

CPE 325: Intro to Embedded Computer System

Lab09

Synchronous Serial Communication

Submitted by: Gianna Foti and Austin Mullins

Date of Experiment: 4 November 2024

Report Deadline: 7 November 2024

Lab Section: CPE353-02

Demonstration Deadline: 7 November 2024

Introduction

In this lab, we focused on establishing SPI communication between two MSP430F5529LP Experimenter Boards by modifying only the master program while keeping the provided slave demo code unchanged. This approach allowed us to concentrate on enhancing the master's functionality to communicate with the user via UART for dynamic control over communication parameters, including frequency and cycle count, while ensuring seamless interaction with the pre-existing slave setup. By loading the unmodified slave code and making targeted edits to the master, we enabled the master to interpret user inputs through UART, allowing real-time adjustments to LED blink frequency and cycle settings. This setup provided insight into master-slave coordination, where the master could control the communication flow without requiring changes on the slave side, illustrating efficient SPI and UART integration in embedded systems.

Theory Topics

1. SPI vs UART
 - a. Serial communication protocols like SPI and UART facilitate data exchange between devices, but they function in distinct ways. SPI (Serial Peripheral Interface) uses synchronous communication, where all connected devices rely on a shared clock signal, making it well-suited for communication between components located close to each other on the same circuit board. In contrast, UART (Universal Asynchronous Receiver-Transmitter) operates asynchronously, with each device using its own internal clock. For accurate data transfer, both devices must be set to the same baud rate, or communication speed. Despite this fundamental difference, both SPI and UART allow full-duplex communication, enabling simultaneous data transmission and reception
2. Serial Communication Types
 - a. The MSP430 microcontroller is equipped with four distinct serial communication protocols: SPI (Serial Peripheral Interface), UART (Universal Asynchronous

Receiver-Transmitter), I²C (Inter-Integrated Circuit), and Infrared communication. Our coursework primarily focused on UART and SPI. The I²C protocol operates using a shared bus system for data transmission, while Infrared communication relies on electromagnetic waves. During our lab exercises, we worked with UART in Labs 8 and 10 and explored SPI communication in Lab 9.

Part 1 Slave:

Program Description:

In this lab setup, the SPI Master and SPI Slave programs enable communication between two MSP430F5529LP Experimenter Boards, establishing a data exchange system with visual feedback through LEDs. The slave program was given to us as a demo and did not need modification.

Part 2 Master:

Program Description:

This program controls the blink pattern of an LED on the MSP430 based on user-defined frequency and cycle settings entered via UART. The program features an interactive prompt over UART at 115,200 baud, enabling users to specify the LED's blink frequency (1-10 Hz) and the number of cycles. Additionally, the program includes SPI communication with a slave device to verify data transmission, using LED feedback to indicate successful data exchange.

Process:

1. First, I configured the UART and SPI modules on the MSP430 to enable communication:
 - a. Using the UART_setup function, I set up the UART module at 115,200 baud with SMCLK as the clock source, enabling reliable data exchange with a terminal for user input.

- b. I then wrote `SPI_Master_UCB0_Setup` to configure the MSP430 as an SPI master. This setup used a 3-wire SPI mode with a clock polarity setting, allowing communication with a slave device on specific data lines.
2. Next, I implemented the user interaction for setting LED blink parameters:
 - a. In the `Prompt` function, I added UART prompts to guide the user through setting an LED blink frequency (1-10 Hz) and the number of cycles for blinking.
 - b. I created a loop to validate that the entered frequency stayed within the required range.
 - i. This approach ensures that only valid data is processed, reducing error potential during runtime.
3. Then, I calculated the delay needed for precise LED control:
 - a. Based on the frequency input, I calculated a delay in cycles, stored in the variable `i`, to control the LED's on/off timing accurately. This calculation used the formula $((1.0 / \text{NumFrequency}) / 2.0) * 10000$, which derived the half-period delay, ensuring the LED would blink according to the selected frequency.
4. Afterward, I integrated SPI data exchange within the LED blink loop:
 - a. Each time the LED blinked, the master sent `MST_Data` to the slave over SPI. Upon receiving data back from the slave, I checked if it matched the expected value (`SLV_Data`).
 - b. If correct, the program toggled LED1 as feedback; otherwise, it turned LED1 off.
 - i. This real-time SPI data verification confirmed proper communication with the slave.
5. I added a heartbeat indicator on LED2 to show program activity:
 - a. To demonstrate that the program was actively responding, I toggled LED2 as a "heartbeat" indicator within the main loop.
 - b. This visual feedback ensured the user that the system was functioning and processing commands as expected.

