EEL4742C: Embedded Systems

Experiment Eight:

Simple I2C Implementation

Ryan Koons 11/02/2020

Introduction: This lab entirely focuses on I2C but will also require functions from previous labs. Additionally, this lab will dive into the introduction of the booster pack as well as the light sensor that is attached to it. I2C is similar to UART in a sense but also has its differences. The main ones being that it operates on a bus platform. More specifically, typically one master and all devices are attached to SDA and SCL lines. These lines are what is used for timing and transmitting data. Each device on the line has an address (typically 7 bit) that the master will then use to communicate. The last bit of that byte is typically used to denote read or write. The read and write processes (as far as transmitting data, and acknowledging transfers) will be explained in more detail in the next section. For now, it is much easier to think about data being transmitted one bit at a time, with 2 bytes being the total amount written or read. This makes sense the internal registers of the i2c device are typically 16 bits. Internal registers are the devices within a particular I2c device, that a master typically is able to write or read from. Also, for a write instruction, the first one always goes to the pointer to designate where to write to.

9.1 I2C Transmission: NOTE: A lot of similar functions and #defines from previous labs are used during this lab; since, these concepts have been explained thoroughly in previous labs, they will not be covered in detail during this lab. The main focus of this lab is on the implementation of I2C.

This lab will involve attaching the Boosterboard to the MSP430 launchpad; this will allow for I2C communication between the light sensor on the booster board and the MSP430.

During initialization, the code executes its typical instructions, defines some integers for counting/displaying incrementation (loop timing etc.) and for the manufacturer and device ID's, defines some characters for print formatting, and calls the UART and I2C init. Functions.

Within I2C Initialization, the code first enters reset state so that no communication is currently taking place and can be configured (reset mode is the default). Next the GPIO pins have multiple functions, therefore they need to be diverted for I2C compatibility. The corresponding bits in the configuration registers p4SEL 1 and p4SEL0 should be set and cleared to achieve I2C functionality. The next step of configuration involves selecting UC Mode 3 for I2c, Master Mode, and SMCLK for UCSSSEL. This defines the clock to be used for the communication, as well as the type of communication, and the master. Then the clock divider to be used for the 1.000 MHz clock is selected. The last step of initialization is to exit reset mode.

