

SISTEMA SPI SIMPLE

Stellaris Lunchpad LM4F120

Descripción breve

Sistema simple de comunicación SPI, el cual está conformado por un maestro y un esclavo, donde un dato es enviado desde la pc hacia el maestro y el maestro envía un dato a través del bus spi el cual es retornado al ser recibido por el esclavo.

Juan Miguel Vargas Sánchez
jmvarsa@gmail.com

Comunicación SPI entre las tarjetas LM4F120 y MSP430

El bus SPI o “Serial Peripheral Interface” por sus siglas en ingles es un estándar de comunicación para el intercambio de información entre circuitos integrados en sistemas electrónicos principalmente. Un sistema de comunicación SPI está conformado por un maestro y un esclavo aunque también es posible agregar más de un esclavo.

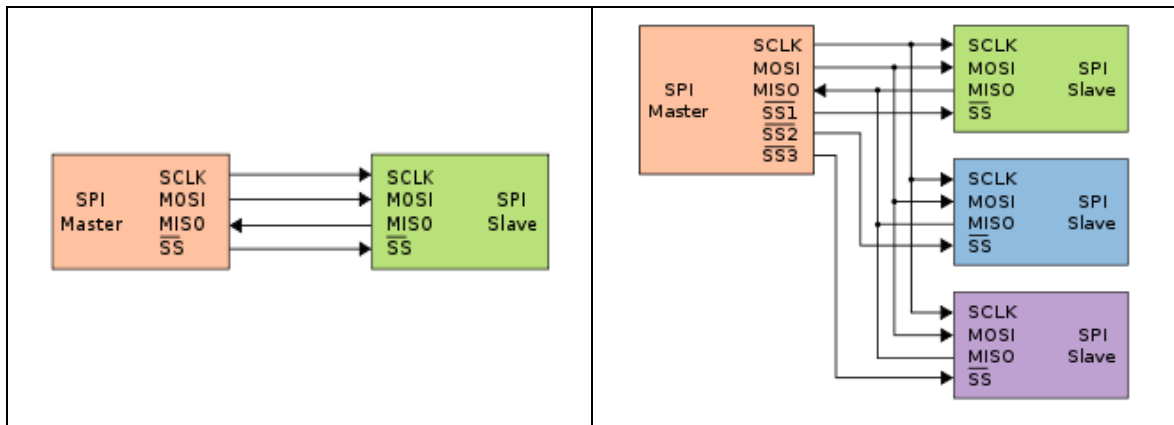


Figura 1. Bus SPI simple del lado izquierdo y bus SPI compuesto del lado derecho.

Para la comunicación maestro esclavo/s son necesarias cuatro líneas principalmente y una más para cuando se está trabajando con más de un esclavo las cuales se describen a continuación:

- Línea de reloj, el reloj es generado por el maestro y sirve para la sincronización entre este y el/los esclavo/s.
- Línea MOSI (Master Output Slave Input), es la salida de datos del maestro hacia el esclavo.
- Línea SOMI (Slave Output Master Input), es la salida de datos del esclavo hacia el maestro.
- Línea SS/SSTE, la cual sirve para seleccionar el esclavo al cual se le va a enviar la información.

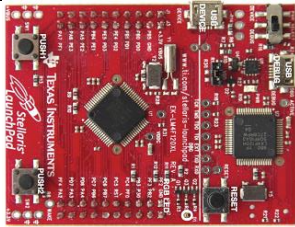
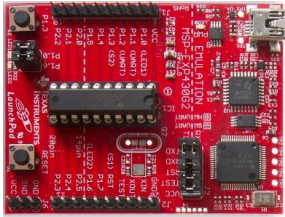

La tarjeta Stellaris LM4F120 cuenta con 4 módulos SSI (Synchronous Serial Interface) con los cuales se puede establecer una comunicación SPI, dichos módulos pueden trabajar como maestros o esclavos según sea requerido.

La comunicación se puede hacer entre tarjetas del mismo o diferente modelo y para esta práctica se ha desarrollado un ejemplo de un sistema SPI simple con una tarjeta Stellaris LM4F120 y una Launchpad MSP430 actuando como Maestro y esclavo respectivamente, cabe mencionar que en este documento se mostrara todo el código requerido para el funcionamiento de dicho sistema pero haciendo

especial énfasis en el código que corresponde a la tarjeta LM4F120.

Hardware y software

El hardware utilizado durante la práctica se muestra en la siguiente tabla:

Stellaris LM4F120	
Launchpad MSP430	
USB to TTL D-SUN-V3.0	

Cabe mencionar que para la comunicación seria se usa un el USB to TTL para una implementación rápida y así evitar el uso de un cable serial y el circuito integrado MAX232.

El software utilizado en esta práctica fue el Hercules el cual sirve para hacer el envío y recepción de datos por comunicación serial entre la pc y la tarjeta LM4F120.

