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

a) 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.

Figure 1: Good Sensor Good Calibration

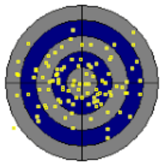


Figure 2: Bad Sensor

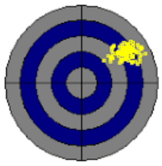


Figure 3: Good Sensor Bad Calibration

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 $I(M)$, extra calibration is required to compensate for resistor tolerances and ADC-precision.

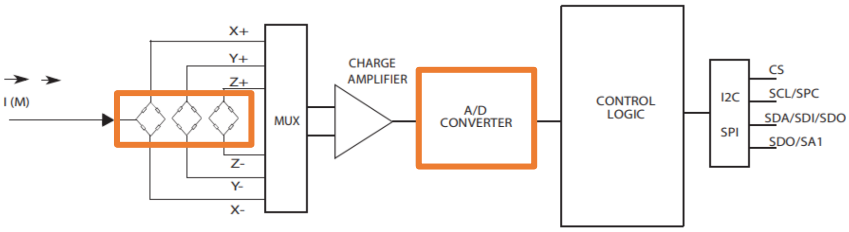


Figure 4: LIS3MDL Block Diagram (Symbolic Wheatstone Bridges and A/D-Converter)

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.

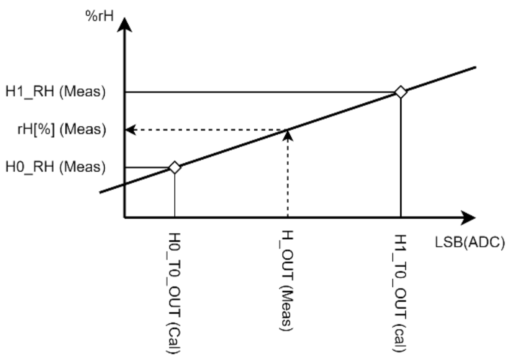


Figure 5: Linear Interpolation Diagram for HTS221 Humidity Sensor

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

$$H\_rH[\%] = \frac{(H1\_RH - H0\_rH) \cdot (H\_OUT - H0\_T0\_OUT)}{H1\_T0\_OUT - H0\_T0\_OUT} + H0\_RH$$

b) 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], 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 bit is read.
- **Simplex:** Transmitting and receiving data takes place on the same line, meaning only one peripheral can talk at a time.

The 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 high period and the idle state. This topology is called an *Open Collector* output.

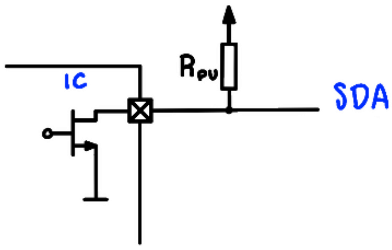


Figure 6: I2C Open Collector Output

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 ($\tau = R \cdot C$) but affect the current consumption.

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] I2C is also often called Two Wire Interface (TWI) due to its former d

c) Describe UART. What does the abbreviation mean? How does communication via UART work?

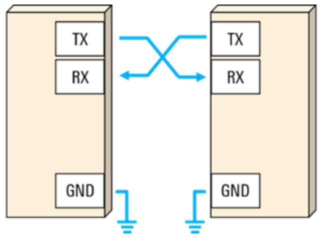


Figure 7: UART Interconnection

UART 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.**

A UART Transmission has following Format:

Start Bit	Data Frame	Parity Bits	Stop Bits
1 bit	5 to 9 Data Bits	0 to 1 bit	1 to 2 bits

Table 2: 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 as follows:

- 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 2^x. 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)`	$2^0 = 1$	`0b 0000 0001`	`0x01`
`(1<<7)`	$2^7 = 128$	`0b 1000 0000`	`0x80`
`(1<<6) | (1<<3)`	$2^6 + 2^3 = 72$	`0b 0100 1000`	`0x48`

Table 3: 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, p. 25]*	**OM0 = 1** **OM1 = 1**	`(1<<6) | (1<<5)`
Set data output rate to 5Hz *Datasheet Table 22 [DocID024204 Rev 6, p. 25]*	**DO0 = 1** **DO1 = 1** **DO2 = 0**	`(1<<3) | (1<<2)`