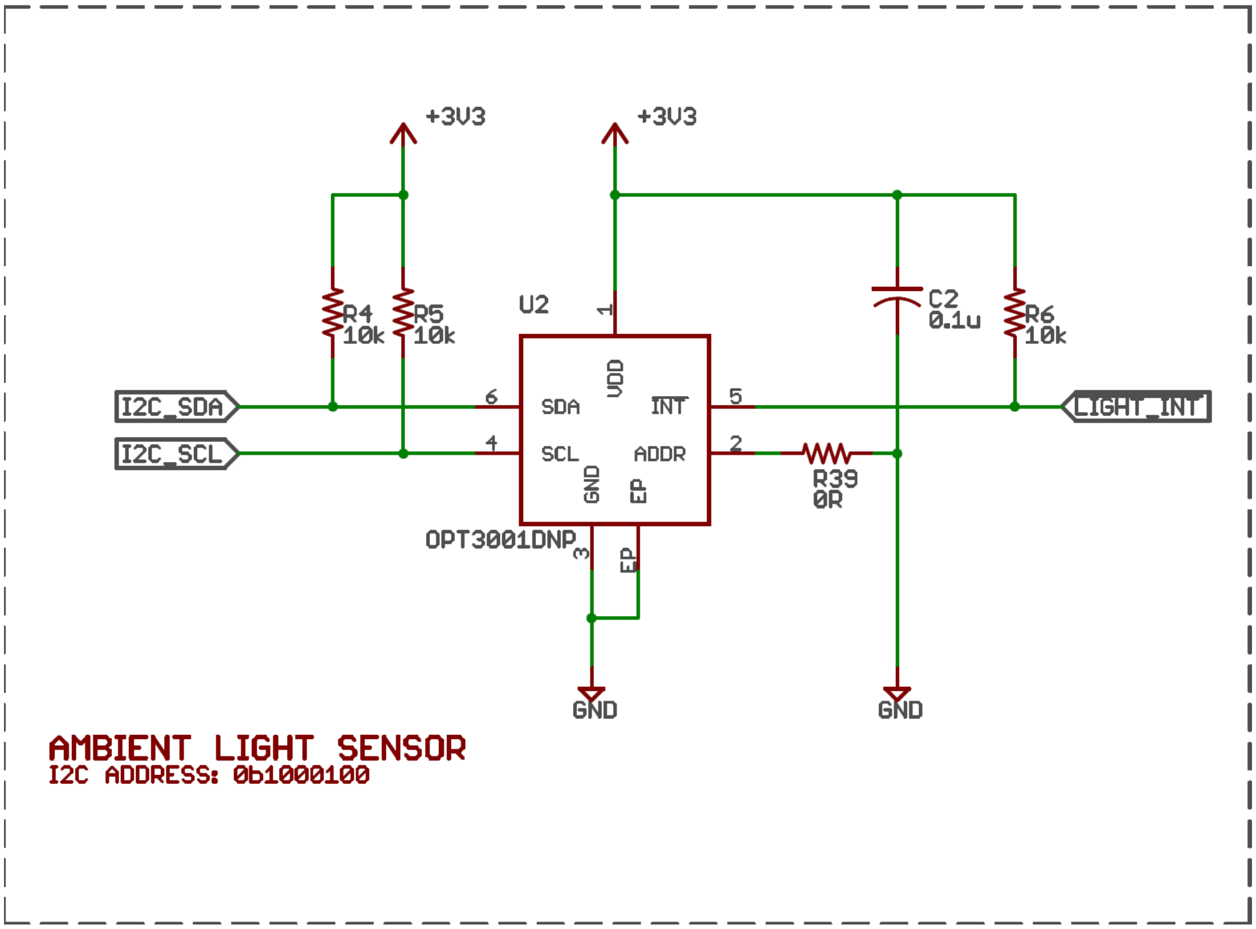
The program also uses a provided I2C read and I2C write function. For the read function the first parameter passed is the devices I2C address (typically 7-bits), the next is the internal register that it wants to write to, and the last unsigned int is where the result should be stored in the main memory. This function will read two bytes from the internal register; this intuitively makes sense since the register is 16-bit. For the write function, the first parameter is also the I2C device address, and the second is the internal register, however now the last is the data that it is to be written to that register. This function writes a 16-bit value to the internal register.

Both functions follow standard I2C protocol for communication. That is, for a I2C read, the master sends a start signal, and then then the i2c device address and R/W bit. The device always acknowledges the request. For sending back the data, the device sends back one byte at a time, waiting for an Ack between each byte. Whenever the master no longer needs data, it doesn't acknowledge it and then sends the stop signal. For the write function, everything operates in a similar fashion—first master sends start, then write a bit at a time with ack from device from in between. Here no nack is necessary, the master simply sends the stop signal. Also, in writing, the first byte always goes to the pointer register, which then defines which internal register that the master most likely will want to write repeated data to; this allows for consecutive writes without having to redefine the internal register every time.

Within the main, the rest of the code is a loop (about 1 second delay) that prints out '0x' and then the manufacturer ID, followed by a new line (and carriage return) and the deviceID. Both of these ID's are returned as integers from the sensor, and converted into hexadecimal from another function (to be explained later). I2c read is used to retrieve these values, by first passing the sensor's device address 0x44, than the internal register for the manufacturer ID which is 0x7E. The internal register for the device ID is 0x7F. The sensor's data sheet should be consulted for finding the internal registers address. These registers should be read in a loop and the converted hexadecimal result should be displayed on the com terminal with an incrementing counter.

The UART Hex functions take in a 16-bit integer, and converts into hex characters to display on the com window. The function essentially shifts to the right multiple times to achieve each set of 4 bits. These 4 bits are then ANDed to ensure no other more significant bits are included. Then result is either greater than or less than 10. If it is greater than, then the hexadecimal value is a letter, otherwise the hexadecimal is a number. The result is passed to the uart write char function.

