```
#include "ds1306_rtc_driver.h"

void spi_rtc_ds1306_config(void)
{
    // unselect PA1
    PORTA &= ~_BV(1);
    // enable SPI, Master, CPOL = 1 CPHA = 1, fck/8
    SPCR = _BV(SPE) | _BV(MSTR) | _BV(CPOL) | _BV(CPHA) | _BV(SPR0);
    SPSR = _BV(SPI2X);
    // clear any old data
    char temp = SPSR;
    temp = SPDR;
}

void write_rtc(unsigned char reg_rtc, unsigned char data_rtc)
{
    // select slave
    PORTA |= _BV(1);
    // send register
    SPDR = reg_rtc;
    // wait for transmission complete
    while ( !(SPSR & _BV(SPIF)) ) {
    }
    // send data
    SPDR = data_rtc;
    // wait for transmission complete
    while ( !(SPSR & _BV(SPIF)) ) {
    }
    // unselect slave
    PORTA &= ~_BV(1);
    // clear SPIF bit in SPSR
    char temp = SPDR;
}

unsigned char read_rtc(unsigned char reg_rtc)
{
    // select slave
    PORTA |= _BV(1);
    // send register
    SPDR = reg_rtc;
    // wait for transmission complete
    while ( !(SPSR & _BV(SPIF)) ) {
    }
    // send dummy data
    SPDR = 0x00;
    // wait for transmission complete
    while ( !(SPSR & _BV(SPIF)) ) {
    }
    // read recieved data
    char temp = SPDR;
    // unselect slave
    PORTA &= ~_BV(1);
    return temp;
}
```

```
void block_write_rtc (volatile unsigned char *array_ptr, unsigned char
strt_addr, unsigned char count)
{
    // select slave
    PORTA |= _BV(1);
    // send register
    SPDR = strt_addr;
    // wait for transmission complete
    while ( !(SPSR & _BV(SPIF)) ) {
    }
    for (char i = 0; i < count; i++) {
        // send data
        SPDR = array_ptr[i];
        // wait for transmission complete
        while ( !(SPSR & _BV(SPIF)) ) {
        }
    }

    // unselect slave
    PORTA &= ~_BV(1);
    // clear SPIF bit in SPSR
    char temp = SPDR;
}

void block_read_rtc (volatile unsigned char *array_ptr, unsigned char
strt_addr, unsigned char count)
{
      // select slave
    PORTA |= _BV(1);
    // send register
    SPDR = strt_addr;
    // wait for transmission complete
    while ( !(SPSR & _BV(SPIF)) ) {
    }

    for (char i = 0; i < count; i++) {
        // send dummy data
        SPDR = 0x00;
        // wait for transmission complete
        while ( !(SPSR & _BV(SPIF)) ) {
        }
        // read recieved data
        array_ptr[i] = SPDR;
    }

    // unselect slave
    PORTA &= ~_BV(1);
}

void unlock_rtc()
{
    unsigned char temp = read_rtc(0x0F);
    write_rtc(0x8F, (temp & 0xBF) );
}

void lock_rtc()
{
```

```
    unsigned char temp = read_rtc(0x0F);
    write_rtc(0x8F, (temp | 0x40) );
}
```

```
/*****************************************************************
 * File:        Name of file
 * Author:      Bryant Gonzaga
 * Created:     Date file was first created
 * Modified:    Date file was last modified
 *
 * Notes:
 *  Processor specific, libraries need
 *
 * Description:
 *  A full description of what can be found in this file
 *
 * How To:
 *  If necessary add some instructions on how to use the file.
 *****************************************************************/

#include "fsm_state_tables.h"

state present_state;

void fsm(state ps, key key)
{
    /* Find the index for task and next state */
    int i = 0;
    while ((ps_transitions_ptr[ps][i].key_val != key) &&
           (ps_transitions_ptr[ps][i].key_val != eol)) {
        i++;
    }

    /* Execute Task/Output/Funcntion */
    ps_transitions_ptr[ps][i].task_ptr();

    /* Update Present State to Next State */
    present_state = ps_transitions_ptr[ps][i].next_state;
}
```

```
/*****************************************************************
 * File:        fsm_tasks.c
 * Author:      Bryant Gonzaga
 * Created:     4/12/2018
 * Modified:    4/12/2018
 *
 * Notes:
 *  Processor specific, libraries need
 *
 * Description:
 *  A full description of what can be found in this file
 *
 * How To:
 *  If necessary add some instructions on how to use the file.
