# Lab 7

# Inter-Integrated Circuit (I2C) Communication

In this lab, we will learn the I2C communication interface and apply it by communicating with the light sensor that's on the on Educational BoosterPack board.

## 7.1 I2C Transmission

I2C stands for Inter-Integrated Circuit and is designed to communicate between multiple chips over a short distance, e.g. chips that are on the same PCB. I2C has two wires, the Serial Data (SDA) and Serial Clock (SCL) and is based on a bus topology. The term bus means that multiple devices are connected to the same set of wires as shown in Figure 7.1. This topology is flexible since it's possible to add new devices without having to use extra pins at the MCU. Each device on the bus is assigned a 7-bit address so that the devices can be distinguished from one another. One of the devices, usually the MCU, is designated as the master and has two main responsibilities: the master generates the clock signal and shares it with the other devices; secondly, transmissions (read or write) can only be initiated by the master. The two wires of I2C are pulled-up to Vcc via pull-up resistors. Therefore, they read high if no action is done.

Otherwise, devices must actively pull the wires to low to transmit a value of zero. I2C is considered to be synchronous since the clock signal is shared between the devices.

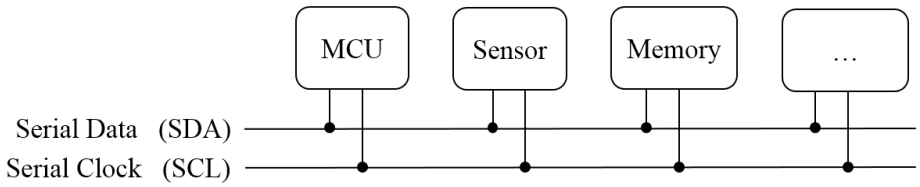


Figure 7.1: The I2C Bus

Figure 7.2 illustrates the read and write operations of the I2C protocol. A frame is delimited by the Start and Stop signals, which have unique patterns that can be distinguished from data bits. The top part of the figure illustrate a read operation in which the master reads two bytes from device address 0x22. The master starts by transmitting the Start signal, followed by a byte that contains the 7-bit address and the R/W' (read/write') bit. This bit is interpreted as 1:read and 0:write. Next, the device acknowledges receiving the request. The device then transmits the first byte (0x12) and the master acknowledges (Ack) it. The device then transmits the second byte (0x34). The master doesn't acknowledge the last byte (Nack = No Ack) so that the device stops transmitting. This enables the master to transmit the Stop signal. The data received is 0x1234 if we consider the first byte to be the most significant byte (MSB).

The bottom part of the figure illustrates a write operation in which the master writes two bytes to device address 0x33. The master starts by transmitting the Start signal, followed by the byte that contains the address and the R/W' bit. Next, the device acknowledges receiving the request. The master then transmits the data bytes which are acknowledged by the device. The master terminates the exchange by transmitting the Stop signal. The data received from the device is 0xABCD if we consider the first byte to be the most significant byte. Note that all the bytes are acknowledged by the device in a write operation.

```
Master                                                    Device
  ->        ->        <-    <-      ->    <-      ->      ->
Start / Address,R / Ack / Data / Ack / Data / Nack / Stop
          0x22              0x12          0x34

Master                                                    Device
  ->        ->        <-    ->      <-    ->      <-      ->
Start / Address,W / Ack / Data / Ack / Data / Ack / Stop
          0x33              0xAB          0xCD
```

Figure 7.2: I2C read (top) and write (bottom) operations

### I2C Device with Internal Registers

An I2C device, e.g. a sensor, usually has multiple internal registers organized in a layout as shown in Figure 7.3. Let's assume that the device in the figure is a temperature sensor. The sensor's address is 0x22 and the sensor has multiple internal registers. Each register is 16-bit and has an 8-bit address. Register 0x50 contains the model number and always reads as 0x1234. Such a register is non-writable and is used for discovery or testing purposes. Register 0x60 is the configuration register, to which we write in order to configure the sensor. Register 0x70 contains the sensor's result and is read by the MCU. All such details about a sensor are outside the scope of I2C and are described in the sensor's data sheet.

