Experiment 9: Inter-Integrated Circuit (I2C) Communication

1.0 Introduction

Serial interface formats are convenient because they require a small number of physical connections. They are often limited in the number of devices that can be connected at once, or they require cumbersome control logic to enable and disable devices. The Inter-Integrated Circuit (I2C) is a synchronous serial interface that uses an addressing scheme to select between multiple communication targets on a shared bus. Although it is generally slower than SPI, the flexibility of easily connecting multiple devices is useful when communication speed is not critical. In this lab, you will gain experience using and testing an I2C device. You will examine I2C protocol using an oscilloscope to identify correct and incorrect interface signals.

Step		Points
4.0	Prelab	40
6.1	Wiring	10
6.2	Setting up I2C	10
6.3	Complete the I2C methods	20
6.4	Reading the temperature sensor	15
6.5	Writing to the EEPROM	10
7.0	Post lab submission	*
Total		105

***All lab points are contingent on submission of code completed in lab.**

2.0 Objectives

1. To understand the concepts of I2C interfacing and communication
2. To learn how to use STM32F0 ADC peripherals to interact with an I2C device

3.0 Equipment and Software

1. Standard ECE362 Software Toolchain (Installation instructions available in Experiment 0: Getting Started)
2. STM32F0-DISCOVERY development board (For procurement instructions, see Experiment 0: Getting Started)

4.0 Prelab

In answering prelab questions, you should consult the following materials:
Your textbook
Lecture notes
ECE 362 web reference document page
https://i2c.info/i2c-bus-specification
https://www.i2c-bus.org/
https://learn.adafruit.com/i2c-addresses?view=all

5.0 Background

5.1 I2C Communication

An I2C interface allows many devices to share a two-wire bus and refer to specific neighbors by their addresses. Each device uses an open-drain output driver for each signal with the expectation that external pull-up resistors will bring the signals back to the high state when no driver is pulling them low. Devices may act as a master. A master is responsible for driving a clock signal (SCL) that is seen by all other devices that share the bus. Synchronous data is conveyed by the SDA signal during the high phase of the SCL signal and the SDA signal changes while SCL is in the low phase. An exception is when SDA changes state while SCL is still in the high state. Every I2C transaction begins with a START condition represented by the SDA line transitioning from high to low while SCL is still high. Every I2C transaction ends with a STOP condition represented by the SDA line transitioning from low to high while SCL is still high.

Between the START and STOP conditions, one or more packets of nine data bits are transferred. An initial packet sent by a master to a particular slave device contains a 7-bit address followed by a one-bit intent. The intent indicates whether the transaction is going to read from the slave (represented by a '1' bit) or write to the slave (with a '0' bit). The selected slave device, whose address matches the address in the initial packet, replies to the master by sending a '0' bit in reply as the ninth bit of the packet. The ninth bit is called an acknowledgement (ACK). If the slave device is not ready, if the slave is terminating a multi-packet transaction, or if there is no such device that matches the address, then nothing will pull the SDA line low. This is called a negative acknowledgement (NACK).

The master sends the initial packet containing the slave address and intent, and the slave sends the ACK bit. If the intent to write is indicated, multiple subsequent nine-bit exchanges can occur. If the intent to read is indicated, then multiple nine-bit exchanges can also occur, but this time the selected slave device will send the eight data bits and the master will respond with an ACK bit.

5.2 STM32 I2C interface

Your STM32 development system contains two independent I2C channels, each of which have two signals that can be routed to the external pins. Each channel uses seven main I/O registers to orchestrate the I2C transactions:

1) The I2C_TIMINGR register is used to select a prescale value and four other values to configure the duration of the SCL signal's high and low periods, as well as the expected setup and hold times for SDA with respect to SCL.
2) The I2C_CR1 register is used to set up long-term device configuration parameters as well as to enable the device (with the PE bit).
3) The I2C_CR2 register is used by a master to start or stop a transaction as well as to indicate the slave address and number of bytes to send to or receive from the slave device.

4) The I2C_TXDR register is used to send a byte to the selected target device.
5) The I2C_RXDR register is used to receive a byte from the selected target device.
6) The I2C_ISR register shows the transaction status. E.g., completed, NACK, STOP, etc.
7) The I2C_ICR register is used to clear any condition indicated by the ISR register.