Table 4: 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 caps out sooner. *Datasheet Table 24 [DocID024204 Rev 6, p. 25]*	**FS1 = 1** **FS0 = 0**	`(1<<6)`

Table 5: 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, p. 26]*	**MD0 = 0** **MD1 = 0**	`0`

Table 6: 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, p. 27]*	**OMZ0 = 1** **OMZ1 = 1**	`(1<<3) | (1<<2)`

Table 7: 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 8: 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*

2. Apply the previously configured sensitivity. In this case, the sensitivity has been set to
   `int16_t x_val = raw_x / sensitivity` *Formula 2*

3. The sensitivity is calculated as follows:
   $$sensitivity\left[\frac{LSB}{Gs}\right] = \frac{(2^{15}-1)\,LSB}{12\,Gs} = 2730$$ *Formula 3*

Because the resolution of the sensor data is 15 Bit, a full register (`0b 0111 1111 1111 1111`) contains $2^{15} - 1$ times the value of the least significant bit. As configured, the full register represents $12\ Gs$ which sets the sensitivity to the calculated value in LSB/Gs. 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* 2730 *when it measures* $1\ Gs$ *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 and C, where the same peripheral is used.

LIS3MDL Interface
Header (Inc/lis3mdl.h)
Source File (Src/lis3mdl.c)
Register Map (Inc/lis3mdl_registers.h)

Figure 8: 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.

```
C_SOURCES =  \
Src/main.c \
Src/lis3mdl.c \
Src/gpio.c \
…
```

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.

```
typedef struct {
    I2C_HandleTypeDef* hi2c;
    uint8_t address;
    uint8_t status;
    int16_t sensitivity;
    int16_t x; // x-axes magnetic field strength in mGs
    int16_t y; // y-axes magnetic field strength in mGs
    int16_t z; // z-axes magnetic field strength in mGs
} LIS3MDL_HandleTypeDef;
```

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 9: 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] (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.

```c
#define LIS3MDL_SAD (0b0011110 << 1)

int main(void)
{
  volatile uint32_t err_cnt = 0; // Debug Variable to Count Errors
  LIS3MDL_HandleTypeDef hlis3mdl;

  err_cnt += LIS3MDL_Init(&hlis3mdl, &hi2c2, LIS3MDL_SAD) != HAL_OK;

  char tx_buf[45] = { 0 };
  const char* sensor_msg = "X: %5d mGs, Y: %5d mGs, Z: %5d mGs\n";

  while (1)
  {
    err_cnt += LIS3MDL_ReadStatus(&hlis3mdl) != HAL_OK;

    if (hlis3mdl.status & ZYXDA) // Check if data is available on all axes
    {
      err_cnt += LIS3MDL_ReadXYZ(&hlis3mdl) != HAL_OK;

      // Transmit Gyroscope Data via UART
      sprintf(tx_buf, sensor_msg, hlis3mdl.x, hlis3mdl.y, hlis3mdl.z);
      err_cnt += HAL_UART_Transmit(&huart4, (uint8_t*)tx_buf, sizeof(tx_buf), HAL_MAX_DELAY) != HAL_OK;
      memset(tx_buf, 0, sizeof(tx_buf));
    }
  }
}
```

