Lab 8 Report

EEL4742C - 00446

Yousef Awad

October 2025

Contents

Introduction

8.1 Using the ADC SAR-Type

```
#include <msp430fr6989.h>
#include <stdint.h>
#include <string.h>

// UART Channels are P3.4 and P3.5 for transmit and recieve respectively
#define transmit BIT4
#define recieve BIT5

// WE LOVE DEFINES
#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

// Global variables for states of runway1 and 2
volatile int red_state = 0; // runway 1 state
volatile int green_state = 0; // runway 2 state
volatile int blink_state = 0;

// Functions provided by the lab
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;
}

int i2c_read_word(unsigned char i2c_address, unsigned char i2c_reg, unsigned int * data)
{
  unsigned char byte1=0, byte2=0; // Intialize to ensure successful reading
  UCB1I2CSA = i2c_address; // Set address
  UCB1IFG &= ~UCTXIFG0;
  // Transmit a byte (the internal register address)
```

```
  UCB1CTLW0 |= UCTR;
  UCB1CTLW0 |= UCTXSTT;
  while((UCB1IFG & UCTXIFG0)==0) {} // Wait for flag to raise
  UCB1TXBUF = i2c_reg; // Write in the TX buffer
  while((UCB1IFG & UCTXIFG0)==0) {} // Buffer copied to shift
    register; Tx in progress; set Stop bit
  // Repeated Start
  UCB1CTLW0 &= ~UCTR;
  UCB1CTLW0 |= UCTXSTT;
  // Read the first byte
  while((UCB1IFG & UCRXIFG0)==0) {} // Wait for flag to raise
  byte1 = UCB1RXBUF;
  // Assert the Stop signal bit before receiving the last byte
  UCB1CTLW0 |= UCTXSTP;
  // Read the second byte
  while((UCB1IFG & UCRXIFG0)==0) {} // Wait for flag to raise
  byte2 = UCB1RXBUF;
  while((UCB1CTLW0 & UCTXSTP)!=0) {}
  while((UCB1STATW & UCBBUSY)!=0) {}
  *data = (byte1 << 8) | (byte2 & (unsigned int)0x00FF);
  return 0;
}

int i2c_write_word(unsigned char i2c_address, unsigned char i2c_reg
    , unsigned int data)
{
  unsigned char byte1, byte2;

  UCB1I2CSA = i2c_address;              // Set I2C address

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

  UCB1IFG &= ~UCTXIFG0;

  // Write 3 bytes
  UCB1CTLW0 |= (UCTR | UCTXSTT);

  while( (UCB1IFG & UCTXIFG0) == 0) {}
  UCB1TXBUF = i2c_reg;

  while( (UCB1IFG & UCTXIFG0) == 0) {}
  UCB1TXBUF = byte1;

  while( (UCB1IFG & UCTXIFG0) == 0) {}
  UCB1TXBUF = byte2;

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

  UCB1CTLW0 |= UCTXSTP;
  while( (UCB1CTLW0 & UCTXSTP) != 0 ) {}
  while((UCB1STATW & UCBUSY)!=0) {}

  return 0;
}

// Reverses a given string
```

```
104 void strrev(char *str)
105 {
106   unsigned int i = 0;
107   unsigned int j = strlen(str) - 1;
108   char temp;
109   while (i < j)
110   {
111     temp = str[i];
112     str[i] = str[j];
113     str[j] = temp;
114     i++;
115     j--;
116   }
117 }
118
119 // Converts an unsigned 16-bit integer to a null-terminated string
      (base 10).