 *****************************************************************/

#include "fsm_tasks.h"

/* Global Variables */
static unsigned char temp_time_data[7]; // temp data for time
static unsigned char temp_alarm_data[4]; // temp data for alarm

/* Global Variables */
unsigned char key_code;         // stores the last pressed key value
unsigned char rtc_time_data[7]; // stores full date and time
unsigned char rtc_alarm_data[4]; // stores full date and time for
alarm

extern state present_state;

static unsigned char time_input_state;
static unsigned char alarm_input_state;

//TODO: maybe find a better way to check if confirm key is a valid
input
unsigned char confirm_valid;

void alarm_input_init()
{
    alarm_input_state = 0;
    dsp_alarm_setter_fn();
}

void time_input_init()
{
    time_input_state = 0;
    dsp_time_setter_fn();
}

void alarm_input_handler_fn()
{
    /* Check Key is in range */
    if (key_code > 9) {
        return;
    }
```

```
    unsigned char temp = 0xFF;

    // TODO: only minumul checks are done. Create a more robust check
of imputs.

    /* Handle Key Input */
    switch (alarm_input_state) {
        case 0:     // Hours
            if (key_code < 3) {
                temp = 2;
            }
            break;
        case 1:
            if (key_code < 10) {
                temp = 2;
            }
            break;
        case 2:     // Minutes
            if (key_code < 6) {
                temp = 1;
            }
            break;
        case 3:
            if (key_code < 10) {
                temp = 1;
            }
            break;
        case 4:    // Seconds
            if (key_code < 6) {
                temp = 0;
            }
            break;
        case 5:
            if (key_code < 10) {
                temp = 0;
                confirm_valid = 1;
            }
            break;
        case 6:
            if (key_code < 4) {
                temp_alarm_data[key_code] |= _BV(7);
            }
        default:
            alarm_input_state = 0;
            break;
    }

    /* On Good Key Input */
    if (temp != 0xFF) {
        /* Update temp Time Array */
        if ( (alarm_input_state % 2) == 0) {
            temp_alarm_data[temp] = (key_code << 4);
        } else {
            temp_alarm_data[temp] |= (key_code & 0x0F);
        }
        /* Increment Handler State */
        alarm_input_state = (alarm_input_state + 1) % 7;
```

```
        /* Display Prompt */
        dsp_alarm_setter_fn();

        temp_alarm_data[3] |= _BV(7);
    }
}

void time_input_handler_fn()
{
    /* Check Key is in range */
    if (key_code > 9) {
        return;
    }

    unsigned char temp = 0xFF;

    // TODO: only minumul checks are done. Create a more robust check
of imputs.

    /* Handle Key Input */
    switch (time_input_state) {
        case 0:     // Month
            if (key_code < 2) {
                temp = 5;
            }
            break;
        case 1:
            if ( (temp_time_data[5] >> 4) == 0) {
                if (key_code != 0) {
                    temp = 5;
                }
            } else if ( (temp_time_data[5] >> 4) == 1) {
                if (key_code < 3) {
                    temp = 5;
                }
            }
            break;
        case 2:     // Day
            if (key_code < 4) {
                temp = 4;
            }
            break;
        case 3:
            temp = 4;
            break;
        case 4:     // Year
            temp = 6;
            break;
        case 5:
            temp = 6;
            break;
        case 6:     // Day of the week
            temp = 3;
            break;
        case 7:     // 12 or 24 hr choice
            if (key_code == 1) {
                temp = 12;
```

```
            } else if (key_code == 2) {
                temp = 24;
