EEL4742C: Embedded Systems

Experiment Eleven:

SPI and LCD

Ryan Koons 11/14/2020

Introduction: Before interacting with the Boosterpack LCD, first SPI communication has to be set-up. SPI is not a standardized method so it has a lot of different possible configurations. For For example, SPI can be configured as full-duplex or half duplex. For this lab, the LCD is not transmitting any data, therefore half-duplex is selected. Like other protocols, this method is synchronous and involves a master propagating the clock signal for other devices. SPI typically is a 4-wire interface, but can also be three wires (depending on whether or not there is data coming in). The fourth wire is the chip select signal which is essentially how a device is able to communicate. If the master enables this bit then the device can communicate, otherwise it is a high impedance line. The chip select is necessary whenever there are multiple slave devices and the master needs to switch between which device it wants to communicate with. However, for one device this pin can always be low(running all the time), and this is the three wire implementation. SPI is built around the concept of shift registers and d-latches/flip-flops. The shift registers allow all of the data to be swapped, and the flip flop ensures the MSB of the data does not change during transmission. Lastly, SPI also has different mode based on the polarity and phase. Polarity changes the clock idle signal (low and high), and phase changes when to latch and when to communicate (based on leading and trailing edge). The second half of the lab will also dive into the API for using the LCD on the boosterpack, and how to create a context to display all sorts of things on the display. This will be covered in ore detail in 11.2.

11.1 SPI Config. And Welcome Screen: The first section of this lab requires populating two previously defined functions in order to configure the SPI communication and print a welcome screen on the LCD. First, the launchpad and Boosterpack documentation was observed to determine all of the necessary pin diversions in order to configure SPI. P1.4 needed to be diverted from GPIO to serial clock for SPI. This was done by modifying the corresponding "SEL" registers defined in the chip's data sheet. The guide instructed to clear BIT4 in P1SEL1 and set bit 4 in P1SEL0. For SIMO, similar instructions were found (except the GPIO pin this time was P1.6. The reset pin was determined to be P9.4 and needed to be assigned as an output in the P9DIR register. It also needed both BIT4's cleared in P9SEL1 and P9SEL0. The data and command pin (P2.3) needed to also be set as an output and have both BIT3's cleared in P2SEL1 and P2SEL0. The last step for the initial SPI function was to set the chip select as an output and clear BIT5 in P2SEL1 and P2SEL0. Again, all of these Pin diversions were determined from the chip data sheet.

The next function in this section dealt more directly with the SPI configuration for the LCD. The control/config. Registers for the SPI setup were found in the family User Guide around pg. 812. First, the eUSCI was placed in reset to setup config. "UCB0CTLW0 |= UCCKPH|UCMSB|UCMST|UCMODE_0|UCSYNC|UCSSEL_3;" was used to configure the first control register with the corresponding clock phase, transmit MSB first, set the microcontroller to be SPI master, select a 3-pin SPI setup, select synchronous mode, and select SMCLK. The polarity was also cleared cautiously to be thorough. Next the clock divider was set to 1 by

assigning a value of 0 to UCB0RW, and reset state was exited. The last step was to set the chip select (P2.5) and Data/command bit (P2.3) to 0 by clearing with the appropriate bitmasks.

Analysis: This configuration corresponds to a half-duplex mode; this is clear since the LCD is not outputting any data back to the microcontroller.