Note: Below denotes the value of pull up resistors on the I2C bus for the light sensor. (10k)



9.1 CCS Code:

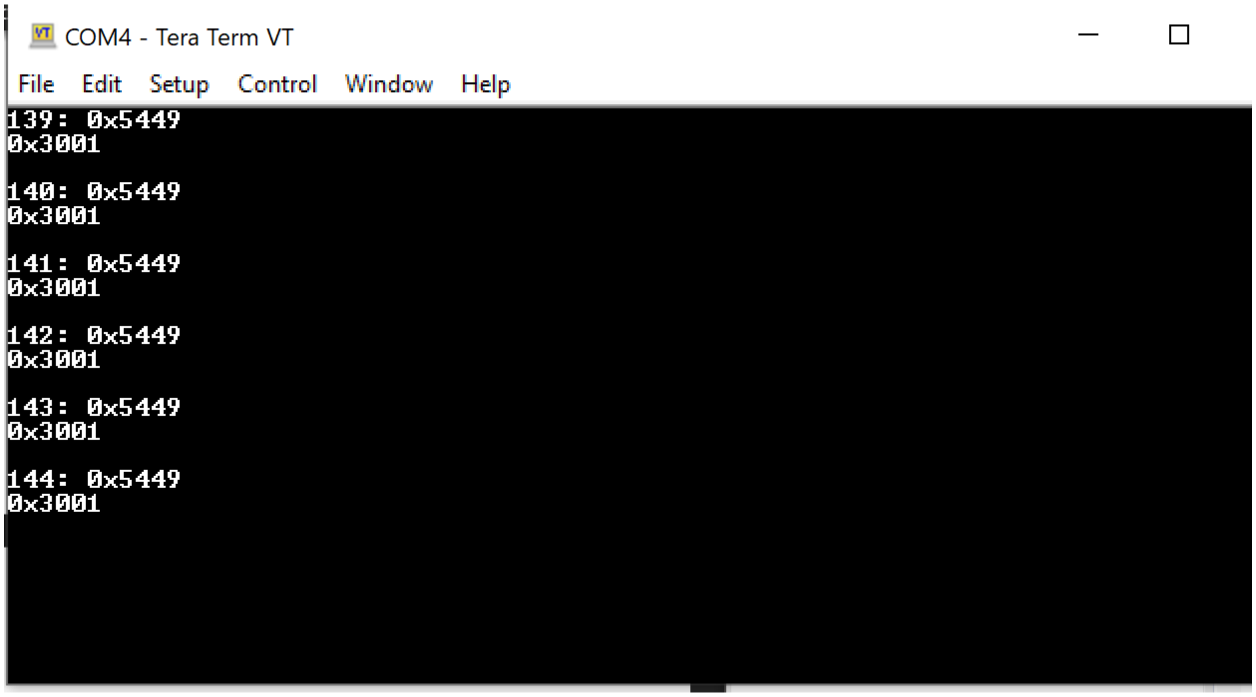
```
#include <msp430.h>
#define FLAGS UCA1IFG //Contains the Tx and Rx flags
#define RXFLAG UCRXIFG //Rx Flag
#define TXFLAG UCTXIFG //Tx Flag
#define TXBUFFER UCA1TXBUF// Tx Buffer
#define RXBUFFER UCA1RXBUF// Rx Buffer


void Initialize_I2C(void);
int i2c_read_word(unsigned char, unsigned char, unsigned int*);
int i2c_write_word(unsigned char, unsigned char, unsigned int);
void init_UART(void);
void uart_write_char(unsigned char ch);
void uart_write_uint16(unsigned int n);
void uart_write_uint16_hex(unsigned int n);
```

```
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5;       //enable GPIO Pins

    unsigned int manufacturerID=0;
    unsigned int deviceID=0;
    unsigned int count=0;
    unsigned int loop=0;
    unsigned int i =0;
    unsigned char nl='\n';
    unsigned char cr='\r';
    unsigned char colon=':';
    unsigned char space=' ';

     init_UART(); //Initialize
     Initialize_I2C();
for(;;){
    for(loop=0; loop<=65534; loop++){

        uart_write_uint16(count);
        uart_write_char(colon);
        uart_write_char(space);

    //read manufacturer ID and Device ID registers
    i2c_read_word(0x44,0x7E, &manufacturerID);
    i2c_read_word(0x44,0x7F, &deviceID);
    uart_write_uint16_hex(manufacturerID);
    uart_write_char(nl);
    uart_write_char(cr);
    uart_write_uint16_hex(deviceID);
    uart_write_char(nl);
    uart_write_char(cr);
    uart_write_char(nl);
    uart_write_char(cr);
    count++;

    for( i=0; i<65000; i++){} //delay loop
            for( i=0; i<65000; i++){} //delay loop
            for( i=0; i<30000; i++){} //delay loop
            for( i=0; i<10000; i++){} //delay loop
    for( i=0; i<65000; i++){} //delay loop
    //i2c_read_word(0x22, 0x50, &data) read two bytes from register 0x50 on I2C
device 0x22, store total in data (pass via uart)
    //unsigned int data= 0xABCD; i2c_write_word(0x22, 0x60, data); wrte 0xABCD to
register 0x50on I2C device 0x22
    }
}
    return 0;
}

void uart_write_uint16_hex(unsigned int n)
{
    int digit;
    unsigned char ch;
    uart_write_char('0');
    uart_write_char('x');

    digit = (n>>12) & 0x000F;

    if(digit<10) ch='0'+digit;
    else ch = 'A' +digit-10;
    uart_write_char(ch);

    digit =(n>>8) & 0x000F;
    if(digit<10) ch='0'+digit;
        else ch = 'A' +digit-10;
        uart_write_char(ch);

    digit=(n>>4) & 0x000F;
    if(digit<10) ch='0'+digit;
        else ch = 'A' +digit-10;
        uart_write_char(ch);

    digit=n & 0x000F;