            }
            break;
        case 8:     // Hours
            if (key_code < 3) {
                temp = 2;
            }
            break;
        case 9:
                temp = 2;
            break;
        case 10:     // Minutes
            if (key_code < 6) {
                temp = 1;
            }
            break;
        case 11:
                temp = 1;
            break;
        case 12:    // Seconds
            if (key_code < 6) {
                temp = 0;
            }
            break;
        case 13:
            temp = 0;
            confirm_valid = 1;
            break;
        default:
            time_input_state = 0;
            break;
    }

    /* On Good Key Input */
    if (temp != 0xFF) {
        /* Update temp Time Array */
        if (time_input_state == 6) {

        } else if (time_input_state == 7) {

        } else if ( (time_input_state % 2) == 0) {
            temp_time_data[temp] = (key_code << 4);
        } else {
            temp_time_data[temp] |= (key_code & 0x0F);
        }
        /* Increment Handler State */
        time_input_state = (time_input_state + 1) % 15;
        /* Display Prompt */
        dsp_time_setter_fn();
    }
}

void dsp_alarm_setter_fn()
{
    /* Init LCD */
    init_lcd_dog();     // setup spi configuration
```

```
    clear_dsp();        // clear ram buffer

    /* save what is to be displayed */
    switch (alarm_input_state) {
        case 0:
        case 1:
            printf("Input hour:\n");
            printf("%d%d",
                ( (temp_alarm_data[2] >> 4) & 0x03 ) ,    // Hour
                (  temp_alarm_data[2]       & 0x0F )      // Hour
            );
            break;
        case 2:
        case 3:
            printf("Input minutes:\n");
            printf("%d%d",
                ( (temp_alarm_data[1] >> 4) & 0x0F ) ,    // Minute
                (  temp_alarm_data[1]       & 0x0F )      // Minute
            );
            break;
        case 4:
        case 5:
            printf("Input seconds:\n");
            printf("%d%d",
                ( (temp_alarm_data[0] >> 4) & 0x0F ) ,    // Second
                (  temp_alarm_data[0]       & 0x0F )      // Second
            );
            break;
        case 6:
            printf("Is this correct?");
            printf("Time: %d%d:%d%d:%d%d",
                ( (temp_alarm_data[2] >> 4) & 0x03 ) ,    // Hour
                (  temp_alarm_data[2]       & 0x0F ) ,    // Hour
                ( (temp_alarm_data[1] >> 4) & 0x0F ) ,    // Minute
                (  temp_alarm_data[1]       & 0x0F ) ,    // Minute
                ( (temp_alarm_data[0] >> 4) & 0x0F ) ,    // Second
                (  temp_alarm_data[0]       & 0x0F )      // Second
            );
            break;
        default:
            break;
    }
    /* Print Chosen Message */
    update_lcd_dog();
}

void dsp_time_setter_fn()
{
    /* Init LCD */
    init_lcd_dog();     // setup spi configuration
    clear_dsp();        // clear ram buffer

    /* save what is to be displayed */
    switch (time_input_state) {
        case 0:
        case 1:
            printf("Input month:\n");
```

```
    printf("%d%d",
        ( (temp_time_data[5] >> 4) & 0x03 ) ,    // Month
        (  temp_time_data[5]       & 0x0F )      // Month
    );
    break;
case 2:
case 3:
    printf("Input day:\n");
    printf("%d%d",
        ( (temp_time_data[4] >> 4) & 0x03 ) ,    // Day
        (  temp_time_data[4]       & 0x0F )      // Day
    );
    break;
case 4:
case 5:
    printf("Input year:\n");
    printf("%d%d",
        ( (temp_time_data[6] >> 4) & 0x0F ) ,    // Year
        (  temp_time_data[6]       & 0x0F )      // Year
    );
    break;
case 6:
    printf("Select day:");                    // Day of week
    break;
case 7:
    printf("Select mode:\n");
