Ali Asadi
Github root directory: https://github.com/aliasadivet/Lab_Submition-.git

Lab08

Task 00: Execute provided code

Task 01:

Modified code:

```
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include "inc/hw_memmap.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/ssi.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "driverlib/adc.h"
//! This example shows how to configure the SSI0 as SPI Master. The code will
//! send three characters on the master Tx then polls the receive FIFO until
//! 3 characters are received on the master Rx.
//!
//! This example uses the following peripherals and I/O signals. You must
//! review these and change as needed for your own board:
//! - SSI0 peripheral
//! - GPIO Port A peripheral (for SSI0 pins)
//! - SSI0Clk - PA2
//! - SSI0Fss - PA3
//! - SSI0Rx - PA4
//! - SSI0Tx - PA5
//!
//! The following UART signals are configured only for displaying console
//! messages for this example. These are not required for operation of SSI0.
//! - UART0 peripheral
//! - GPIO Port A peripheral (for UART0 pins)
//! - UART0RX - PA0
//! - UART0TX - PA1
//!
//! This example uses the following interrupt handlers. To use this example
//! in your own application you must add these interrupt handlers to your
//! vector table.
//! - None.
//
//*****************************************************************************
//*****************************************************************************
//
// Number of bytes to send and receive.
//
//*****************************************************************************
#define NUM_SSI_DATA 3
//*****************************************************************************
```

```c
//
// This function sets up UART0 to be used for a console to display information
// as the example is running.
//
//*****************************************************************************
void InitConsole(void)
{
     // Enable GPIO port A which is used for UART0 pins.
     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
     // Configure the pin muxing for UART0 functions on port A0 and A1.
     // This step is not necessary if your part does not support pin muxing.
     // TODO: change this to select the port/pin you are using.
     GPIOPinConfigure(GPIO_PA0_U0RX);
     GPIOPinConfigure(GPIO_PA1_U0TX);
     // Enable UART0 so that we can configure the clock.
     SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
     // Use the internal 16MHz oscillator as the UART clock source.
     UARTClockSourceSet(UART0_BASE, UART_CLOCK_PIOSC);
     // Select the alternate (UART) function for these pins.
     GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
     // Initialize the UART for console I/O.
     UARTStdioConfig(0, 115200, SysCtlClockGet());
}
char* atoi(int value, char *buffer)
{
    char buff[10];
    int i = 0, j;
    do {
        buff[i++] = value % 10;
        value /= 10;
    }while(value);

    for(j = 0; j < i; j++)
        buffer[j] = buff[i-j-1]|0x30;
    buffer[j] = 0;
    return buffer;
}
//*****************************************************************************
//
// Configure SSI0 in master Freescale (SPI) mode. This example will send out
// 3 bytes of data, then wait for 3 bytes of data to come in. This will all be
// done using the polling method.
//
//*****************************************************************************
int main(void)
{

     uint32_t pui32DataRx[NUM_SSI_DATA];
     uint32_t ui32Index;
     uint32_t ui32ADC0Value[4];
     volatile uint32_t ui32TempAvg;
     volatile uint32_t ui32TempValueC;
     volatile uint32_t ui32TempValueF;
     char buffer[16];

SysCtlClockSet(SYSCTL_SYSDIV_12|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);
     SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
     ADCHardwareOversampleConfigure(ADC0_BASE, 32);
     ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
```

```
ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_TS);
ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_TS);
ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_TS);
ADCSequenceStepConfigure(ADC0_BASE,1,3,ADC_CTL_TS|ADC_CTL_IE|ADC_CTL_END);
ADCSequenceEnable(ADC0_BASE, 1);
// The SSI0 peripheral must be enabled for use.
SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
// For this example SSI0 is used with PortA[5:2]. The actual port and pins
// used may be different on your part, consult the data sheet for more