120 void custom_itoa(uint16_t number, char *buffer)
121 {
122   unsigned int i = 0;
123
124   // Handle the special case of 0
125   if (number == 0)
126   {
127     buffer[i++] = '0';
128     buffer[i] = '\0';
129     return;
130   }
131
132   // Process individual digits
133   while (number > 0)
134   {
135     int remainder = number % 10;
136     buffer[i++] = remainder + '0'; // Convert digit to its ASCII
        character
137     number = number / 10;
138   }
139
140   buffer[i] = '\0'; // Null-terminate the string
141
142   // The digits are in reverse order, so we need to reverse the
      string
143   strrev(buffer);
144 }
145
146 // Configures ACLK to 32 KHz crystal
147 void config_ACLK_to_32KHz_crystal()
148 {
149   // By default, ACLK runs on LFMODCLK at 5MHz/128 = 39 KHz
150
151   // Reroute pins to LFXIN/LFXOUT functionality
152   PJSEL1 &= ~BIT4;
153   PJSEL0 |= BIT4;
154
155   // Wait until the oscillator fault flags remain cleared
156   CSCTL0 = CSKEY; // Unlock CS registers
157   do
```

```
158   {
159     CSCTL5 &= ~LFXTOFFG; // Local fault flag
160     SFRIFG1 &= ~OFIFG; // Global fault flag
161   }
162   while((CSCTL5 & LFXTOFFG) != 0);
163 
164   CSCTL0_H = 0; // Lock CS registers
165   return;
166 }
167 
168 void initialize_uart(void)
169 {
170   // Configuring the pins to use backchannel uart
171   P3SEL1 &= ~(transmit | recieve);
172   P3SEL0 |= (transmit | recieve);
173 
174   // Setting the clock to SMCLK
175   UCA1CTLW0 |= UCSSEL_2;
176 
177   // Setting the dividers and enabling oversampling
178   UCA1BRW = 6;
179   // setting the modulators and such
180   UCA1MCTLW = UCBRF3 | UCBRS5 | UCOS16;
181 
182   // Exiting the reset state
183   UCA1CTLW0 &= ~UCSWRST;
184 }
185 
186 void uart_write_char(volatile unsigned char ch)
187 {
188   while (!(FLAGS & TXFLAG))
189   {
190     // Wait for transmission that is ongoing to complete
191   }
192 
193   TXBUFFER = ch;
194   return;
195 }
196 
197 unsigned char uart_read_char(void)
198 {
199   if (!(FLAGS & RXFLAG))
200   {
201     return 0; // no byte was recieved
202   }
203 
204   // Return the buffer
205   volatile unsigned char return_char = RXBUFFER;
206   return return_char;
207 }
208 
209 void uart_write_string(char *string)
210 {
211   unsigned int i; // counter
212   for (i = 0; i < strlen(string); i++)
213   {
214     uart_write_char(string[i]);
```

```
  }
  return;
}

void uart_write_uint16 (uint16_t number)
{
  // Converting the number via snprintf
  char buffer[6]; // 5 characters is the max amount of characters
    for 65,536
  custom_itoa(number, buffer);
  uart_write_string(buffer);
  return;
}

void uint16_to_4hex(unsigned int given_uint, char output[5])
{
  static const char hex[] = "0123456789ABCDEF"; // All possible
    hexes
  output[0] = hex[(given_uint >> 12) & 0xF];
  output[1] = hex[(given_uint >> 8) & 0xF];
  output[2] = hex[(given_uint >> 4) & 0xF];
  output[3] = hex[(given_uint >> 12) & 0xF];
  output[4] = hex[(given_uint) & 0xF];
  return;
}

int main(void)
{
  // Enabling the leds and other stuff
  WDTCTL   = WDTPW | WDTHOLD; // Stop WDT
  PM5CTL0 &= ~LOCKLPM5;       // Enable GPIO pins

  // doing what the function says
  initialize_uart();

  // yup, whatever it says
  initialize_i2c();

  // Actual logic for selection
  for (;;)
  {
    unsigned int manufacturerID = 0;
    unsigned int deviceID       = 0;

    i2c_read_word(0x44, 0x7E, &manufacturerID);
    i2c_read_word(0x44, 0x7F, &deviceID);

    // Converting the ids to ascii
    char man_ascii[3];
    man_ascii[0] = (char)((manufacturerID >> 8) & 0xFF); // High
    byte
    man_ascii[1] = (char)(manufacturerID & 0xFF); // Low byte
    man_ascii[2] = '\0';

    // transmitting the data
    uart_write_string("Manufacturer ID: ");
    uart_write_string(man_ascii);
```

```
    uart_write_char('\n');

    // Converting the device id.
