CPE 325: Intro to Embedded Computer Systems

100

Lab09

Synchronous Serial Communications

Submitted by: Michael Agnew

Date of Experiment: 3/18/2024

Report Deadline: 3/19/2024

Demonstration Deadline: 3/25/2024

Introduction:

The main topics gone over throughout this lab are synchronous serial communication, SPI communication) and the use of a master-slave interface on the given circuit board. On the board that was used for this lab, there are two different chips, one of which being the MSP430FG4618, and the other the MSP430F2013. The 2013 is subordinate to the 4618 and acts as a sort of subroutine to it. The main idea behind this lab is to have the master code on the 4816 act as a user interface with UART through MobaXTerm and pass the needed information to the slave code on the 2013 which controls the blink rate of an LED.

Theory:

SPI Vs UART: SPI communication and UART communication share some similarities but also some differences. Some of the similarities are that UART and SPI both transmit the required data sequentially, or in other words, one singular bit at a time. Also, UART and SPI are very often used for embedded systems such as the one used for this lab. As for differences, UART communication is made for making asynchronous circuits with different clock speeds able to communicate with one another. Synchronous communication however, which SPI uses, is used for processors that share the same or similar clock on the same device. In this SPI communication, it makes use of a serial connection where data is moved and intercepted by several devices by means of a shared clock. Another difference between UART and SPI is that UART communication typically has two devices that work together semi-equally. Whereas in SPI communication, there is typically a master-slave dynamic between the two systems.

DMA Controller: DMA stands for “direct memory access”. This is an extremely useful controller and provides many benefits to one trying to use it. One of the biggest advantages of using DMA is that it acts independently of the CPU. Normally, the CPU is in charge of managing data transfer and assignment between UART, SPI communication or so on. But, in the case of DMA, this controller can offload the workload from the processor and free up memory for the CPU to focus on other tasks. This can boost the performance of the CPU and is extremely useful for embedded systems. Another advantage of this is that the use of DMA can save power by offloading work from the processor and allow the processor to go into low power mode. This can potentially save a lot of time, energy and money for whoever is trying to make this program. As opposed to programmed input output, DMA interacts directly between whatever peripherals are being used and the memory in the device. This is better than regular input/output because the CPU is free with DMA and can operate separately which boosts performance greatly.

Problem 01:

The first and only problem in this lab was to configure a master and slave code with SPI communication and UART implementation in order to blink an LED at a certain rate depending on user input through UART. To start the program, the code is initialized with a beacon (LED) pause rate of 50. What this means is that the beacon will be on for one time slot (32 ms) and then it will be turned off on the next time slot, before remaining off for *P* times slots, with *P* being the number of pauses that is input by the user.

Master: The master code for this project is essentially just a user interface. It outputs the program’s prompt for the amount of pause, and informs the user if the pause rate chosen is not within the proper bounds of 1-100. Then, it also takes in ? and - as input, if ? is selected, then it tells the user what the current pause rate of the beacon is, and if **-** is input, then the beacon is turned off until a different pause rate is entered. A flowchart was required to be made for this program which is shown below:

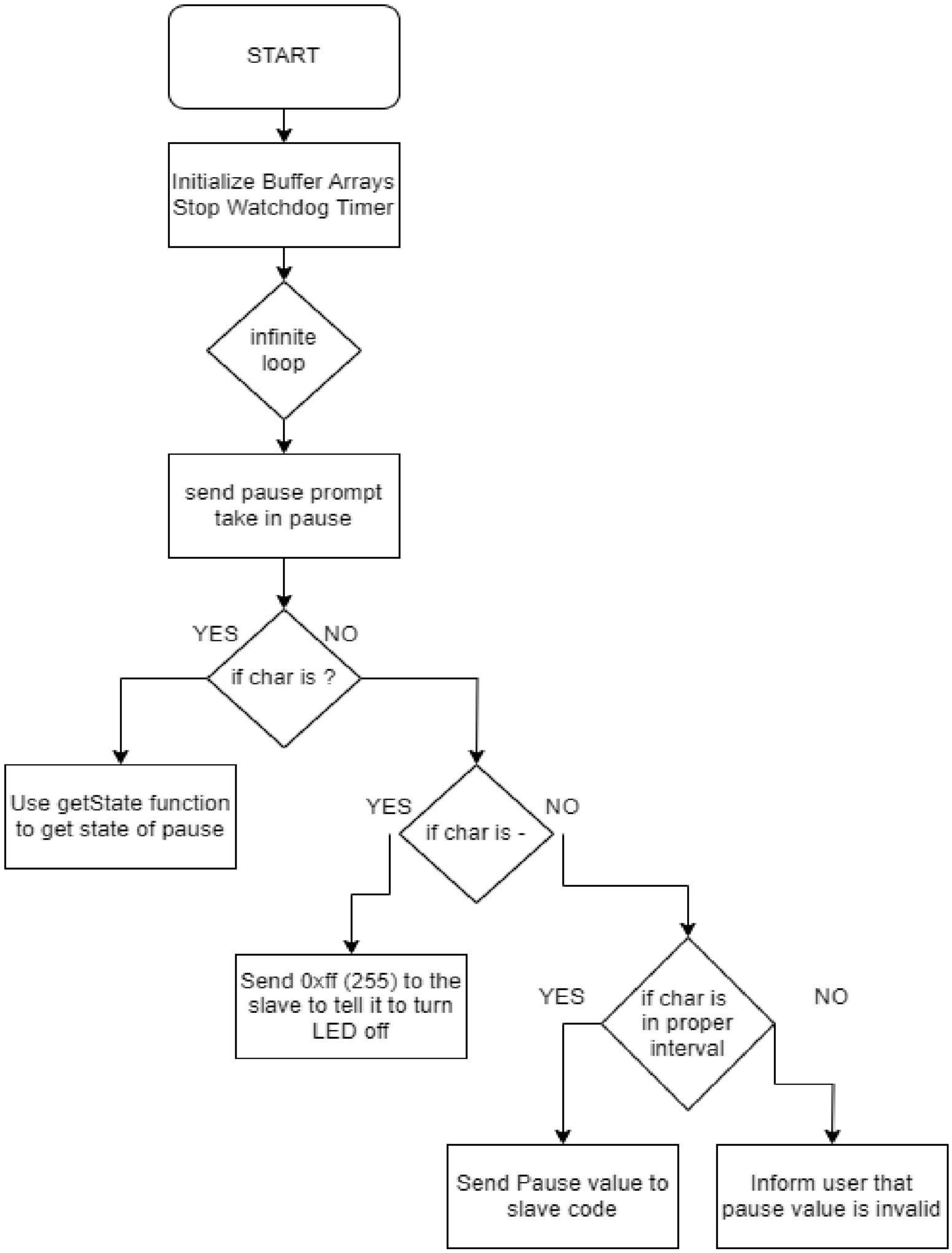

Figure 1. Flowchart for Master Code.

Slave: The role of the slave code is to take in the pause rate from the master and use it to control the blink rate of the beacon. If the slave code receives 255 from the master, then that is because - was input, in which case the slave code turns off the beacon. Whereas, if the proper pause in the given interval is reached, then it enters into the watchdog ISR, which occurs every 32 ms, and blinks the LED at the proper rate. A flowchart was required for this as well which is shown here:

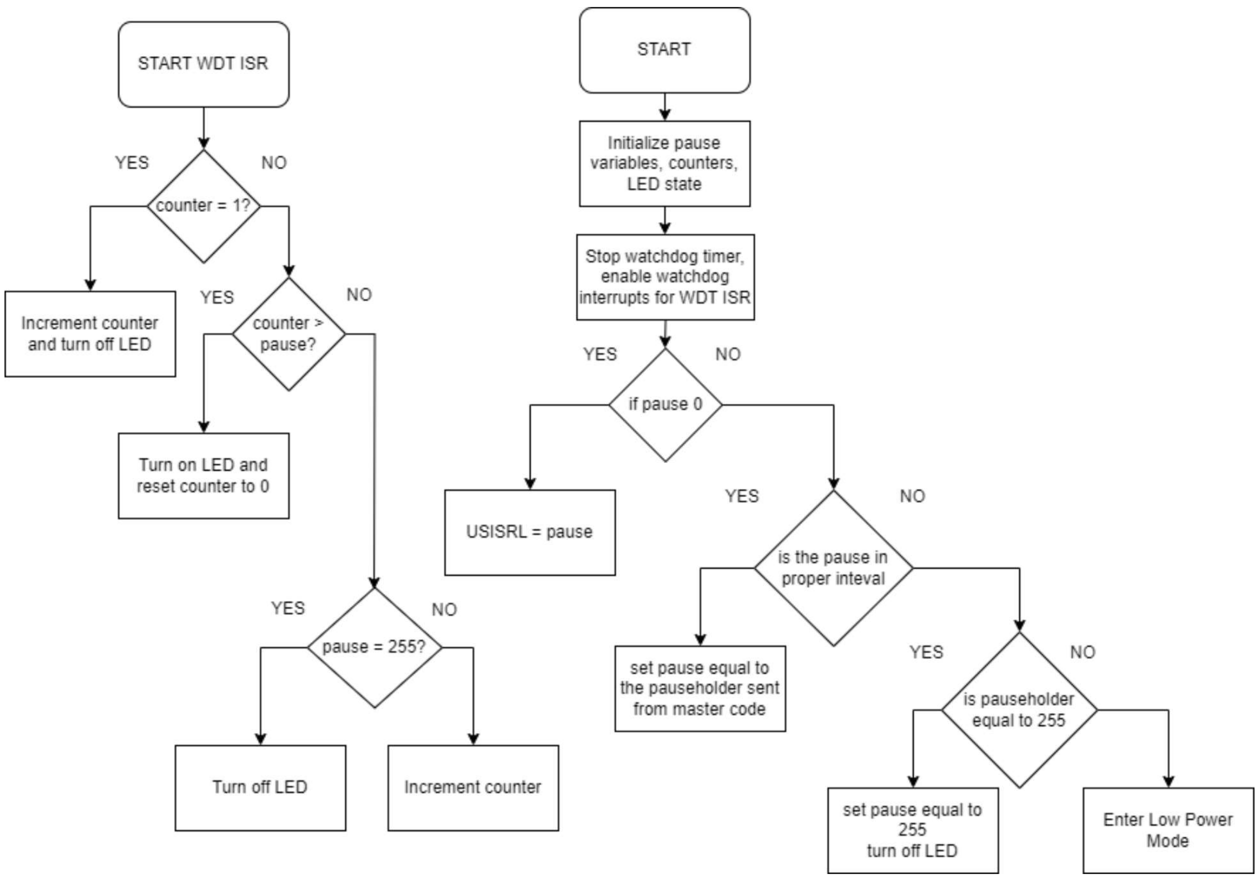

Figure 2. Flowchart for Slave Code.

Output: Another requirement for the completion of this lab was for the output to be screenshot and documented. The following is the appropriate output for the program:

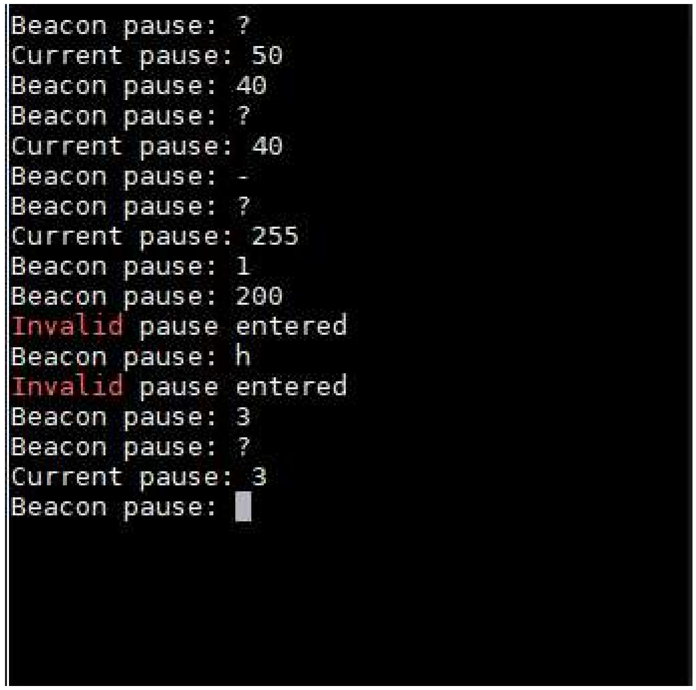

Figure 3. UART Terminal Output for Program.

Conclusion:

Fortunately, there were no issues encountered over the course of this lab. There were various things that were learned such as SPI communication, the similarities and differences between it and UART connection, and the way that the master-slave interface works on the black circuit board. In this lab, the means of incorporating master-slave SPI communication was done so and the appropriate output and LED blinking process was completed in the proper way. This lab is very useful for anyone wanting to learn how embedded systems interact with each other through serial SPI communication and interfacing with user input through UART.

