EEL 4742C: Embedded Systems

Christopher Badolato

11/18/2019

EEL4742-0011

Lab 9 Inter-Integrate Circuit (12C) Communication

Introduction

In this lab we will be implementing inter-integrated circuit communication (I2C) using the Educational booster pack with the MSP430. In the first part of this experiment we will read the device ID and the manufacturers ID each second and displayed to the terminal using UART. In the second part, the booster pack contains a light sensor that will be read from every second and displayed on the terminal.

Part 9.1: I2C Transmission

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| I2C works by using two busses, one for data SDA, and one for a clock SCL. Using this two-bus system we can sync the clock of the MSP and the booster pack, in doing so, we can read or write data to or from the booster pack.  The I2C transmission can read two bytes at time. The master (MSP430) will first send a start signal. Then the master sends the address of the device and whether we are meant to read or write with the device to that device. Once the master has received an acknowledge, the master can start sending or receiving data with the device as we mention before, 2 bytes at a time.   * If we are receiving data, the master device will send a NACK signal as well as a stop to signal. * If we are sending data, the device will acknowledge it has received the data, but will immediately stop, no NACK   We can interact with multiple internal registers of the device that will give us data or allow us to send data between the device and the master.  In the code we were given the I2C initialize function and the I2c read functions. We need to implement the UART write, and 32Khz clock that way we can write our value directly to the terminal.  Using the i2c\_read\_word function we will send the device location address (0x44) and the address of the value for the manufacturer’s ID (0x7E) and device ID (0x7F).  The code prints to the terminal the manufacturer’s ID and device ID every second.  What is the address of the Manufacturer ID register? What value does this register return?   |  | | --- | | 0x7E  This value returns the value store on the device pointed to by address. It will return the manufacturer ID |   What does this value mean?   |  | | --- | | It is a pointer to where the value is being store on the device. |   What is the address of the Device ID register? What value does this register return?   |  | | --- | | 0x7F  This value returns the value store on the device pointed to by address. It will return the device ID |   What is the light sensor’s I2C address?   |  | | --- | | 0x00 for configuration  0x01 for the results |   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.   |  | | --- | | The resistors are set to pull-up to high are set to 10k ohms  A map with text  Description automatically generated | |  | |
| //Christopher Badolato  //11/12/2019  //Lab 9.1  //EEL 4742 0011  //I2c intro  **#include** <msp430.h>  **#include** <stdio.h>  **#define** FLAGS UCA1IFG // Contains the transmit & receive flags  **#define** RXFLAG UCRXIFG // Receive flag  **#define** TXFLAG UCTXIFG // Transmit flag  **#define** TXBUFFER UCA1TXBUF // Transmit buffer  **#define** RXBUFFER UCA1RXBUF // Receive buffer  **#define** redLED BIT0 // Red LED at P1.0  **#define** greenLED BIT7 // Green LED at P9.7  // Configure eUSCI in I2C master mode  **void** **Initialize\_I2C**(**void**) {  // Enter reset state before the configuration starts...  