    char dev_ascii[5];
    uint16_to_4hex(deviceID, dev_ascii);
    uart_write_string("Device ID: ");
    uart_write_string(dev_ascii);
    uart_write_char('\n');

    __delay_cycles(1000000); // delay of 1 million cycles
  }
}
```

8.2 Reading the X- and Y- Coordinates of the Joystick

```c
#include <msp430fr6989.h>
#include <stdint.h>
#include <string.h>

// UART Channels are P3.4 and P3.5 for transmit and recieve respectively
#define transmit BIT4
#define recieve BIT5

// WE LOVE DEFINES
#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

// Global variables for states of runway1 and 2
volatile int red_state = 0; // runway 1 state
volatile int green_state = 0; // runway 2 state
volatile int blink_state = 0;

// Functions provided by the lab
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;
}

int i2c_read_word(unsigned char i2c_address, unsigned char i2c_reg, unsigned int * data)
{
  unsigned char byte1=0, byte2=0; // Intialize to ensure successful reading
  UCB1I2CSA = i2c_address; // Set address
  UCB1IFG &= ~UCTXIFG0;
  // Transmit a byte (the internal register address)
  UCB1CTLW0 |= UCTR;
```

```
  UCB1CTLW0 |= UCTXSTT;
  while((UCB1IFG & UCTXIFG0)==0) {} // Wait for flag to raise
  UCB1TXBUF = i2c_reg; // Write in the TX buffer
  while((UCB1IFG & UCTXIFG0)==0) {} // Buffer copied to shift
    register; Tx in progress; set Stop bit
  // Repeated Start
  UCB1CTLW0 &= ~UCTR;
  UCB1CTLW0 |= UCTXSTT;
  // Read the first byte
  while((UCB1IFG & UCRXIFG0)==0) {} // Wait for flag to raise
  byte1 = UCB1RXBUF;
  // Assert the Stop signal bit before receiving the last byte
  UCB1CTLW0 |= UCTXSTP;
  // Read the second byte
  while((UCB1IFG & UCRXIFG0)==0) {} // Wait for flag to raise
  byte2 = UCB1RXBUF;
  while((UCB1CTLW0 & UCTXSTP)!=0) {}
  while((UCB1STATW & UCBBUSY)!=0) {}
  *data = (byte1 << 8) | (byte2 & (unsigned int)0x00FF);
  return 0;
}

int i2c_write_word(unsigned char i2c_address, unsigned char i2c_reg
    , unsigned int data)
{
  unsigned char byte1, byte2;

  UCB1I2CSA = i2c_address;              // Set I2C address

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

  UCB1IFG &= ~UCTXIFG0;

  // Write 3 bytes
  UCB1CTLW0 |= (UCTR | UCTXSTT);

  while( (UCB1IFG & UCTXIFG0) == 0) {}
  UCB1TXBUF = i2c_reg;

  while( (UCB1IFG & UCTXIFG0) == 0) {}
  UCB1TXBUF = byte1;

  while( (UCB1IFG & UCTXIFG0) == 0) {}
  UCB1TXBUF = byte2;

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

  UCB1CTLW0 |= UCTXSTP;
  while( (UCB1CTLW0 & UCTXSTP) != 0 ) {}
  while((UCB1STATW & UCBUSY)!=0) {}

  return 0;
}

// Reverses a given string
void strrev(char *str)
```

```
{
  unsigned int i = 0;
  unsigned int j = strlen(str) - 1;
  char temp;
  while (i < j)
  {
    temp = str[i];
    str[i] = str[j];
    str[j] = temp;
    i++;
    j--;
  }
}

// Converts an unsigned 16-bit integer to a null-terminated string (base 10).