    if(digit<10) ch='0'+digit;
            else ch = 'A' +digit-10;
            uart_write_char(ch);


      return;

}


// 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 92
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 ***************************
```

```c
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) {}
//********** 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;
}

//function for properly Tx 16-bit int (0-65535) to PC
void uart_write_uint16(unsigned int n)
{
 int digit; //digit for parsing through

 //check ten=thousandths place
 if(n>=10000)
 {
     //modulo 10 ensures no digits past 16-bit int allowed (out of constraint)
     digit = (n/10000) % 10; //takes leftmost digit first
     uart_write_char('0' + digit); //pass correct ascii value for digit to computer
 }

//check thousandths place
 if(n>=1000)
  {
      digit = (n/1000) % 10; //takes next leftmost digit
      uart_write_char('0' + digit); //print next digit on screen
  }
//modulo is also used to hack off other digits of int that is being displayed
 //(i.e. ten thousandths place that already printed to PC)
 //Hundredths place
 if(n>=100)
  {
      digit = (n/100) % 10;//take next digit
      uart_write_char('0' + digit); //print
  }

//tenths place
 if(n>=10)
  {  //process repeats similarly
      digit = (n/10) % 10;
      uart_write_char('0' + digit);
  }

//ones place always exectures
 digit = n%10; //get rid of all other digits
 uart_write_char('0'+ digit); //print digit on  PC

 return;

}


//UART config
//9600 Baud, 8-bit data, LSB first, no parity, 1 stop bit, no flow CTL
//Initial clock: SMCLK @ 1.000 MHz with Oversampling (16x)
void init_UART(void)
{
    //Divert GPIO pins for UART
        P3SEL1 &= ~(BIT4|BIT5);  //Using 3.4--UCA1TXD, 3.5--UCA1RXD
        P3SEL0 |= (BIT4|BIT5);

        //Use SMCLK, other settings default
        UCA1CTLW0 |= UCSSEL_2;

        //Configure clock dividers and modulators
        //UCBR=6, UCBRF=8, UCBRS=0x20, UCOS16=1 (over-sampling)
        UCA1BRW =6;
        //bit masking for UCBRS and UCBRF done using 0-7 bit number
        UCA1MCTLW = UCBRS5|UCBRF3|UCOS16;


        //exit reset state, to begin Tx/Rx
        UCA1CTLW0 &= ~UCSWRST;
}

//TXFLAG: 1 When ready to transmit, 0 during transmission
void uart_write_char(unsigned char ch)
{
    //wait for any ongoing transmission  to complete
    while((FLAGS & TXFLAG) ==0 ){}

    //Write byte to the transmit buffer
    TXBUFFER= ch;
```

```
}



// 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)
//^ IS THIS WRONG!!***************************

UCB1BRW = 8;
// Exit the reset mode
UCB1CTLW0 &= ~UCSWRST;
}
```

