EN 173 Lab Guides

Lab 8

Version 2025, 2025-01-20

Communicating with a digital temperature sensor

The TMP102 is a **digital** temperature sensor that communicates using the I^2C protocol. It has an accuracy of 0.5°C and a resolution of 0.0625°C. Its operating temperature range is -40°C to +125°C.

The Sparkfun breakout board for the TMP102 that you will be using has integrated pull-up resistors so you will not need to include those externally. The connections between the TMP102 and the nRF52840 are given in Table 1.

Table 1. TMP102 to nRF52840 DK connections

Signal	nRF52840 pin
VCC	VDD (via power bus)
GND	GND (via ground bus)
SDA	p0.26
SCL	p0.27
ALT (alert)	no connection
ADD0 (address)	GND (via ground bus)

On some versions of the Sparkfun breakout board for the TMP102 the ADD0 pin is internally connected to the ground bus. If that is the case, you will not need to make the external connection. You can check this by looking for a set of pads on the board labeled ADD0. If there is a solder bridge between two pads, then the connection is made. Do not attempt to connect the ADD0 pin if it is already connected internally (and definitely do not connect it to the power bus).

The TMP102 has four possible I^2C addresses that are selected by how ADD0 is connected. If it is connected to the ground bus, then the 7-bit address is 0x48. If it is connected to the power bus, then the address is 0x49. If it is connected to SDA, then the address is 0x4A. If it is connected to SCL, then the address is 0x4B. In your circuit it is connected to ground so the 7-bit I^2C address you will use is 0x48.

Program 3 reads temperatures from the TMP102 and then outputs them to the computer via USB. The TMP102 has four possible registers: one for configuration, one for reading the temperature, one for configuring a low temperature alert, and one for configuring a high temperature alert. You will only use the first two of those registers today.

1. Create a new application.
2. Edit `prj.conf` to include the following:

   Program 1. The I^2C and logging modules (including float support) must be enabled.

   ```
   CONFIG_I2C=y
   CONFIG_LOG=y
   ```

```
CONFIG_CBPRINTF_FP_SUPPORT=y
```

3. Generate a build configuration and create an overlay.
4. Edit the overlay file.

Program 2. The TMP102 is configured as an I^2C device with an address of 0x48.

```
&i2c0 {
  tempsensor: tmp102@48 { ①
    compatible = "i2c-device";
    reg = <0x48>; ②
    status = "okay";
  };
};
```

① The device name is `tmp102` and the address is 0x48.

② The address is 0x48.

5. Enter Program 3 into your `main.c` file.

Program 3. Read temperature from TMP102 and send to computer

```
#include <zephyr/kernel.h>
#include <zephyr/device.h>
#include <zephyr/drivers/i2c.h>
#include <zephyr/logging/log.h>

#define SLEEP_TIME_MS 1000
#define TMP_NODE DT_NODELABEL(tempsensor)

LOG_MODULE_REGISTER(Lab8A, LOG_LEVEL_INF);

const struct i2c_dt_spec tempSensor = I2C_DT_SPEC_GET(TMP_NODE);

int main(void) {
    int err;
    uint8_t buffer[3];
    int16_t rawTemp;
    float tempC;

    if (!i2c_is_ready_dt(&tempSensor)) {
        LOG_ERR("I2C bus not ready");
        return -1;
    }

    buffer[0] = 0x01; // Configuration register
    buffer[1] = 0x60; ①
    buffer[2] = 0xA0; ②
    err = i2c_write_dt(&tempSensor, buffer, 3);
    if (err) {
        LOG_ERR("I2C write failed");
        return -1;
    }
    k_msleep(SLEEP_TIME_MS);

    while (true) {
        buffer[0] = 0x00; // Temperature register
        err = i2c_write_read_dt(&tempSensor, buffer, 1, buffer, 2);
        if (err) {
            LOG_ERR("I2C write-read failed");
            return -1;
        }

        rawTemp = (buffer[0] << 4) | (buffer[1] >> 4); ③
        tempC = rawTemp * 0.0625; ④
        LOG_INF("Temperature: %.1f C", tempC);

        k_msleep(SLEEP_TIME_MS);
    }
}
```

① Bits 0-4 of the first configuration byte configure the alert feature (which is not used in this program). Bits 5 and 6 are fixed at 1. Bit 7 is used to configure a power-saving one-shot conversion mode.

② Bits 0-3 are not used. Bit 4 enables 13-bit resolution rather than default 12 bit resolution. Bit 5 is set at 1 for writes. Bits 6 and 7 set the conversion rate (we are using the 4 Hz setting).

