Geovanni Portillo
Github root directory: https://github.com/portig1/submissions_E2

Date Submitted: October 20, 2019

Task 01:

Youtube Link: https://youtu.be/UgNqeK3ALyI

Modified Schematic (if applicable):

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"
#include "utils/uartstdio.c"
#include "driverlib/adc.h"

//*****************************************************************************
//
//! \addtogroup ssi_examples_list
//! <h1>SPI Master (spi_master)</h1>
//!
//! 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            1 //Only sending Temperature value one at a time

//*****************************************************************************
//
// 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.
    // TODO: change this to whichever GPIO port you are using.
    //
    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.
    // TODO: change this to select the port/pin you are using.
    //
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    //
    // Initialize the UART for console I/O.
    //
    UARTStdioConfig(0, 115200, 16000000);
}
```

Geovanni Portillo
Github root directory: https://github.com/portig1/submissions_E2

```
uint32_t getADCtemp(void);

//*****************************************************************************
//
// 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)
{
#if defined(TARGET_IS_TM4C129_RA0) ||                                         \
    defined(TARGET_IS_TM4C129_RA1) ||                                         \
    defined(TARGET_IS_TM4C129_RA2)
    uint32_t ui32SysClock;
#endif

    uint32_t pui32DataTx[NUM_SSI_DATA];
    uint32_t pui32DataRx[NUM_SSI_DATA];
    uint32_t ui32Index;
    uint32_t pui32ADCDataTx;
    uint32_t pui32ADCDataRx;

    //
    // Set the clocking to run directly from the external crystal/oscillator.
    // TODO: The SYSCTL_XTAL_ value must be changed to match the value of the
    // crystal on your board.
    //
#if defined(TARGET_IS_TM4C129_RA0) ||                                         \
    defined(TARGET_IS_TM4C129_RA1) ||                                         \
    defined(TARGET_IS_TM4C129_RA2)
    ui32SysClock = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                                       SYSCTL_OSC_MAIN |
                                       SYSCTL_USE_OSC), 25000000);
#else
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_16MHZ);
#endif

    //
            // The ADC0 peripheral must be enabled for use.
            //
            SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
            SysCtlDelay(3);

            //
            // Enable sample sequence 3 with a processor signal trigger.  Sequence 3
            // will do a single sample when the processor sends a singal to start the
            // conversion.  Each ADC module has 4 programmable sequences, sequence 0
            // to sequence 3.  This example is arbitrarily using sequence 3.
            //
            ADCSequenceConfigure(ADC0_BASE, 3, ADC_TRIGGER_PROCESSOR, 0);

            //
```

```
            // Configure step 0 on sequence 3.  Sample the temperature sensor
            // (ADC_CTL_TS) and configure the interrupt flag (ADC_CTL_IE) to be set
            // when the sample is done.  Tell the ADC logic that this is the last
            // conversion on sequence 3 (ADC_CTL_END).  Sequence 3 has only one
            // programmable step.  Sequence 1 and 2 have 4 steps, and sequence 0 has
            // 8 programmable steps.  Since we are only doing a single conversion
using
            // sequence 3 we will only configure step 0.  For more information on the
            // ADC sequences and steps, reference the datasheet.
            //
            ADCSequenceStepConfigure(ADC0_BASE, 3, 0, ADC_CTL_TS | ADC_CTL_IE |
                                     ADC_CTL_END);

            //
            // Since sample sequence 3 is now configured, it must be enabled.
            //
            ADCSequenceEnable(ADC0_BASE, 3);

            //
            // Clear the interrupt status flag.  This is done to make sure the
            // interrupt flag is cleared before we sample.
            //
            ADCIntClear(ADC0_BASE, 3);


    //
    // 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");

    //
    // 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.
    // TODO: change this to whichever GPIO port you are using.
    //
    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_5 | GPIO_PIN_4 | GPIO_PIN_3 |
                   GPIO_PIN_2);

    //
    // 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.
    //
#if defined(TARGET_IS_TM4C129_RA0) ||                                         \
    defined(TARGET_IS_TM4C129_RA1) ||                                         \
    defined(TARGET_IS_TM4C129_RA2)
    SSIConfigSetExpClk(SSI0_BASE, ui32SysClock, SSI_FRF_MOTO_MODE_0,
                       SSI_MODE_MASTER, 1000000, 8);
#else
    SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_0,
                       SSI_MODE_MASTER, 1000000, 8);
#endif

    //
    // Enable the SSI0 module.
    //
    SSIEnable(SSI0_BASE);
    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.
//
pui32DataTx[0] = 's';
pui32DataTx[1] = 'p';
pui32DataTx[2] = 'i';
pui32ADCDataTx = getADCtemp();
//
// Display indication that the SSI is transmitting data.
//
UARTprintf("\nTemperature Sent: ");