9.1 Output (Tera Term):



```
139: 0x5449
0x3001

140: 0x5449
0x3001

141: 0x5449
0x3001

142: 0x5449
0x3001

143: 0x5449
0x3001

144: 0x5449
0x3001
```

9.2 Reading Measurements from the Light Sensor: The purpose of this section is to properly be able to configure the light sensor and display its readings on the PC. After initialization, the sensor needs to be properly configured. Configuration is necessary in order to set the resolution and range of the light sensor. It is a 16 bit result, that has a 4-bit exponent and a 12 bit result. Using the sensor date the desired configuration can be achieved: exponent of 7 (LSB worth 1.28 resolution). This shows that the max value that can be achieved is 5,241 lux. There is a tradeoff between resolution and range, increasing one will cause a decrease in the other. The lab manual provides the bit fields and the following needs to be used to achieve the desired 16bit configuration register layout (all 16-bits converted into hex for the i2c_write function. The following bit field was used and then converted into hex: 0111, 0, 11, 0,0,0,0,0, 0,1,00. (see Table 11 in sensor's data sheet for more info.)

```
"//config ALS--Rn 0x7, CT=0x0, M=0x3, ME=0x1
     i2c_write_word(0x44,0x01, 0x7604);"
```

Thus, the I2c write function is used to configure the light sensor. The light sensor also has a mask exponent result, which will be used since the exponent remains fixed during the program (it can be auto selected based on range and resolution). The address of the configuration register on the sensor was 0x01 and this can be found in the chip's data sheet. The program runs a delay loop (about one second) and prints the result from the light sensor each time. This result is multiplied by 1.28 & type-casted as a float to display appropriately on the PC display. Testing shows that completely covering the light sensor resulted in a value of 0, and shining a flashlight directly into it reported the max value of 5241 lux. So, testing behaved understandably, and data matches the expectation.