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

[2] *Datasheet Table 11 [DocID024204 Rev 6, p. 17]*

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 10: 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]*).

```
HAL_StatusTypeDef LIS3MDL_Init(LIS3MDL_HandleTypeDef* hlis3mdl, I2C_HandleTypeDef* hi2c, uint8_t address)
{
    hlis3mdl->hi2c = hi2c;
    hlis3mdl->address = address;
    hlis3mdl->status = 0;
    hlis3mdl->x = 0;
    hlis3mdl->y = 0;
    hlis3mdl->z = 0;

    // Check if the device is connected
    uint8_t whoami = 0;
    if (LIS3MDL_ReadRegister(hlis3mdl, LIS3MDL_WHO_AM_I, &whoami) != HAL_OK) return HAL_ERROR;
    if (whoami != LIS3MDL_WHO_AM_I_VALUE) return HAL_ERROR;

    // Configure device
    uint8_t cfg_data[4] = {
        DO_5HZ | OM_UHP | TEMP_EN,  // CTRL_REG1
        FS_12GAUSS,                 // CTRL_REG2
        MD_CONTINUOUS,              // CTRL_REG3
        OMZ_UHP                     // CTRL_REG4
    };

    // Write configuration data to control registers 1 to 4
    if (LIS3MDL_WriteRegisters(hlis3mdl, LIS3MDL_CTRL_REG1, cfg_data, 4) != HAL_OK) return HAL_ERROR;
    if (__LIS3MDL_Set_Sensitivity(hlis3mdl) != HAL_OK) return HAL_ERROR;
    return HAL_OK;
}
```

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

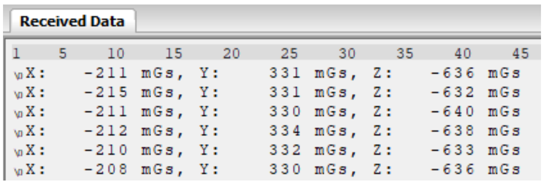
The data is then being concatenated as shown in Formula 1 to obtain the raw sensor data. This value is converted to milli-gauss (Formula 2 with a multiplier of 1000) afterwards and stored in their respective fields of the sensor handle struct.

```c
HAL_StatusTypeDef LIS3MDL_ReadXYZ(LIS3MDL_HandleTypeDef* hlis3mdl)
{
    // Burst read 6 bytes starting from OUT_X_L
    uint8_t data[6] = { 0 };
    if (LIS3MDL_ReadRegisters(hlis3mdl, LIS3MDL_OUT_X_L, data, 6) != HAL_OK) return HAL_ERROR;

    // Combine the 8-bit high and low bytes into 16-bit values
    int16_t raw_x = ((data[1] << 8) | data[0]);
    int16_t raw_y = ((data[3] << 8) | data[2]);
    int16_t raw_z = ((data[5] << 8) | data[4]);

    // Convert to mGs
    hlis3mdl->x = 1000 * raw_x / hlis3mdl->sensitivity;
    hlis3mdl->y = 1000 * raw_y / hlis3mdl->sensitivity;
    hlis3mdl->z = 1000 * raw_z / hlis3mdl->sensitivity;

    return HAL_OK;
}
```

Code Segment 5: LIS3MDL_ReadXYZ()

A.3. Results

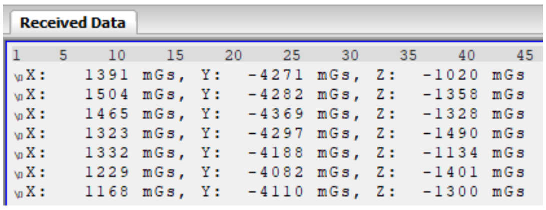
The Terminal HTerm (ver 0.8.1beta) has been used to connect to the COM Port of the Elite-Board and read the transmitted data.



```
Received Data
1    5    10    15    20    25    30    35    40    45
X:   -211 mGs, Y:    331 mGs, Z:   -636 mGs
X:   -215 mGs, Y:    331 mGs, Z:   -632 mGs
X:   -211 mGs, Y:    330 mGs, Z:   -640 mGs
X:   -212 mGs, Y:    334 mGs, Z:   -638 mGs
X:   -210 mGs, Y:    332 mGs, Z:   -633 mGs
X:   -208 mGs, Y:    330 mGs, Z:   -636 mGs
```