// information. GPIO port A needs to be enabled so these pins can be used.
     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
// Configure the pin muxing for SSI0 functions on port A2, A3, A4, and A5.
// This step is not necessary if your part does not support pin muxing.
// TODO: change this to select the port/pin you are using.
GPIOPinConfigure(GPIO_PA2_SSI0CLK);
GPIOPinConfigure(GPIO_PA3_SSI0FSS);
GPIOPinConfigure(GPIO_PA4_SSI0RX);
GPIOPinConfigure(GPIO_PA5_SSI0TX);
// Configure the GPIO settings for the SSI pins. This function also gives
// control of these pins to the SSI hardware. Consult the data sheet to
// see which functions are allocated per pin.
// The pins are assigned as follows:
// PA5 - SSI0Tx
// PA4 - SSI0Rx
// PA3 - SSI0Fss
// PA2 - SSI0CLK
// TODO: change this to select the port/pin you are using.
//
GPIOPinTypeSSI(GPIO_PORTA_BASE,  GPIO_PIN_4 | GPIO_PIN_3 |
GPIO_PIN_2);
GPIOPinTypeSSI(GPIO_PORTA_BASE,  GPIO_PIN_5 );
// Configure and enable the SSI port for SPI master mode. Use SSI0,
// system clock supply, idle clock level low and active low clock in
// freescale SPI mode, master mode, 1MHz SSI frequency, and 8-bit data.
// For SPI mode, you can set the polarity of the SSI clock when the SSI
// unit is idle. You can also configure what clock edge you want to
// capture data on. Please reference the datasheet for more information on
// the different SPI modes.
SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_0,
SSI_MODE_MASTER, 1000000, 8);
// Enable the SSI0 module.
SSIEnable(SSI0_BASE);


// Set up the serial console to use for displaying messages. This is
     // just for this example program and is not needed for SSI operation.
     InitConsole();
     // Display the setup on the console.
     UARTprintf("SSI ->\n");
     UARTprintf(" Mode: SPI\n");
     UARTprintf(" Data: 8-bit\n\n");
while(1)
{
    // Read any residual data from the SSI port. This makes sure the receive
    // FIFOs are empty, so we don't read any unwanted junk. This is done here
    // because the SPI SSI mode is full-duplex, which allows you to send and
    // receive at the same time. The SSIDataGetNonBlocking function returns
    // "true" when data was returned, and "false" when no data was returned.
    // The "non-blocking" function checks if there is any data in the receive
```

```
        // FIFO and does not "hang" if there isn't.

        while(SSIDataGetNonBlocking(SSI0_BASE, &pui32DataRx[0]))
        {
        }

        // Initialize the data to send.
        ADCIntClear(ADC0_BASE, 1);
        ADCProcessorTrigger(ADC0_BASE, 1);

        while(!ADCIntStatus(ADC0_BASE, 1, false))
               {
               }

       ADCSequenceDataGet(ADC0_BASE, 1, ui32ADC0Value);
       ui32TempAvg = (ui32ADC0Value[0] + ui32ADC0Value[1] + ui32ADC0Value[2] +
ui32ADC0Value[3] + 2)/4;
       ui32TempValueC = (1475 - ((2475 * ui32TempAvg)) / 4096)/10;
       ui32TempValueF = ((ui32TempValueC * 9) + 160) / 5;
        // Display indication that the SSI is transmitting data.
       UARTprintf("Sent : Temp - '%d'\n\r", ui32TempValueC);
       atoi(ui32TempValueC, buffer);
        // Send 3 bytes of data.
        for(ui32Index = 0; ui32Index < strlen(buffer); ui32Index++)
        {

            // Send the data using the "blocking" put function. This function
            // will wait until there is room in the send FIFO before returning.
            // This allows you to assure that all the data you send makes it into
            // the send FIFO.
            SSIDataPut(SSI0_BASE, buffer[ui32Index]);
        }

        // Wait until SSI0 is done transferring all the data in the transmit
FIFO.
        while(SSIBusy(SSI0_BASE))
        {
        }

        // Display indication that the SSI is receiving data.
        UARTprintf("\nReceived:\n ");

        // Receive 3 bytes of data.
        for(ui32Index = 0; ui32Index < strlen(buffer); ui32Index++)
        {
            // Receive the data using the "blocking" Get function. This function
            // will wait until there is data in the receive FIFO before
returning.
            SSIDataGet(SSI0_BASE, &pui32DataRx[ui32Index]);

            // Since we are using 8-bit data, mask off the MSB.
            pui32DataRx[ui32Index] &= 0x00FF;

            // Display the data that SSI0 received.
            UARTprintf("'%c' ", pui32DataRx[ui32Index]);
        }
    }

}
```

Grading scheme: 30% Coding, 30% Documentation, 40% Execution/Video.

