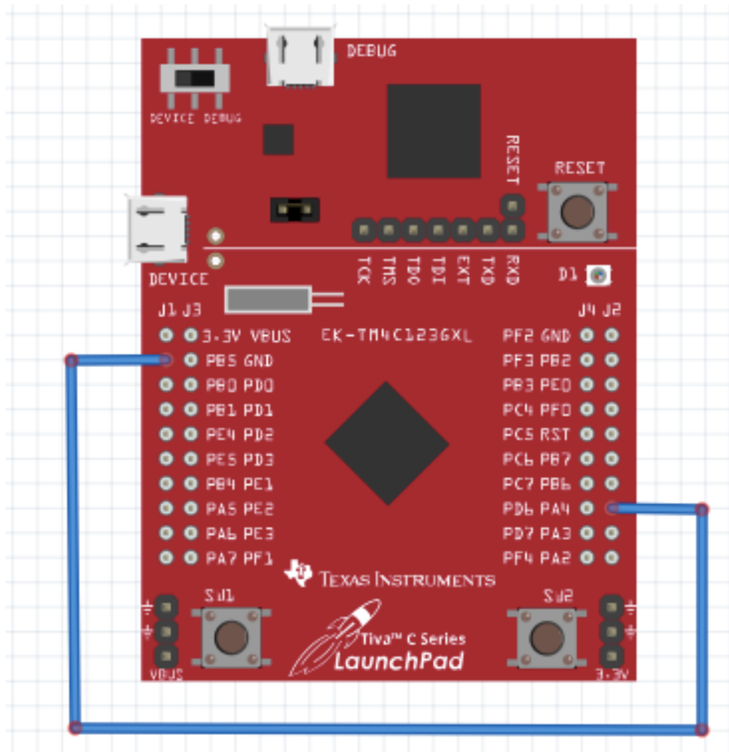
Lab 8: Synchronous Serial Interface (SPI)

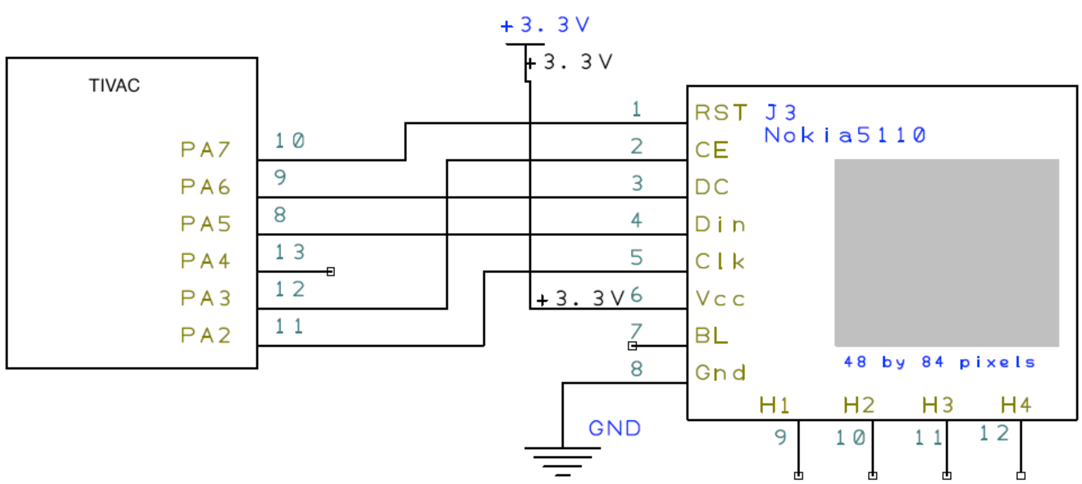
Objective

In this lab you will use the SSI Peripheral to do a Loopback Test and interface a SPI enabled Nokia 5110 LCD display.