6. Finally, I tested and optimized the main program loop:
 - a. The main loop continuously calls Prompt for user input, allowing the user to adjust the frequency and cycle count without restarting the program.
 - b. Testing various frequency and cycle values helped confirm the program's stability and ensured it met the intended design for both UART communication and SPI feedback.

Flowchart Master:

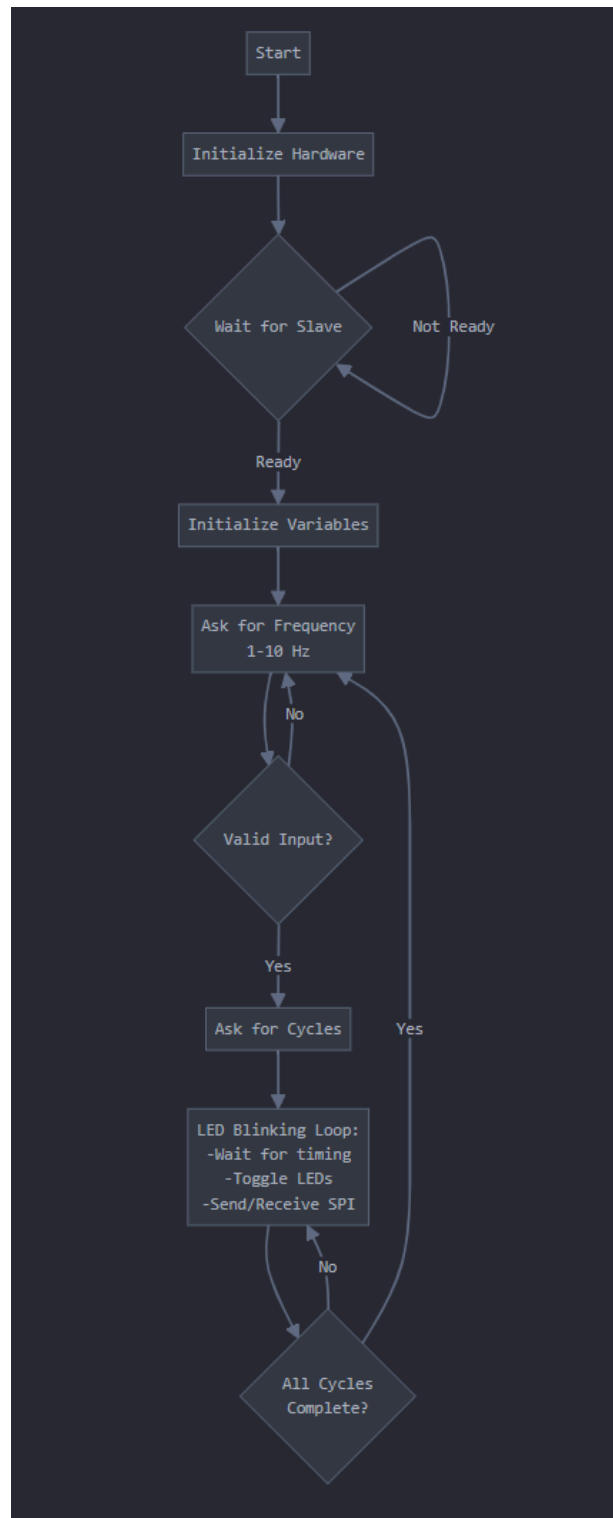
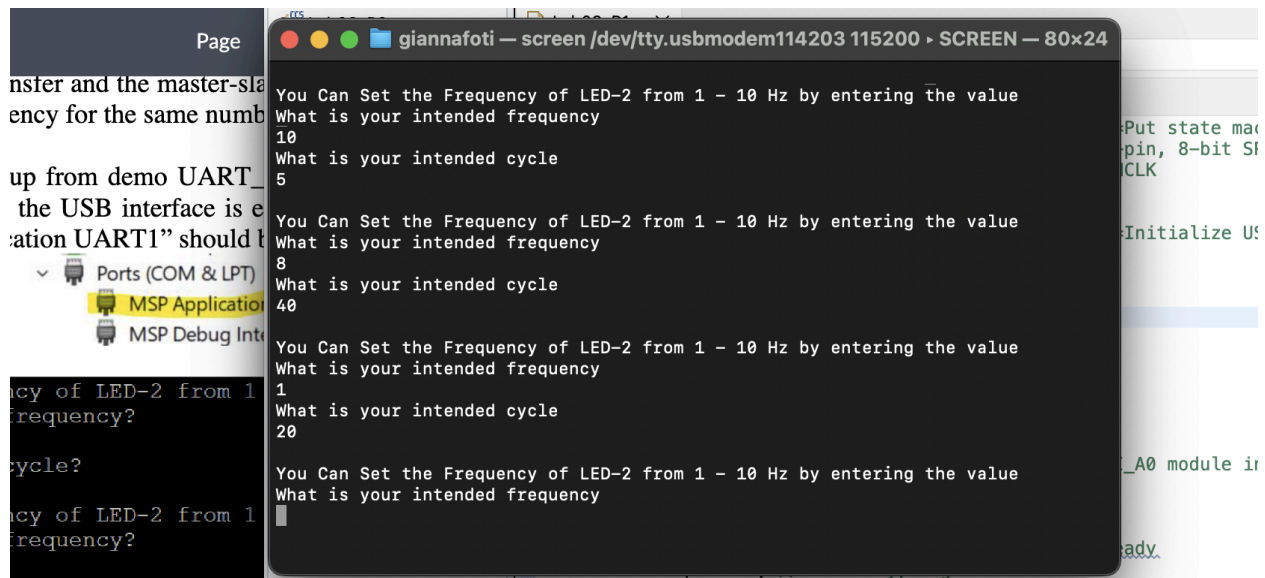


