Prof. Dr. Marco Zimmerling Max Granzow Jonas Kubicki
https://nes-lab.org/

Networked and Low-Power Embedded Systems
WS 2025/26

Lab 4: Sensing & I2C

Introduction

In today's lab, we will interface with an external accelerometer, that we connect to the development kit over GPIO. Acceleration sensors, often employed in various applications such as motion sensing, gaming, and automotive systems, provide a rich source of data for understanding and responding to changes in an object's velocity and orientation.

We communicate with the accelerometer via the widely used synchronous serial communication protocol I2C (Inter-Integrated Circuit), also known as Two-Wire Interface (TWI). The protocol's simplicity and efficiency makes it an excellent choice for connecting sensors and other peripherals. I2C employs a master-slave architecture where a master device (in our case, the nRF52 SOC) initiates and controls communication with one or more slave devices (e.g. the accelertion sensor). Each device on the I2C bus has a unique address, allowing the master to selectively communicate with specific slaves. The "two-wire" designation refers to the fact that I2C uses only two communication lines:

- *Serial Clock Line (*`SCL`*):* This line carries clock pulses generated by the master to synchronize data transmission.
- *Serial Data Line (*`SDA`*):* This bidirectional line carries the actual data between the master and slave devices.

I2C is known for its scalability, supporting data rates ranging from a few kilobits per second to several megabits per second, depending on the specific implementation and hardware capabilities. This flexibility makes I2C suitable for various applications, from low-power, low-speed sensor interfaces to high-speed data transfers in more complex systems.

1 Configuring the I2C (TWI) Peripheral

We start with the configuration of the I2C/TWI peripheral on the nRF52 chip. The nRF52840 has three sligthly different TWI peripherals. A basic TWI peripheral is described in the datasheet in Section 6.29. Section 6.31 describes an extended TWI interface for I2C master devices

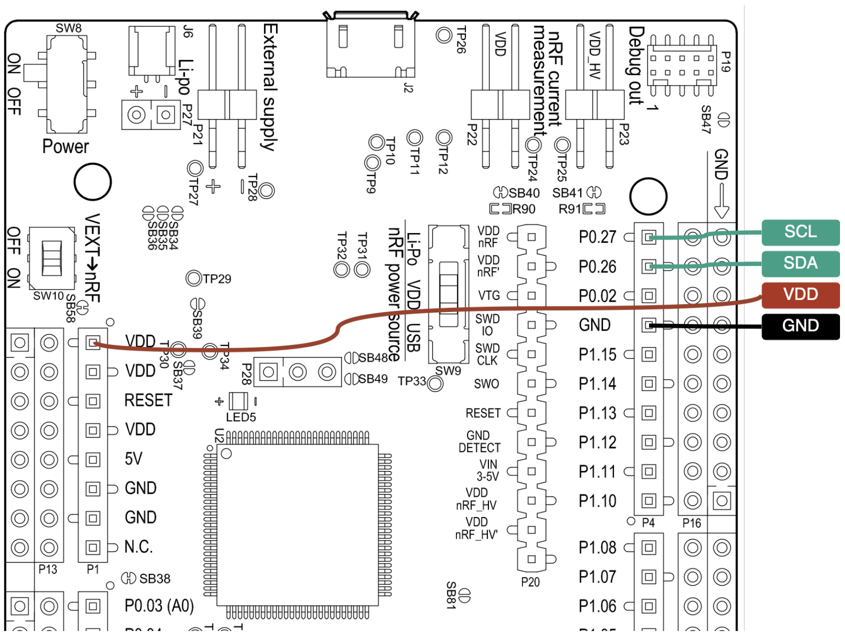

Figure 1: Pin mapping of the development kit.

(TWIM) and Section 6.32 for slave devices (TWIS). In this lab we will use the TWIM peripheral.

> The TWIM peripheral utilizes Nordic's EasyDMA functionality, which provides peripherals easy access to RAM via DMA transfers. This allows us to use the TWI communication buffers for transmissions and receptions without the CPU.