Appendix:

Master:

```
/*----------------------------------------------------------------------------
--------
 * File:        Master.c
 * Function:    Implements a serial user interface with UART
communication. Also uses SPI communication with a slave chip and informs
 *              it of what rate to blink an LED.
 * Description: This code performs LED blinking at a rate given by the
user.
 * Input:       User input through serial communication
 * Output:      LEDs on the circuit board.
 * Author(s):   Michael Agnew, ma0133@uah.edu
 * Date:        March 18th, 2024

*----------------------------------------------------------------------------
-----*/

#include <msp430.h>
#include <stddef.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>


char BeaconArray[] = "Beacon pause: "; // output the prompt for beacon
pause rate
char PauseBuffArray[5]; // make an array to store the pausey boy
char PauseHolder; // hold the pause close hold it to your heart

void main(void)
{

    WDTCTL = WDTPW + WDTHOLD;     // Stop watchdog timer
    unsigned char pause;    // make a pause char

    UART_setup(); // call uart setup function
    SPI_setup(); // call spi setup function

    while(1) // infinite loop
    {
        UART_putStr(BeaconArray); // Send prompt
        UART_getStr(PauseBuffArray,5); // get the value of the pause
with a limit of 5 digits
```

```
        if (!(strcmp(PauseBuffArray,"?"))) // if the prompt is a
question mark then we get the state
        {
            SPI_getState(); // call get state function so that we can
get the state lol
        }
        else if (!(strcmp(PauseBuffArray,"-"))) // if minus then we need
to return 255 to tell the subroutine to turn off the LED
        {
            pause = 0xFF;  // set pause equal to 255
            SPI_putChar(pause); // send pause to the subroutine
        }
        else
        {
            pause = (unsigned char)atoi(PauseBuffArray); // convert
pause from character to integer
            if ((pause>0) && (pause<101)) // check the range of the
pause that was given
            {
                SPI_setState(pause); // if it is in the correct range
then send it to the subroutine
            }
            else
            {
                UART_putStr("\r\nInvalid pause entered"); // if it
outside of the correct bounds then inform the user that their decision
sucks
            }
        }
        UART_putStr("\r\n"); // make a newline afterwards
    }
}

/* getChar function definition to take in chars from spi communication
*/
char SPI_getChar(void)
{
    while(!(IFG2 & UCB0RXIFG)); // Wait until character is ready to be
received
    IFG2 &= ~UCB0RXIFG;
    UCB0TXBUF = 0x80;     // Dummy write to start SPI
    while (!(IFG2 & UCB0RXIFG));   // USCI_B0 TX buffer ready?
    return UCB0RXBUF;
}

/* getState function definition (for getting the current pause) */
void SPI_getState(void)
{
    unsigned int k = 0;
    char pauseStr[20];
    SPI_putChar(0x00);
    for (k=0;k < 1000; k++); // Give slave time to load transmit
```

```
register
    PauseHolder = SPI_getChar();
    sprintf(pauseStr,"\r\nCurrent pause: %u",PauseHolder);
    UART_putStr(pauseStr);
}

/* putChar function definition, waits for character to transmit and then
puts the new one into the tx buffer */
void SPI_putChar(char yay)
{
    while (!(IFG2 & UCB0TXIFG));  // Wait for previous character to
transmit
    UCB0TXBUF = yay;                // Put character into tx buffer
}

/* setState function that literally just sends the pause value to the
subroutine */
void SPI_setState(char state)
{
    SPI_putChar(state);
}

/* spi setup function that sets up the spi communication */
void SPI_setup(void)
{
    UCB0CTL0 = UCMSB + UCMST + UCSYNC;// Sync. mode, 3-pin SPI, Master
mode, 8-bit data
    UCB0CTL1 = UCSSEL_2 + UCSWRST; // SMCLK and Software reset
    UCB0BR0 = 0x02;                // Data rate = SMCLK/2 ~= 500kHz
    UCB0BR1 = 0x00;
    P3SEL |= BIT1 + BIT2 + BIT3;   // P3.1,P3.2,P3.3 option select
    UCB0CTL1 &= ~UCSWRST;          // **Initialize USCI state machine**
}

/* getChar function that waits for things to be ready and then sends
character */
char UART_getChar(void)
{
    while (!(IFG2 & UCA0RXIFG)); // Wait until a character is ready to
be read from Rx buffer
    IFG2 &= ~UCA0RXIFG;
    return UCA0RXBUF;
}

/* getStr function that gets the string from UART and puts it into array
*/
void UART_getStr(char* ReceiveArray, int limit)
{
    char yay;
    unsigned int h = 0;

    yay = UART_getChar();
    while ((yay != '\r') & (h < limit-1))
```

```
    {
        ReceiveArray[h++] = yay; // Store received character in receive
buffer
        UART_putChar(yay); // Echo character back
        yay = UART_getChar(); // Get next character
    }
    ReceiveArray[h] = (char)0x00; // Terminate string with null
character
}

void UART_putChar(char yay)
{
    while (!(IFG2 & UCA0TXIFG));  // Wait for previous character to
transmit
    UCA0TXBUF = yay;                // Put character into tx buffer
}

void UART_putStr(char* message)
{
    int u;
    for(u = 0; message[u] != 0; u++)
    {
        UART_putChar(message[u]);
    }
}

void UART_setup(void)
{
    P2SEL |= BIT4 + BIT5;          // Set UC0TXD and UC0RXD to transmit
and receive data
    UCA0CTL1 |= UCSWRST;           // Software reset
    UCA0CTL0 = 0;                  // USCI_A0 control register
    UCA0CTL1 |= UCSSEL_2;          // Clock source SMCLK - 1048576 Hz
    UCA0BR0 = 54;                  // Baud rate - 1048576 Hz / 19200
    UCA0BR1 = 0;
    UCA0MCTL = 0x0A;               // remainder 10
    UCA0CTL1 &= ~UCSWRST;          // Clear software reset
}


/* FUNCTION PROTOTYPES */

void SPI_setup(void);

char SPI_getChar(void);

void SPI_putChar(char);

void SPI_getState(void);

void SPI_setState(char);

char UART_getChar(void);
```

```
void UART_getStr(char*, int);

void UART_putChar(char);

void UART_putStr(char*);

void UART_setup(void);
```

