Student Name Github root directory:
(insert link here)

Date Submitted:

Task 00: Execute provided code

Youtube Link: https://www.youtube.com/watch?v=XXE-jX9o03c

Task 01:

Youtube Link:

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 "driverlib/adc.h"
#include "driverlib/rom.h"

#define TARGET_IS_BLIZZARD_RB1
//*****************************************************************************
//
//! \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)
```

```c
//! - 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

//*****************************************************************************
//
// 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);
}
//*****************************************************************************
//
// 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 tempval;

    //
    // 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_TM4C123_RA0) ||                                        \
defined(TARGET_IS_TM4C123_RA1) ||                                        \
defined(TARGET_IS_TM4C123_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
```

```
    //

    // 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_TM4C123_RA0) ||                                       \
defined(TARGET_IS_TM4C123_RA1) ||                                           \
defined(TARGET_IS_TM4C123_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]))
    {
}
    volatile uint32_t ui32TempAvg;
volatile uint32_t ui32TempValueC;
volatile uint32_t ui32TempValueF;

          uint32_t ui32ADC0Value[4];

          ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
```

```
        // number of samples to be averaged  32 for task 2
          ROM_ADCHardwareOversampleConfigure(ADC0_BASE, 64);

          //configure the ADC Sequencer ( ADC0, sample sequencer 1, processor
triggers the sequence, highest priority)
          ROM_ADCSequenceConfigure(ADC0_BASE, 2, ADC_TRIGGER_PROCESSOR, 0);


          // configure the four steps in the sequencer, 0-2 on sequencer 1 to
sample temp (ADC_CTL_TS), ADC0, sequencer 1, step 0-2...
          ROM_ADCSequenceStepConfigure(ADC0_BASE, 2, 0, ADC_CTL_TS);
          ROM_ADCSequenceStepConfigure(ADC0_BASE, 2, 1, ADC_CTL_TS);
ROM_ADCSequenceStepConfigure(ADC0_BASE, 2, 2, ADC_CTL_TS);





          // The last must sample the temp and configure the interrupt flag to be
set when sample is done. Tell ADC logic that this is the last conversion on seq 1

ROM_ADCSequenceStepConfigure(ADC0_BASE,2,3,ADC_CTL_TS|ADC_CTL_IE|ADC_CTL_END);
          // Enable the Sequencer 1 adc
          ROM_ADCSequenceEnable(ADC0_BASE, 2);

          while(1)
              {// Clear the ADC interrup status flag
                       ROM_ADCIntClear(ADC0_BASE, 2);
                      // Trigger ADC conversion with software
                      ROM_ADCProcessorTrigger(ADC0_BASE, 2);


                     // waith for the conversion to complete
while(!ROM_ADCIntStatus(ADC0_BASE, 2, false))
                     {
                     }

          // we can read the ADC value from the ADC sample sequencer 1 FIFO
ROM_ADCSequenceDataGet(ADC0_BASE, 2, ui32ADC0Value);
                     // calculate the average of the temperature sensor data
ui32TempAvg = (ui32ADC0Value[0] + ui32ADC0Value[1] + ui32ADC0Value[2] +
ui32ADC0Value[3] + 2)/4;                         // calculate celsius value
                     ui32TempValueC = (1475 - ((2475 * ui32TempAvg)) / 4096)/10;
                     // calculate farenheit value
                     ui32TempValueF = ((ui32TempValueC * 9) + 160) / 5;
```

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

```
    //
    // Initialize the data to send.
    //
    tempval = ui32TempValueF;   //
pui32DataTx[1] = ui32TempValueF;
// pui32DataTx[2] =ui32TempValueF;

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

    //
    // Send 3 bytes of data.
    //
    for(ui32Index = 0; ui32Index < NUM_SSI_DATA; ui32Index++)
    {
        //
        // Display the data that SSI is transferring.
        //
        UARTprintf("'%c' ", tempval);

        //
        // 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, tempval);
    }

    //
    // 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 < 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, tempval);

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

         //
         // Display the data that SSI0 received.
         //
         UARTprintf("'%c' ", tempval);
     }
    }
    //
    // Return no errors
//      return(0);
}}
```

Task 01:

Youtube Link:
https://www.youtube.com/watch?v=YOzWPmajGds