//
// Send 3 bytes of data.
//
for(ui32Index = 0; ui32Index < NUM_SSI_DATA; ui32Index++)
{
    //
    // Display the data that SSI is transferring.
    //

    //UARTprintf("'%c' ", pui32DataTx[ui32Index]);
    UARTprintf("%d ", pui32ADCDataTx);

    //
    // 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, pui32DataTx[ui32Index]);
    SSIDataPut(SSI0_BASE, pui32ADCDataTx);
}
UARTprintf("\n");
//
// 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("\nTemperature Received:  ");

//
// Receive 3 bytes of data.
//
for(ui32Index = 0; ui32Index < NUM_SSI_DATA; 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("%d ", pui32DataRx[ui32Index]);
     }
    UARTprintf("\n");
    }
    //
    // Return no errors
    //
    return(0);
}

uint32_t getADCtemp(void) {
    //
         // This array is used for storing the data read from the ADC FIFO. It
         // must be as large as the FIFO for the sequencer in use.  This example
         // uses sequence 3 which has a FIFO depth of 1.  If another sequence
         // was used with a deeper FIFO, then the array size must be changed.
         //
         uint32_t ADCValues[1];

         //
         // These variables are used to store the temperature conversions for
         // Celsius and Fahrenheit.
         //
         uint32_t TempValueC ;
         uint32_t TempValueF ;

        //
        // Trigger the ADC conversion.
        //
        ADCProcessorTrigger(ADC0_BASE, 3);

        //
        // Wait for conversion to be completed.
        //
        while(!ADCIntStatus(ADC0_BASE, 3, false))
        {
        }

        //
        // Clear the ADC interrupt flag.
        //
        ADCIntClear(ADC0_BASE, 3);
```

```
        //
        // Read ADC Value.
        //
        ADCSequenceDataGet(ADC0_BASE, 3, ADCValues);

        //
        // Use non-calibrated conversion provided in the data sheet. I use floats in
intermediate
        // math but you could use intergers with multiplied by powers of 10 and
divide on the end
        // Make sure you divide last to avoid dropout.
        //
        TempValueC = (uint32_t)(147.5 - ((75.0*3.3 *(float)ADCValues[0])) / 4096.0);

        //
        // Get Fahrenheit value.  Make sure you divide last to avoid dropout.
        //
        TempValueF = ((TempValueC * 9) + 160) / 5;

        //
        // This function provides a means of generating a constant length
        // delay.  The function delay (in cycles) = 3 * parameter.  Delay
        // 250ms arbitrarily.
        //
        SysCtlDelay(80000000 / 12);
        return TempValueF;
}
```

Task 02:

Youtube Link: https://youtu.be/6mzuEv5ZfLU

Modified Schematic (if applicable):

Modified Code:

```
/*
 * main.c
 */
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/fpu.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
```

Geovanni Portillo
Github root directory: https://github.com/portig1/submissions_E2

```
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "driverlib/ssi.h"
#include "utils/uartstdio.h"

#define NUM_LEDS 1
uint8_t frame_buffer[NUM_LEDS*3];
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) {

    FPULazyStackingEnable();

    // 80MHz
    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);

    uint32_t i = 0;
   // fill_frame_buffer(255, 0, 0, NUM_LEDS); //R
    while(1)
    {
        fill_frame_buffer(255, 0, 0, NUM_LEDS); //R
        send_data(frame_buffer, NUM_LEDS);
        SysCtlDelay(800000); // delay more then 50us
```

Geovanni Portillo
Github root directory: https://github.com/portig1/submissions_E2

```
          fill_frame_buffer(0, 255, 0, NUM_LEDS); //G
          send_data(frame_buffer, NUM_LEDS);
          SysCtlDelay(800000); // delay more then 50us

        fill_frame_buffer(0, 0, 255, NUM_LEDS); //B
        send_data(frame_buffer, NUM_LEDS);
        SysCtlDelay(800000); // delay more then 50us

        fill_frame_buffer(255, 255, 0, NUM_LEDS); //RG
        send_data(frame_buffer, NUM_LEDS);
        SysCtlDelay(800000); // delay more then 50us

          fill_frame_buffer(255, 0, 255, NUM_LEDS); //RB
          send_data(frame_buffer, NUM_LEDS);
          SysCtlDelay(800000); // delay more then 50us

          fill_frame_buffer(0, 255, 255, NUM_LEDS); //GB
          send_data(frame_buffer, NUM_LEDS);
          SysCtlDelay(800000); // delay more then 50us

          fill_frame_buffer(40, 255, 255, NUM_LEDS); //RGB
          send_data(frame_buffer, NUM_LEDS);
          SysCtlDelay(800000); // delay more then 50us
    }

        return 0;
}

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]);
                }
        }

        SysCtlDelay(50000); // delay more then 50us
}

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;
        }
}
```

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

Geovanni Portillo
Github root directory: https://github.com/portig1/submissions_E2

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