Conexiones

Las conexiones entre las tarjetas se listan a continuación:

Conexión	LM4F120	MSP430
Reloj	PB4	P1.4
Transmisión Maestro	PB7 (Tx)	P1.2 (Rx)
Recepción Maestro	PB6 (Rx)	P1.1 (Tx)

Tabla 1. Conexiones entre Maestro y esclavo.

Conexión	LM4F120	MSP430
Ground	GND	GND
Transmisión (lm4f120)	PE1 (Tx)	RXD
Recepción (lm4f120)	PE0 (Rx)	TXD

Tabla 2. Conexión entre lm4f120 y TTL.

El sistema SPI simple está conformado por una tarjeta Stellaris LM4F120 como maestro y una tarjeta MSP430 como esclavo como se muestra en la figura 1.

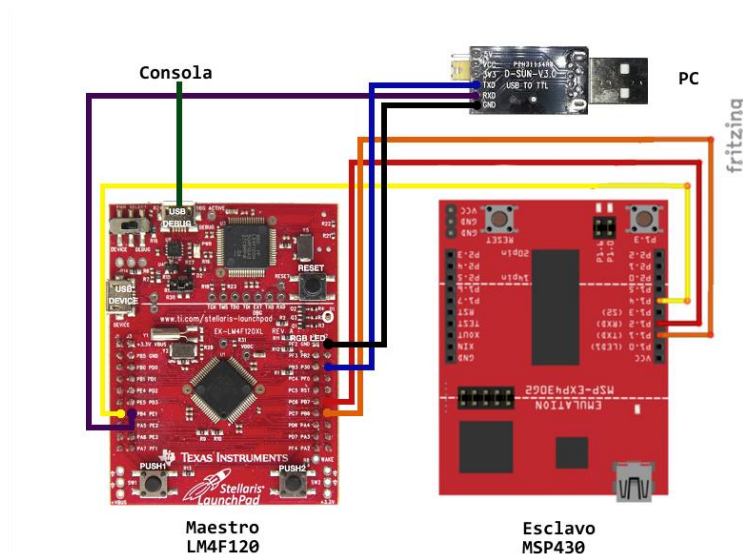


Figura 1. Diagrama general del sistema de comunicación SPI simple.

El Maestro recibe un dato desde la PC a través de la comunicación UART, este dato es enviado a través del bus SPI hacia el esclavo y por último el esclavo toma el dato recibido y lo manda de regreso por el bus SPI, por último este dato es desplegado por el maestro en la consola de información.

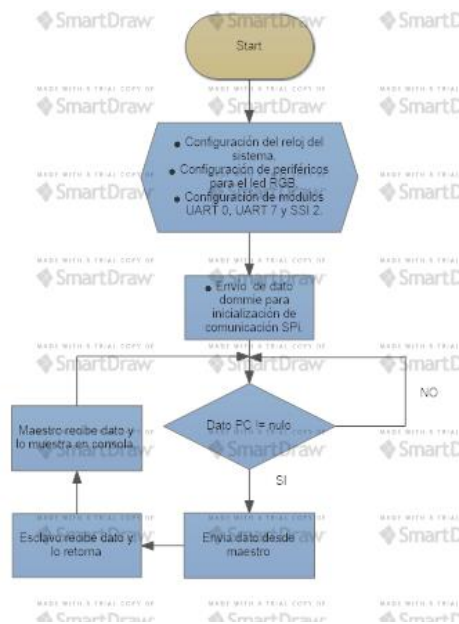


Figura 2. Diagrama de flujo del sistema de comunicación SPI simple.

El resultado final de las conexiones se muestra en la figura 3

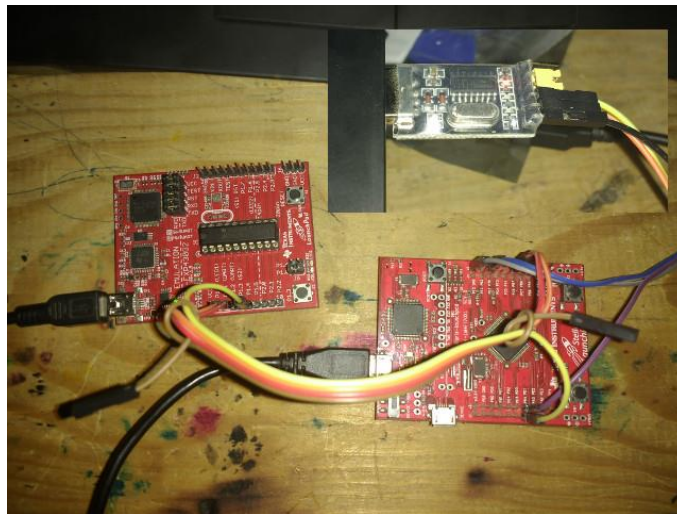


Figura 3. Conexión de tarjetas y ttl reales.

Al iniciarse el sistema se imprimen información útil para saber que el sistema está trabajando correctamente.