Figure 1: Program 2 Flowchart

Program Output:



The screenshot shows a terminal window titled "giannafoti — screen /dev/tty.usbmodem114203 115200 • SCREEN — 80x24". The terminal displays a series of prompts and user inputs for configuring LED-2. The prompts are: "You Can Set the Frequency of LED-2 from 1 - 10 Hz by entering the value", "What is your intended frequency?", and "What is your intended cycle?". The user inputs are: 10, 5, 8, 40, 1, and 20. The terminal also shows a list of ports on the left, including "Ports (COM & LPT)", "MSP Application", and "MSP Debug Interface".

```
Page
nster and the master-slave
ency for the same numb

up from demo UART_
the USB interface is e
ation UART1" should b
  Ports (COM & LPT)
  MSP Application
  MSP Debug Interface

ncy of LED-2 from 1
frequency?
cycle?
ncy of LED-2 from 1
frequency?

You Can Set the Frequency of LED-2 from 1 - 10 Hz by entering the value
What is your intended frequency
10
What is your intended cycle
5
You Can Set the Frequency of LED-2 from 1 - 10 Hz by entering the value
What is your intended frequency
8
What is your intended cycle
40
You Can Set the Frequency of LED-2 from 1 - 10 Hz by entering the value
What is your intended frequency
1
What is your intended cycle
20
You Can Set the Frequency of LED-2 from 1 - 10 Hz by entering the value
What is your intended frequency