By using these registers, any typical I2C transaction can be conducted.

5.3 Viewing an I2C transaction on your lab station oscilloscope

The oscilloscope at your lab station can be used as a high-level monitor for several serial protocols, including I2C. To do so, use the following procedure:

1) Connect the channel 1 and 2 probes of the scope to the SCL and SDA signals. Make sure that the ground clips of both probes are attached to zero volts. (Use a rigid device, such as a LED plugged into the ground strip of a breadboard, to attach the ground clips. Never attach ground clips to your STM32 development board. You will almost certainly slip, short power and ground pins, and destroy an on-board diode if you do this.)
2) While continuous I2C communication is in operation, press the "Auto Scale" button in the upper right of the scope controls. This should allow you to see the square waves. If you press the "Single" button in the upper right corner of the scope controls, you will freeze the display on one snapshot of the communication. You may use the "x pos" dial in the upper middle of the scope controls to slide the captured waveforms to the left or right. Press the "Run/Stop" button to continue scanning.
3) Press the "Serial" button (located either between channels 1 & 2, or in the gray box on the middle right of the controls). Choose Serial 1, set the mode for I2C, and choose a 7-bit address size. Set the signals to tell the scope which channel is connected to SDA and SCL. Now, when you press "Run/Stop" and "Single" the snapshot will show a third row at the bottom of the screen which interprets the I2C information for you. It uses notation such as "S50WaP" to indicate a single packet that begins with a START bit (S), followed by the 7-bit address 0x50, followed by a low bit which indicates the intent to write (W), followed by an acknowledgement (a), followed by a STOP bit (P). By using this interpreter, you can avoid having to analyze the waveforms by hand.

You will use the scope to view multiple steps of this lab experiment.

6.0 Experiment

6.1 Wiring

Refer to the datasheet for the 24AA32AF I2C EEPROM on the ECE 362 reference web page. Take the 24AA32AF from your lab kit and connect the V_{CC} and V_{SS} pins of the EEPROM to 3V and ground, connect the A0, A1, A2 pins to ground, and connect the WP pin to allow normal read/write operation. Connect the SCL and SDA pins of the EEPROM to the SCL and SDA pins of the I2C1 channel of your STM32. For this experiment, use PB6 and PB7 for I2C1. Connect a 1K resistor to each of these signals to act as a pull-up.

Refer to the datasheet for the TC74A0-3.3VAT (the one in your lab kit is in the TO-220 package) on the ECE362 reference web page. Connect the V_{DD} of the TC74 to 3V, GND to GND, and SCL and SDA to the corresponding pins on your STM32 development board.

Finally, you will use the serial port for reporting information. Configure that exactly as you did for homework 4.

Uncomment **`test_wiring()`** in your main program, and open the serial port with a terminal program. If you configured everything correctly it will display "Wiring tested!! Completed correctly, proceed to the next step".

Show your completed circuit to your TA before going on.

6.2 Setting up I2C

For this section, you will need to complete the following function:

1) **`init_I2C1()`**: This function should do the following:
    1. Enable clock to GPIOB.
    2. Configure PB6 and PB7 to alternate functions I2C1_SCL and I2C1_SDA.
    3. Enable the clock to I2C1 in the RCC.
    4. Set I2C1 to 7 bit mode.
    5. Enable NACK generation for I2C1.
    6. Configure the I2C1 timing register so that PRESC is 4, SCLDEL is 3 and SDADEL is 1 and SCLH is 3 and SCLL is 9.
    7. Disable own address1 and own address 2 and set the 7 bit own address to 1.
    8. Enable I2C1.

Uncomment **`prob2()`** in your main program, and open the serial port with a terminal program. If you configured everything correctly it will display "6.2 done!".

Demonstrate this to your TA.

6.3 Complete the I2C methods

For this section, you will need to complete the following functions:

1) **`i2c1_start()`**: This method has two parameters: the I2C slave address and the direction of transfer. It does not return any value. It should perform the following:
    a. Clear the SADD bits in I2C1_CR2.
    b. Set the SADD address in I2C1_CR2.
    c. Check the direction bit with 'RD' (see #define). If dir = = RD, then set the RD_WRN bit in CR2 register.
    d. Set the START bit in CR2 register.
2) **`i2c1_stop()`**: This method has no parameters. It does not return any value. It should perform the following:

a. Check if the STOPF flag is set in ISR register. If so, return.
b. Set the STOP bit in CR2.
c. Wait for the STOPF flag to be set in ISR register.
d. Clear the STOPF flag by writing to the STOPCF bit in the ICR register.