Download the source code from Moodle and follow the steps below to complete the code in the file `task1.c`. Remember the steps from lab 1 to compile and flash your program. Compile the source code with `$ make task1`.

1. The configuration of the TWIM0 peripheral is simple. We only need to specify the mapping of the two signals SCL and SDA to physical pins, and the desired operating frequency. Figure 1 shows the mapping of pins on the accelerometer breakout board and the development kit. The registers of the TWIM peripheral can be found in Section 6.31.7 on page 473 ff. Use Figure 1 to find out the pins for the two signals and configure the `PSEL.SDA` and `PSEL.SCL` registers accordingly. (Tip: This step is similar to the UART configuration from the last lab.) Configure the TWIM peripheral to operate at a frequence of 250 kbps.

2. Implement the `twi_write()` function such that we can send data from the master to the slave device. Refer to Section 6.31.2 in the datasheet. The accelerometer on the accelerometer breakout board is a NXP MMA8452Q with device address `0x1D`. Note that I2C uses 7 bit addresses as the 8th bit is used to indicate whether the transfer should be a read or write operation. Fortunately, the TWIM peripheral of the nRF52 chip already takes over several steps in the I2C transfer, for example, the write bit is automatically set after the address when starting a write operation via `STARTTX`.

   As described at the end of Section 6.31.2, the I2C write transfer should be stopped as fast as possible after the last byte has been sent. The nRF52 chip offers in this case (as well as for many other cases and peripherals) a convinient “shortcut” functionality, where certain events can be linked to certain tasks, which are then executed with minimal delay. Look at the `SHORTS` register and enable the corresponding shortcut.

In addition to the address and the shortcut, we only need to specify the number of bytes we want to write (`TXD.MAXCNT` register) and a pointer (`TXD.PTR` register) to the location of our transmission data.

3. To read data from the slave device, see Section 6.31.3 and implement the `twi_read()` function, which works very similarly to the `twi_write()` function.
4. Some I2C peripherals use a repeated start sequence. Instead of ending a transmission after a write from the master to the slave, another start condition is sent with a following read action. The accelerometer we use requires a repeated start condition to access some of its registers. Details are outlined in Section 6.31.4 and can be used to implement the `twi_write_read()` function. It is a combination of a write and a following read step as in the steps before. You can use the available shortcuts for the I2C interations.

2 Inspecting the I2C Transfer

We will now communicate with the accelerometer and check our implementation from task 1 using the LA. The I2C communication with the accelerometer is described in its datasheet `MMA8452Q`, which you can find on Moodle, in Section 5.10.

To configure the accelerometer, we need to write to the sensor’s registers. How write accesses work are depicted in Figure 2.