void custom_itoa(uint16_t number, char *buffer)
{
  unsigned int i = 0;

  // Handle the special case of 0
  if (number == 0)
  {
    buffer[i++] = '0';
    buffer[i] = '\0';
    return;
  }

  // Process individual digits
  while (number > 0)
  {
    int remainder = number % 10;
    buffer[i++] = remainder + '0'; // Convert digit to its ASCII character
    number = number / 10;
  }

  buffer[i] = '\0'; // Null-terminate the string

  // The digits are in reverse order, so we need to reverse the string
  strrev(buffer);
}

// Configures ACLK to 32 KHz crystal
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;
}

void initialize_uart(void)
{
  // Configuring the pins to use backchannel uart
  P3SEL1 &= ~(transmit | recieve);
  P3SEL0 |= (transmit | recieve);

  // Setting the clock to SMCLK
  UCA1CTLW0 |= UCSSEL_2;

  // Setting the dividers and enabling oversampling
  UCA1BRW = 6;
  // setting the modulators and such
  UCA1MCTLW = UCBRF3 | UCBRS5 | UCOS16;

  // Exiting the reset state
  UCA1CTLW0 &= ~UCSWRST;
}

void uart_write_char(volatile unsigned char ch)
{
  while (!(FLAGS & TXFLAG))
  {
    // Wait for transmission that is ongoing to complete
  }

  TXBUFFER = ch;
  return;
}

unsigned char uart_read_char(void)
{
  if (!(FLAGS & RXFLAG))
  {
    return 0; // no byte was recieved
  }

  // Return the buffer
  volatile unsigned char return_char = RXBUFFER;
  return return_char;
}

void uart_write_string(char *string)
{
  unsigned int i; // counter
  for (i = 0; i < strlen(string); i++)
  {
    uart_write_char(string[i]);
  }
```

```
  return;
}

void uart_write_uint16 (uint16_t number)
{
  // Converting the number via snprintf
  char buffer[6]; // 5 characters is the max amount of characters
    for 65,536
  custom_itoa(number, buffer);
  uart_write_string(buffer);
  return;
}

void uint16_to_4hex(unsigned int given_uint, char output[5])
{
  static const char hex[] = "0123456789ABCDEF"; // All possible
    hexes
  output[0] = hex[(given_uint >> 12) & 0xF];
  output[1] = hex[(given_uint >> 8) & 0xF];
  output[2] = hex[(given_uint >> 4) & 0xF];
  output[3] = hex[(given_uint >> 12) & 0xF];
  output[4] = hex[(given_uint) & 0xF];
  return;
}

int main(void)
{
  // Enabling the leds and other stuff
  WDTCTL   = WDTPW | WDTHOLD; // Stop WDT
  PM5CTL0 &= ~LOCKLPM5;       // Enable GPIO pins

  // doing what the function says
  initialize_uart();

  // yup, whatever it says
  initialize_i2c();

  // Actual logic for selection
  for (;;)
  {
    unsigned int light = 0;

    // Writing the configuration to the light sensor
    /*
    *  The configuration register is:
    *  RN [15:12] (R/W)- b1100 is reset
    *  CT [11] (R/W)- b1 is reset
    *  M[ 10:9] (R/W) - b00 is reset
    *  OVF [8] (R) - b0 is reset
    *  CRF [7] (R) - b0 is reset
    *  FH [6] (R) - b0 is reset
    *  FL [5] (R) - b0 is reset
    *  L [4] (R/W) - b1 is reset
    *  POL [3] (R/W) - b0 is reset
    *  ME [2] (R/W) - b0 is reset
    *  FC [1:0] (R/W) - b00 is reset
    *
```

```
271     *  R/W is Read/Write
272     *  R is Read
273     */
274     i2c_write_word(0x44, 0x01, 0x7614);
275     // Reading the value of the light sensor
276     i2c_read_word(0x44, 0x00, &light);
277
278     // Converting the gathered reading to the proper value
279     int correctedLight = light * 1.28;
280
281     // writing to the serial console what the light sensor found
282     uart_write_string("Lux: ");
283     uart_write_uint16(correctedLight);
284     uart_write_char('\n');
285
286     __delay_cycles(1000000); // delay of 1 million cycles
287   }
288 }
```

8.3 Application: Platform Balancing Control

```
#include <msp430.h>
#include <msp430fr6989.h>
#include <string.h>
#include <stdint.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

#define BUT1 BIT1
#define BUT2 BIT2

// The array has the shapes of the digits (0 to 9)
// Complete this array...