3) **`i2c1_senddata()`**: This method has two parameters: a pointer to the first element of an array ('data') and the size of that array. It sends the array contents over the I2C bus. It returns success or failure of sending the data. Use constants FAIL and SUCCESS (see the #define statements) for the return value to indicate failure or success. It should perform the following:
    a. Clear the NBYTES of CR2.
    b. Set the NBYTES bits to the parameter size.
    c. Write a 'for' loop that iterates 'size' number of times.
        i. Initialize a variable timeout to 0.
        ii. Wait for I2C_ISR_TXIS to be 1.
            1. While waiting for the bit to be set increment timeout.
            2. If timeout exceeds 1000000,
            3. Return FAIL
        iii. Set TXDR to data[i]; where i is the ith iteration of the 'for' loop.
    d. Wait until TC flag is set or the NACK flag is set.
    e. If NACKF flag is set, return FAIL.
    f. Else return success.
4) **`i2c1_readdata()`**: This method has two parameters: a pointer to the first element of an array ('data') and the size of that array. It fills the array with data from the I2C bus. It returns success or failure for receiving the data. Use constants FAIL and SUCCESS (see the #define statements) for the return value to indicate failure or success. It should perform the following:
    a. Clear the NBYTES of CR2.
    b. Set the NBYTES bits to the parameter size.
    c. Write a 'for' loop that iterates 'size' number of times.
        i. Initialize a variable timeout to 0.
        ii. Wait for I2C_ISR_RXNE to be 1.
            1. While waiting for the bit to be set, increment timeout.
            2. If timeout exceeds 1000000,
            3. Return FAIL.
        iii. Read RXDR to data[i]; where i is the ith iteration of the 'for' loop.
    d. Wait until TC flag is set or the NACK flag is set.
    e. Else return success.

Uncomment **`prob3()`** in your main program, and open the serial port using a terminal program. If you configured everything correctly it will display "6.3 done!".

6.4 Reading the temperature sensor

For this section, you will complete the **`read_temperature()`** method, which will return the temperature as a 32 bit signed integer (even though it is read from the sensor as an 8-bit signed integer).

This method should do the following:

1) Call **`I2C1_waitidle()`**.

2) Call **`I2C1_start()`** with address of TC74A0 (see datasheet), with WR as the direction.

3) Call **`I2C1_senddata(data,1)`** where data is an array whose single element is initialized to 0.

4) Call **`I2C1_start()`** but this time with RD. The technical jargon for this is “repeated start”.

5) Call **`I2C1_readdata(&temp, 1)`** where temp is pre-initialized to -20.

6) Call **`I2C1_stop().`**

7) Return **`temp`**.

Uncomment **`prob4()`** in you main program, open the serial port. If you configured everything correctly it will display “6.4 done! Temperature in C = <value>”.

Demonstrate this to your TA.

6.5 Write values into the EEPROM

For this section, you will complete the **`write_eeprom()`** method, which will write the “data” at the address “addr”.

This method should do the following:

1) Initialize an array **`write_buf[3]`** where the first element is the higher byte of parameter ‘wr_addr’, the second element is the lower byte of ‘wr_addr’, and the third element is ‘data’.

2) Call **`I2C1_waitidle()`**.

3) Call **`I2C1_start()`** with I2C address of the EEPROM (see datasheet), with WR as the direction.

4) Call **`I2C1_senddata(write_buf, 3)`**.

5) Call **`I2C1_stop()`**.

6) Wait for 5ms.

Uncomment **`prob5()`** in you main program, and open the serial port using a terminal program. It will open an interactive serial terminal where you can read and write to the EEPROM. If you can write and read the same value at the same address, your **`write_EEPROM()`** works correctly.

Demonstrate this to your TA.

6.6 Complete post lab and submit code

Submit your code and complete the post lab on the website. The portal for post-lab will close 10 minutes after your scheduled lab.