B | Changing |
| T0\_degC\_x8 | Lower Output calibration value (times 8) | 0x32 | Constant |
| T1\_degC\_x8 | Upper Output calibration value (times 8) | 0x33 | Constant |
| T1/T0 msb | Most significant bits for Tx\_degC\_x8 | 0x35 | Constant |
| T0\_OUT | Lower input calibration value | 0x3C-0x3D | Constant |
| T0\_OUT | Upper input calibration value | 0x3E-0x3F | Constant |

Table 16: Temperature Calibration parameter description

How the values are processed to be ready to be used in Formula 5 is shown in Code Segment 12.

Again, the ADC-reading is interpolated with the calibration values.

Ein Bild, das Mond, Dunkelheit, Schwarz, Astronomisches Objekt enthält.

Automatisch generierte Beschreibung

Figure 15: Linear Interpolation Diagram for HTS221 Temperature Sensor

The temperature is interpolated from input and output calibration values. Interpolation can be achieved with following formular (derived from Figure 15):

|  |  |  |
| --- | --- | --- |
|  |  | Formula 5 |

## C.2. Implementation

### Function Description

As most functions are already documented in the Function Description of Task B, only new functions will be listed here.

|  |
| --- |
| HTS221\_ReadTemperature(HTS221\_HandleTypeDef\* hhts221) |
| Reads the temperature register and converts is into °C with a factor of 10 for one more decimal place precision. The result is stored in the temp field of the HTS221-Struct.  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |

### Main Function

The main code works the same as in Task B. The Sensor is initialized, and the status register will be read indefinitely. If the temperature data available bit is set, the temperature is read and calculated. The result of the measurement is the temperature in °C multiplied by a factor of ten. This allows to get one decimal place extra precision. This is again transmitted via UART to the serial monitor.



Code Segment 11: Task C - Relevant code in main.c

### Most Relevant Sensor Functions

Following code complements the \_\_HTS221\_Get\_Calibration() function for the temperature calibration. Since the temperature output calibration values Tx\_degC are stored with a factor of 8, a right shift operation by 3 achieves a division by 8. The value for the 2 MSBs of the output calibration values are stored in another register and must be masked out and concatenated as shown in the code. The input calibration values Tx\_out are again stored in two registers. A concatenation to a 16-bit signed integer is therefore required.



Code Segment 12: HTS221 - Get Calibration values for humidity

Finally, the HTS221\_ReadTemperature() function reads the upper and lower data register, concatenates them and applies Formula 5. A temporary variable has been introduced to shorten the expression.



Code Segment 13: HTS221\_ReadTemperature()

## C.3. Results

|  |  |
| --- | --- |
| Figure 16: Task C - Results | This measurement has taken place in a warm attic room. Artificial temperature variations show consistent measurements. |

## C.4. Discussion

The code could be reused 1:1. The only difference was the main function.

# Task D: Read Temperature & Pressure Values from the LPS22HH Sensor

## D.1. Calculations

### Control Register Configuration

CTRL\_REG2:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **CTRL\_REG1 (**0x10**)** | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | ODR2 | ODR1 | ODR0 | EN\_LPFP |  |  |  | | | |
| **Operation Description** | | **Bits** | **Value** |
| An adequate choice for the Output data rate is 1Hz for a calm reading on the serial monitor | | **ODR0 = 1**  **ODR1 = 0** | (1<<4) = 0x10 |

Table 17: LPS22HH - CTRL\_REG1

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **CTRL\_REG2 (**0x11**)** | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | BOOT | INT\_H\_L | PP\_OD | IF\_ADD\_INC | 0 | SWRESET | LOW\_NOISE\_EN | ONE\_SHOT | | | |
| **Operation Description** | | **Bits** | **Value** |
| Important here is to set the IF\_ADD\_INC Bit to 1, which allows the address to increment when reading or writing multiple bytes. This ensures, that burst reading/writing is possible. Since this bit is set by default, no action is required. | | **IF\_ADD\_INC = 1** | (1<<4) = 0x10 |

Table 18: LPS22HH - CTRL\_REG2

Any configuration of the other bits in the control registers are irrelevant for this task.

