Final Project: Security Keypad Lock

John McAvoy
Email: mcavoyj5@students.rowan.edu

December 22, 2018

Contents

1 Project Description

This project was destined to be a security keypad, similar to the keypads found on digital safes and doors. The MSP430G2553 launchpad was used as well as a custom 12-button keypad that is used to enter and set combinations. The security keypad is able to set combinations, lock and unlock an electric lock, and communicate through UART.

2 Final Code

2.1 src/msp430g2553_keypad_lock.c

2.1.1 main

```c
/**
 * @file msp430g2553_keypad_lock.c
 * @author John McAvoy
 * @date 11 Dec 2018
 * @desc main program that uses MSP430 as keypad combinational lock
 */

#include <msp430.h>
#include <stdio.h>
#include "../lib/keypad.h" // setup_keypad_pins, handle_keypress
#include "../lib/security.h" // setup_lock_pins, setup_timeout_timer, lock
#include "../lib/uart.h" // setup_uart

int main(void)
{

  WDTCTL = WDTPW + WDTHOLD;                 // Stop watchdog timer

  setup_lock_pins();
  setup_keypad_pins();
  setup_timeout_timer();
  setup_uart();

  char welcome_message[27] = "Keypad Lock - John McAvoy\n\n";
  send_bytes(welcome_message, 27);

  __bis_SR_register(GIE);       // enable interrupts
}

```

Listing 1: src/msp430g2553_keypad_lock.c::main()

The main function starts by stopping the watch dog timer. Then, it calls the setup functions for the lock pins, keypad pins, timeout timer, and UART. Next, it sends a welcome message to UART and then enables the global interrupt.

2.1.2 Timer0_A1

```c

// Timer A0 interrupt service routine
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=TIMER0_A1_VECTOR
__interrupt void Timer0_A1 (void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(TIMER0_A1_VECTOR))) Timer0_A1 (void)
#else
#error Compiler not supported!
#endif
{
    stop_timeout_timer();
    lock(); // lock
}
```

Listing 2: src/msp430g2553_keypad_lock.c::Timer0_A1()

The Timer0_A1 interrupt service routine is called every time the timeout timer finishes its cycle. The service routine stops the timeout timer and then locks triggers the lock state.

2.1.3 USCI0RX_ISR

```c

#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCI0RX_ISR(void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(USCIAB0RX_VECTOR))) USCI0RX_ISR (void)
#else
#error Compiler not supported!
#endif
{
    unsigned char data = UCA0RXBUF;
}
```

Listing 3: src/msp430g2553_keypad_lock.c::USCI0RX_ISR()

The USCI0RX interrupt service routine was not used, however this project could use this in the future to send commands via UART to the device.

2.1.4 Port_1

```

// Port 1 interrupt service routine
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(PORT1_VECTOR))) Port_1 (void)
#else
#error Compiler not supported!
#endif
{
    handle_key_press();
    clear_key_interupt_flags();
}

```

Listing 4: src/msp430g2553_keypad_lock.c::Port_1()

The Port_1 interrupt service routine is called every time one of keypad keys that is connected to a P1 port is pressed. The service routine calls `handle_key_press` .

2.1.5 Port_2

```
// Port 2 interrupt service routine
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=PORT2_VECTOR
__interrupt void Port_2(void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(PORT2_VECTOR))) Port_2 (void)
#else
#error Compiler not supported!
#endif
{
    handle_key_press();
    clear_key_interupt_flags();
}
```

Listing 5: src/msp430g2553_keypad_lock.c::Port_2()

The Port_2 interrupt service routine calls the same functions as the Port_1 interrupt service routine. This service will run whenever the keypad keys connected to P2 are pressed.