Figure 9: Task A - Sensor Results in Idle State

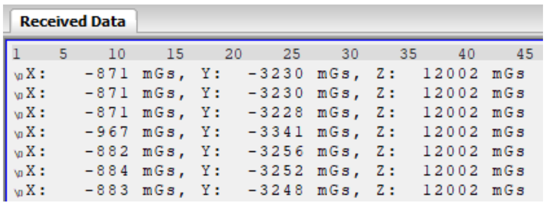
In this measurement example, the board is laid flat on the table. Here, the magnetic field of the Earth is measured.



```
Received Data
1    5    10    15    20    25    30    35    40    45
X:   1391 mGs, Y:  -4271 mGs, Z:  -1020 mGs
X:   1504 mGs, Y:  -4282 mGs, Z:  -1358 mGs
X:   1465 mGs, Y:  -4369 mGs, Z:  -1328 mGs
X:   1323 mGs, Y:  -4297 mGs, Z:  -1490 mGs
X:   1332 mGs, Y:  -4188 mGs, Z:  -1134 mGs
X:   1229 mGs, Y:  -4082 mGs, Z:  -1401 mGs
X:   1168 mGs, Y:  -4110 mGs, Z:  -1300 mGs
```

Figure 10: 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.



```
Received Data
1    5    10    15    20    25    30    35    40    45
X:   -871 mGs, Y:  -3230 mGs, Z:  12002 mGs
X:   -871 mGs, Y:  -3230 mGs, Z:  12002 mGs
X:   -871 mGs, Y:  -3228 mGs, Z:  12002 mGs
X:   -967 mGs, Y:  -3341 mGs, Z:  12002 mGs
X:   -882 mGs, Y:  -3256 mGs, Z:  12002 mGs
X:   -884 mGs, Y:  -3252 mGs, Z:  12002 mGs
X:   -883 mGs, Y:  -3248 mGs, Z:  12002 mGs
```

Figure 11: Task A - Sensor Results with Magnet (Close Distance)

Here, the magnet is held in immediate proximity ($< 1cm$) to the sensor. As configured, the measurement saturates at $12\ Gs$.

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 12: 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

Addr	Variable	Format (1)	b7	b6	b5	b4	b3	b2	b1	b0
0x28	H_OUT	s(16)	H7	H6	H5	H4	H3	H2	H1	H0
0x29			H15	H14	H13	H12	H11	H10	H9	H8
0x30	H0_rH_x2	u(16)	H0.7	H0.6	H0.5	H0.4	H0.3	H0.2	H0.1	H0.0
0x31	H1_rH_x2	u(16)	H1.7	H1.6	H1.5	H1.4	H1.3	H1.2	H1.1	H1.0
0x36	H0_T0_OUT	s(16)	7	6	5	4	3	2	1	0
0x37			15	14	13	12	11	10	9	8
0x3A	H1_T0_OUT	s16)	7	6	5	4	3	2	1	0
0x3B			15	14	13	12	11	10	9	8

Table 13: 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 14: 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.

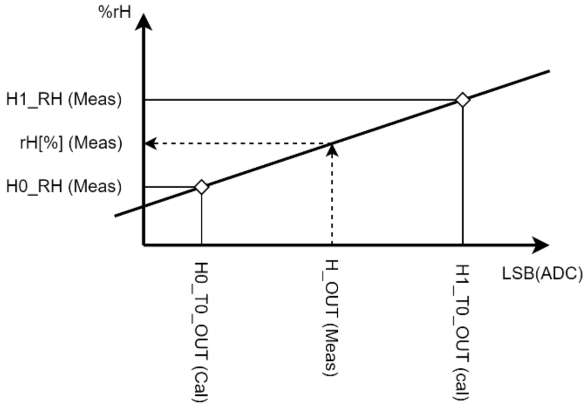


Figure 12: Linear Interpolation Diagram for HTS221 Humidity Sensor

The humidity is interpolated from input and output calibration values. Interpolation can be achieved with following formular (derived from Figure 12):

$$H\_rH[\%] = \frac{(H1\_rH - H0\_rH) \cdot (H\_OUT - H0\_T0\_OUT)}{H1\_T0\_OUT - H0\_T0\_OUT} + H0\_rH \qquad \text{Formula 4}$$