```

Figure 2: Program 2 Output

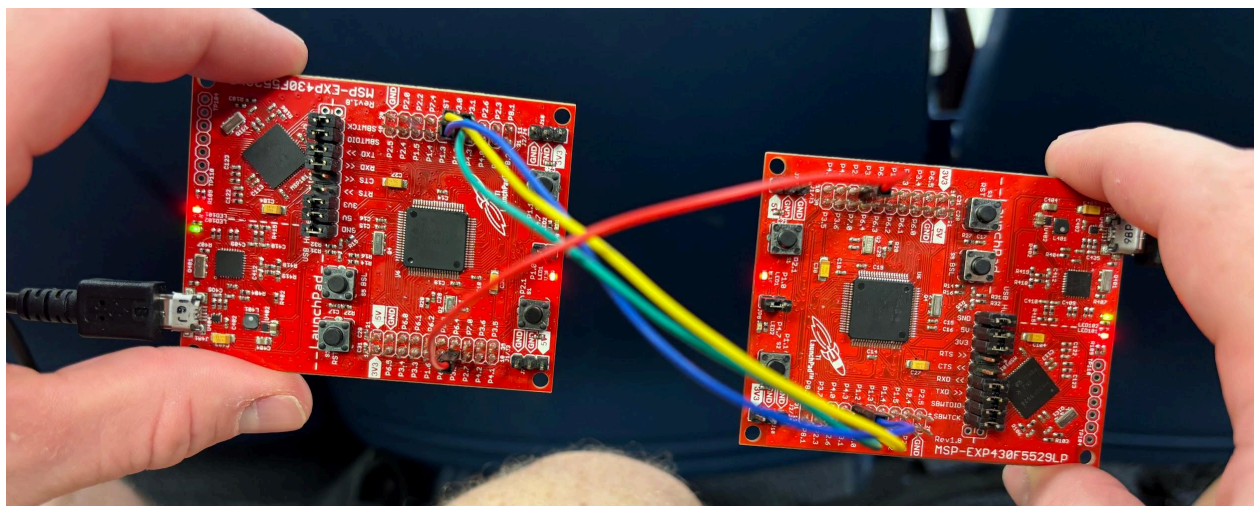


Figure 3: Connected Boards

Conclusion

In conclusion, we successfully established SPI communication between two MSP430F5529LP Experimenter Boards using the UCB0 module, with specific pins configured for SPI signals and a handshaking mechanism via P1.2. By physically connecting the boards and following the specified

initialization steps, we enabled a reliable data exchange where the master board sends data to the slave, and the slave echoes it back. Through programming both master and slave boards, we verified communication through the LED indicators. LED1 on the slave remained on when receiving and echoing data, while LED2 on both boards blinked at a rate of 0.5Hz, indicating data transmission and echo response. Additionally, we implemented user-controlled parameters for communication frequency (1-10 Hz) and cycle count, allowing flexible testing of data exchange rates. This experiment reinforced the fundamental concepts of SPI communication, synchronization, and handshaking between master and slave devices in embedded systems. It also demonstrated how UART can be integrated to allow real-time user interaction, making this setup adaptable for various real-world applications requiring precise communication and user control.

Appendix

Table 1: Program 1 Slave (note: same as demo)

```
#include <msp430.h>
/*
void SPI_Slave_UCA0_Setup(void) {
    P3SEL |= BIT3+BIT4; // P3.3,4 option select
    P2SEL |= BIT7; // P2.7 option select
    UCA0CTL1 |= UCSWRST; // **Put state machine in reset**
    UCA0CTL0 |= UCSYNC+UCCKPL+UCMSB; // 3-pin, 8-bit SPI slave,
    // Clock polarity high, MSB
    UCA0CTL1 &= ~UCSWRST; // **Initialize USCI state machine**
}
*/
void SPI_Slave_UCB0_Setup(void) {

    P3SEL |= BIT0+BIT1+BIT2; // P3.3,4 option select
    //P2SEL|= BIT7; // P2.7 option select
    UCB0CTL1 |= UCSWRST; // **Put state machine in reset**
    UCB0CTL0 |= UCSYNC+UCCKPL+UCMSB; // 3-pin, 8-bit SPI slave,
    // Clock polarity high, MSB
    UCB0CTL1 &= ~UCSWRST; // **Initialize USCI state machine**
}

int main(void) {

    WDTCTL = WDTPW+WDTHOLD;
    SPI_Slave_UCB0_Setup();
```



```

P1DIR |= BIT2 + BIT0; // Set P1.0 and P1.2 as outputs
P1OUT |= BIT2 + BIT0; // LED1 is on, P1.2 is set

__delay_cycles(100);
//P1OUT&= ~BIT2;
// LED is on, P1.2 is off

// P4.7 is heartbeat of the application (toggles on each received char)
P4DIR |= BIT7;
P4OUT = 0;

for(;;) {

    while(!(UCB0IFG&UCRXIFG));
    // wait for a new character
    while(!(UCB0IFG&UCTXIFG));
    // new character is received, is TXBUF ready?

    UCB0TXBUF = UCB0RXBUF; // echo character back if ready

    P4OUT ^= BIT7;
}
}