9.2 CCS Code:

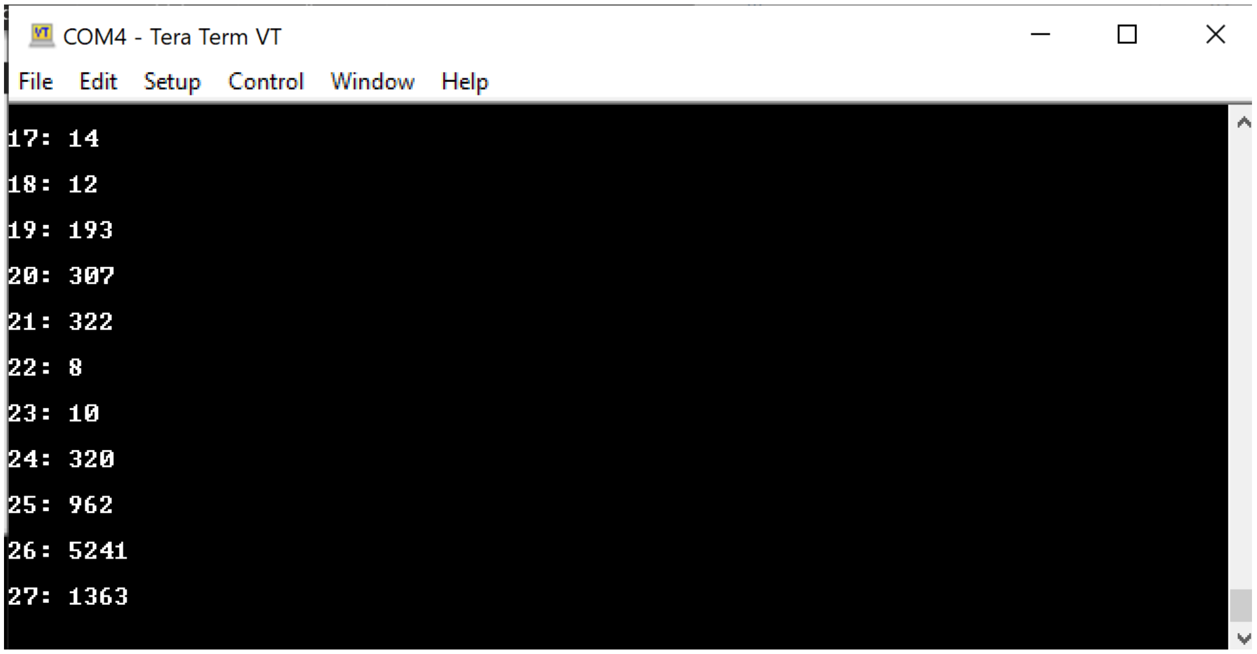
```
#include <msp430.h>
#include <stdio.h>
#define FLAGS UCA1IFG //Contains the Tx and Rx flags
#define RXFLAG UCRXIFG //Rx Flag
#define TXFLAG UCTXIFG //Tx Flag
#define TXBUFFER UCA1TXBUF// Tx Buffer
#define RXBUFFER UCA1RXBUF// Rx Buffer


void Initialize_I2C(void);
int i2c_read_word(unsigned char, unsigned char, unsigned int*);
int i2c_write_word(unsigned char, unsigned char, unsigned int);
void init_UART(void);
void uart_write_char(unsigned char ch);
void uart_write_uint16(unsigned int n);
```

```
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5;       //enable GPIO Pins

     unsigned int data=0;
    unsigned int count=0;
    unsigned int loop=0;
    unsigned int i =0;
    unsigned char nl='\n';
    unsigned char cr='\r';
    unsigned char colon=':';
    unsigned char space=' ';

     init_UART(); //Initialize
     Initialize_I2C();

     //config ALS--Rn 0x7, CT=0x0, M=0x3, ME=0x1
     i2c_write_word(0x44,0x01, 0x7604);

for(;;){
    for(loop=0; loop<=65534; loop++){

        uart_write_uint16(count);
        uart_write_char(colon);
        uart_write_char(space);
//print current measurement
        i2c_read_word( 0x44, 0x00, &data);
        data = (float)data*1.28;
        uart_write_uint16(data);

    uart_write_char(nl);
    uart_write_char(cr);
    uart_write_char(nl);
    uart_write_char(cr);
    count++;

    for( i=0; i<65000; i++){} //delay loop
              for( i=0; i<65000; i++){} //delay loop
              for( i=0; i<30000; i++){} //delay loop
              for( i=0; i<10000; i++){} //delay loop
      for( i=0; i<65000; i++){} //delay loop

    //i2c_read_word(0x22, 0x50, &data) read two bytes from register 0x50 on I2C
device 0x22, store total in data (pass via uart)
    //unsigned int data= 0xABCD; i2c_write_word(0x22, 0x60, data); wrte 0xABCD to
register 0x50on I2C device 0x22
    }
}
    return 0;
}


// Read a word (2 bytes) from I2C (address, register)
```

```c
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 92
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) {}
//********** 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;
}

//function for properly Tx 16-bit int (0-65535) to PC
void uart_write_uint16(unsigned int n)
{
 int digit; //digit for parsing through

 //check ten=thousandths place
 if(n>=10000)
 {
     //modulo 10 ensures no digits past 16-bit int allowed (out of constraint)
     digit = (n/10000) % 10; //takes leftmost digit first
     uart_write_char('0' + digit); //pass correct ascii value for digit to computer
 }

//check thousandths place
 if(n>=1000)
  {
      digit = (n/1000) % 10; //takes next leftmost digit
      uart_write_char('0' + digit); //print next digit on screen
  }
//modulo is also used to hack off other digits of int that is being displayed
 //(i.e. ten thousandths place that already printed to PC)

 //Hundredths place
 if(n>=100)
  {
      digit = (n/100) % 10;//take next digit
      uart_write_char('0' + digit); //print
  }

//tenths place
 if(n>=10)
  {  //process repeats similarly
      digit = (n/10) % 10;
      uart_write_char('0' + digit);
  }

//ones place always exectures
 digit = n%10; //get rid of all other digits
 uart_write_char('0'+ digit); //print digit on  PC

 return;

}


//UART config
//9600 Baud, 8-bit data, LSB first, no parity, 1 stop bit, no flow CTL
//Initial clock: SMCLK @ 1.000 MHz with Oversampling (16x)
void init_UART(void)
{
    //Divert GPIO pins for UART
        P3SEL1 &= ~(BIT4|BIT5);  //Using 3.4--UCA1TXD, 3.5--UCA1RXD
        P3SEL0 |= (BIT4|BIT5);

        //Use SMCLK, other settings default
        UCA1CTLW0 |= UCSSEL_2;

        //Configure clock dividers and modulators
        //UCBR=6, UCBRF=8, UCBRS=0x20, UCOS16=1 (over-sampling)
        UCA1BRW =6;
        //bit masking for UCBRS and UCBRF done using 0-7 bit number
        UCA1MCTLW = UCBRS5|UCBRF3|UCOS16;


        //exit reset state, to begin Tx/Rx
        UCA1CTLW0 &= ~UCSWRST;
}

//TXFLAG: 1 When ready to transmit, 0 during transmission
void uart_write_char(unsigned char ch)
{
    //wait for any ongoing transmission  to complete
    while((FLAGS & TXFLAG) ==0 ){}

    //Write byte to the transmit buffer
    TXBUFFER= ch;
}



// 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)
//^ IS THIS WRONG!!***************************

UCB1BRW = 8;
// Exit the reset mode
UCB1CTLW0 &= ~UCSWRST;
}
```