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 15: 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.

```
int main(void)
{
  uint32_t err_cnt = 0; // Debug variable to count errors
  HTS221_HandleTypeDef hhts221; // Sensor data is stored in this struct

  err_cnt += HTS221_Init(&hhts221, &hi2c2, HTS221_SAD) != HAL_OK;

  char tx_buf[50] = { 0 };
  char* sensor_msg = "Relative Humidity: %2d.%1d%%\n";
  while (1)
  {
    err_cnt += HTS221_ReadStatus(&hhts221);

    if (hhts221.status & H_DA) // Check if humidity data is available
    {
      // H_DA Bit is cleared after reading humidity data
      err_cnt += HTS221_ReadHumidity(&hhts221) != HAL_OK;

      // Transmit humidity data via UART
      sprintf(tx_buf, sensor_msg, hhts221.humidity / 10, hhts221.humidity % 10);
      err_cnt += HAL_UART_Transmit(&huart4, (uint8_t*)tx_buf, sizeof(tx_buf), HAL_MAX_DELAY) != HAL_OK;
      memset(tx_buf, 0, sizeof(tx_buf));
      HAL_Delay(200);
    }
  }
}
```

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.

```
HAL_StatusTypeDef HTS221_Init(HTS221_HandleTypeDef* hhts221, I2C_HandleTypeDef* hi2c, uint8_t address)
{
    hhts221->hi2c = hi2c;
    hhts221->address = address;
    hhts221->status = 0;
    hhts221->temperature = 0;
    hhts221->humidity = 0;

    // Check if Sensor is connected
    uint8_t who_am_i;
    if (HTS221_ReadRegister(hhts221, HTS221_WHO_AM_I, &who_am_i)) return HAL_ERROR;
    if (who_am_i != HTS221_WHO_AM_I_VAL) return HAL_ERROR;

    // Configure Sensor
    if (HTS221_WriteRegister(hhts221, HTS221_CTRL_REG1, ODR0 | PD) != HAL_OK) return HAL_ERROR;
    if (__HTS221_Get_Calibration(hhts221) != HAL_OK) return HAL_ERROR;
    return HAL_OK;
}
```

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.

```
uint8_t i2c_buf[4];
// i2c_buf[0:3] <- 0x30 - 0x33 (H0_rH_x2, H1_rH_x2)
if (HTS221_ReadRegisters(hhts221, HTS221_H0_rH_x2, i2c_buf, 2) != HAL_OK) return HAL_ERROR;
hhts221->cal.H0_rH = i2c_buf[0] >> 1; // Divide by 2
hhts221->cal.H1_rH = i2c_buf[1] >> 1;

// i2c_buf[0:1] <- 0x36 - 0x37 (H0_T0_OUT)
// i2c_buf[2:3] <- 0x3A - 0x3B (H1_T0_OUT)
if (HTS221_ReadRegisters(hhts221, HTS221_H0_T0_OUT_L, i2c_buf, 2) != HAL_OK) return HAL_ERROR;
if (HTS221_ReadRegisters(hhts221, HTS221_H1_T0_OUT_L, i2c_buf + 2, 2) != HAL_OK) return HAL_ERROR;
hhts221->cal.H0_T0_out = ((uint16_t)i2c_buf[1] << 8 | i2c_buf[0]);
hhts221->cal.H1_T0_out = ((uint16_t)i2c_buf[3] << 8 | i2c_buf[2]);
```

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).

```
HAL_StatusTypeDef HTS221_ReadRegisters(HTS221_HandleTypeDef* hhts221, uint8_t reg, uint8_t* data, uint16_t size)
{
    // Set MSB of sub address to 1 for auto-increment
    return HAL_I2C_Mem_Read(hhts221->hi2c, hhts221->address, 0x80 | reg, I2C_MEMADD_SIZE_8BIT, data, size,
       I2C_TIMEOUT);
}
```