    printf("1: 12 hr\n2: 24 hr");
    break;
case 8:
case 9:
    printf("Input hour:\n");
    printf("%d%d",
        ( (temp_time_data[2] >> 4) & 0x03 ) ,    // Hour
        (  temp_time_data[2]       & 0x0F )      // Hour
    );
    break;
case 10:
case 11:
    printf("Input minutes:\n");
    printf("%d%d",
        ( (temp_time_data[1] >> 4) & 0x0F ) ,    // Minute
        (  temp_time_data[1]       & 0x0F )      // Minute
    );
    break;
case 12:
case 13:
    printf("Input seconds:\n");
    printf("%d%d",
        ( (temp_time_data[0] >> 4) & 0x0F ) ,    // Second
        (  temp_time_data[0]       & 0x0F )      // Second
    );
    break;
case 14:
    printf("Is this correct?");
    printf("Date: %d%d/%d%d/%d%d\n",
        ( (temp_time_data[5] >> 4) & 0x03 ) ,    // Month
        (  temp_time_data[5]       & 0x0F ) ,    // Month
```

```
                ( (temp_time_data[4] >> 4) & 0x03 ) ,    // Day
                (  temp_time_data[4]       & 0x0F ) ,    // Day
                ( (temp_time_data[6] >> 4) & 0x0F ) ,    // Year
                (  temp_time_data[6]       & 0x0F )      // Year
            );
            printf("Time: %d%d:%d%d:%d%d",
                ( (temp_time_data[2] >> 4) & 0x03 ) ,    // Hour
                (  temp_time_data[2]       & 0x0F ) ,    // Hour
                ( (temp_time_data[1] >> 4) & 0x0F ) ,    // Minute
                (  temp_time_data[1]       & 0x0F ) ,    // Minute
                ( (temp_time_data[0] >> 4) & 0x0F ) ,    // Second
                (  temp_time_data[0]       & 0x0F )      // Second
            );
            break;
        default:
            break;
    }
    /* Print Chosen Message */
    update_lcd_dog();
}

void confirm_alarm_fn()
{
        if (confirm_valid) {
        /* Transfer Temp Data to Permenant Array */
        for (unsigned char i = 0; i < 4; i++) {
            rtc_alarm_data[i] = temp_alarm_data[i];
        }
        /* Initialize ds1306 IC */
        spi_rtc_ds1306_config();
        /* unlock mem */
        unlock_rtc();
        /* Send Time to DS1306 */
        block_write_rtc(rtc_alarm_data, 0x87, 4);
        /* lock mem */
        lock_rtc();
        /* confirm no longer valid */
        confirm_valid = 0;
        /* display data */
        dsp_all_fn();
    }

}

void confirm_time_fn()
{
    if (confirm_valid) {
        /* Transfer Temp Data to Permenant Array */
        for (unsigned char i = 0; i < 7; i++) {
            rtc_time_data[i] = temp_time_data[i];
        }
        /* Initialize ds1306 IC */
        spi_rtc_ds1306_config();
        /* unlock mem */
        unlock_rtc();
        /* Send Time to DS1306 */
        block_write_rtc(rtc_time_data, 0x80, 7);
```

```
        /* lock mem */
        lock_rtc();
        /* confirm no longer valid */
        confirm_valid = 0;
        /* display data */
        dsp_all_fn();
    }
}

void dsp_all_fn()
{
     /* Get Data */
    block_read_rtc(rtc_time_data, 0x00, 7); // get the time data from
rtc

    /* Init LCD */
    init_lcd_dog();     // setup spi configuration
    clear_dsp();        // clear ram buffer

    /* save what is to be displayed */
    printf("Date: %d%d/%d%d/%d%d\n",
            ( (rtc_time_data[5] >> 4) & 0x03 ) ,    // Month
            (  rtc_time_data[5]       & 0x0F ) ,    // Month
            ( (rtc_time_data[4] >> 4) & 0x03 ) ,    // Day
            (  rtc_time_data[4]       & 0x0F ) ,    // Day
            ( (rtc_time_data[6] >> 4) & 0x0F ) ,    // Year
            (  rtc_time_data[6]       & 0x0F )      // Year
    );

    printf("Time: %d%d:%d%d:%d%d\n",
            ( (rtc_time_data[2] >> 4) & 0x03 ) ,    // Hour
            (  rtc_time_data[2]       & 0x0F ) ,    // Hour
            ( (rtc_time_data[1] >> 4) & 0x0F ) ,    // Minute
            (  rtc_time_data[1]       & 0x0F ) ,    // Minute