const unsigned char LCD_Shapes[10] = {0xFC,0x60,0xDB,0xF3,0x67,0xB7
    ,0xBF,0xE0, 0xFF, 0xF7} ;

volatile int number = 0;

volatile unsigned int LuxValue = 0;

volatile int time = 0;
volatile int current_time = 50000;


char HH_c[] = "00";
char MM_c[] = "00";
char SS_c[] = "00";

volatile int prevBase;

void uart_write_char(volatile unsigned char ch)
{
  while (!(FLAGS & TXFLAG))
  {
    // Wait for transmission that is ongoing to complete
  }

  TXBUFFER = ch;
  return;
}

void uart_write_string(char *string)
{
  unsigned int i; // counter
  for (i = 0; i < strlen(string); i++)
  {
    uart_write_char(string[i]);
  }
```

```
  return;
}

// Reverses a given string
void strrev(char *str)
{
  unsigned int i = 0;
  unsigned int j = strlen(str) - 1;
  char temp;
  while (i < j)
  {
    temp = str[i];
    str[i] = str[j];
    str[j] = temp;
    i++;
    j--;
  }
}

// Converts an unsigned 16-bit integer to a null-terminated string (base 10).
void custom_itoa(uint16_t number, char *buffer)
{
  unsigned int i = 0;

  // Handle the special case of 0
  if (number == 0)
  {
    buffer[i++] = '0';
    buffer[i] = '\0';
    return;
  }

  // Process individual digits
  while (number > 0)
  {
    int remainder = number % 10;
    buffer[i++] = remainder + '0'; // Convert digit to its ASCII character
    number = number / 10;
  }

  buffer[i] = '\0'; // Null-terminate the string

  // The digits are in reverse order, so we need to reverse the string
  strrev(buffer);
}

void uart_write_uint16 (uint16_t number)
{
  // Converting the number via snprintf
  char buffer[6]; // 5 characters is the max amount of characters for 65,536
  custom_itoa(number, buffer);
  uart_write_string(buffer);
  return;
```

```
}

unsigned char uart_read_char(void)
{
  if (!(FLAGS & RXFLAG))
  {
    return 0; // no byte was recieved
  }

  // Return the buffer
  volatile unsigned char return_char = RXBUFFER;
  return return_char;
}

void initialize_uart(void)
{
  // Configuring the pins to use backchannel uart
  P3SEL1 &= ~(transmit | recieve);
  P3SEL0 |= (transmit | recieve);

  // Setting the clock to SMCLK
  UCA1CTLW0 |= UCSSEL_2;

  // Setting the dividers and enabling oversampling
  UCA1BRW = 6;
  // setting the modulators and such
  UCA1MCTLW = UCBRF3 | UCBRS5 | UCOS16;

  // Exiting the reset state
  UCA1CTLW0 &= ~UCSWRST;
}

// Configures ACLK to 32 KHz crystal
void config_ACLK_to_32KHz_crystal(void)
{
  // 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;
}

// Functions provided by the lab
void initialize_i2c(void)
{
```

```
165   // Configure the MCU in Master mode
166   // Configure pins to I2C functionality
167   // (UCB1SDA same as P4.0) (UCB1SCL same as P4.1)
168   // (P4SEL1=11, P4SEL0=00) (P4DIR=xx)
169   P4SEL1 |= (BIT1|BIT0);
170   P4SEL0 &= ~(BIT1|BIT0);
171   // Enter reset state and set all fields in this register to zero
172   UCB1CTLW0 = UCSWRST;
173   // Fields that should be nonzero are changed below