TIVAC – SPI LOOPBACK TEST WIRING



TIVAC – NOKIA 5110 LCD WIRING



Procedure

Hardware

48X84 pixels matrix LCD controller/driver PCD8544

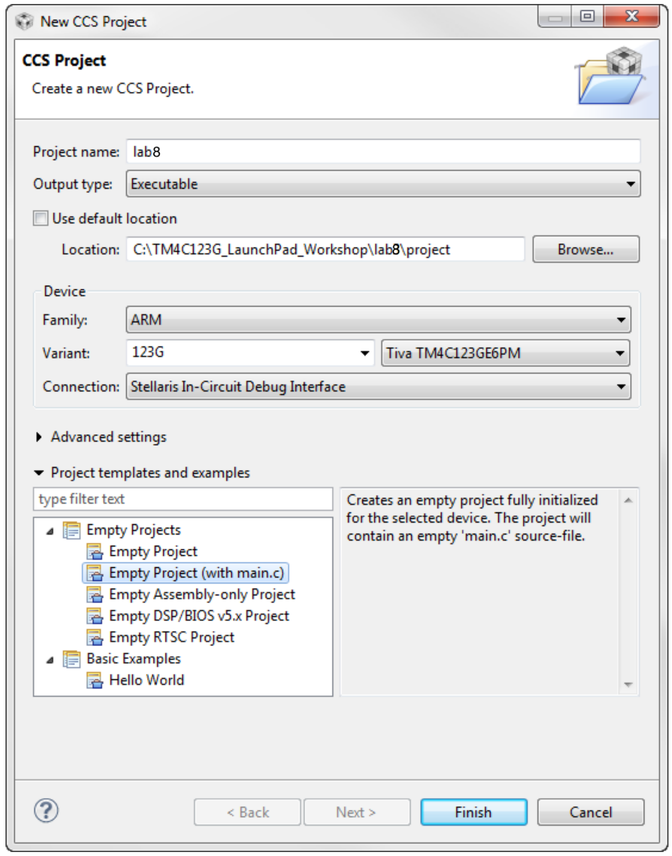
The PCD8544 is a low power CMOS LCD controller/driver, designed to drive a graphic display of 48 rows and84 columns. All necessary functions for the display are provided in a single chip, including on-chip generation of LCD supply and bias voltages, resulting in a minimum of external components and low power consumption. The PCD8544 interfaces to microcontrollers through a serial bus interface.

- Single chip LCD controller/driver
- 48 row, 84 column outputs
- Display data RAM 48 ´ 84 bits
- On-chip:
  - Generation of LCD supply voltage (external supply also possible)
  - Generation of intermediate LCD bias voltages
  - Oscillator requires no external components (external clock also possible).
- External RES (reset) input pin
- Serial interface maximum 4.0 Mbits/s
- CMOS compatible inputs
- Mux rate: 48
- Logic supply voltage range VDD to VSS: 2.7 to 3.3 V

Procedure

Create lab8 Project

1. ► Maximize Code Composer. On the CCS menu bar select File → New → CCS Project. Make the selections shown below. Make sure to uncheck the “Use default location” checkbox and select the correct path to the project folder as shown. In the variant box, just type “123G” to narrow the results in the right-hand box. In the Project templates and examples window, select Empty Project (with spi_main.c – also refered to main.c). Click Finish.



When the wizard completes, click the ▷ next to lab8 in the Project Explorer pane to expand the project. Note that Code Composer has automatically added a mostly empty `main.c` file to your project.

Note: We placed a file called `main.txt` in the `lab8/project` folder which contains the final code for the lab. If you run into trouble, you can refer to this file.

1. ► Expand the project and open `main.c` for editing. Place the following includes at the top of the file:

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

We're going to need all the regular include files along with the ones that give us access to the SSI peripheral.

2. ► Leave a line for spacing and add the template for main() below:

```
int main(void)
```

```
{
}
```

3. ► Insert the next line as the first ones before the `main`(). We'll need these variables for temporary data and index purposes.

```
#define NUM_SSI_DATA            3
```

4. ► Leave a line for spacing and set the clock to 50MHz as we've done before:

```
   SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC |
                        SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);
```

5. ► Space down a line and add the next two lines. Since SSI0 is on GPIO port A, we'll need to enable both peripherals:

```
// The SSI0 peripheral and port A must be enabled for use.
// Enable the SSI0 peripheral
SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
// The SSI0 peripheral is on Port A and pins 2,3,4 and 5.
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
```

6. ► Space down a line and add the following four lines. These will configure the muxing and GPIO settings to bring the SSI functions out to the pins.

```
// This function/s configures the pin muxing on port A pins 2,3,4
and 5
    GPIOPinConfigure(GPIO_PA2_SSI0CLK);
    GPIOPinConfigure(GPIO_PA3_SSI0FSS);
    GPIOPinConfigure(GPIO_PA4_SSI0RX);
    GPIOPinConfigure(GPIO_PA5_SSI0TX);
    GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_5 | GPIO_PIN_4 |
                                    GPIO_PIN_3 |GPIO_PIN_2);
   GPIOPinWrite(GPIO_PORTA_BASE,GPIO_PIN_4,GPIO_PIN_4);
```