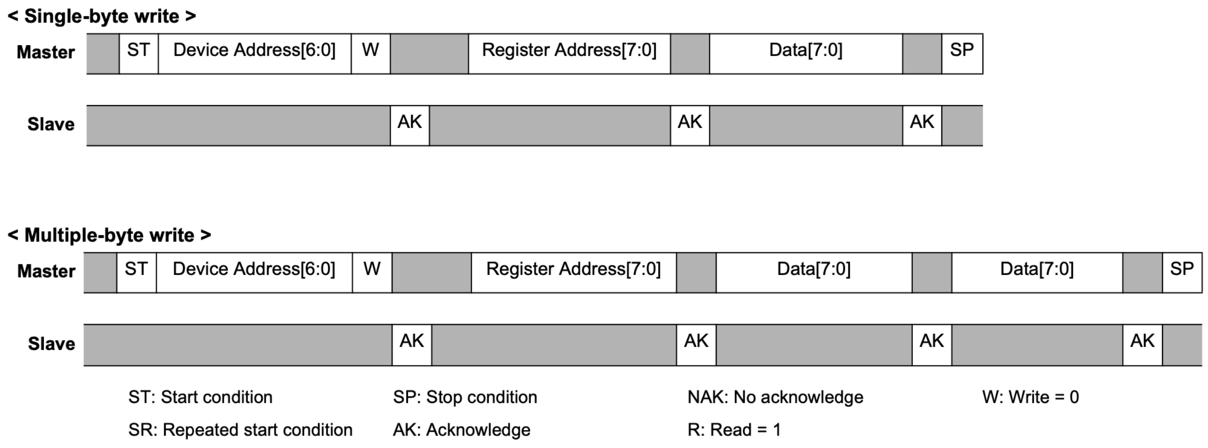

Figure 2: I2C write operation for the MMA8452Q accelerometer.

As always in I2C, the first byte that we transmit contains the address of the slave device and the read/write bit, which is set to 0 in this case to signal a write operation. The second byte that we transmit specifies the register to which we want to write. An overview of all registers and their addresses can be found in the datasheet of the sensor on page 19 ff. Finally, the third byte contains the value that is to be written to the specified register.

Read accesses are shown in Figure 3.

To read the contents of registers, we must first carry out a write transfer with the register address that we want to read. Then the master continues the I2C transfer with a repeated start

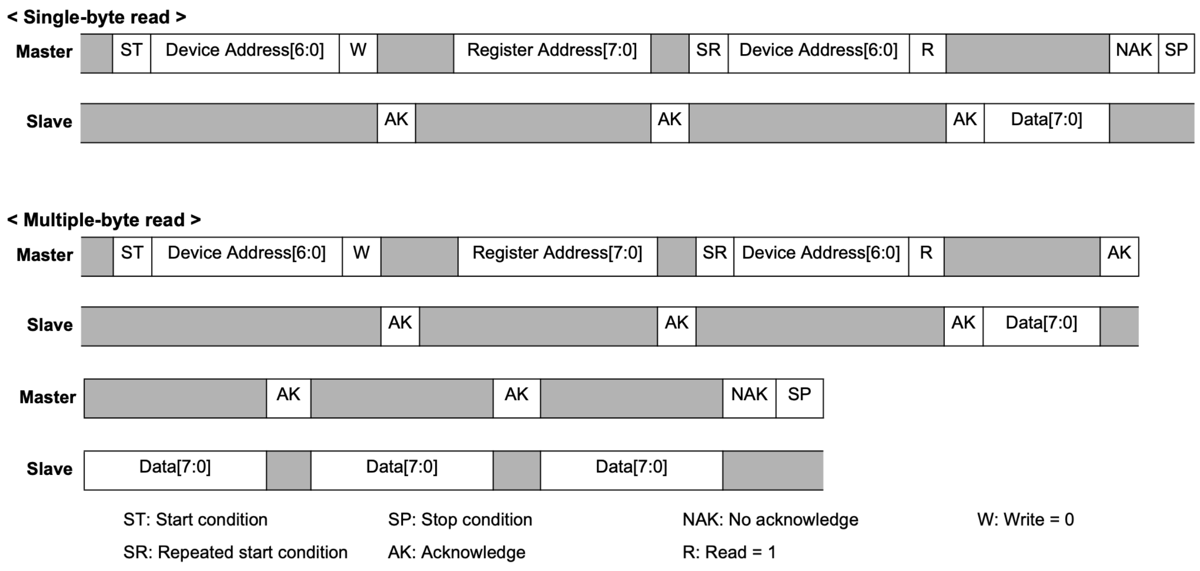

Figure 3: I2C read operation for the MMA8452Q accelerometer.

condition, but this time with the read/write bit set to 1 to perform a read operation. The slave then sends the register contents byte by byte. Our sensor supports multi-byte reads and automatically increments the register addresses and returns their contents until the master stops the transfer. Using the TWIM peripheral on our nRF52 chip, the stop condition is automatically generated when the specified number of bytes (`RXD.MAXCNT`) have been read.

With this knowledge you can now carry out the following steps and complete the code in the file `task2.c`. Compile the source code with `$ make task2`.