UCB1CTLW0 |= UCSWRST;  // Divert pins to I2C functionality  P4SEL1 |= (BIT1|BIT0);  P4SEL0 &= ~(BIT1|BIT0);  // Keep all the default values except the fields below...  // (UCMode 3:I2C) (Master Mode) (UCSSEL 1:ACLK, 2,3:SMCLK)  UCB1CTLW0 |= UCMODE\_3 | UCMST | UCSSEL\_3;  // Clock divider = 8 (SMCLK @ 1.048 MHz / 8 = 131 KHz)  UCB1BRW = 8;  // Exit the reset mode  UCB1CTLW0 &= ~UCSWRST;  }  ////////////////////////////////////////////////////////////////////  // 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;  }  **void** **config\_ACLK\_to\_32KHz\_crystal**() {  // By default, ACLK runs on LFMODCLK at 5MHz/128 = 39 KHz  // Reroute pins to LFXIN/LFXOUT functionality  PJSEL1 &= ~BIT4;  PJSEL0 |= BIT4;  // Wait until the oscillator fault flags remain cleared  CSCTL0 = CSKEY; // Unlock CS registers  **do** {  CSCTL5 &= ~LFXTOFFG; // Local fault flag  SFRIFG1 &= ~OFIFG; // Global fault flag  } **while**((CSCTL5 & LFXTOFFG) != 0);  CSCTL0\_H = 0; // Lock CS registers  **return**;  }  // Configure UART to the popular configuration  // 9600 baud, 8-bit data, LSB first, no parity bits, 1 stop bit  // no flow control  // Initial clock: SMCLK @ 1.048 MHz with oversampling  **void** **Initialize\_UART2**(**void**){  // Divert pins to UART functionality  P3SEL1 &= ~(BIT4|BIT5);  P3SEL0 |= (BIT4|BIT5);  // Use ACLK clock; leave other settings default  UCA1CTLW0 |= UCSSEL\_1;  // Configure the clock dividers and modulators  UCA1BRW = 6;  UCA1MCTLW = UCBRS1 | UCBRS2 | UCBRS3 | UCBRS5 | UCBRS6 | UCBRS7;  // Exit the reset state (so transmission/reception can begin)  UCA1CTLW0 &= ~UCSWRST;  }  **void** **uart\_write\_char**(**unsigned** **char** ch){  // Wait for any ongoing transmission to complete  **while** ((FLAGS & TXFLAG) == 0 ) {}  // Write the byte to the transmit buffer  TXBUFFER = ch;  }  // The function returns the byte; if none received, returns NULL  **unsigned** **char** **uart\_read\_char**(**void**){  **unsigned** **char** temp;  // Return NULL if no byte received  **if**( (FLAGS & RXFLAG) == 0)  **return** NULL;  // Otherwise, copy the received byte (clears the flag) and return it  temp = RXBUFFER;  **return** temp;  }  **uart\_write\_uint16**(**unsigned** **int** currentValue){  **int** number;  **if**(currentValue >= 10000){  number = currentValue/10000;  uart\_write\_char('0'+ number);  }  **if**(currentValue >= 1000){  number = (currentValue/1000) % 10;  uart\_write\_char('0'+ number);  }  **if**(currentValue >= 100){  number = (currentValue/100) % 10;  uart\_write\_char('0'+ number);  }  **if**(currentValue >= 10){  number = (currentValue/10) % 10;  uart\_write\_char('0'+ number);  }  number = currentValue % 10;  uart\_write\_char('0' + number);  }  **int** **main**(**void**){  **unsigned** **int** manualID;  **unsigned** **int** devID;  **unsigned** **int** value;  WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer  PM5CTL0 &= ~LOCKLPM5; // Enable the GPIO pins  //32khz clock  config\_ACLK\_to\_32KHz\_crystal();  //configure uart  Initialize\_UART2();  Initialize\_I2C();  i2c\_read\_word(0x44, 0x7E, &manualID);  i2c\_read\_word(0x44, 0x7F, &devID);  //ACLK, div 1, upmode  TA0CTL = TASSEL\_1 | ID\_0 | MC\_1 | TACLR;  TA0CCR0 = (65536 - 1);  **for**(;;){  **for**(value = 0; value <= 65535; value++){  //write the current integer  uart\_write\_uint16(value);  uart\_write\_char(' ');  uart\_write\_uint16(manualID);  uart\_write\_char(' ');  uart\_write\_uint16(devID);  uart\_write\_char('\n');  //return  uart\_write\_char('\r');  **while**((TA0CTL & TAIFG) == 0){}  TA0CTL &= ~TAIFG;  }  }  } |