```

Table 2: Program 1 Master

```

/*-----
* File: Lab09_Master1.c
* Description: This program uses UART and SPI communication on the MSP430 to control an LED
*              based on user input for frequency and cycle count. It provides an interactive
*              prompt at 115,200 baud, allowing users to set the LED blink frequency (1-10 Hz)
*              and the number of cycles. The program also uses SPI to communicate with a slave
*              device for data verification, providing feedback through LEDs.
* Input: User-defined frequency and cycle count via UART
* Output: LED blink pattern, data verification feedback through LED status
* Author: Gianna Foti
*-----*/

#include <msp430.h>
unsigned char MST_Data, SLV_Data; // Variables for master and slave data in SPI
unsigned char temp; // Temporary variable
void UART_setup(); // UART setup function declaration
void UARTA1_putchar(char c); // Function to send a character via UART
char UART_getChar(); // Function to receive a character via UART
void PrintString(char * msg); // Function to print a string via UART
void UART_getLine(char* buf, int limit); // Function to get user input line via UART
void Prompt(); // Function to prompt user for frequency and cycle count

```

```

void SPI_Master_UCB0_Setup(); // SPI setup function declaration
char Frequency[10]; // Buffer for frequency input from user
char Cycle[10]; // Buffer for cycle count input from user
void SPI_Master_UCB0_Setup(void) {
    // Configure P3.0, P3.1, P3.2 as SPI pins (SIMO, SOMI, CLK)
    P3SEL |= BIT0 + BIT1 + BIT2;
    UCB0CTL1 |= UCSWRST; // Put state machine in reset during setup
    UCB0CTL0 |= UCMST + UCSYNC + UCCKPL + UCMSB; // Configure as 3-pin, 8-bit SPI master
    UCB0CTL1 |= UCSSEL_2; // Select SMCLK as clock source
    UCB0BR0 = 18; // Set baud rate divisor
    UCB0BR1 = 0;
    UCB0CTL1 &= ~UCSWRST; // Initialize USCI state machine
}

int main(void) {
    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
    P1OUT = 0; // Initialize output
    P1DIR |= BIT0; // Set P1.0 as output for LED1
    P4DIR |= BIT7; // Set P4.7 as output for LED2
    P4OUT &= ~BIT7; // Turn off LED2
    P1IN &= ~BIT2; // Initialize P1.2 input
    UART_setup(); // Set up UART communication
    SPI_Master_UCB0_Setup(); // Set up SPI communication
    // Wait for slave device to be ready
    while (!(P1IN & BIT2)); // Wait until P1.2 goes high (slave ready signal)
    P1OUT |= BIT0; // Light LED1 when slave is ready
    __delay_cycles(1000000); // Delay for visual indication
    P1OUT &= ~BIT0; // Turn off LED1
    MST_Data = 0x01; // Initialize master data to 1
    SLV_Data = 0x00; // Initialize expected slave data to 0
    for (;;) {
        Prompt(); // Continuously prompt user for input
    }
}

void Prompt() {
    PrintString("\r\nYou Can Set the Frequency of LED-2 from 1 - 10 Hz by entering the value\r\n");
    PrintString("What is your intended frequency\r\n");
    UART_getLine(Frequency, 10); // Get frequency input from user
    // Ensure frequency is within 1-10 Hz range
    while (atoi(Frequency) < 1 || atoi(Frequency) > 10) {
        PrintString("Error: You can set the Frequency of LED-2 from 1 - 10 Hz by entering the value\r\n");
        UART_getLine(Frequency, 10); // Re-prompt user for valid frequency
    }
    PrintString("What is your intended cycle\r\n");
    UART_getLine(Cycle, 10); // Get cycle count input from user
    int NumFrequency = atoi(Frequency); // Convert frequency input to integer
    int NumCycle = atoi(Cycle); // Convert cycle count input to integer
    double g = ((1.0 / NumFrequency) / 2.0) * 10000; // Calculate delay period based on frequency
    int h = 0; // Cycle counter
    int f = 0; // Inner loop counter
    // Loop to blink LED and perform data verification for the specified cycle count
    while (h < NumCycle * 2) { // Blink twice per cycle
        for (f = 0; f < (int)g; f++); // Delay loop based on frequency
        P1OUT ^= BIT0; // Toggle LED1
    }
}