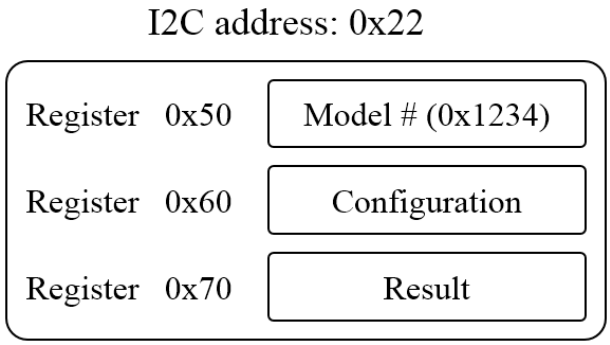


Figure 7.3: An I2C device with internal registers

The updated read and write operations that access the internal registers of an I2C device are shown in Figure 7.4. In the top part of the figure, the master reads the Model # register, which is register 0x50 on I2C address 0x22. The master starts with a write operation because, first, it needs to specify the internal address 0x50. Once the master transmits 0x50 and the device acknowledges it, the master puts a Repeated Start signal (there's no Stop Signal prior), then follows up with a read request to device 0x22. The master then reads two bytes and receives 0x1234.

I2C devices usually remember the last internal register that was read. This means if the master wants to read the Model # register again, it wouldn't have to transmit the value 0x50 again. It can immediately

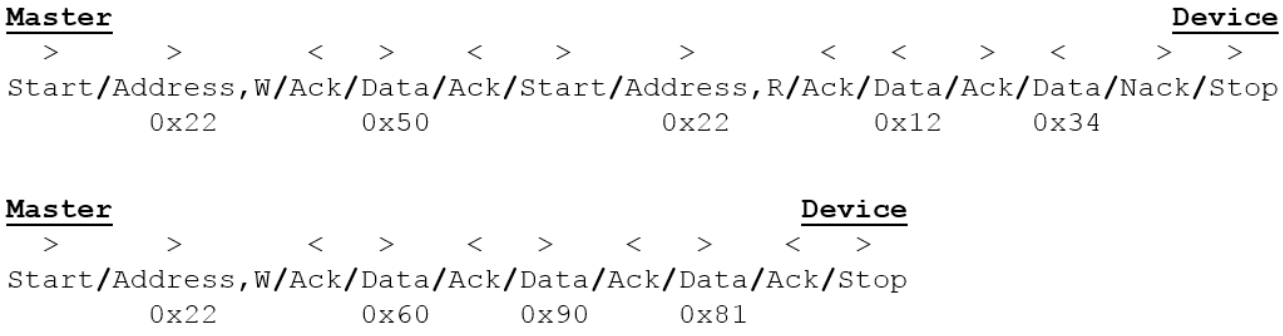


Figure 7.4: Read (top) and write (bottom) operations with the device's internal registers

start with a read request and read two bytes. The sensor will provide the content of register 0x50 since this was the last register that was read. If the device supports such a feature, it would be described in its data sheet.

In the bottom part of Figure 7.4, the master writes the value 0x9081 to the configuration register address. In this operation, the master writes three bytes back-to-back. The first byte is interpreted by the device as the internal register. Therefore, the first byte written is 0x60 (designates the configuration register) and it's followed by the two bytes 0x90 and 0x81.

**I2C Modes and the eUSCI Module**

The I2C protocol supports multiple modes, each designating a maximum clock frequency. The modes are described in the official I2C specs document[1] and are summarized below. In order to use a specific mode, both the master and the device(s) should support it. Most MCUs and I2C devices support the Standard Mode and the Fast Mode but not all devices support speeds beyond that since more sensitive filters in the circuitry would be required.

```
Standard Mode         up to 100 KHz
Fast Mode             up to 400 KHz
Fast Mode Plus        up to 1 MHz
High Speed Mode       up to 3.4 MHz
Ultra Fast Mode       up to 5 MHz     (unidirectional)
```

The eUSCI module in our MCU supports only the Standard Mode and the Fast Mode as described in the FR6xx Family User's Guide `(slau3670)` on p. 819. Therefore, when using our MCU as I2C master, we should limit the I2C clock frequency to 400 KHz. The I2C clock can be configured by using either SMCLK (e.g. 1 MHz) or ACLK (e.g. 32 KHz) and then applying a divider to it. Modulators are not used with I2C since it's synchronous and, therefore, it's not essential to be very close to the target frequency as it wouldn't affect the correctness of communication. eUSCI supports 'device mode' in which the MCU oeprates as a regular device and responds to an address that we configure in the code.