③ The temperature is a 12-bit value with the MSB in the first byte and the lower 4 bits in the second byte. To assemble the 12-bit value, the MSB is shifted left 4 bits and then the lower 4 bits are ORed in after the LSB is shifted right 4 bits.

④ The temperature is a 12-bit value with a resolution of 0.0625°C per bit.

Exercise 8.1

Connect the TMP102 temperature sensor as described and upload Program 3. Verify that everything is working properly.

Demonstrate that you have successfully constructed and coded an I^{2}C interface to the TMP102.

Exercise 8.2

You will use the *Analog Discovery*'s logic analyzer to observe the I^{2}C between the nRF52840 and the TMP102.

1. Connect a ground flywire (on the *AD*) to the ground bus strip on the breadboard.
2. Connect flywire **0** (pink) to the same row as the SCL connection on the breaboard.
3. Connect flywire **1** (green) to the same row as the SDA connection on the breadboard.

4. . Start *Waveforms* and launch the **Logic** app.
5. Click on the green plus sign to add a logic channel to be monitored.
6. Select I2C for the logic protocol. The default settings are fine.
7. Next, change the Trigger type from **Simple** to **Protocol**. Select I2C.
8. Set Base to 100 us/div and then enter 400 us as the Position.
9. Click **[Run]** to begin repeated acquisitions.
10. You should observe the I^2C communication between the nRF52840 and the TMP102. The SCL line should be a square wave with a frequency of 100 kHz. The SDA line should be a square wave with a frequency of 100 kHz and should change when the SCL line is low. The data should be sent in 8-bit bytes with the MSB first. The first byte is the address of the TMP102 (0x48) followed by a write bit (0) and then the first register address for the temperature values (0x00). After a brief pause, the nRF52840 will send a repeated start condition followed by the address of the TMP102 with a read bit (1). The TMP102 will then send the two bytes of temperature data. The nRF52840 will then send a stop condition.
11. Heat the TMP102 with your finger and observe the temperature values change.
12. Now change the address of the TMP102 in the overlay file to 0x49. Rebuild and upload the program. Observe the I^2C communication again. The address should now be 0x49. However, that is not the actual address of the TMP102 so it does not send an acknowledgement bit. This failure to acknowledge is known as a NACK (not acknowledge) and is a common source of I^2C communication errors. The `i2c_write_dt` command will return an error if it does not receive an acknowledge bit and the program ends.

Your Turn

Digital compass

In this assignment you will communicate with a digital compass using I^2C. Two versions of instructions are provided: one for the HMC5883 and one for the LIS3MDL. You will only need to complete the one that corresponds to the sensor you have.

Assignment 8.1

1. Access the assignment link on Blackboard.
2. Declare the following in your main function:
   a. `uint8_t buffer[6]`
   b. `int16_t magComponents[3]`
   c. `float magMicroT[3]`
3. Configure the sensor as described below, selecting the correct tab for the sensor you are using.

HMC5883

The digital compass (HMC5883) has a 7-bit address of 0x1E. You will read the three magnetic field components from the compass.

> This breakout board does not include pull-up resistors for SDA or SCL so you must provide them externally. A low- to mid-kilohm resistor (for example, 2.2 kΩ) is required connecting SDA to the power bus. The same is true of SCL.

Configuration registers (not needed for this program)

There are two configuration registers with addresses of 0x00 and 0x01. You can use those to select the number of samples averaged per measurement output, the data output rate, the sensor gain, and some other less important things. The defaults (1 sample per output, 15 Hz data rate, and 0.092 μT multiplier) are fine for this program so no need to write anything!

Mode register

The address of the mode register is 0x02. Sending it a data byte of 0x01 will place it into single measurement mode (which will then require the mode to be reset before each new measurement). In this program you want continuous measurement mode, so send it a data byte of 0x00.

Data output registers

The data output register addresses begin at 0x03. Read 6 bytes from this to get the three magnetic field components (two bytes per component).

LIS3MDL

In this assignment you will communicate with a digital compass using I^2C. The digital compass (LIS3MDL) has a 7-bit address of 0x1C (or 0x1E if the solder jumper AD1 on the back of the board is connected). You will read the three magnetic field components from the compass and then print those.

There are weak (10 kΩ) pull-up resistors integrated into Adafruit’s breakout board for the LIS3MDL so you might not need any external. If you do run into problems with communication you might find external 4.7 kΩ resistors for SDA and SCL do the trick.

Control registers

There are five control registers on the LIS3MDL that configure its features.