1. Copy the code of `twi_read()`, `twi_write()`, and `twi_write_read()` from task 1.
2. Define two buffers (i.e. arrays) above the start of the `main()` function. The data to be transmitted should be stored in one buffer, while the other buffer stores the received data.
3. Read the chip id (register `WHO_AM_I`) of our sensor. Print the read value (the UART peripheral is already configured) and verify that it is `0x2A`.
4. Take another look at the pin mapping of the accelerometer and the development kit in Figure 1 and connect the LA to `GND` (use the black cable in the bottom row on the left) and connect two of the cables in the top row to `SCL` and `SDA`. Start the `Logic 2` software, wait until the LA is detected, and record for about 5 s. Go to the “Analyzers” menu on the right-hand side, remove any existing analyzers and add the I2C analyzer. Examine the signals and check that everything works as you expect it to.

3 Recording and Processing Sensor Data

We have now prepared everything to actually read out the sensor values of the accelerometer. The goal is to measure the acceleration in all three axes (X,Y,Z) every 0.5 s. In addition, the current orientation of the board is to be determined and output. Differentiate between 6 different states, depending on which axis is pointing up or down (i.e., `X UP`, `X DOWN`, `Y UP`, `Y DOWN`, `Z UP`, and `Z DOWN`), see Figure 4.

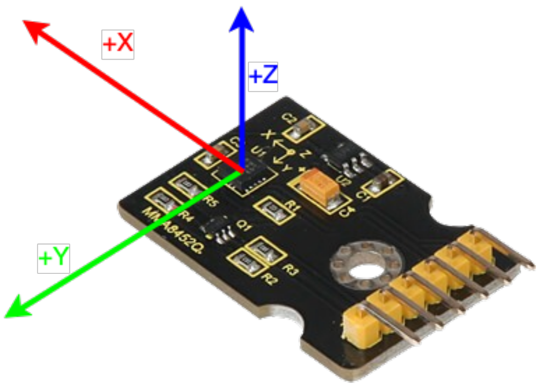

Figure 4: Orientation of the accelerometer on the breakout board.

Follow the steps below and complete the code in the file `task3.c`. Compile the source code with `$ make task3`.

1. Copy the code of `twi_read()`, `twi_write()`, and `twi_write_read()` from task 1.
2. The accelerometer uses different power modes and goes into sleep mode after boot-up. No sensor values can be obtained in this mode. We must therefore first switch the accelerometer to normal operating mode. See the `CTRL_REG1` register on page 19 in the sensors's datasheet and perform an I2C write transfer to put the sensor into active power mode.
3. The accelerometer stores the current acceleration values in 6 different registers, which are `OUT_X_MSB`, `OUT_X_LSB`, `OUT_Y_MSB`, `OUT_Y_LSB`, `OUT_Z_MSB`, and `OUT_Z_LSB`. In the code, the `TIMER` peripheral is already setup to trigger an interrupt every 0.5 s. Use the ISR `TIMER1_IRQHandler()` and read the sensor values by performing an I2C read transfer of 6 bytes. In this way, all values can be read at once and the accelerometer ensures that the values are not changed during this read transfer (data integrity). As mentioned in the datasheet, each sensor value is 12 bit wide and the LSB and MSB registers of each axis must be combined in the following way.

   ```
   // The following calculation applies to all axes.
   // NOTE: Depending on the orientation, the acceleration can be
   // positive or negative (data type!).
   ACC_X = ACC_X_MSB << 4 | ACC_X_LSB >> 4;
   if (ACC_X > 2047) ACC_X = ACC_X - 4096;
   ```

   Output the acceleration values of all axes in the ISR and see how they change when moving and rotating the device.
4. Now determine the orientation of the device based on the observed values. Implement the corresponding logic in the ISR as well and output the current orientation as one of the following states: `X UP`, `X DOWN`, `Y UP`, `Y DOWN`, `Z UP`, and `Z DOWN`.