**The Light Sensor**

The light sensor on the BoosterPack is the Texas Instruments model OPT3001. The sensor's spectral response closely matches the human eye's since it's designed for user experience applications such as light automation, street lights control, traffic lights, cameras and display backlight control. Download the sensor's data sheet by searching for its identifier, `sbos681b`, in a search engine. You will need to use the data sheet later in this lab.

As described in the data sheet, the light sensor supports the I2C Standard Mode (100 KHz), Fast Mode (400 KHz) and the High Speed Mode (up to 2.6 MHz). It turns out the sensor also requires a minimum

[1] Document: "UM10204: I2C-bus specification and user manual"

I2C frequency of 10 KHz. Therefore, based on joint requirements of the eUSCI module and the sensor, we can configure an I2C clock frequency in the range of 10-400 KHz.

**The BoosterPack & LaunchPad Setup**

In this lab, we will use the Educational BoosterPack sensor board which contains the TI OPT3001 light sensor and we'll attach it to the LaunchPad board using the 40-pin connector. Make sure the orientation is correct when you connect the two boards. Download the BoosterPack User's Guide `(slau599a)` document and look on p. 5 at the figure illustrating the 40-pin connector. The connector is organized into four rows, called jumpers J1 to J4, and they have a total of 40 pins labelled 1 to 40. The figure on p. 5 shows that the light sensor is connected to the pins of J1 that are shown below.

```
Serial clock (SCL):    I2C_SCL at J1.9
Serial data (SDA):     I2C_SDA at J1.10
```

Note the positions of J1.9 and J1.10 as the bottom two pins of the J1 jumper. Next, we'll look at the LaunchPad User's Guide `(slau627a)` on p. 17 for these two jumper pin locations. They correspond to the pins below. I2C is implemented in the Channel B of eUSCI.

```
UCB1SCL / P4.1  -->  eUSCI Module 1 Channel B clock / Port 4.1
UCB1SDA / P4.0  -->  eUSCI Module 1 Channel B data  / Port 4.0
```

By default, these pins are configured as P4.0/P4.1. To configure these pins to the I2C functionality, we'll look in the chip's data sheet `(slas789c)` on p. 103, where we find the following.

```
P4DIR=xx   P4SEL1=11   P4SEL0=00    LCDSz=00
```

The LCDSz bits are zero by default. The code below configures the pins to the I2C functionality.

```
// Configure pins to I2C functionality
// (UCB1SDA same as P4.0) (UCB1SCL same as P4.1)
// (P4SEL1=11, P4SEL0=00) (P4DIR=xx)
P4SEL1 |= (BIT1|BIT0);
P4SEL0 &= ~(BIT1|BIT0);
```

**eUSCI Module Configuration**

At this point, we are ready to configure the eUSCI module for operation as I2C master. The configuration registers in I2C mode are found in the Family User's Guide `(slau367o)`, Chapter 32, starting on p. 841.

The function below performs the configuration. It starts by configuring the I2C pins then engages the reset state of the eUSCI module since the configuration should be modified only in the reset state. The function then selects the I2C mode, master mode and SMCLK as the clock source. The default frequency of SMCLK is 1 MHz (1,000,000 Hz). The clock is divided by 8, resulting in a frequency of 125 KHz, which is supported by both the MCU and the light sensor. The function finally exits the reset state so the communication can begin.

```
void Initialize_I2C(void) {
    // Configure the MCU in Master mode

    // Configure pins to I2C functionality
    // (UCB1SDA same as P4.0) (UCB1SCL same as P4.1)
    // (P4SEL1=11, P4SEL0=00) (P4DIR=xx)
    P4SEL1 |= (BIT1|BIT0);
    P4SEL0 &= ~(BIT1|BIT0);

    // Enter reset state and set all fields in this register to zero
    UCB1CTLW0 = UCSWRST;

    // Fields that should be nonzero are changed below
    // (Master Mode: UCMST) (I2C mode: UCMODE_3) (Synchronous mode: UCSYNC)
    // (UCSSEL 1:ACLK, 2,3:SMCLK)
    UCB1CTLW0 |= UCMST | UCMODE_3 | UCSYNC | UCSSEL_3;

    // Clock frequency: SMCLK/8 = 1 MHz/8 = 125 KHz
    UCB1BRW = 8;
    // Chip Data Sheet p. 53 (Should be 400 KHz max)

    // Exit the reset mode at the end of the configuration
    UCB1CTLW0 &= ~UCSWRST;
}
```