            ( (rtc_time_data[0] >> 4) & 0x0F ) ,    // Second
            (  rtc_time_data[0]       & 0x0F )      // Second
    );

    // TODO: Check for 12 HR or 24 HR setting

    printf("Alarm 0:");

    /* Actually send data */
    update_lcd_dog();
}

void error_fn()
{
}

void nop_fn()           // not sure if needed yet
{

}
```

```
/*********************************************
 * File:    humidicon.c
 * Author:  Bryant Gonzaga
 * Date:    3/6/2018
 *********************************************/

#include "humidicon.h"

extern unsigned char humidicon_byte0;
extern unsigned char humidicon_byte1;
extern unsigned char humidicon_byte2;
extern unsigned char humidicon_byte3;
extern unsigned long int humidity;
extern unsigned long int temperature;

void spi_humidicon_config()
{
    // unselect PA0
    DDRA |= _BV(0);
    // enable SPI, Master, CPOL = 1 CPHA = 1, fck/64
    SPCR = _BV(SPE) | _BV(MSTR) | _BV(CPOL) | _BV(CPHA) | _BV(SPR1) |
_BV(SPR0);
    // clear any old data
    char temp = SPSR;
    temp = SPDR;
}

unsigned char read_humidicon_byte()
{
    // write to data register to start sclk
    SPDR = 0x00;
    // wait for transmission complete
    while ( !(SPSR & _BV(SPIF)) ) {
    }
    // clear SPIF bit in SPSR
    char temp = SPDR;

    return temp;
}

void read_humidicon()
{
    // select slave
    PORTA &= ~_BV(0);

    humidicon_byte3 = read_humidicon_byte();
    humidicon_byte2 = read_humidicon_byte();
    humidicon_byte1 = read_humidicon_byte();
    humidicon_byte0 = read_humidicon_byte();

    // unselect slave
    PORTA |= _BV(0);

    // sshh hhhhh - hhhh hhhh - tttt tttt - tttt ttxx
    unsigned int temp = humidicon_byte0 + ( ((int) humidicon_byte1) <<
8);
```

```
    unsigned int humi = humidicon_byte2 + ( ((int) humidicon_byte3) <<
8);
    temp >>= 2; // fix shifted number
    humi &= 0x3FFF;

    humidity = compute_scaled_rh(humi);
    temperature = compute_scaled_temp(temp);
}

//*****************************************************************
**********
// Function : unsigned int compute_scaled_rh(unsigned int rh)
// Date and version : version 1.0
// Target MCU : ATmega128A
// Author : Ken Short
// DESCRIPTION
// Computess scaled relative humidity in units of 0.01% RH from the
raw 14-bit
// realtive humidity value from the Humidicon.
//
//
// Modified
//*****************************************************************
**********
long int compute_scaled_rh(unsigned int rh)
{
    long int temp = ( (long) rh ) * 10000;
    long int tmpo = temp / 16382;
    return tmpo;
}

//*****************************************************************
**********
// Function : unsigned int compute_scaled_temp(unsigned int temp)
// Date and version : version 1.0
// Target MCU : ATmega128A
// Author : Ken Short
// DESCRIPTION
// Computess scaled temperature in units of 0.01 degrees C from the
raw 14-bit
// temperature value from the Humidicon
//
//
// Modified
//*****************************************************************
**********
long int compute_scaled_temp(unsigned int temp)
{
    long int tmp = ( (long) temp ) * 16500;
    long int tmo = (tmp / 16382) - 4000;
    return tmo;
}
```

```
#include "lcd_dog_c_driver.h"


void delay_30uS()
{
    __delay_cycles(240);
}

void delay_40mS()
{
    __delay_cycles(320000);
}

void init_spi_lcd()
{
    // Enable SPI, Master, fck/64,
    SPCR = _BV(SPE) | _BV(MSTR) | _BV(CPOL) | _BV(CPHA) | _BV(SPR1) |
_BV(SPR0);

    char temp = SPSR;
    temp = SPDR;
}

int lcd_spi_transmit_CMD(char cmd)
{
    // clear rs for command
    LCD_PORT &= ~_BV(RS);
    // select slave
    LCD_PORT &= ~_BV(SS_bar);
    // send command
    SPDR = cmd;
    // wait for transmission complete
    while ( !(SPSR & _BV(SPIF)) ) {
    }
    // clear SPIF bit in SPSR
    char temp = SPDR;
    // unselect slave
    LCD_PORT |= _BV(SS_bar);