7. Next we need to configure the SPI port on SSI0 for the type of operation that we want. Given that there are two bits (SPH – clock polarity and SPO – idle state), there are four modes (0-3). ► Leave a line for spacing and add the next two lines after the last. Then double-click on `SSI_FRF_MOTO_MODE_0` and press F3 to see all four definitions in `ssi.h`:

```
// Configure and enable the SSI port for SPI master mode.
    SSIClockSourceSet(SSI0_BASE,SSI_CLOCK_SYSTEM);
    SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(),
              SSI_FRF_MOTO_MODE_0,SSI_MODE_MASTER, 1000000, 8);
```

The API specifies the SSI module, the clock source (this is hard wired), the mode, master or slave, the bit rate and the data width.

8. ► Enable SSI Module

```
SSIEnable(SSI0_BASE);
```

9. Initialize UART0 using the following

```
// 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, 16000000);
```

10. Display the setup on the console.

11. 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.

12. Initialize the data to send.

```
UARTprintf("SSI ->\n");
UARTprintf("  Mode: SPI\n");
UARTprintf("  Data: 8-bit\n\n");
```

13. Display indication that the SSI is transmitting data.

```
UARTprintf("Sent:\n  ");
```

14. Send 3 bytes of data.

```
for(ui32Index = 0; ui32Index < NUM_SSI_DATA; ui32Index++)
 {
   // Display the data that SSI is transferring.
   UARTprintf("'%c' ", pui32DataTx[ui32Index]);
   SSIDataPut(SSI0_BASE, pui32DataTx[ui32Index]);
 }
```

15. Wait until SSI0 is done transferring all the data in the transmit FIFO.

```
while(SSIBusy(SSI0_BASE))
 {
 }
```

16. Display indication that the SSI is receiving data.

```
UARTprintf("\nReceived:\n  ");
```

17. // Receive 3 bytes of data.

```
for(ui32Index = 0; ui32Index < NUM_SSI_DATA; ui32Index++)
    {
   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]);
     }
    }
```

18. ► Save your work.

Build and Load

19. ► Build and load the code. Make sure to connect the PINs If you have errors, compare your spi_`main.c` to the code below:

```
#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"

//! 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

//*****************************************************************************
//
// 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, 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)
{
    uint32_t pui32DataTx[NUM_SSI_DATA];
    uint32_t pui32DataRx[NUM_SSI_DATA];
    uint32_t ui32Index;
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_16MHZ);
    // 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.
SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_0,
                   SSI_MODE_MASTER, 1000000, 8);
// 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';
    // 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' ", pui32DataTx[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, pui32DataTx[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 < 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("'%c' ", pui32DataRx[ui32Index]);
     }
    }
    // Return no errors
    return(0);
}
```

Run and Test

20. ► Run the code by clicking the Resume button. You should see Accelerometer data in serial terminal.

21. When you're done, ► click the Terminate button to return to the CCS Edit perspective.
22. ► Right-click on lab8 in the Project Explorer pane and close the project.
23. ► Disconnect your LaunchPad board from the USB port.

Follow the submission guideline to be awarded points for this Lab.

Task 1: Modify the supplied code to transmit and receive the Internal Temperature and verify the results.

Task 2: Display the z-axis results in Nokia5110 GLCD. If task is not working, display the Lab 5 – Temperature on the LCD as: "Temperature: 72.92 F, 20.34 F". Update every sec. using the timer.

Follow the submission guideline to be awarded points for this Lab.

Submit the following for all Labs:

1. In the document, for each task submit the modified or included code (only) with highlights and justifications of the modifications. Also include the comments.
2. Create a Github repository with a random name (no CPE/403, Lastname, Firstname). Place all labs under the root folder TIVAC, sub-folder named LABXX, with one document and one video link file for each lab, place modified c files named as LabXX-TYY.c.
3. If multiple c files or other libraries are used, create a folder LabXX-TYY and place these files inside the folder.
4. The folder should have a) Word document (see template), b) source code file(s) with startup_ccs.c and other include files, c) text file with youtube video links (see template).