### The Programming Model

The programming model of I2C consists of initiating the Start and Stop signals, listening to the acknowledgements (Acks) from the device and reading or writing bytes from/to the device. The detailed procedures are shown in the FR6xx Family User's Guide on p. 824-832 and are implemented by the functions in Appendix A. These functions assume the setup in Figure 7.3 where the I2C device has internal registers. Each internal register is 16-bit and has an 8-bit address, which is the setup used by the OPT3001 light sensor. Both of these functions return zero when they return successfully.

```
int i2c_read_word(unsigned char i2c_address, unsigned char i2c_reg, unsigned
    int * data);
```

```
int i2c_write_word(unsigned char i2c_address, unsigned char i2c_reg,
    unsigned int data);
```

The first function reads two bytes from an internal register on the I2C device. The third parameter is passed by reference and will have the data that was read from the device. The code below is an example that reads register 0x50 on I2C address 0x22. The parameter 'data' is passed by reference using the & sign.

```
// Reading two bytes from register 0x50 on I2C device 0x22
unsigned int data;
...
i2c_read_word(0x22, 0x50, &data);
```

The second function writes a 16-bit value to a register on the I2C device. The code in the example below writes 0xABCD to register 0x60 on I2C device 0x22.

```
// Writing 0xABCD to register 0x50 on I2C device 0x22
unsigned int data = 0xABCD;
...
i2c_write_word(0x22, 0x60, data);
```

**Reading the Manufacturer ID and Device ID Registers**

We are now ready to communicate with the light sensor using I2C. You need to start by finding the sensor's I2C address. As described in the sensor's data sheet `(sbos681b)`, it's possible to configure the sensor to multiple addresses using its address pin. In your report, describe what addresses are possible and how the configuration is done. The actual address of the sensor depends on the wiring of the BoosterPack board, therefore, look at the schematics in the BoosterPack User's Guide `(slau599a)` to find out how the light sensor is wired. Find two pieces of information from the schematic: the I2C address and the value of the pull-up resistors used on the I2C lines. Take a screenshot of the schematics portion showing this information.

The light sensor contains two internal 16-bit registers called Manufacturer ID and Device ID. These registers always return the same values and are helpful for testing purposes. Look in the sensor's data sheet `(sbos681b)` for a summary of the sensor's internal registers. You should be able to find the internal (8-bit) addresses of these two registers and the values they're supposed to return.

Write a code that combines the I2C initialization function above and the read and write functions in Appendix A and perform I2C transmissions that read the Manufacturer ID and Device ID registers. Add the UART functions to your code and transmit the data received from the sensor to the terminal application

on the PC so you can observe the values. Read the two registers from the sensor continuously in an infinite loop and add a delay loop to slow down the readings to about one per second.

Perform the following and submit the answers in your report:

- Write the program and demo it to the TA.
- What possible addresses can the sensor have and how is the address chosen?
- What is the address of the sensor as wired on the BoosterPack board?
- What is the value of the pull-up resistors on the I2C wires? Include a screenshot of the schematics highlighting the I2C address and the pull-up resistors.
- What are the addresses of the Device ID and Manufacturer ID registers and what values are they supposed to return?
- Submit the code in your report.

## 7.2 Reading Measurements from the Light Sensor

At reset, the light sensor starts in a low-power shutdown state so that it draws a very small current. It was possible to read the Manufacturer ID and Device ID registers in this state. However, in order to read light measurements, the sensor should be configured by writing to its configuration register. The sensor measures a lux value, which is the unit of measuring light intensity.