```

```

// SPI communication with slave device
while (!(UCB0IFG & UCTXIFG)); // Wait for SPI TX buffer ready
UCB0TXBUF = MST_Data; // Send master data to slave
while (!(UCB0IFG & UCRXIFG)); // Wait for data back from slave
// Check if received data matches expected slave data
if (UCB0RXBUF == SLV_Data) {
    P1OUT |= BIT0; // Light LED1 if data matches
} else {
    P1OUT &= ~BIT0; // Turn off LED1 if data does not match
}
P4OUT ^= BIT7; // Toggle LED2 as a heartbeat indicator
// Update data values for next transmission
MST_Data = (MST_Data + 1) % 50;
SLV_Data = (SLV_Data + 1) % 50;
h++; // Increment cycle counter
}
}

void UART_setup(void) {
    // Configure UART communication on USCI_A1
    P4SEL |= BIT4 + BIT5; // Set P4.4 and P4.5 for UART
    UCA1CTL1 |= UCSWRST; // Put state machine in reset during setup
    UCA1CTL0 = 0; // Default UART configuration
    UCA1CTL1 |= UCSSEL_2; // Select SMCLK as UART clock source
    UCA1BR0 = 0x09; // Set baud rate to 115200 (SMCLK/115200)
    UCA1BR1 = 0x00;
    UCA1MCTL = 0x02; // Modulation settings
    UCA1CTL1 &= ~UCSWRST; // Initialize USCI state machine
}

void UARTA1_putchar(char c) {
    while (!(UCA1IFG & UCTXIFG)); // Wait until TX buffer is ready
    UCA1TXBUF = c; // Transmit character via UART
}

char UART_getChar() {
    while (!(UCA1IFG & UCRXIFG)); // Wait until RX buffer has data
    return UCA1RXBUF; // Return received character
}

void PrintString(char * msg) {
    // Print a string character-by-character via UART
    register int i = 0;
    for (i = 0; i < strlen(msg); i++) {
        UARTA1_putchar(msg[i]); // Send each character
    }
}

void UART_getLine(char* buf, int limit) {
    // Get a line of input from UART with character limit
    int i = 0;
    char ch;
    while (1) {
        ch = UART_getChar(); // Get character from UART
        if (ch == '\r' || ch == '\n') { // Enter key pressed
            buf[i] = '\0'; // Null-terminate string
            PrintString("\r\n"); // Print newline
            break;
        }
    }
}

```

```
} else if (ch == '\b' || ch == 127) { // Backspace pressed
    if (i > 0) {
        i--; // Move back buffer pointer
        PrintString("\b\b"); // Clear character on screen
    }
} else if (i < limit - 1) { // Normal character input
    buf[i++] = ch;
    UARTA1_putchar(ch); // Echo character back to UART
}
}
}
```