Part 9.2: Reading Measurements from the Light Sensor

|  |  |  |  |
| --- | --- | --- | --- |
| In this part of the lab we will be using the code from the previous part to print right to the terminal using data from the booster pack. We will be accessing the sensor this time with the i2c\_write\_word to configure the setting of the sensor to ensure we are receiving the data we actually want. Then we will read the data from the booster pack with this configuration set.  First we must send our i2c configuration with the i2c\_write\_word() function to address 0x01. In this case I created a mask variable and store the values (in binary) onto the mask.  Using the code below I shifted on the values we intend to send to the configuration register  R1,r2,r3 represent the value of our exponent in our case we set the exponent to 1.28 which means, for every 1 value we receiving it will actually represent 1.28 in lux.  **unsigned** **int** mask = 0, r0 , r1, r2, CT = 0, M1, M0, ME, r3;  r0 = 1<<12;  r1 = 1<<13;  r2 = 1<<14;  r3 = 0 << 15;  CT << 11;  M1 = 1<<10;  M0 = BIT9;  ME = BIT2;  mask |= (r0|r1|r2|CT|M1|M0|ME);  Once we have this mask set, we can send it to the configuration register on the booster pack to let us know which value to send back. The code below will send our mask to the configuration register  i2c\_write\_word(0x44, 0x01, mask);  What is the address of the configuration register on the sensor?   |  | | --- | | 0x01 |   What configuration value (hex) did you write to the sensor? Show how this value is formatted  into bit fields.   |  | | --- | | 0111 0110 0000 0100  0x7604 |   Does the data make sense based on what you expected?   |  | | --- | | Yes, we must multiply the mantissa value sent back by 1.28 to print to correct lux value to the terminal. | |
| //Christopher Badolato  //11/12/2019  //Lab 9.2  //EEL 4742 0011  //I2c intro  **#include** <msp430.h>  **#include** <stdio.h>  **#define** FLAGS UCA1IFG // Contains the transmit & receive flags  **#define** RXFLAG UCRXIFG // Receive flag  **#define** TXFLAG UCTXIFG // Transmit flag  **#define** TXBUFFER UCA1TXBUF // Transmit buffer  **#define** RXBUFFER UCA1RXBUF // Receive buffer  **#define** redLED BIT0 // Red LED at P1.0  **#define** greenLED BIT7 // Green LED at P9.7  // Configure eUSCI in I2C master mode  **void** **Initialize\_I2C**(**void**) {  // Enter reset state before the configuration starts...  UCB1CTLW0 |= UCSWRST;  // Divert pins to I2C functionality  P4SEL1 |= (BIT1|BIT0);  P4SEL0 &= ~(BIT1|BIT0);  // Keep all the default values except the fields below...  // (UCMode 3:I2C) (Master Mode) (UCSSEL 1:ACLK, 2,3:SMCLK)  UCB1CTLW0 |= UCMODE\_3 | UCMST | UCSSEL\_3;  // Clock divider = 8 (SMCLK @ 1.048 MHz / 8 = 131 KHz)  UCB1BRW = 8;  // Exit the reset mode  UCB1CTLW0 &= ~UCSWRST;  }  ////////////////////////////////////////////////////////////////////  // 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) {}  //\*\*\*\*\*\*\*\*\*\* 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;  }  ////////////////////////////////////////////////////////////////////  // 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;  }  **void** **config\_ACLK\_to\_32KHz\_crystal**() {  // By default, ACLK runs on LFMODCLK at 5MHz/128 = 39 KHz  // Reroute pins to LFXIN/LFXOUT functionality  PJSEL1 &= ~BIT4;  PJSEL0 |= BIT4;  // Wait until the oscillator fault flags remain cleared  CSCTL0 = CSKEY; // Unlock CS registers  **do** {  CSCTL5 &= ~LFXTOFFG; // Local fault flag  SFRIFG1 &= ~OFIFG; // Global fault flag  } **while**((CSCTL5 & LFXTOFFG) != 0);  CSCTL0\_H = 0; // Lock CS registers  **return**;  }  // Configure UART to the popular