2.2 lib/uart.c

```c
// uart.h

#ifndef UART_H
#define UART_H

#include <msp430.h>
#include <stdint.h>

void setup_uart();

void send_bytes(char *bytes, uint8_t length);

#endif // UART_H
```

Listing 6: lib/uart.h

2.2.1 setup_uart

```c
// uart.c

#include <msp430.h>
#include <stdint.h>

void setup_uart() {
    DCOCTL = 0;                               // Select lowest DCOx and MODx settings
    BCSCTL1 = CALBC1_1MHZ;                    // Set DCO
    DCOCTL = CALDCO_1MHZ;
    P1SEL = BIT1 + BIT2 ;                     // P1.1 = RXD, P1.2=TXD
    P1SEL2 = BIT1 + BIT2 ;                    // P1.1 = RXD, P1.2=TXD
    UCA0CTL1 |= UCSSEL_2;                     // SMCLK
    UCA0BR0 = 104;                            // 1MHz 9600
    UCA0BR1 = 0;                              // 1MHz 9600
    UCA0MCTL = UCBRS0;                        // Modulation UCBRSx = 1
    UCA0CTL1 &= ~UCSWRST;                     // **Initialize USCI state machine**
    IE2 |= UCA0RXIE;                          // Enable USCI_A0 RX interrupt
}
```

Listing 7: lib/uart.c::setup_uart()

Setup UART configures the UART to use P1.1 as RXD and P1.2 as TXD, uses the 1MHx clock and sets the baud rate at 9600.

2.2.2 send_bytes

```c

void send_bytes(char *bytes, uint8_t length) {
    while(length--) {
        while (!(IFG2&UCA0TXIFG));                // USCI_A0 TX buffer ready?
        UCA0TXBUF = *bytes; //Write the character
        bytes++; //Increment the bytes pointer to point to the next character
    }
}
```

Listing 8: lib/uart.c::send_bytes()

Send bytes takes a pointer to a byte array and loops through the length of the array. For each character in the array, the loop waits for the TX buffer to be ready, then it writes the byte to `UCA0TXBUF` which sends the byte via UART.

2.3 lib/security.c

```c
/**
 * @file security.h
 * @author John McAvoy
 * @date 11 Dec 2018
 * @desc provides security function logic for keylock
 */

#ifndef SECURITY_H
#define SECURITY_H

#include <msp430.h>
#include <stdint.h>
#include "Key.h"

#define UNLOCKED ( 1 << 3 ) // P2.3
#define TIMEOUT  ( 1 << 4 ) // P2.1
#define LOCKED   ( 1 << 5 ) // P2.5
#define PASSCODE_LENGTH 4

/**
 * @func setup_lock_pins
 * @desc initializes I/O pins for handling the lock/unlock
 */
void setup_lock_pins();

/**
 * @func setup_timeout_timer
 * @desc initializes the 3s timeout timer
 */
void setup_timeout_timer();

/**
 * @func setup_timeout_timer
 * @desc sets the new passcode
 * @param new_passcode - Key array of new passcode sequence
 */
void set_passcode(Key new_passcode[]);

/**
 * @func lock
 * @desc sets the state to locked
 */
void lock();

/**
 * @func lock
 * @desc sets the state to unlocked
 */
void unlock();

/**
 * @func start_timeout_timer
 * @desc starts the timeout timer
 */
void start_timeout_timer();

/**
 * @func stop_timeout_timer
 * @desc stops the timeout timer
 */
void stop_timeout_timer();

/**
 * @func handle_combiation
 * @desc takes in the entered combination and handles appropriate response
 * @param combination - Key array of the entered combination
 */
void handle_combiation(Key combination[]);

#endif // SECURITY_H
```

Listing 9: lib/security.h

2.3.1 setup_lock_pins

```c
/**
 * @file security.c
 * @author John McAvoy
 * @date 11 Dec 2018
 * @desc provides security function logic for keylock
 */

#define DEBUG 1
#ifdef DEBUG
#include    <stdio.h>
#include    "uart.h"
#endif

#include "security.h"
#include "Key.h"


Key passcode[PASSCODE_LENGTH] = { KEY_1, KEY_2, KEY_3, KEY_4 }; // initial passcode: 1-2-3-4
Key input_buffer[PASSCODE_LENGTH] = { KEY_ERR, KEY_ERR, KEY_ERR, KEY_ERR }; // holds input keys

uint8_t input_counter = 0; // tracks input count

/**
 * @func setup_lock_pins
 * @desc initializes I/O pins for handling the lock/unlock
 */
void setup_lock_pins() {

    // configure pins to I/O mode
    P2SEL &= ~(LOCKED + UNLOCKED);
    P2SEL2 &= ~(LOCKED + UNLOCKED);

    // configure pins to outputs
    P2DIR |= LOCKED + UNLOCKED;

    lock();
}
```

Listing 10: lib/security.c::setup_lock_pins()

The lock, unlock, and timeout pins are set to I/O mode and set to output direction. Then, the lock state is set.