The light sensor returns a 16-bit result that consists of a 4-bit exponent (leftmost bits) and a 12-bit result (called mantissa). The 4-bit exponent specifies the value of the LSB bit. This setup is shown in the data sheet `(sbos681b)` in Table 9. We will set the exponent to 7, which makes the LSB worth 1.28. This means every time the (12-bit) result goes up by 1, the reading (lux value) goes up by 1.28. A 12-bit value varies between 0 and 4,095, which means the lux value varies from 0 to (4,095 x 1.28) 5,241 lux. This is the full range when the exponent is set to 7. This is a reasonable configuration since a well lit room registers a few hundred lux. If we shine a flash light close to the sensor, the lux value can reach the thousands.

As Table 9 shows, it's possible to configure the sensor so the full range is up to 83,865 lux. However, the LSB's worth, therefore the steps, become larger. Note that if the exponent is set to 12, the sensor automatically determines the most suitable exponent and returns it with the result.

Configure the sensor based on the setup below. Find the configuration register's address and layout in the data sheet. Based on the layout, derive the hex value that should be written to the configuration register.

```
RN=0111b=7          The LSB bit is worth 1.28
CT=0                Result produced in 100 ms
M=11b=3             Continuous readings
ME=1                Mask (hide) the Exponent from the result
```

When the last field ME (Mask Exponent) is set to 1, the result register will always have zeros in the leftmost 4-bit field that would normally have the exponent field. We chose a fixed exponent of 7 in our configuration, therefore, we know that the 4-bit exponent field will be 7 in every reading. Having this field set to zero makes our job easier since we only need to multiply the result by 1.28 to get the lux value. Otherwise, we would have to shift out the exponent field from the result. Note that if a variable exponent is chosen (where the sensor chooses the exponent with every reading), then it would be necessary to inspect the exponent field in the result in order to determine the multiplier.

Modify the previous code so that it reads the sensor repeatedly in an infinite loop. The delay loop should pace the readings to about one per second. Transmit the data over UART to the PC. Include an incrementing counter so you can see that the transmission is ongoing.

Interpret the data to check if the readings make sense. A well lit room should register 100 to 200 lux. A desktop lit by a desk lamp should register around 400 lux. Cover the sensor with your finger and check if the readings approach zero. Alternatively, shine your phone's flash light and check if the lux value increases as you move the phone closer to the sensor. The reading should max out at 5,241 lux. The sensor should be more responsive when the light is directed at it straight rather than from the sides.

Perform the following and submit the answers in your report:

- Write the program and demo it to the TA.
- What is the address of the configuration register on the sensor?
- What configuration value (hex) did you write to the sensor? Show how this value is formatted into bit fields.
- Do the sensor's readings seem reasonable and consistent?
- Submit the code in your report.

### 7.3 Application: Lux Logger

As an embedded engineer, you are in charge of implementing a lux logger. A sample output of the application is shown below. The logger takes a sensor reading every minute and prints it to the terminal along with a timestamp. At the start, low/high limit values are established which are -+10 lux from the current reading. When the reading goes out of range, a message of Up or Down is printed and the low/high limits are updated then. Based on the sample below, the first reading is 501 lux, therefore, the low/high limits are set to 491 lux and 511 lux, respectively. When the reading reached 513 lux, the message Up was printed and the low/high limits were updated to 503 lux and 523 lux. Later on, the reading reached 503 lux and the message Down was printed. Finally, implement a feature that enables the end user to set the time. When button S2 is pressed, the user can enter the time as shown in the sample. The keystrokes don't show on the monitor but the MCU confirms the time that was set.

One requirement in this code is to maintain the time using the timer via interrupt and updating the time each second.

```
*** Lux Logger ***

12:00   501 lux   <Up>

Enter the time...(3 or 4 digits then hit Enter)
Time is set to 12:54

12:54   501 lux
12:55   509 lux
12:56   510 lux
12:57   513 lux   <Up>
12:58   513 lux
12:59   514 lux
1:00    514 lux
1:01    510 lux
1:02    506 lux
1:03    503 lux   <Down>
1:04    501 lux
1:05    501 lux
```

Perform the following and submit the answers in your report:

- Write the program and demo it to the TA.
- Submit the code in your report.