174   // (Master Mode: UCMST) (I2C mode: UCMODE_3) (Synchronous mode:
        UCSYNC)
175   // (UCSSEL 1:ACLK, 2,3:SMCLK)
176   UCB1CTLW0 |= UCMST | UCMODE_3 | UCSYNC | UCSSEL_3;
177   // Clock frequency: SMCLK/8 = 1 MHz/8 = 125 KHz
178   UCB1BRW = 8;
179   // Chip Data Sheet p. 53 (Should be 400 KHz max)
180   // Exit the reset mode at the end of the configuration
181   UCB1CTLW0 &= ~UCSWRST;
182 }
183
184 int i2c_read_word(unsigned char i2c_address, unsigned char i2c_reg,
        unsigned int *data)
185 {
186   unsigned char byte1=0, byte2=0; // Intialize to ensure successful
        reading
187   UCB1I2CSA = i2c_address; // Set address
188   UCB1IFG &= ~UCTXIFG0;
189   // Transmit a byte (the internal register address)
190   UCB1CTLW0 |= UCTR;
191   UCB1CTLW0 |= UCTXSTT;
192   while(!(UCB1IFG & UCTXIFG0))
193   {
194     // Wait for flag to raise
195   }
196   UCB1TXBUF = i2c_reg; // Write in the TX buffer
197   while(!(UCB1IFG & UCTXIFG0))
198   {
199     // Buffer copied to shift register; Tx in progress; set Stop
        bit
200   }
201   // Repeated Start
202   UCB1CTLW0 &= ~UCTR;
203   UCB1CTLW0 |= UCTXSTT;
204   // Read the first byte
205   while(!(UCB1IFG & UCRXIFG0))
206   {
207     // Wait for flag to raise
208   }
209   byte1 = UCB1RXBUF;
210   // Assert the Stop signal bit before receiving the last byte
211   UCB1CTLW0 |= UCTXSTP;
212   // Read the second byte
213   while(!(UCB1IFG & UCRXIFG0))
214   {
215     // Wait for flag to raise
216   }
217   byte2 = UCB1RXBUF;
```

```
218   while(UCB1CTLW0 & UCTXSTP)
219   {
220   }
221   while(UCB1STATW & UCBBUSY)
222   {
223   }
224   *data = (byte1 << 8) | (byte2 & (unsigned int)0x00FF);
225   return 0;
226 }
227
228 int i2c_write_word(unsigned char i2c_address, unsigned char i2c_reg
      , unsigned int data)
229 {
230   unsigned char byte1, byte2;
231
232   UCB1I2CSA = i2c_address; // Set I2C address
233
234   byte1 = (data >> 8) & 0xFF; // MSByte
235   byte2 = data & 0xFF;        // LSByte
236
237   UCB1IFG &= ~UCTXIFG0;
238
239   // Write 3 bytes
240   UCB1CTLW0 |= (UCTR | UCTXSTT);
241
242   while(!(UCB1IFG & UCTXIFG0))
243   {
244     // Wait
245   }
246   UCB1TXBUF = i2c_reg;
247
248   while(!(UCB1IFG & UCTXIFG0))
249   {
250     // Wait
251   }
252   UCB1TXBUF = byte1;
253
254   while(!(UCB1IFG & UCTXIFG0))
255   {
256     // Wait
257   }
258   UCB1TXBUF = byte2;
259
260   while(!(UCB1IFG & UCTXIFG0))
261   {
262     // Wait
263   }
264
265   UCB1CTLW0 |= UCTXSTP;
266   while ( UCB1CTLW0 & UCTXSTP)
267   {
268     // Wait
269   }
270   while (UCB1STATW & UCBUSY)
271   {
272     // Wait
273   }
```

```
274
275   return 0;
276 }
277
278 void update_clock_numbers(unsigned int n)
279 {
280   // A1 & A2 hours
281   // A3 & A4 Mins
282   // A5 & A6 seconds
283   // assume numbers
284   // divide by # secs in hours
285   // divide by # secs in hours
286   // divide by #
287
288   unsigned int HH = 0;
289   unsigned int MM = 0;