    return 0;
}

int lcd_spi_transmit_DATA(char data)
{
    // set rs for data
    LCD_PORT |= _BV(RS);
    // select slave
    LCD_PORT &= ~_BV(SS_bar);
    // clear SPIF bit in SPSR
    char temp = SPSR;
    temp = SPDR;
    // send data
    SPDR = data;
```

```
    // wait for transmission
    while ( !(SPSR & _BV(SPIF)) ) {
    }
    // clear SPIF bit in SPSR
    temp = SPDR;
    // unselect slave
    LCD_PORT |= _BV(SS_bar);

    return 0;
}

void init_lcd_dog(void)
{
    init_spi_lcd();
    // start up delay
    delay_40mS();
    // function set 1
    lcd_spi_transmit_CMD(0x39);
    delay_30uS();
    // function set 2
    lcd_spi_transmit_CMD(0x39);
    delay_30uS();
    // bias set
    lcd_spi_transmit_CMD(0x1E);
    delay_30uS();
    // power ctrl
    lcd_spi_transmit_CMD(0x50);
    delay_30uS();
    // follower ctrl
    lcd_spi_transmit_CMD(0x6C);
    delay_40mS();
    // contrast set
    lcd_spi_transmit_CMD(0x77);
    delay_30uS();
    // display on
    lcd_spi_transmit_CMD(0x0c);
    delay_30uS();
    // clear display
    lcd_spi_transmit_CMD(0x01);
    delay_30uS();
    //entry mode
    lcd_spi_transmit_CMD(0x06);
    delay_30uS();
}

void update_lcd_dog(void)
{
    init_spi_lcd();

    // init DDRAM addr-ctr
    lcd_spi_transmit_CMD(0x80) ;
    delay_30uS();
    // send data
    for (char i = 0; i < 16; i++) {
        lcd_spi_transmit_DATA(dsp_buff_1[i]);
        delay_30uS();
```

```
    }

    // init DDRAM addr-ctr
    lcd_spi_transmit_CMD(0x90) ;
    delay_30uS();
    // send data
    for (char i = 0; i < 16; i++) {
        lcd_spi_transmit_DATA(dsp_buff_2[i]);
        delay_30uS();
    }

    // init DDRAM addr-ctr
    lcd_spi_transmit_CMD(0xA0) ;
    delay_30uS();
    // send data
    for (char i = 0; i < 16; i++) {
        lcd_spi_transmit_DATA(dsp_buff_3[i]);
        delay_30uS();
    }
}
```

```
//*****************************************************************
*******
//
// File Name            : lcd_ext.c
// Title                : LCD Utilities
// Date                 : 02/07/10
// Version              : 1.0
// Target MCU           : ATmega128 @  MHz
// Target Hardware      ;
// Author               : Ken Short
// DESCRIPTION
// The file contains two functions that make it easier for a C
// program to use the LCD display. The function clear_dsp() clears the
display
// buffer arrays. When followed by the function update_dsp(), the
// display is blanked.
//
// The function putchar() puts a single character, passed to it as an
argument,
// into the display buffer at the position corresponding to the value
of
// variable index. This putchar function replaces the standard putchar
funtion,
// so a printf statement will print to the LCD
//
// Warnings             : none
// Restrictions         : none
// Algorithms           : none
// References           : none
//
// Revision History     : Initial version
//
//
//*****************************************************************
******

#include "lcd.h"


static char index;    // index into display buffer

//*****************************************************************
**********
// Function             : void clear_dsp(void)
// Date and version     : 02/07/10, version 1.0
// Target MCU           : ATmega128
// Author               : Ken Short
// DESCRIPTION
// Clears the display buffer. Treats each 16 character array
separately.
// NOTE: update_dsp must be called after to see results
//
// Modified
//*****************************************************************
**********
```

```
/*****************************************************************
 * File:        rtc_sys.c
 * Author:      Bryant Gonzaga
 * Created:     4/11/2018
 * Modified:    4/12/2018
 *
 * Notes:
 *  Intended for ATmega128.
 *
 * Description:
 *  This is a table driven fsm that can set the alarm 0 for an RTC.