2.3.2 setup_timeout_timer

```

/**
 * @func setup_timeout_timer
 * @desc initializes the timeout timer
 */
void setup_timeout_timer() {
    CCTL0 = CCIE;
    TACTL = TASSEL_2 + MC_1 + TACLR;    //ACLK, up mode, clear
    CCR0 = 3200;
}
```

Listing 11: lib/security.c::setup_timeout_timer()

The timeout timer is used to delay combination inputs. The timer is configured to use the 32kHz clock to count to 3200 with the capture compare interrupt enabled. This makes the timeout delay 1 second.

2.3.3 set_passcode

```

/**
 * @func setup_timeout_timer
 * @desc sets the new passcod* @param new_passcode - Key array of new passcode sequence
 */
void set_passcode(Key new_passcode[]) {
    for(uint8_t i = 0 ; i < PASSCODE_LENGTH; i++) {
        passcode[i] = new_passcode[i];
    }
}
```

Listing 12: lib/security.c::set_passcode()

`set_passcode` is used to read a `Key` array, loop through the array and set the passcode to the `new_passcode` .

2.3.4 lock

```c

/**
 * @func lock
 * @desc sets the state to locked
 */
void lock() {
    P2OUT &= ~UNLOCKED;
    P2OUT |= LOCKED;
}
```

Listing 13: lib/security.c::lock()

The lock state sets the unlocked pin off and the locked pin on.

2.3.5 unlock

```c

/**
 * @func lock
 * @desc sets the state to unlocked
 */
void unlock() {
    P2OUT &= ~LOCKED;
    P2OUT |= UNLOCKED;
}
```

Listing 14: lib/security.c::unlock()

The unlock state sets the unlocked pin on and the locked pin off.

2.3.6 start_timeout_timer

```c

/**
 * @func start_timeout_timer
 * @desc starts the timeout timer
 */
void start_timeout_timer() {
    TA0CTL = MC_2; // continuous mode
    P2OUT |= TIMEOUT; // turn on timeout LED
}
```

Listing 15: lib/security.c::start_timeout_timer()

The timeout timer is turned on by seting TA0CTL into continuous mode. When the timer is started, the timout pin is set high.

2.3.7 stop_timeout_timer

```

/**
 * @func stop_timeout_timer
 * @desc stops the timeout timer
 */
void stop_timeout_timer() {
    TA0CTL = MC_0; // halt
    TAR = 0; // clear timer
}
```

Listing 16: lib/security.c::stop_timeout_timer()

The timeout timer is turned off by setting TACTL to halt and TAR .

2.3.8 handle_combination

```
/**
 * @func handle_combiation
 * @desc takes in the entered combination and handles appropriate response
 * @param combination - Key array of the entered combination
 */
void handle_combiation(Key combination[]) {
    uint8_t passcode_entered = 1; // true
    for(uint8_t i = 0; i < PASSCODE_LENGTH; i++) {
        if(combination[i] != passcode[i])
            passcode_entered = 0; // false
    }

    if(passcode_entered)
        unlock();
    else
        lock();

    start_timeout_timer();
}
```

Listing 17: lib/security.c::handle_combination()

2.4 lib/keypad.c

```c
#ifndef KEYPAD_H
#define KEYPAD_H

#include <msp430.h>
#include "Key.h"

#define A0 ( 1 << 0 ) // P1.0
#define A1 ( 1 << 3 ) // P1.3
#define A2 ( 1 << 4 ) // P1.4
#define A3 ( 1 << 5 ) // P1.5
#define A9 ( 1 << 7 ) // P1.7

#define A4 ( 1 << 0 ) // P2.0
#define A5 ( 1 << 2 ) // P2.1
#define A6 ( 1 << 1 ) // P2.2
#define A7 ( 1 << 6 ) // P2.6
#define A8 ( 1 << 7 ) // P2.7

#define P1KEYS ( A0 | A1 | A2 | A3 | A9 )
#define P2KEYS ( A4 | A5 | A6 | A7 | A8 )
/**
 * @func setup_keypad_pins
 * @desc initializes the appropriate I/O pins for reading the keypad
 */
void setup_keypad_pins();

/**
 * @func clear_key_interupt_flags
 * @desc clears keypad port interrupt flags
 */
void clear_key_interupt_flags();


/**
 * @func get_key_pressed
 * @desc determines the pressed key based on A3..A0 pins
 * @returns Key enum corresponding to the key pressed
 */
Key get_key_pressed();

/**
 * @func handle_key_press
 * @desc handles reading the combination endetered
 */
void handle_key_press();

/**
 * @func send_combination_in
 * @desc sends current combination enter via uart
 */
void send_combination_in();
#endif // KEYPAD_H
```