The default settings of control register 1 (with an address of 0x20) are okay and include operating the x and y axes in low-power mode and having an output data rate of 10 Hz.

The default settings of control register 2 (with an address of 0x21) are also okay. The most important is the full-scale selection of ±4 gauss, where 1 gauss = 0.0001 tesla. The Earth's magnetic field falls in the range of 0.25 to 0.65 gauss. If you were using this magnetometer to measure fields generated by other sources, there are also settings of 8, 12, and 16 gauss.

You need to change the settings of control register 3 (with an address of 0x22) to enable continuous conversion mode. The default is single conversion mode. You can do this by writing 0x00 to this register.

Data output registers

The data output register addresses begin at 0x28. Read 6 bytes from this to get the three magnetic field components (two bytes per component).

4. The I^{2}C interface must read in data from the sensor as data type `uint8_t`. That is why you needed `buffer`.
5. The raw magnetic components are stored as `int16_t`. The data type `int16_t` encodes integers using 16 bits and includes both positive and negative numbers (along with other particular details about how it is encoded).

HMC5883

1. Next, combine pairs of data words to get the x, y, and z components using our bitshift and OR method. Assign these to `magComponents`. There are two very important things to note:
   a. The MSB comes first in each pair, the opposite of the accelerometer! Therefore,
      `magComponent[0] = buffer[0]<<8 | buffer[1];`
      (element 0 and 1 are flipped compared to the accelerometer).
   b. You also need to know that Honeywell (the manufacturer of the HMC5883L) is sadistic and ordered them x, **z**, and then y. You don't want to know how long it took me to figure that out! You want your `magComponents`` to have the traditional ordering (x, y, z).
2. You can now calculate the actual magnetic field (in μT) by multiplying the raw data by 0.092. This is the default multiplier for the HMC5883L.

LIS3MDL

1. Next, combine pairs of data words to get the x, y, and z components using our bitshift and OR method. Assign these to `magComponents`. The LIS3MDL uses little endian format so the first byte is the LSB and the second byte is the MSB.
   `magComponent[0] = data[1]<<8 | data[0];`
2. You can now calculate the actual magnetic field (in μT) by dividing the raw data by 6.842. This is the default multiplier for the LIS3MDL.

6. Print these three components to the computer.

7. Test your digital compass. Are the components plausible?

Take a video demonstrating successful completion of this project and upload that to Blackboard. Also, don't forget to push your final code to the remote repository.

To get truly accurate results you would also need to apply a temperature-dependent calibration factor and trigger a calibration (see the product data sheet if you are truly curious but the uncompensated results are good enough for this class).

Tap Detection, Revisited

In this assignment you will revisit the tap detection program from Lab 7. You will change the program so that communication takes place using I^{2}C.

Project requirements

- You will configure an accelerometer to detect single and double taps. Internal nRF52840 LEDs will be used to indicate whether a single tap or a double tap event has occurred.
- The majority of your code will be spent configuring the accelerometer prior to putting it into Measure mode and entering a `while` loop.
- The ADXL345 interrupts will be configured to set INT2 high when either a single or double tap event has occurred. INT2 will be connected to nRF52840 P0.04 (configured as a GPIO).
- Inside of the `while` loop, you will check to see if the accelerometer has indicated something happened (P0.04 switches from high to low). If it did, you will communicate with the ADXL345 via **I^{2}C** and determine whether a single or double tap was detected. Note: when a double tap occurs the ADXL345 will also say a single tap occurred (because a double tap consists of at least one tap). We want **our** single tap indicator to mean that one and only one tap was detected so be careful with the logic.
- It will turn on LED1 if a single tap was detected (but not one that was part of a double tap) and LED2 if a double tap was detected. These will remain lit for 1 second.

Tasks

1. Access the GitHub Classroom link for this assignment on Blackboard.
2. Follow the usual steps for getting started with a repository from GitHub Classroom.
3. The configuration of the accelerometer should occur once, outside of the `while` loop.
4. Don't forget to set the power control register to Measure mode as the last step before entering the `while` loop.
5. Inside of the `while` loop, your program should check to see if there is an alert on P0.04. If there is, then communicate with the accelerometer and read the interrupt source byte.
6. Set the LEDs if a tap has been detected and then keep them on for 1 second.

7. When you are satisfied with its operation, finish documenting your code with comments and a detailed `README.md`.

   When your program and circuit are working successfully, remember to push the commits to the remote repository. Also, take a video of its successful operation (along with your reflection) and upload this to Blackboard.