 *****************************************************************/

/* Include Libraries */
#include <iom128.h>
#include <intrinsics.h>

/* Include Personal Libraries */
#include "ds1306_rtc_driver.h"
#include "humidicon.h"
#include "fsm_defs.h"
#include "lcd.h"

/* FSM Function in fsm.c */
extern void fsm(state ps, key key_val);

/* MAGIC NUMBER */
#define INIT_MAGIC_NUM  0x3E65

__flash unsigned int* rtc_init_done;

/* Function Prototypes */
void init_rtc_sys();

int main(void)
{
    /* Initialize Ports */
    DDRA = 0x03;        // slave select for rtc and humidicon
    DDRB = 0xFF;        // spi pins
    DDRC = 0xF0;        // for keypad
    DDRD = 0xF0;        // external interrupts

    /* Deselect slaves */
    PORTA &= ~_BV(1);   // deselect rtc
    PORTA |= _BV(0);    // deselect humidicon
    PORTB |= _BV(0);    // deselect lcd

    /* Activate Internal Pullups */
    PORTC = 0x0F;   // for keypad
    PORTD = 0x05;   // for keypad interrupt

    /* Clear the LCD */
    init_lcd_dog();
    clear_dsp();
    update_lcd_dog();
```

```
    /* Init RTC Chip */
    unlock_rtc();
    write_rtc(0x8F, 0x04);
    lock_rtc();

    /* Initialize Interrupts */
    EICRA = 0x3C;
    EIMSK = 0x07;
    __enable_interrupt();

    /* Initialize RTC */
    init_rtc_sys();     // initialize the time and date

    while (1) {
    }
}

void init_rtc_sys()
{
    /* check if system has already been initialized */
    if (*rtc_init_done == INIT_MAGIC_NUM) {
        fsm(display_state, rtc_1hz_key);
        return;
    }
    /* Fisrt Power Up Ever */
    fsm(display_state, set_time_key);
    *rtc_init_done = INIT_MAGIC_NUM;
}
```

```
void clear_dsp(void)
{
  // assuming buffers might not be contiguous
  for(char i = 0; i < 16; i++)
    dsp_buff_1[i] = ' ';

  for(char i = 0; i < 16; i++)
    dsp_buff_2[i] = ' ';

  for(char i = 0; i < 16; i++)
    dsp_buff_3[i] = ' ';

  index = 0;
}


//*****************************************************************
**********
// Function             : int putchar(int c)
// Date and version     : 02/07/10, version 1.0
// Target MCU           : ATmega128
// Author               : Ken Short
// DESCRIPTION
// This function displays a single ascii chararacter c on the lcd at
the
// position specifieb by the global variable index
// NOTE: update_dsp must be called after to see results
//
//
// Modified
//*****************************************************************
**********

int putchar(int c)
{
    if (c == '\f') {
        index = 0;
        return 0;
    }
    if (index < 16) {
        if (c == '\b') {
            if (index == 0) {
                return 0;
            } else {
                dsp_buff_1[--index] = ' ';
            }
        } else if (c == '\n' ||  c == '\r') {
            index = 16;
        } else {
            dsp_buff_1[index++] = (char)c;
        }
    } else if (index < 32) {
        if (c == '\b') {
            if (index == 16) {
                index = 15;
                dsp_buff_1[index] = ' ';
            } else {
```

```
                dsp_buff_2[--index - 16] = ' ';
            }
        } else if (c == '\n' ||  c == '\r') {
            index = 32;
        } else {
            dsp_buff_2[index++ - 16] = (char)c;
        }
    } else if (index < 48) {
        if (c == '\b') {
            if (index == 32) {
                index = 31;
                dsp_buff_2[15] = ' ';
            } else {
                dsp_buff_3[--index - 32] = ' ';
            }
        } else if (c == '\n' ||  c == '\r') {
            index = 0;
        } else {
            dsp_buff_3[index++ - 32] = (char)c;
        }
    } else {
        index = 0;
        dsp_buff_1[index++] = (char)c;
  }

    return c;
}
```

```
/*****************************************************************
 * File:        rtc_sys_isr.c
 * Author:      Bryant Gonzaga
 * Created:     4/12/2018
 * Modified:    4/12/2018
 *
 * Notes:
 *  Intended for ATmega128.
 *
 * Description:
 *  Interrupt subroutines.