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.

```
HAL_StatusTypeDef HTS221_ReadHumidity(HTS221_HandleTypeDef* hhts221)
{
    uint8_t data[2] = { 0, 0 };

    // Burst read humidity registers (Status Register is cleared automatically)
    if (HTS221_ReadRegisters(hhts221, HTS221_TEMP_OUT_L, data, 2) != HAL_OK) return HAL_ERROR;
    int16_t h_out = (int16_t)((uint16_t)data[1] << 8 | (uint16_t)data[0]);

    // Interpolate humidity using formula (1) from the Technical Note [TN1218, p. 2]
    int32_t tmp32 = (int32_t)(h_out - hhts221->cal.H0_T0_out) * (int32_t)(hhts221->cal.H1_rH - hhts221->cal.H0_rH);
    hhts221->humidity = 10 * (tmp32 / (hhts221->cal.H1_T0_out - hhts221->cal.H0_T0_out)) + hhts221->cal.H0_rH * 10;

    if (hhts221->humidity > 999) hhts221->humidity = 999; // Clamp to 99.9%
    return HAL_OK;
}
```

Code Segment 10: HTS221_ReadHumidity()

B.3. Results

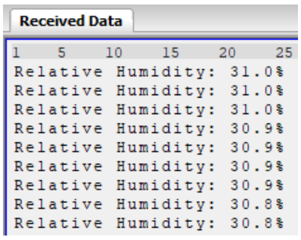


Figure 13: Task B - Result

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

Adr	Variable	Format	b7	b6	b5	b4	b3	b2	b1	b0
Output registers										
28	H_OUT	(s16)	H7	H6	H5	H4	H3	H2	H1	H0
29			H15	H14	H13	H12	H11	H10	H9	H8
2A	T_OUT	(s16)	T7	T6	T5	T4	T3	T2	T1	T0
2B			T15	T14	T13	T12	T11	T10	T9	T8
Calibration registers										
30	H0_rH_x2	(u8)	H0.7	H0.6	H0.5	H0.4	H0.3	H0.2	H0.1	H0.0
31	H1_rH_x2	(u8)	H1.7	H1.6	H1.5	H1.4	H1.3	H1.2	H1.1	H1.0
32	T0_degC_x8	(u8)	T0.7	T0.6	T0.5	T0.4	T0.3	T0.2	T0.1	T0.0
33	T1_degC_x8	(u8)	T1.7	T1.6	T1.5	T1.4	T1.3	T1.2	T1.1	T1.0
34	Reserved	(u16)								
35	T1/T0 msb	(u2),(u2)	Reserved				T1.9	T1.8	T0.9	T0.8
36	H0_T0_OUT	(s16)	7	6	5	4	3	2	1	0
37			15	14	13	12	11	10	9	8
38	Reserved									
39										
3A	H1_T0_OUT	(s16)	7	6	5	4	3	2	1	0
3B			15	14	13	12	11	10	9	8
3C	T0_OUT	(s16)	7	6	5	4	3	2	1	0
3D			15	14	13	12	11	10	9	8
3E	T1_OUT	(s16)	7	6	5	4	3	2	1	0
3F			15	14	13	12	11	10	9	8

Table 16: 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 17: 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.

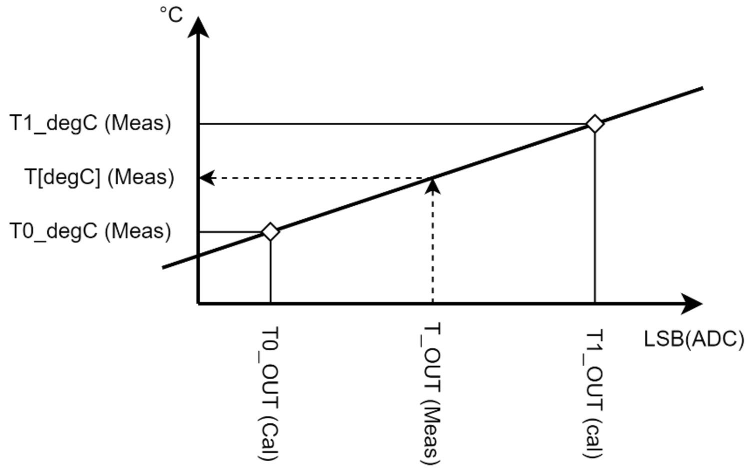


Figure 14: Linear Interpolation Diagram for HTS221 Temperature Sensor

The temperature is interpolated from input and output calibration values. Interpolation can be achieved with following formular (derived from Figure 14):

$$T[°C] = \frac{(T1\_degC - T0\_degC) \cdot (T\_OUT - T0\_OUT)}{T1\_OUT - T0\_OUT} + T0\_degC \qquad \textit{Formula 5}$$