11.1 CCS Code:

```
void HAL_LCD_PortInit(void)
{
    ////////////////////////////////////
    // Configuring the SPI pins
    ////////////////////////////////////
    // Divert UCB0CLK/P1.4 pin to serial clock
    P1SEL1 &= ~BIT4;
    P1SEL0 |= BIT4;
    //LCDs??????

    // Divert UCB0SIMO/P1.6 pin to SIMO
    P1SEL1 &= ~BIT6;
    P1SEL0 |= BIT6;

    // OK to ignore UCB0STE/P1.5 since we'll connect the display's enable bit to low
(enabled all the time)
    // OK to ignore UCB0SOMI/P1.7 since the display doesn't give back any data

    ////////////////////////////////////////////
    // Configuring the display's other pins
    ////////////////////////////////////////////
    // Set reset pin as output
    P9DIR |= BIT4;  //set as output and clear the rest
    P9SEL1 &= ~BIT4;
    P9SEL0 &= ~BIT4;
    // Set the data/command pin as output
        P2DIR |= BIT3;  //set as output and clear the rest
        P2SEL1 &= ~BIT3;
        P2SEL0 &= ~BIT3;

    // Set the chip select pin as output
        P2DIR |= BIT5;  //set as output and clear the rest
        P2SEL1 &= ~BIT5;
        P2SEL0 &= ~BIT5;

    return;
}
```

```
void HAL_LCD_SpiInit(void)
{
    //////////////////////////
    // SPI configuration
    //////////////////////////

    // Put eUSCI in reset state while modifying the configuration
    UCB0CTLW0 |= UCSWRST;

        // Set clock phase to "capture on 1st edge, change on following edge"--set 1
        //x designates channel?
        // Set clock polarity to "inactive low"--low/0 by default
        // Set data order to "transmit MSB first"--set 1
        // Set MCU to "SPI master"--set 1
        // Set SPI to "3 pin SPI" (we won't use eUSCI's chip select)--select 0
        // Set module to synchronous mode
        // Set clock to SMCLK
    UCB0CTLW0 |= UCCKPH|UCMSB|UCMST|UCMODE_0|UCSYNC|UCSSEL_3;
    UCB0CTLW0 &= ~UCCKPL;//clear polarity just in case

    // Set clock divider to 1 (SMCLK is from DCO at 8 MHz; we'll run SPI at 8 MHz)
    UCB0BRW = 0; //set clock divider to 1

    // Exit the reset state at the end of the configuration
    UCB0CTLW0 &= ~UCSWRST;

    // Set CS' (chip select) bit to 0 (display always enabled)
    // Set DC' bit to 0 (assume data)
    //Formatting GPIO Pins
    P2OUT &= ~BIT5;  //set bit to 0
    P2OUT &= ~BIT3;  //set bit to 0

    return;
       }
```

11.2 LCD API: This section required creating two screens on the LCD using the API that can be toggled using the button on the MSP430 Launchpad. The majority of this section involveds consulting the TI API in order to take advantage of their pre-defined functions. The rest of this section required getting the “switching” of the two screens to reliably work. Additionally,

screen two was configured to display an 8-bit, 1 second timer with appropriate roll-back-to-zero behavior.

The program begins yy defining flag variables, a uint8_t variable or the counter, and a Graphics_Context g_sContext Graphic library for the API. This variable type is necessary and is usually passed by reference in order to interact with the API and display all kinds of images/objects on the display. These were all defined as global variables, so every function and ISR could interact with them. This was done since the majority of the display code will be handled by an ISR that triggers whenever the button is pressed. Since this method was selected, it is important to recall that the button is not debounced and therefore will occasionally registers a double input (switch between both screens rapidly). Sharp and quick button presses are recommended for the best operational experience.

Within the main, the LEDs and the button are configured. Here the backlight is also initiated. This was found to be P2.6, and the initiation involved setting the bit as an output and clearing the corresponding bits in the P2SEL1 and P2SEL0 registers. Lastly, the documentation (LCD, boosterpack, and chip data sheet) revealed that clearing the bit in the output register would result in the lowest brightness (OFF) and vice versa for the highest brightness. So, this bit was set to achieve the highest brightness. The next part of the main configured the timing and interrupts. Similar code from previous labs was used to accomplish this result. ACLK was configured to 32kHz, and Up mode was used (set to 32768-1) to achieve a perfect one second timer. Then interrupts were enabled so that an ISR can perform a specific task (update the counter) each second. Enabling the interrupts implies configuring CCIFG, CCIE, TAIFG, and enabling the global interrupt bit (TAIE was not enabled since no specific action needed to be taken during a rollback event).

The final part of the main involved configuring the SPI clock, and showing the default screen (screen 1) on the display. Screen 1 was an image of the UCF logo. This output was achieved by calling the LCD init, set orientation, init context functions and then creating a definition for the image “extern const tImage logo4BPP_UNCOMP;”. All of this was achieved by passing the global &g_sContext by reference. Additionally, the Graphics_drawImage function was used to print the graphic to the display. Then, an infinite while loop executed and the rest of the code was handled by ISR’s.

The first ISR involves button 1 and is used to toggle between screen 1 and screen 2. As soon as the ISR detects that button 1 was pressed, it first toggles the state flag (0->1) etc. The state flag is used to determine which screen should be showing on the display. At first state 0 (screen 1) is showing by default, so when the button is pressed now state 1 (screen 2) should be displayed. State 0 is identical to the screen inn the main code and just displays the UCF mage once more. State 1 declares a context and runs through the init. Steps before printing several required objects to the screen. First, it sets a new background and foreground color using predefines colors in the glib library. Then it clears the screen, and draws the “welcome to…” string on the display. The next thing to draw is an outline circle using the function Graphics_drawCircle and passing the context, x, y coordinates, and the radius of the circle. Then, the color was changed using the foreground function. Then a filled circle was printed, and once more the color was changed. Typically, the color was changed between each unique shape that was drawn on the display. For the filled and unfilled rectangles, a struct needed to be created that defined the x and y coordinates of the top left and bottom right corners of the

rectangle. Then this struct was passed by reference to the corresponding API function. Throughout, this function state was also redefined so that it wouldn't randomly change from other interrupts.