Slave:

```
/*--------------------------------------------------------------------------
--------
 * File:         Slave.c
 * Function:     Implements LED blinking based on parameters passed from
the master code.
 * Description: This code performs LED blinking at a rate given by the
user.
 * Input:        None
 * Output:       LEDs on the circuit board.
 * Author(s):    Michael Agnew, ma0133@uah.edu
 * Date:         March 18th, 2024

*--------------------------------------------------------------------------
-----*/

#include "msp430x20x3.h"

#define LED_ON()       P1OUT |= 0x01
#define LED_OFF()      P1OUT &= ~0x01

unsigned int juan;

unsigned char pause = 50;

unsigned char PauseHolder;

void main(void)
{
    sysInit();        // System Initialize function call (sets up the
system for our purposes)
    SPI_setup();      // SPI_Setup function call (sets up spi
communication)
    initLED();        // initialize the LED with the correct parameters
and initialize it to be on
```

```
    while(1)
    {

        if (PauseHolder == 0)
        {
            USISRL = pause;
        }
        else if (PauseHolder <= 100)
        {
            pause = PauseHolder;
        }
        else if (PauseHolder == 0xff)
        {
            pause = 0xff;
            LED_OFF();
        }
        _BIS_SR(LPM0_bits + GIE); // LPM0 w/ Interrupt
    }
}

void initLED(void)
{
    P1DIR |= BIT0;                  // P1.0 as output - LED3
    LED_ON();
}

void SPI_setup(void)
{
    USICTL0 |= USISWRST;            // Set UCSWRST -- needed for
re-configuration process
    USICTL0 |= USIPE5 + USIPE6 + USIPE7 + USIOE; // SCLK-SDO-SDI port
enable,MSB first
    USICTL1 = USIIE;                // USI Counter Interrupt enable
    USICTL0 &= ~USISWRST;           // **Initialize USCI state machine**
    USICNT = 8;                     // Load bit counter, clears IFG
}

void sysInit(void) // system initialization (from demo code)
{
    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
    WDTCTL = WDT_MDLY_32;     // 32 ms interval mode
    IE1 |= WDTIE;                // enable watchdog interrupts
    BCSCTL1 = CALBC1_1MHZ;          // Set DCO
    DCOCTL = CALDCO_1MHZ;
}



#pragma vector = USI_VECTOR // this is also from demo code
__interrupt void USI_ISR(void)
{
    PauseHolder = USISRL;             // Read new command
    USICNT = 8;                     // Load bit counter for next TX
```

```
    _BIC_SR_IRQ(LPM0_bits);       // Exit from LPM0 on RETI
}


#pragma vector = WDT_VECTOR
__interrupt void WDT_ISR(void)
{
    if (juan == 1) // juan
    {
        LED_OFF(); // turn off led
        juan++;
    }
    else if (juan > pause)
    {
        LED_ON(); // turn on led
        juan = 0; // reset counter
    }
    else if (pause == 0xff)
    {
        LED_OFF();
    }
    else
    {
        juan++; // increment counter

    }
}

/* FUNCTION PROTOTYPES */

void initLED(void);

void SPI_setup(void);

void sysInit(void);
```