Listing 18: lib/keypad.h

2.4.1 setup_keypad_pins

```c
/**
 * @file keypad.c
 * @author John McAvoy
 * @date 11 Dec 2018
 * @desc provides functions related to reading the keypad inputs
 */

#include <msp430.h>
#include <stdio.h>
#include "keypad.h"
#include "security.h"
#include "uart.h"
#include "Key.h"

/**
 * @func setup_keypad_pins
 * @desc initializes the appropriate I/O pins for reading the keypad
 */
void setup_keypad_pins() {
    // configure pins to I/O mode
    P1SEL &= ~(P1KEYS);
    P1SEL2 &= ~(P1KEYS);
    P1REN &= ~(P1KEYS);

    P2SEL &= ~(P2KEYS);
    P2SEL2 &= ~(P2KEYS);
    P2REN &= ~(P2KEYS);

    // configure pins to inputs
    P1DIR &= ~(P1KEYS);
    P2DIR &= ~(P2KEYS);

    // interrupt on low-to-high
    P1IES &= ~(P1KEYS);
    P2IES &= ~(P2KEYS);

    // enables KF interrupt
    P1IE |= (P1KEYS);
    P2IE |= (P2KEYS);
}
```

Listing 19: lib/keypad.c::setup_keypad_pins()

The keypad pins are connected to I/O pins in Port_1 and Port_2. All of the pins are set to I/O mode and the direction is set to input. Also, the Port_1 and Port_2 interrupts are also enabled for each of the keypad keys.

2.4.2 clear_key_interrupt_flags

```c

/**
 * @func clear_key_interupt_flags
 * @desc clears keypad port interrupt flags
 */
void clear_key_interupt_flags() {
    P1IFG &= ~(P1KEYS);
    P2IFG &= ~(P2KEYS);
}
```

Listing 20: lib/keypad.c::setup_keypad_pins()

The keypad key interrupts are cleared by setting the corresponding bits `P1IFG` and `P2IIFG` low.

2.4.3 get_key_pressed

```c

/**
 * @func get_key_pressed
 * @desc determines the pressed key based on A3..A0 pins
 * @returns Key enum corresponding to the key pressed
 */
Key get_key_pressed() {

    if(P1IN & A9) return KEY_9;
    if(P2IN & A8) return KEY_8;
    if(P2IN & A7) return KEY_7;
    if(P2IN & A6) return KEY_6;
    if(P2IN & A5) return KEY_5;
    if(P2IN & A4) return KEY_4;
    if(P1IN & A3) return KEY_3;
    if(P1IN & A2) return KEY_2;
    if(P1IN & A1) return KEY_1;
    if(P1IN & A0) return KEY_0;
    return KEY_ERR; // default
}
```

Listing 21: lib/keypad.c::get_key_pressed

The `get_key_pressed` function returns the corresponding `Key` that is pressed by using bit masks on `P1` and `P2` .

2.4.4 handle_key_press

```c

static Key combination_in[PASSCODE_LENGTH];
static uint8_t in_counter = 0;

void handle_key_press() {
    Key key_in = get_key_pressed();
    if(key_in != KEY_ERR){
        combination_in[in_counter++] = key_in;
        send_combination_in();
        if (in_counter == PASSCODE_LENGTH) {
            handle_combiation(combination_in);
            in_counter = 0;
        }
    }
}
```

Listing 22: lib/keypad.c::handle_key_press()

The `handle_key_press` function gets the `Key` that is pressed and adds it to the `combination_in` array. When the `combination_in` is full, then the `combination_in` is passed to `handle_combination` .