The last ISR involved operating/updating the counter on the display. This ISR is called every second. Another flag Done was used so that the counter instantaneously knows when it should and shouldn't be printing the counter value to the display; the counter should always be running but only print out on screen 2. To correctly display each digit of the counter, if statements were used. For example, if the counter is only one digit, then print two spaces and then the digit. The same routine follow for other digit sizes, this ensures that digits do not write on top of each other. The previous context, and sprint are used to achieve this. Lastly, Graphics_drawString Centered with a fixed width font is used to display the string. Other fonts will leave behind pixels that don't get erased correctly.

11.2 CCS Code:

```
// Code to print to the LCD pixel display on the Educational
BoosterPack

#include <msp430fr6989.h>
#include "Grlib/grlib/grlib.h"          // Graphics library (grlib)
#include "LcdDriver/lcd_driver.h"       // LCD driver
#include <stdio.h>
#include <stdint.h>                    //Uint library

#define redLED BIT0
#define greenLED BIT7
#define button BIT1
#define TACTL TA0CTL    //necessary for timer config
#define TACCR0 TA0CCR0
#define BUT1 BIT1 //P1.1

void convert_32khz();  //convert to 32kHz function

volatile uint8_t n=0;  // timer/counter variable
int state=0;  //current state for lcd
int done=0;   //second toggle-for correctly counting
Graphics_Context g_sContext;        // Declare a graphic library context

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;     // Stop the Watchdog timer
    PM5CTL0 &= ~LOCKLPM5;         // Disable GPIO power-on default high-impedance
mode

    P1DIR |= redLED;    P1OUT &= ~redLED;
    P9DIR |= greenLED;  P9OUT &= ~greenLED;
    P1DIR &= ~button; P1REN|=button; P1OUT|=button; // button, resistor, pullup
    P1IE |= BUT1; //1:enable interrupts
    P1IES |= BUT1; //1: interrupt on falling edge
    P1IFG &= ~BUT1; //0: Clear interrupt flags
```

```
//init backlight
            P2DIR |= BIT6;  //set as output and clear the rest
            P2SEL1 &= ~BIT6;
            P2SEL0 &= ~BIT6;
    //P2OUT &= ~BIT6; //select lowest brightness--OFF
    P2OUT |= BIT6; //select highest brightness--ON

    //Configure ACLK to 32kHz
    convert_32khz();
    //Configure Channel 0 for up mode with interrupt
    TACCR0= (32768-1); // 1 second period @ 32kHz
    TA0CCTL0 |= CCIE; //enable Channel 0 CCIE Bit
    TA0CCTL0 &= ~CCIFG; //clear channel 0 CCIFG Bit
    //Configure Timer_A: ACLK, Divide by 1, up Mode, clear TAR, enable interrupt for
rollback event (leaves TAIE=0)
    TACTL = TASSEL_1 | ID_0 | MC_1 | TACLR;
    TACTL &= ~TAIFG; //Clear flag at start

    //lastly enable GIE, avoid raising interrupts too soon
    _enable_interrupts();

    // Configure SMCLK to 8 MHz (used as SPI clock)
    CSCTL0 = CSKEY;                 // Unlock CS registers
    CSCTL3 &= ~(BIT4|BIT5|BIT6);    // DIVS=0
    CSCTL0_H = 0;                   // Relock the CS registers

    //set default state

////////////////////////////////////////////////////////////////////////////////
///////

      Crystalfontz128x128_Init();            // Initialize the display
      Crystalfontz128x128_SetOrientation(0); // Set the screen orientation
      Graphics_initContext(&g_sContext, &g_sCrystalfontz128x128); // Initialize the
context
      extern const tImage logo4BPP_UNCOMP;  //definition for image
//display  image
      Graphics_drawImage( &g_sContext, &logo4BPP_UNCOMP, 0,0);

while(1){} //infinite loop

}


#pragma vector = PORT1_VECTOR //Link the ISR to the Vector
__interrupt void P1_ISR()
{
if( (P1IFG&BUT1) !=0 ) {  //when button1 is pressed
    //toggle state flags
if( state ==0){ state=1;}




else if( state==1){ state=0;}
```

```
P1IFG &= ~BUT1; //clear flag (shared)
if( state==0  ) //first screen
    {
    done=0;  //secondtoggle, dont worry about counter

    P1OUT |= redLED;
    P9OUT &= ~greenLED;


////////////////////////////////////////////////////////////////////////////////
///////
        extern const tImage logo4BPP_UNCOMP;
//image stuff
       Graphics_drawImage( &g_sContext, &logo4BPP_UNCOMP, 0,0);
       state=0;

        return;
    }



else if( state==1)
    {
    done=1;

    P9OUT |= greenLED;
    P1OUT &= ~redLED;


////////////////////////////////////////////////////////////////////////////////
///////
                Graphics_Context g_sContext;        // Declare a graphic library
context
                Crystalfontz128x128_Init();         // Initialize the display

                // Set the screen orientation
                Crystalfontz128x128_SetOrientation(0);

                // Initialize the context
                Graphics_initContext(&g_sContext, &g_sCrystalfontz128x128);

                // Set background and foreground colors first before clearing the
screen
                      Graphics_setBackgroundColor(&g_sContext,
GRAPHICS_COLOR_TURQUOISE);
                      Graphics_setForegroundColor(&g_sContext,
GRAPHICS_COLOR_DARK_CYAN    );

                // Clear the screen
                Graphics_clearDisplay(&g_sContext);

////////////////////////////////////////////////////////////////////////////////
///////
                state=1;
```