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* hts221)` 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

Table 18: Task C - Function Description (additional Functions only)

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 10. This allows to get one extra decimal place of precision. This is again transmitted via UART to the serial monitor.

```
int main(void)
{
  uint32_t err_cnt = 0; // Debug variable to count errors
  HTS221_HandleTypeDef hhts221; // Sensor data is stored in this struct

  err_cnt += HTS221_Init(&hhts221, &hi2c2, HTS221_SAD) != HAL_OK;

  char tx_buf[23] = { 0 };
  char* sensor_msg = "Temperature: %2d.%1d degC\n";

  while (1)
  {

    err_cnt += HTS221_ReadStatus(&hhts221);

    if (hhts221.status & T_DA) // Check if temperature data is available
    {
      // T_DA Bit is cleared after reading humidity data
      err_cnt += HTS221_ReadTemperature(&hhts221) != HAL_OK;

      // Transmit tempearture data via UART
      sprintf(tx_buf, sensor_msg, hhts221.temperature / 10, hhts221.temperature % 10);
      err_cnt += HAL_UART_Transmit(&huart4, (uint8_t*)tx_buf, sizeof(tx_buf), HAL_MAX_DELAY) != HAL_OK;
      memset(tx_buf, 0, sizeof(tx_buf));
      HAL_Delay(200);
    }
  }
}
```

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.

```
uint8_t i2c_buf[4], t_msb; // Temporary buffer for I2C data
// i2c_buf[0:2] <- 0x32 - 0x33 (T0_degC_x8, T1_degC_x8)
// t_msb <- 0x35 (T1_T0_MSB)
if (HTS221_ReadRegisters(hhts221, HTS221_T0_degC_x8, i2c_buf, 2) != HAL_OK) return HAL_ERROR;
if (HTS221_ReadRegister(hhts221, HTS221_T1_T0_MSB, &t_msb) != HAL_OK) return HAL_ERROR;
hhts221->cal.T0_degC = (((uint16_t)(t_msb & 0x03) << 8) | i2c_buf[0]) >> 3; // Prepend MSB and Divide by 8
hhts221->cal.T1_degC = (((uint16_t)(t_msb & 0x0C) << 6) | i2c_buf[1]) >> 3;

// i2c_buf[0:3] <- 0x3C - 0x3E, (T0_OUT, T1_OUT)
if (HTS221_ReadRegisters(hhts221, HTS221_T0_OUT_L, i2c_buf, 4) != HAL_OK) return HAL_ERROR;
hhts221->cal.T0_out = ((uint16_t)i2c_buf[1] << 8 | i2c_buf[0]);
hhts221->cal.T1_out = ((uint16_t)i2c_buf[3] << 8 | i2c_buf[2]);
```

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.

```
HAL_StatusTypeDef HTS221_ReadTemperature(HTS221_HandleTypeDef* hhts221)
{
    uint8_t data[2] = { 0, 0 };

    // Burst read temperature registers (Status Register is cleared automatically)
    if (HTS221_ReadRegisters(hhts221, HTS221_TEMP_OUT_L, data, 2) != HAL_OK) return HAL_ERROR;
    uint16_t t_out = ((int16_t)data[1] << 8 | (int16_t)data[0]);

    // Interpolate temperature using formula (2) from the Technical Note [TN1218, p. 6]
    // Multiply by 10 to get one decimal place
    uint32_t tmp32 = (int32_t)(t_out - hhts221->cal.T0_out) * (int32_t)(hhts221->cal.T1_degC - hhts221->cal.T0_degC);
    hhts221->temperature = 10 * (tmp32 / (hhts221->cal.T1_out - hhts221->cal.T0_out)) + hhts221->cal.T0_degC * 10;

    return HAL_OK;
}
```

Code Segment 13: HTS221_ReadTemperature()