## Student Q&A

1. The light sensor has an address pin that allows customizing the I2C address. How many addresses are possible? What are they and how are they configured? Look in the sensor's data sheet.
2. According to the light sensor's data sheet, what should be the value of the pull-up resistors on the I2C wires? Did the BoosterPack use the same values?
3. What I2C clock frequency do each of the eUSCI module and the sensor support?

# Appendix A

# I2C Functions

```
////////////////////////////////////////////////////////////////////
////////    Function Headers    ////////////////////////////////////
////////////////////////////////////////////////////////////////////
int i2c_read_word(unsigned char, unsigned char, unsigned int*);   //
int i2c_write_word(unsigned char, unsigned char, unsigned int);   //
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
// Read a word (2 bytes) from I2C (address, register)
int i2c_read_word(unsigned char i2c_address, unsigned char i2c_reg, unsigned
    int * data) {
    unsigned char byte1, byte2;

    // Initialize the bytes to make sure data is received every time
    byte1 = 111;
    byte2 = 111;

    //********** Write Frame #1 ***************************
    UCB1I2CSA = i2c_address;    // Set I2C address

    UCB1IFG &= ~UCTXIFG0;
    UCB1CTLW0 |= UCTR;          // Master writes (R/W bit = Write)
    UCB1CTLW0 |= UCTXSTT;       // Initiate the Start Signal
```

```
    while ((UCB1IFG & UCTXIFG0) ==0) {}

    UCB1TXBUF = i2c_reg;        // Byte = register address

    while((UCB1CTLW0 & UCTXSTT)!=0) {}

    if(( UCB1IFG & UCNACKIFG )!=0) return -1;

    UCB1CTLW0 &= ~UCTR;         // Master reads (R/W bit = Read)
    UCB1CTLW0 |= UCTXSTT;       // Initiate a repeated Start Signal
    //*****************************************************

    //********** Read Frame #1 ***************************
    while ( (UCB1IFG & UCRXIFG0) == 0) {}
    byte1 = UCB1RXBUF;
    //*****************************************************

    //********** Read Frame #2 ***************************
    while((UCB1CTLW0 & UCTXSTT)!=0) {}
    UCB1CTLW0 |= UCTXSTP;       // Setup the Stop Signal

    while ( (UCB1IFG & UCRXIFG0) == 0) {}
    byte2 = UCB1RXBUF;

    while ( (UCB1CTLW0 & UCTXSTP) != 0) {}
    //*****************************************************

    // Merge the two received bytes
    *data = (    (byte1 << 8) | (byte2 & 0xFF)    );
    return 0;
}




////////////////////////////////////////////////////////////////////
// Write a word (2 bytes) to I2C (address, register)
int i2c_write_word(unsigned char i2c_address, unsigned char i2c_reg,
    unsigned int data) {
    unsigned char byte1, byte2;

    byte1 = (data >> 8) & 0xFF;  // MSByte
    byte2 = data & 0xFF;         // LSByte
```

```
    UCB1I2CSA = i2c_address;     // Set I2C address

    UCB1CTLW0 |= UCTR;           // Master writes (R/W bit = Write)
    UCB1CTLW0 |= UCTXSTT;        // Initiate the Start Signal

    while ((UCB1IFG & UCTXIFG0) ==0) {}

    UCB1TXBUF = i2c_reg;        // Byte = register address

    while((UCB1CTLW0 & UCTXSTT)!=0) {}

    while ((UCB1IFG & UCTXIFG0) ==0) {}

    //********** Write Byte #1 **************************
    UCB1TXBUF = byte1;
    while ( (UCB1IFG & UCTXIFG0) == 0) {}

    //********** Write Byte #2 **************************
    UCB1TXBUF = byte2;
    while ( (UCB1IFG & UCTXIFG0) == 0) {}

    UCB1CTLW0 |= UCTXSTP;
    while ( (UCB1CTLW0 & UCTXSTP) != 0) {}

    return 0;
}
```