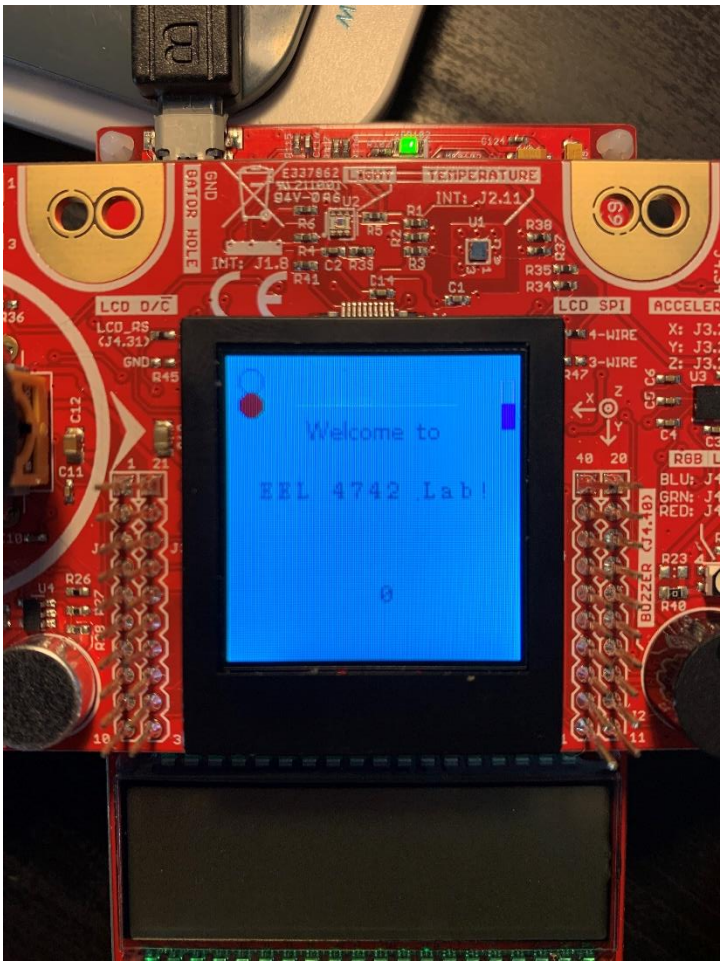
```
                char mystring[20];

                GrContextFontSet(&g_sContext, &g_sFontCmss12);
                Graphics_drawStringCentered(&g_sContext, "Welcome to",
AUTO_STRING_LENGTH, 64, 30, OPAQUE_TEXT);

                GrContextFontSet(&g_sContext, &g_sFontCmtt12);
                sprintf(mystring, "EEL 4742 Lab!");
                Graphics_drawStringCentered(&g_sContext, mystring, AUTO_STRING_LENGTH,
64, 55, OPAQUE_TEXT);


                //draw outline circle
                Graphics_drawCircle(&g_sContext, 10,10, 5); //x,y, radius

                Graphics_setForegroundColor(&g_sContext, GRAPHICS_COLOR_FIRE_BRICK
); //change foreground before filled circle

                //draw filled circle
                Graphics_fillCircle(&g_sContext, 10, 20, 5);
                Graphics_setForegroundColor(&g_sContext,  GRAPHICS_COLOR_PINK);
//change foreground before filled circle
                //draw unfilled rectangle
            struct Graphics_Rectangle R1 = {120, 10, 125, 20}; //struct rectangle for
API
            Graphics_drawRectangle( &g_sContext, &R1 );

            struct Graphics_Rectangle R2 = {120, 20, 125, 30}; //struct rectangle for
API
            //new color
            Graphics_setForegroundColor(&g_sContext,  GRAPHICS_COLOR_MEDIUM_ORCHID);
//change foreground before filled circle
            Graphics_fillRectangle(&g_sContext, &R2);

            Graphics_setForegroundColor(&g_sContext,  GRAPHICS_COLOR_LINEN); //change
foreground before filled circle
            Graphics_drawLineH(&g_sContext, 30, 100, 20);

            state=1;

            return;
    }
}

}



//**********ISR********//
//**********************//
#pragma vector = TIMER0_A0_VECTOR //Link the ISR to the Vector
__interrupt void T0A0_ISR()
{
//Toggle both LEDs
```

```
    if(  done==1){


        Graphics_Context g_sContext;        // Declare a graphic library context
        char mystring[20];
        Crystalfontz128x128_SetOrientation(0);
                        // Initialize the context
                        Graphics_initContext(&g_sContext, &g_sCrystalfontz128x128);


        if(n<10) {sprintf(mystring, "  %d", n);}

        else if(n<100) {sprintf(mystring, " %d", n);}

        else { sprintf(mystring, "%d", n);}

        Graphics_setBackgroundColor(&g_sContext, GRAPHICS_COLOR_TURQUOISE);
        Graphics_setForegroundColor(&g_sContext, GRAPHICS_COLOR_DARK_CYAN     );


        GrContextFontSet(&g_sContext, &g_sFontFixed6x8);


        Graphics_drawStringCentered(&g_sContext, mystring, AUTO_STRING_LENGTH, 64,
100, OPAQUE_TEXT);
        //done=0;

    }

    //print rectangle, print #
    n++;  //increment counter
P1OUT ^= BIT0;
P9OUT ^= BIT7;
//Hardware clears the flag (CCIFG in TA0CCTL0)
}


//Function for Converting ACLK to 32Khz
void convert_32khz()
{
//By default, ACLK runs on LFMODCLK at 5Mhz/128= 39 Khz
//Reroute pins to LFXIN/LFXOUT functionality
PJSEL1 &= ~BIT4;
PJSEL0 |= BIT4;
//wait until oscillator fault flags are cleared
CSCTL0 = CSKEY; //unlock CS registers
do {
CSCTL5 &= ~LFXTOFFG; //local fault flag
SFRIFG1 &= ~OFIFG; //global fault flag
} while((CSCTL5 & LFXTOFFG) != 0 );
CSCTL0_H = 0; //Lock CS registers
return;
```