 *****************************************************************/

//*****************************************************************
*******
//
// File Name            : keyscan_isr.c
// Title                : Keypad scan interrupt service routine
// Date                 : 02/07/10
// Version              : 1.0
// Target MCU           : ATmega128 @  MHz
// Target Hardware      ;
// Author               : Ken Short
// DESCRIPTION
// When keypad interrupt occurs, key matirx is scanned and is encoded
using
// a table lookup. The keypad is connected to PORTC. See diagram in
laboratory
// description.
//
// Warnings             : none
// Restrictions         : none
// Algorithms           : none
// References           : none
//
// Revision History     : Initial version
//
//
//*****************************************************************
******

//#define debug   //this can be uncommented to remove delays for
simulation

/* Include Libraries */
#include <iom128.h>         //Atmega128 definitions
#include <intrinsics.h>     //Intrinsic functions.
#include <avr_macros.h>     //Useful macros.

#include "fsm_defs.h"
#include "ds1306_rtc_driver.h"
/*
* Port pin numbers for columns and rows of the keypad
*/
//PORT Pin Definitions.
#define COL1  7   //pin definitions for PortB
```

```
#define COL2  6
#define COL3  5
#define COL4  4
#define ROW1  3
#define ROW2  2
#define ROW3  1
#define ROW4  0

#define INT0  0   //pin definitions for PortD

/* Lookup table declaration */
const char tbl[16] = {1, 2, 3, 15, 4, 5, 6, 14, 7, 8, 9, 13, 10, 0,
11, 12};

/* FSM Function in fsm.c */
extern void fsm(state ps, key key);

//=============================================================//
//                    Static Functions                         //
//=============================================================//
static void check_release(void)
{
#ifndef debug
  while(!TESTBIT(PIND,INT0));     //Check that keypad key is released.

  __delay_cycles(50000);          //Delay (.05secs) / (1 / 1MHz)
cycles.

  while(!TESTBIT(PIND,INT0));     //Check that key has stopped
bouncing.
#endif
}

static key keycode_to_keyenum(unsigned char key_code)
{
    if (key_code < 10) {
        return num_keys;
    }
    switch (key_code) {
        case  SET_ALARM_KEY:
            return set_alarm_key;
        case  SET_TIME_KEY:
            return set_time_key;
        case  CONFIRM_KEY:
            return confirm_key;
        case  CANCEL_KEY:
            return cancel_key;
    }
    return eol;
}

//=============================================================//
//                Interrupt Subroutines                        //
//=============================================================//
/**
 * Interrupt subroutine for key presses.
 */
```

```
#pragma vector=INT0_vect          // Declare vector location.
__interrupt void keypad_isr(void)   // Declare interrupt function
{
  extern char key_code;                   // Holds key_code
  extern state present_state;

  //Note: TESTBIT returns 0 if bit is not set and a non-zero number
otherwise.

  if(!TESTBIT(PINC,ROW1))         //Find Row of pressed key.
    key_code = 0;
  else if(!TESTBIT(PINC,ROW2))
    key_code = 4;
  else if(!TESTBIT(PINC,ROW3))
    key_code = 8;
  else if(!TESTBIT(PINC,ROW4))
    key_code = 12;

  DDRC = 0x0F;                    //Reconfigure PORTC for Columns.
  PORTC = 0xF0;

#ifndef debug
  __delay_cycles(256);            //Let PORTC settle.
#endif

  if(!TESTBIT(PINC,COL1))         //Find Column.
    key_code += 0;
  else if(!TESTBIT(PINC,COL2))
    key_code += 1;
  else if(!TESTBIT(PINC,COL3))
    key_code += 2;
  else if(!TESTBIT(PINC,COL4))
    key_code += 3;

  DDRC = 0xF0;                    //Reconfigure PORTC for Rows.
  PORTC = 0x0F;

  key_code = (tbl[key_code]);

  fsm(present_state, keycode_to_keyenum(key_code));

  check_release();                //Wait for keypad release.
}

/**
 * Interrupt subroutine for the 1Hz output from the rtc.
 */
#pragma vector=INT1_vect
__interrupt void rtc_1hz_isr()
{
    /* Display New Data */
    fsm(present_state, rtc_1hz_key);
}

/**
 * Interrupt subroutine for alarm 0, generated by the rtc.
 */
```

```
#pragma vector=INT2_vect
__interrupt void alarm_zero_isr()
{
}
```