Ali Asadi
Github root directory: https://github.com/aliasadivet/Lab_Submition-.git

Task02

Modified code:

```
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_memmap.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/ssi.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"

#define NUM_LEDS 8
uint8_t frame_buffer[NUM_LEDS*3];
void reset();
void send_data(uint8_t* data, uint8_t num_leds);
void fill_frame_buffer(uint8_t r, uint8_t g, uint8_t b,uint32_t num_leds);

static volatile uint32_t ssi_lut[] = {
    0b100100100,
    0b110100100,
    0b100110100,
    0b110110100,
    0b100100110,
    0b110100110,
    0b100110110,
    0b110110110
    };


int main(void) {

    SysCtlClockSet(SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL | SYSCTL_XTAL_16MHZ
|SYSCTL_OSC_MAIN);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlDelay(50000);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
    SysCtlDelay(50000);
    GPIOPinConfigure(GPIO_PA5_SSI0TX);
    GPIOPinConfigure(GPIO_PA2_SSI0CLK);
    GPIOPinConfigure(GPIO_PA4_SSI0RX);
    GPIOPinConfigure(GPIO_PA3_SSI0FSS);
    GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_5);
    GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_2);
    GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_4);
    GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_3);
    //20 MHz data rate
    SSIConfigSetExpClk(SSI0_BASE, 80000000, SSI_FRF_MOTO_MODE_0, SSI_MODE_MASTER,
2400000, 9);
    SSIEnable(SSI0_BASE);


    while(1)
    {
```

```
        fill_frame_buffer(255, 0, 0, NUM_LEDS);
        send_data(frame_buffer, NUM_LEDS);
        SysCtlDelay(50000000);
        fill_frame_buffer(0, 255, 0, NUM_LEDS);
        send_data(frame_buffer, NUM_LEDS);
        SysCtlDelay(50000000);
        fill_frame_buffer(0, 0, 255, NUM_LEDS);
        send_data(frame_buffer, NUM_LEDS);
        SysCtlDelay(50000000);
       fill_frame_buffer(255, 255, 0, NUM_LEDS);
        send_data(frame_buffer, NUM_LEDS);
        SysCtlDelay(50000000);
        fill_frame_buffer(0, 255, 255, NUM_LEDS);
        send_data(frame_buffer, NUM_LEDS);
        SysCtlDelay(50000000);
        fill_frame_buffer(255, 0, 255, NUM_LEDS);
        send_data(frame_buffer, NUM_LEDS);
        SysCtlDelay(50000000);
         fill_frame_buffer(255, 255, 255, NUM_LEDS);
        send_data(frame_buffer, NUM_LEDS);
        SysCtlDelay(50000000);

    }

}

void send_data(uint8_t* data, uint8_t num_leds)
{
    uint32_t i, j, curr_lut_index, curr_rgb;

    for(i = 0; i < (num_leds*3); i = i + 3) {
        curr_rgb = (((uint32_t)data[i + 2]) << 16) | (((uint32_t)data[i + 1]) <<
8) | data[i];
        for(j = 0; j < 24; j = j + 3) {
            curr_lut_index = ((curr_rgb>>j) & 0b111);
            SSIDataPut(SSI0_BASE, ssi_lut[curr_lut_index]);
        }
    }
    for(i = 0; i < 100; i++) {
            SSIDataPut(SSI0_BASE, 0);
        }
}

void fill_frame_buffer(uint8_t r, uint8_t g, uint8_t b, uint32_t num_leds)
{
    uint32_t i;
    uint8_t* frame_buffer_index = frame_buffer;
    for(i = 0; i < num_leds; i++) {
        *(frame_buffer_index++) = g;
        *(frame_buffer_index++) = r;
        *(frame_buffer_index++) = b;
    }
}
```