C.3. Results

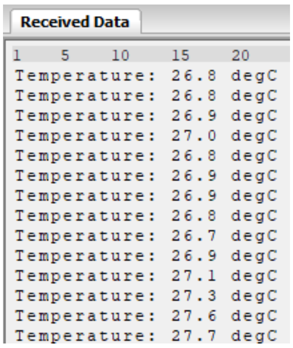


Figure 15: Task C - Results

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 19: 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 20: 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 21: 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 22: 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 23: 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

```
int main(void)
{
  uint32_t err = 0;
  LPS22HH_HandleTypeDef hlps22hh;

  err += LPS22HH_Init(&hlps22hh, &hi2c2, LPS22HH_SAD) != HAL_OK;

  char* sensor_msg = "Pressure: %4d hPa, Temperature: %4d degC\n";
  char tx_buf[60];

  while (1)
  {
    err += LPS22HH_ReadStatus(&hlps22hh) != HAL_OK;

    if (hlps22hh.status & (T_DA | P_DA))
    {
      err += LPS22HH_ReadTemperature(&hlps22hh) != HAL_OK;
      err += LPS22HH_ReadPressure(&hlps22hh) != HAL_OK;

      sprintf(tx_buf, sensor_msg, hlps22hh.pressure, hlps22hh.temp);
      HAL_UART_Transmit(&huart4, (uint8_t*)tx_buf, strlen(tx_buf), HAL_MAX_DELAY);
      memset(tx_buf, 0, sizeof(tx_buf));
    }
  }
}
```

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.

```
HAL_StatusTypeDef LPS22HH_Init(LPS22HH_HandleTypeDef* hlps22hh, I2C_HandleTypeDef* hi2c, uint8_t address)
{
    hlps22hh->i2c = hi2c;
    hlps22hh->address = address;
    hlps22hh->status = 0;
    hlps22hh->pressure = 0;
    hlps22hh->temp = 0;

    // Check WHO_AM_I Register
    uint8_t who_am_i = 0;
    if (LPS22HH_ReadRegister(hlps22hh, LPS22HH_WHO_AM_I, &who_am_i)) return HAL_ERROR;
    if (who_am_i != LPS22HH_WHO_AM_I_VALUE) return HAL_ERROR;

    // Set Output Data Rate to 1Hz
    if (LPS22HH_WriteRegister(hlps22hh, LPS22HH_CTRL_REG1, ODR_1HZ)) return HAL_ERROR;

    return HAL_OK;
}
```

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).

```
HAL_StatusTypeDef LPS22HH_ReadPressure(LPS22HH_HandleTypeDef* hlps22hh)
{
    uint8_t data[3] = { 0, 0, 0 };
    if (LPS22HH_ReadRegisters(hlps22hh, LPS22HH_PRESS_OUT_XL, data, 3)) return HAL_ERROR;

    int32_t raw_press = ((uint32_t)data[2] << 16) | ((uint16_t)data[1] << 8) | data[0];
    hlps22hh->pressure = raw_press / P_LSB;

    return HAL_OK;
}
```

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).

```
HAL_StatusTypeDef LPS22HH_ReadTemperature(LPS22HH_HandleTypeDef* hlps22hh)
{
    uint8_t data[2] = { 0, 0 };
    if (LPS22HH_ReadRegisters(hlps22hh, LPS22HH_TEMP_OUT_L, data, 2)) return HAL_ERROR;

    int16_t raw_temp = ((uint16_t)data[1] << 8) | data[0];
    hlps22hh->temp = raw_temp / T_LSB;

    return HAL_OK;
}
```

Code Segment 17: LPS22HH_ReadTemperature()

D.3. Results

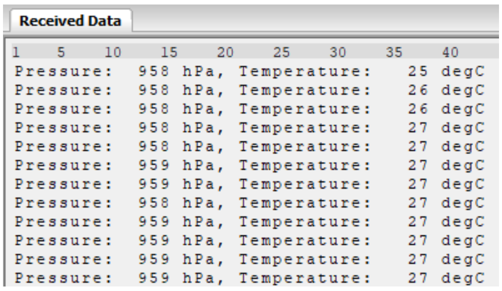


Figure 16: Task D - Results

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

Figures

Code Segments

Tables