### Calculating the Pressure Data

|  |  |  |
| --- | --- | --- |
| **Register Name** | **Description** | **Address** |
| PRESS\_OUT\_XL | Lowest pressure data acquisition register | 0x28 |
| PRESS\_OUT\_L | Lower pressure data acquisition register | 0x29 |
| PRESS\_OUT\_H | Upper pressure data acquisition register | 0x2A |

Table 19: LPS22HH - Output Pressure Registers

There are three pressure sensor data registers, which must be shifted and concatenated in following manner:

press\_lsb = PRESS\_OUT\_H<<16 | PRESS\_OUT\_L<<8 | PRESS\_OUT\_XL

and subsequently divided by the LSB sensitivity, which is 4096:

press\_hPa = press\_lsb / 4096

### Calculating the Temperature Data

|  |  |  |
| --- | --- | --- |
| **Register Name** | **Description** | **Address** |
| TEMP\_OUT\_L | Lowest temperature data acquisition register | 0x2B |
| TEMP\_OUT\_H | Upper temperature data acquisition register | 0x2C |

Table 20: LPS22HH - Output Temperature Registers

The two temperature registers must be concatenated and subsequently divided by the LSB sensitivity, which is 100.

temp\_lsb = TEMP\_OUT\_H<<8 | TEMP\_OUT\_L

temp\_degC = temp\_lsb / 100

## D.2. Implementation

### Function Description

|  |
| --- |
| LPS22HH\_Init(LPS22HH\_HandleTypeDef\* hlps22hh, I2C\_HandleTypeDef\* hi2c, uint8\_t address) |
| Initializes the sensor. Assigns the I2C-Handle and the address to the LPS22HH-Struct. Performs all initial control register configurations.  HAL\_OK: success  HAL\_ERROR: device not connected, address invalid or I2C communication failed |
| LPS22HH\_ReadRegister(LPS22HH\_HandleTypeDef\* hlps22hh, uint8\_t reg, uint8\_t\* data) |
| Reads the provided register and stores the result in a data pointer. Acts as a wrapper function for the HAL\_I2C\_MemRead() function to reduce the number of redundant parameters.  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |
| LPS22HH\_ReadRegisters(LPS22HH\_HandleTypeDef\* hlps22hh, uint8\_t reg, uint8\_t\* data, uint16\_t size) |
| Reads multiple registers starting at the one provided and auto increments the register address by one for each reading operation. Stores the acquired data in a provided data-pointer.  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |
| LPS22HH\_WriteRegister(LPS22HH\_HandleTypeDef\* hlps22hh, uint8\_t reg, uint8\_t data) |
| Writes provided data to a specified register. Acts as a wrapper function for the HAL\_I2C\_MemWrite() function to reduce the number of redundant parameters.  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |
| LPS22HH\_ReadStatus(LPS22HH\_HandleTypeDef\* hlps22hh) |
| Reads the Status register. The result is stored in the status field of the LPS22HH-Struct.  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |
| LPS22HH\_ReadPressure(LPS22HH\_HandleTypeDef\* hlps22hh) |
| Reads the pressure registers and converts it into hPa. The result is stored in the pressure field of the LPS22HH-Struct.  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |
| LPS22HH\_ReadTemperature(LPS22HH\_HandleTypeDef\* hlps22hh) |
| Reads the temperature registers and converts it into °C. The result is stored in the temp field of the LPS22HH-Struct.  HAL\_OK: success  HAL\_ERROR: I2C communication failed or timed out |

Table 21: Task D - Function Implementation

### Main Function

The main function for this task follows the same pattern as before.

* Initialize the sensor
* Continuously read the status register
* Check if pressure and temperature data is available
* Read and transmit the data



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

### Most Relevant Sensor Functions

The initialization function works in the same way as in all previous Tasks, by checking the who am I register and configuring the control register as determined in the calculations.



Code Segment 15: LPS22HH\_Init()

This time there are 3 pressure registers to be read, resulting in a 24-Bit resolution. The raw data is first concatenated and afterwards divided by the LSB value of the data, which is 4096. (Taken from the application note).



Code Segment 16: LPS22HH\_ReadPressure()

The temperature reading is a 16-Bit value which is again concatenated and divided by the LSB value (here 100, taken from the application note).



Code Segment 17: LPS22HH\_ReadTemperature()

## D.3. Results

|  |  |
| --- | --- |
| Figure 17: Task D - Results | This measurement has taken place in a warm attic room. Artificial temperature variations show consistent measurements. |

## D.4. Discussion

Throughout these Tasks we noticed that a consistent code style can save a lot of work. Every I2C peripheral followed the same structure, which drastically shortened the amount of time spent coding.

# Index

I2C *Inter-Integrated Circuit*

SAD *Slave Address*

SCL *Serial Clock*

SDA *Serial Data*

UART *Universal Asynchronous Recieve and Transmit*

UHP *Ultra-High Performance*

## Figures

[Figure 1: Good Sensor Good Calibration 4](#_Toc168699561)

[Figure 2: Bad Sensor 4](#_Toc168699562)

[Figure 3: Good Sensor Bad Calibration 4](#_Toc168699563)

[Figure 4: LIS3MDL Block Diagram (Symbolic Wheatstone Bridges and A/D-Converter) 4](#_Toc168699564)

[Figure 5: Linear Interpolation Diagram for HTS221 Humidity Sensor 4](#_Toc168699565)