290   unsigned int SS = 0;
291
292   unsigned int current_digit = 0;
293   current_digit = n % 3600;    // gets current # hours
294
295   // Seconds Logic
296   unsigned int seconds = n % 60;
297   if (seconds > 0)
298   {
299     SS = seconds % 10;
300   }
301   if (seconds > 10)
302   {
303     SS += (seconds / 10) *10;
304   }
305
306   n /= 60;
307
308   // Minutes Logic
309   unsigned int minutes = n % 60;
310   if (minutes > 0)
311   {
312     // add the one digits to the thing
313     MM += minutes % 10;
314   }
315   if (minutes > 10)
316   {
317     MM += (minutes / 10) * 10;
318   }
319
320   n = n/60;
321
322   // Hours Logic
323   unsigned int hours = n % (60);
324
325   if (hours > 0)
326   {
327     HH += hours % 10;
328   }
329   if (hours > 10)
330   {
```

```
331     HH += (hours / 10) * 10;
332   }
333
334   // Printing
335   uart_write_uint16(HH);
336   uart_write_char(':');
337   uart_write_uint16(MM);
338   uart_write_char(':');
339   uart_write_uint16(SS);
340   uart_write_char('\t');
341 }
342
343 void main(void)
344 {
345   volatile int n;
346   // Stop the Watchdog timer
347   WDTCTL = WDTPW | WDTHOLD;
348
349   // Unlock the GPIO pins
350   PM5CTL0 &= ~LOCKLPM5;
351
352   // Configure the LEDs as output
353   P1DIR |= redLED; // Direct pin as output
354   P1OUT &= ~redLED; // Turn LED Off
355
356   P9DIR |= greenLED; // Direct pin as output
357   P9OUT &= ~greenLED; // Turn LED Off
358
359   //buttons
360   P1DIR &= ~(BUT1 | BUT2);
361   P1REN |=  (BUT1 | BUT2);
362   P1OUT |=  (BUT1 | BUT2);
363
364   P1IES |=  (BUT1 | BUT2); //1: Interrupt on falling edge (0 for
        rising edge)
365   P1IFG &= ~(BUT1 | BUT2); //0: Clear the interrupt flags
366   P1IE  |=  (BUT1 | BUT2); //1: Enable the interrupts
367
368   config_ACLK_to_32KHz_crystal();
369
370   // standard delay is 1 second with interrupts
371   // Configure channel 0 for up mode with interrupts
372   TA0CCR0 = 32768;    // 1 second @ 32kHz
373   TA0CCTL0 |= CCIE;   // Enable channel 0 CCIE1
374   TA0CCTL0 &= ~CCIFG; // Clear Channel 0 CCIFG
375
376   // Use ACLK, divide by 1, up mode, clear TAR
377   TA0CTL = TASSEL_1 | ID_0 | MC_1 | TACLR;
378
379   // Ensure flag is cleared at the start
380   TA0CTL &= ~TAIFG;
381
382  // Enable Global Interrupt bit ( call an intrinsic function)
383  _enable_interrupt();
384  initialize_i2c();
385  initialize_uart();
386
```

```
  i2c_write_word(0x44,0x01, 0x7604);

  P1OUT |= redLED;
  uart_write_string("done with init\n");
  _delay_cycles(600000);
  i2c_read_word(0x44, 0x00, &LuxValue);
  prevBase = LuxValue;

  for (;;)
  {
    _delay_cycles(50000);
    // confirm loop in action
    P1OUT ^= redLED;
  }
}

// one second Timer system
// interrupt for blinking
#pragma vector = TIMER0_A0_VECTOR
__interrupt void TA00_ISR()
{
  P9OUT ^= greenLED;

  if (time == 1)
  {
    i2c_read_word(0x44, 0x00, &LuxValue);
    update_time();
    uart_write_char('\t');