9.2 Output (Tera Term):



```
17: 14
18: 12
19: 193
20: 307
21: 322
22: 8
23: 10
24: 320
25: 962
26: 5241
27: 1363
```

Student Q&A:

1) The sensor's address pin sets the LSBs on the I2C address. According to the sensor data sheet, since the address pin can be assign to GND, VDD, SDA, or SLC. Then it has 4 possible i2c device addresses. They are 1000100 (addr-GND), 1000101 (addr-VDD), 1000110 (SDA), and 1000111(SCL). These addresses can be achieved by tieing the address pin to one of the following pins.
2) The sensor's data sheet does not specify a pull up resistor value; however, the booster pack uses pull-ups of 10k.

Conclusion: After this lab it is now clear how to implement i2c (through use of read and write function), as well as how to use the light sensor on the MSP430 Booster pack. I2C is a very efficient and reliable way to communicate between devices. This lab was primarily performed for writing to configuration registers to set up the light sensor, as well as reading various measurements and print the measurements on the pc. I2C certainly is more difficult to implement than UART. The functions require iterating through for each byte of transmittable data, and acknowledging and not acknowledging. However, once the functions are written, it is very simple to communicate using I2C. Although, it is always important to remember that the first byte always writes to the pointer. This Is done so that a repeated write can be used to the same internal register for the i2c device. It is important to note that multiple masters can be used (as long as they don't interrupt each other); however, this is above the scope of this lab. For now the most important thing is the general understanding of I2C, configuring sensors, and being able to print out the result from the sensor.