[Figure 6: I2C Open Collector Output 5](#_Toc168699566)

[Figure 7: UART Interconnection 6](#_Toc168699567)

[Figure 8: UART Transmission Frame 6](#_Toc168699568)

[Figure 9: LIS3MDL Interface Structure 10](#_Toc168699569)

[Figure 10: Task A - Sensor Results in Idle State 15](#_Toc168699570)

[Figure 11: Task A - Sensor Results with Magnet (Medium Distance) 15](#_Toc168699571)

[Figure 12: Task A - Sensor Results with Magnet (Close Distance) 15](#_Toc168699572)

[Figure 13: Linear Interpolation Diagram for HTS221 Humidity Sensor 17](#_Toc168699573)

[Figure 14: Task B - Result 21](#_Toc168699574)

[Figure 15: Linear Interpolation Diagram for HTS221 Temperature Sensor 23](#_Toc168699575)

[Figure 16: Task C - Results 26](#_Toc168699576)

[Figure 17: Task D - Results 32](#_Toc168699577)

## Code Segments

[Code Segment 1: Adding C-Sources to the Makefile 10](#_Toc168699578)

[Code Segment 2: LIS2MDL\_HandleTypeDef 10](#_Toc168699579)

[Code Segment 3: Task A - Relevant Code in main.c 12](#_Toc168699580)

[Code Segment 4: LIS3MDL\_Init() 13](#_Toc168699581)

[Code Segment 5: LIS3MDL\_ReadXYZ() 14](#_Toc168699582)

[Code Segment 6: Task B - Relevant code in main.c 19](#_Toc168699583)

[Code Segment 7: HTS221\_Init() 20](#_Toc168699584)

[Code Segment 8: HTS221 - Get Calibration values for humidity 20](#_Toc168699585)

[Code Segment 9: HTS221\_ReadRegisters() 21](#_Toc168699586)

[Code Segment 10: HTS221\_ReadHumidity() 21](#_Toc168699587)

[Code Segment 11: Task C - Relevant code in main.c 24](#_Toc168699588)

[Code Segment 12: HTS221 - Get Calibration values for humidity 25](#_Toc168699589)

[Code Segment 13: HTS221\_ReadTemperature() 25](#_Toc168699590)

[Code Segment 14: Task D - Relevant code in main.c 30](#_Toc168699591)

[Code Segment 15: LPS22HH\_Init() 31](#_Toc168699592)

[Code Segment 16: LPS22HH\_ReadPressure() 31](#_Toc168699593)

[Code Segment 17: LPS22HH\_ReadTemperature() 32](#_Toc168699594)

## Tables

[Table 1: I2C Transmission Example 5](#_Toc168699595)

[Table 2: Left-Shift Operation Example 7](#_Toc168699596)

[Table 3: LIS3MDL - CTRL\_REG1 7](#_Toc168699597)

[Table 4: LIS3MDL - CTRL\_REG2 8](#_Toc168699598)

[Table 5: LIS3MDL - CTRL\_REG3 8](#_Toc168699599)

[Table 6: LIS3MDL - CTRL\_REG4 8](#_Toc168699600)

[Table 7: LIS3MDL - 2's complement representation of the sensor data 9](#_Toc168699601)

[Table 8: LIS3MDL - Function Description 11](#_Toc168699602)

[Table 9: LIS3MDL - Transfer when master is writing multiple bytes to slave (2 bytes) 13](#_Toc168699603)

[Table 10: LIS3MDL - Transfer when master is reading multiple bytes of data from slave (2 bytes) 14](#_Toc168699604)

[Table 11: HTS221 - CTRL\_REG1 16](#_Toc168699605)

[Table 12: Register Map for humidity calibration registers (Taken from the AppNote) 17](#_Toc168699606)

[Table 13: Humidity Calibration Value Description 17](#_Toc168699607)

[Table 14: HTS211 - Function description 18](#_Toc168699608)

[Table 15: Register Map for Temperature Calibration 22](#_Toc168699609)

[Table 16: Temperature Calibration parameter description 22](#_Toc168699610)

[Table 17: LPS22HH - CTRL\_REG1 27](#_Toc168699611)

[Table 18: LPS22HH - CTRL\_REG2 27](#_Toc168699612)

[Table 19: LPS22HH - Output Pressure Registers 27](#_Toc168699613)

[Table 20: LPS22HH - Output Temperature Registers 28](#_Toc168699614)

[Table 21: Task D - Function Implementation 29](#_Toc168699615)

1. I2C is also often called Two Wire Interface (TWI) due to its former d [↑](#footnote-ref-1)
2. Datasheet Table 11 [[*DocID024204 Rev 6*](https://www.mouser.com/datasheet/2/389/lis3mdl-1849592.pdf), p. 17] [↑](#footnote-ref-2)