    uart_write_uint16(LuxValue);
    uart_write_string(" lux");

    if (LuxValue > prevBase + 10)
    {
      uart_write_string("\t<Up>\n");
      prevBase = LuxValue;
    }
    else if (LuxValue < prevBase - 10)
    {
      uart_write_string("\t<Down>\n");
      prevBase = LuxValue;
    }
    else
    {
      uart_write_char('\n');
    }
    time = 0;
  }
  time = time + 1;
  current_time = current_time + 1;
}

void update_time()
{
  // if it is 9
  if (MM_c[1] == '9')
  {
    if (MM_c[0] < '5')
```

```
444     {
445       MM_c[0]++;
446       MM_c[1] = '0';
447     }
448     else
449     {
450       //now increase hour by one
451       MM_c[0] = '0';
452       MM_c[1] = '0';
453       // check last possible time, else go up
454       if (HH_c[0] == '2' && HH_c[1] == '3')
455       {
456         HH_c[0] = '0';
457         HH_c[1] = '0';
458       }
459       else if (HH_c[1] == '0')
460       {
461         HH_c[0]++;
462         HH_c[1] = '0';
463       }
464       else
465       {
466         HH_c[1]++;
467       }
468     }
469   }
470   else
471   {
472     MM_c[1]++;
473   }
474   print_time();
475 }
476
477 void print_time()
478 {
479   uart_write_string(HH_c);
480   uart_write_char(':');
481   uart_write_string(MM_c);
482 }
483
484 #pragma  vector = PORT1_VECTOR
485 __interrupt  void set_time()
486 {
487   __delay_cycles(50000);
488   if (P1IFG & BUT2)
489   {
490     TA0CTL &= ~MC_3;
491     uart_write_string("Enter the time...(3 or 4 digits then hit
      Enter)\n");
492     char given_char;
493
494     char time[] = "____";
495     volatile int loc = 0;
496     while (loc < 4)
497     {
498       given_char = uart_read_char();
499       if (given_char ==0)
```

```
500       {
501         continue;
502       }
503
504       if (given_char == 3 || given_char == 27 || given_char == 13)
505       {
506         break;
507       }
508       time[loc] = given_char;
509       loc++;
510     }
511     uart_write_string("got out!\n");
512
513     if (time[3] == 95)
514     {
515       uart_write_string("entered 3");
516       HH_c[0] = 48;
517       HH_c[1] = time[0];
518       MM_c[0] = time[1];
519       MM_c[1] = time[2];
520     }
521     else
522     {
523       uart_write_string("entered 4");
524       HH_c[0] = time[0];
525       HH_c[1] = time[1];
526       MM_c[0] = time[2];
527       MM_c[1] = time[3];
528     }
529
530     uart_write_string("Time set to ");
531     print_time();
532     uart_write_char('\n');
533
534     P1IFG &= ~BUT2;
535     TA0CTL |= MC_1;
536   }
537 }
```

Student Q&A

1

Given: *How many cycles does it take the ADC to convert a 12-bit result? (look in the configuration register that contains ADC12RES).*

2

Given: *In this experiment, we set our reference voltages* $VR+ = AVCC$ *(Analog Vcc) and* $VR- = AVSS$ *(Analog Vss). What voltage values do these signals have? Look in the MCU data sheet (slas789c) in Table 5.3. Assume that Vcc=3.3V and Vss=0.*