configuration  // 9600 baud, 8-bit data, LSB first, no parity bits, 1 stop bit  // no flow control  // Initial clock: SMCLK @ 1.048 MHz with oversampling  **void** **Initialize\_UART2**(**void**){  // Divert pins to UART functionality  P3SEL1 &= ~(BIT4|BIT5);  P3SEL0 |= (BIT4|BIT5);  // Use ACLK clock; leave other settings default  UCA1CTLW0 |= UCSSEL\_1;  // Configure the clock dividers and modulators  UCA1BRW = 6;  UCA1MCTLW = UCBRS1 | UCBRS2 | UCBRS3 | UCBRS5 | UCBRS6 | UCBRS7;  // Exit the reset state (so transmission/reception can begin)  UCA1CTLW0 &= ~UCSWRST;  }  **void** **uart\_write\_char**(**unsigned** **char** ch){  // Wait for any ongoing transmission to complete  **while** ((FLAGS & TXFLAG) == 0 ) {}  // Write the byte to the transmit buffer  TXBUFFER = ch;  }  // The function returns the byte; if none received, returns NULL  **unsigned** **char** **uart\_read\_char**(**void**){  **unsigned** **char** temp;  // Return NULL if no byte received  **if**( (FLAGS & RXFLAG) == 0)  **return** NULL;  // Otherwise, copy the received byte (clears the flag) and return it  temp = RXBUFFER;  **return** temp;  }  **uart\_write\_uint16**(**unsigned** **int** currentValue){  **int** number;  **if**(currentValue >= 10000){  number = currentValue/10000;  uart\_write\_char('0'+ number);  }  **if**(currentValue >= 1000){  number = (currentValue/1000) % 10;  uart\_write\_char('0'+ number);  }  **if**(currentValue >= 100){  number = (currentValue/100) % 10;  uart\_write\_char('0'+ number);  }  **if**(currentValue >= 10){  number = (currentValue/10) % 10;  uart\_write\_char('0'+ number);  }  number = currentValue % 10;  uart\_write\_char('0' + number);  }  **int** **main**(**void**){  **unsigned** **int** result;  **unsigned** **int** exponent, mantissa;  **unsigned** **int** mask = 0, r0 , r1, r2, CT = 0, M1, M0, ME, r3;  r0 = 1<<12;  r1 = 1<<13;  r2 = 1<<14;  r3 = 0 << 15;  CT << 11;  M1 = 1<<10;  M0 = BIT9;  ME = BIT2;  mask |= (r0|r1|r2|CT|M1|M0|ME);  **unsigned** **int** value;  WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer  PM5CTL0 &= ~LOCKLPM5; // Enable the GPIO pins  //32khz clock  config\_ACLK\_to\_32KHz\_crystal();  //configure uart  Initialize\_UART2();  Initialize\_I2C();  i2c\_write\_word(0x44, 0x01, mask);  TA0CTL = TASSEL\_1 | ID\_0 | MC\_1 | TACLR;  TA0CCR0 = (65536 - 1);  **for**(;;){  **for**(value = 0; value <= 65535; value++){  //write the current integer  i2c\_read\_word(0x44, 0x00, &result);  mantissa = (1.28 \* result);  uart\_write\_uint16(value);  uart\_write\_char(' ');  uart\_write\_uint16(mantissa);  uart\_write\_char('\n');  //return  uart\_write\_char('\r');  **while**((TA0CTL & TAIFG) == 0){}  TA0CTL &= ~TAIFG;  }  }  } |

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.

|  |
| --- |
| Four I2C address are possible by connecting the ADDR pin to one of four pins: GND CDD, CDA, or SCL  They are locations where our signals are sent or received.  Each register has an 8-bit address and has a size of 16-bits. |

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 Booster Pack use the same values?

|  |
| --- |
| 10k ohms  Yes the booster pack used the same values. |

Conclusion

In this experiment we have implement Inter-Integrated Circuit (12C) Communication using the MSP430 and the booster pack light sensor. First, we pulled the manufacturers ID and device ID from the booster pack and displayed them on the terminal every second. Then, using similar code, we used the booster pack to sense the light in the room by configuring the I2C to read the values from the sensor register. Then reading the values from the register we printed them to the terminal.