2.4.5 send_combination_in

```c

void send_combination_in() {
    const char buffer_size = (in_counter * 2 ); // 1-2-3-4
    char buffer[buffer_size];

    for(uint8_t i = 0; i < in_counter; i++) {
        buffer[i*2] = key2Char(combination_in[i]);
    }
    for(uint8_t i = 0; i < in_counter / 2; i++) {
        buffer[i*2+1] = '-';
    }

    buffer[buffer_size - 1] = '\n';
    send_bytes(buffer, buffer_size);
}
```

Listing 23: lib/keypad.c::send_combination_in()

The `send_combinaiton_in` function creates an output character buffer that is sends the current combination entered via UART.

2.5 lib/Key.c

```c
/**
 * @file key.h
 * @author John McAvoy
 * @date 11 Dec 2018
 * @desc provides Key enum typdef
 */
#ifndef KEY_H
#define KEY_H

typedef enum Key {
    KEY_0,
    KEY_1,
    KEY_2,
    KEY_3,
    KEY_4,
    KEY_5,
    KEY_6,
    KEY_7,
    KEY_8,
    KEY_9,
    KEY_ERR
} Key;

char key2Char(Key k);

#endif // KEY_H
```

Listing 24: lib/Key.h

2.5.1 key2Char

```c
/**
 * @file key.c
 * @author John McAvoy
 * @date 11 Dec 2018
 * @desc provides Key enum typdef
 */

#include "Key.h"

char key2Char(Key k) {
    switch(k) {
        case KEY_0: return '0';
        case KEY_1: return '1';
        case KEY_2: return '2';
        case KEY_3: return '3';
        case KEY_4: return '4';
        case KEY_5: return '5';
        case KEY_6: return '6';
        case KEY_7: return '7';
        case KEY_8: return '8';
        case KEY_9: return '9';
        case KEY_ERR: return '?';
        default: return '!';
    }
}
```

Listing 25: lib/Key.c::key2Char

The `Key` enum holds all the different types of key categories which makes it useful for passing keycodes between functions instead of numbers. `key2Char` is used to convert a `Key` into a printable character.

3 Unit Tests

3.1 tests/keypad/keypad_test.c

```c
/**
 * @file keypad_test.c
 * @author John McAvoy
 * @date 11 Dec 2018
 * @desc test program for keypad.c
 */

#include <msp430.h>
#include <stdio.h>
#include "../../lib/Key.h"
#include "../../lib/keypad.h"
#include "../../lib/uart.h"

int main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // Stop watchdog timer
  setup_uart();
  setup_keypad_pins();

  char start_message[12] = "Keypad Test\n";
  send_bytes(start_message, 12);

  char p1_reg = P1IN;
  while(1){
      __bis_SR_register(GIE);       // enable interrupts
  }

}

// Port 1 interrupt service routine
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(PORT1_VECTOR))) Port_1 (void)
#else
#error Compiler not supported!
#endif
{
    char buffer[3];
    sprintf(buffer, "%02d\n", get_key_pressed());
    send_bytes(buffer, 3);
    clear_key_interupt_flags();
}

// Port 2 interrupt service routine
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=PORT2_VECTOR
__interrupt void Port_2(void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(PORT2_VECTOR))) Port_2 (void)
#else
#error Compiler not supported!
#endif
{
    char buffer[3];
    sprintf(buffer, "%02d\n", get_key_pressed());
    send_bytes(buffer, 3);
    clear_key_interupt_flags();
}
```

Listing 26: tests/keypad/keypad_test.c

This unit test was used to test the `keypad.c` functions. Each time a key was pressed, its value was sent via UART for debugging.