}

11.2 Output:


Welcome to
EEL 4742 Lab!
255
LCD D/C
LCD_RS
(J4.31)
GND
LCD SPI
4-WIRE
3-WIRE
TEMPERATURE
INT: J2.11
INT: J1.8
GATOR HOLE
BUZZER (J4.10)
BLU: J4
GRN: J4
RED: J4



Student Q&A:

1) The SPI is implemented as half-duplex: the LCD is not outputting any data back to the microcontroller.
2) The SPI clock was configured to 8MHZ for the 11.2 program.
3) The max frequency supported is 16 MHz.
4) The driver is specific to each display; it has to be specifically configured in order to properly interface with the hardware, and create reset signals, and propagate command/data signals.
5) The API is not device specific, it is higher level and operates directly above the driver. So, as long as the driver properly interfaces with the hardware, the API is easily implementable.

Conclusion: After this lab, SPI is much more clearly understood, and now it is also a lot easier to interface with the LCD and print out all sorts of objects on the display. SPI is very versatile and complex, but this lab provided a great half-duplex implementation. This is very helpful for any scenario where the device does not need to transmit data; but, other scenarios would require a more elaborate setup so that the microcontroller can receive incoming data from the device. Since SPI isn't standardized, it also has a lot of mode for communication that involve different polarity, phase, latching, and transmission times. However, any SPI configuration can easily be achieved by consulting the family users guide and the chip's data sheet. The majority of SPI configuration involves locating the corresponding GPIO pins (diverting them), and setting the configuration registers. This lab also delved into how to use the TI API for the LCD. By creating a context variable, a lot of predefined functions can be taken advantage of to achieve various tasks. As long as the proper drivers are installed (device-specific), this API can be implemented on all sorts of displays.