3.2 tests/security/security_test.c

```c
/**
 * @file security_test.c
 * @author John McAvoy
 * @date 11 Dec 2018
 * @desc test program for security_test.c
 */

#include <msp430.h>
#include <stdio.h>
#include "../../lib/Key.h"
#include "../../lib/security.h"
#include "../../lib/uart.h"

int main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // Stop watchdog timer
  setup_uart();
  setup_lock_pins();

  char start_message[25] = "Start of Security Test\n";
  send_bytes(start_message, 20);

  char locking_message[13] = "Locking Test\n";
  send_bytes(locking_message, 13);

  // locking test
  char unlock_message[7] = "Unlock\n";
  send_bytes(unlock_message, 7);
  unlock();
  __delay_cycles(1000000);
  char lock_message[5] = "Lock\n";
  send_bytes(lock_message, 5);
  lock();

  __delay_cycles(1000000);

  char cc_message[20] = "Correct Combination\n";
  send_bytes(cc_message, 20);
  // correct key combination test
  Key correctCombination[4] = {KEY_0, KEY_1, KEY_3, KEY_4};
  handle_combiation(correctCombination);
  __delay_cycles(1000000);
  send_bytes(lock_message, 5);
  lock();

  __delay_cycles(1000000);

  char bc_message[17] = "Bad Combination\n";
  send_bytes(bc_message, 17);
  // bad key combination test
  Key badCombination[17] = {KEY_5, KEY_3, KEY_2, KEY_0};
  handle_combiation(badCombination);

  __delay_cycles(10000000);
  send_bytes(lock_message, 5);
  lock();

  char nc_message[17] = "New Combination\n";
  send_bytes(nc_message, 17);
  // passcode set test
  Key newCombination[4] = {KEY_7, KEY_8, KEY_9, KEY_1};
  set_passcode(newCombination);
  correctCombination[0] = KEY_7;
  correctCombination[1] = KEY_8;
  correctCombination[2] = KEY_9;
  correctCombination[3] = KEY_1;
  send_bytes(cc_message, 20);
  handle_combiation(correctCombination);
  send_bytes(cc_message, 20);
  __delay_cycles(10000000);
  lock();

  uint8_t end_message[21] = "End of Security Test\n";
  send_bytes(end_message, 21);

}
```

Listing 27: tests/keypad/security/security_test.c

This unit test was used to test the `security.c` functions. Various `Key` combinations were sent to the `handle_combination` function to test that they work properly.

3.3 tests/timer/timer_test.c

```c
/**
 * @file timer_test.c
 * @author John McAvoy
 * @date 11 Dec 2018
 * @desc test program for timeout_timer
 */

#include <msp430.h>
#include <stdio.h>
#include "../../lib/security.h"
#include "../../lib/uart.h"

int main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // Stop watchdog timer
  setup_uart();
  setup_lock_pins();
  setup_timeout_timer();

  char start_message[12] = "Timer Test\n";
  send_bytes(start_message, 12);

  start_timeout_timer();

  //__no_operation();
  while(1){
      __bis_SR_register(GIE);       // enable interrupts
  }

}

// Timer A0 interrupt service routine
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=TIMER0_A0_VECTOR
__interrupt void Timer_A (void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(TIMER0_A0_VECTOR))) Timer_A (void)
#else
#error Compiler not supported!
#endif
{
    char message[13] = "Timer Reached";
    send_bytes(message, 13);
}
```

Listing 28: tests/timer/timer_test.c

This unit test was used to make sure that the timeout timer was configured correctly. Each time the interrupt was triggered, a message was via to UART.

3.4 tests/uart/uart_test.c

```c
/**
 * @file msp430g2553_keypad_lock.c
 * @author John McAvoy
 * @date 11 Dec 2018
 * @desc test program for uart.c
 */

#include <msp430.h>
#include <stdio.h>
#include "../../lib/uart.h"

int main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // Stop watchdog timer
  setup_uart();

  const char test_message[16] = "UART Testbench";
  send_bytes(test_message, 16);

  __bis_SR_register(GIE);

  while(1) __no_operation();
}

#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCI0RX_ISR(void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(USCIAB0RX_VECTOR))) USCI0RX_ISR (void)
#else
#error Compiler not supported!
#endif
{
    unsigned char data = UCA0RXBUF;
    unsigned char out_message[15];
    sprintf(out_message, "'%2x' Received\n", data);
    send_bytes(out_message, 15);
}
```

Listing 29: tests/uart/uart_test.c

This unit test was used to test the `uart.c` functions.

4 Conclusions and Future Work

The keypad security lock works correctly, however it can be improved by adding encoding logic to reduce the number of I/O pins used. The MSP430G2553 does not have enough I/O pins for all 12 keys in the number pads. I originally designed the device to use a primary encoder that would reduce the 12 signals from the keypad into a 4-bit signal. I wasn't able to accomplish this circuit because all I had was 2-input OR gates and the circuit was too complicated to fit on a breadboard.