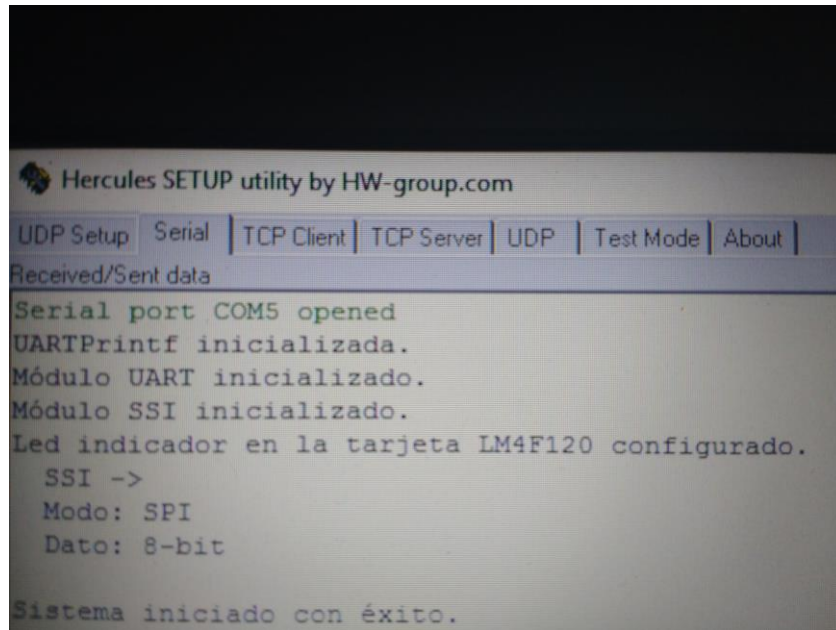


Figura 4. Información de inicialización correcta del sistema.

Como ya se mencionó anteriormente la funcionalidad es muy básica, se hace el envío de un dato desde la pc al maestro (Im4f120) a través de la comunicación serial, el maestro envía este dato por el bus SPI hacia el esclavo (MSP430), al recibir el dato el esclavo retorna el dato recibido hacia el maestro y por último el maestro imprime el dato retornado en la consola de información.

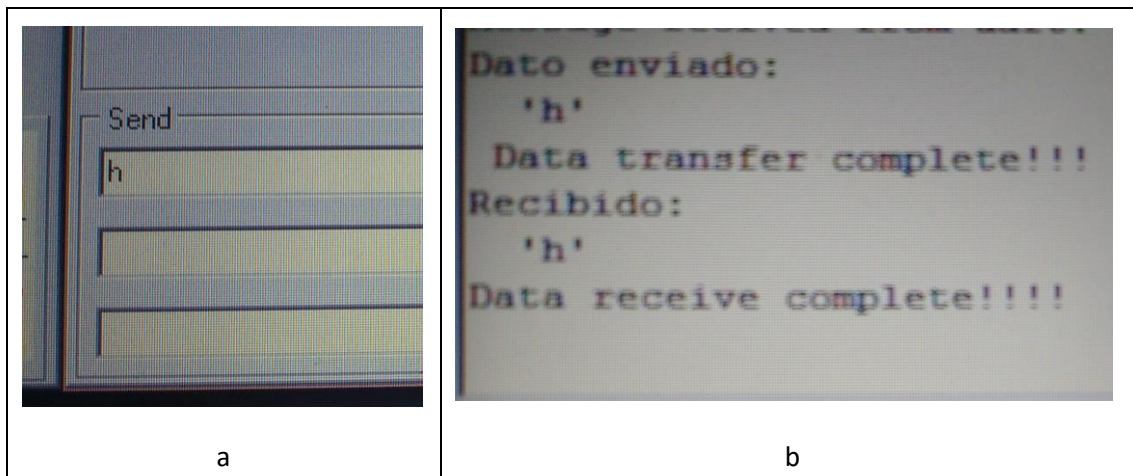


Figura 5. Dato a enviar (a) y consola de información (b).

Explicación del código utilizado en el sistema

El software del sistema de comunicación SSI está conformado por varias parte de código, como ya sabemos lo primero que se debe hacer la declaración de las cabeceras de las librerías necesarias para la ejecución del software así como las variables globales que se requieran, a continuación se muestran las cabeceras y variables declaradas para el actual sistema:

```
#include "inc/hw_memmap.h"
#include "inc/hw_ssi.h"
#include "inc/hw_types.h"
#include "inc/hw_ints.h"
#include "driverlib/pin_map.h"
#include "driverlib/interrupt.h"
#include "driverlib/ssi.h"
#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"

#define NUM_SSI_DATA 3

unsigned long ulDataTx[NUM_SSI_DATA];
unsigned long ulDataRx[NUM_SSI_DATA];
unsigned long ulindex1;
int ulindex2;
```

En este sistema se hace uso del módulo uart para desplegar información durante la ejecución del sistema y hacer el envío de caracteres desde la PC hacia la tarjeta y después ser enviado del maestro al esclavo, dicho dato debe ser regresado por el esclavo al maestro y este último mostrarlo en la terminal donde se imprime el log de información.

Método InitConsole

En este método se hace una configuración básica de la uart 0, donde dicho modulo hace la función de una consola para mostrar los logs de información durante la ejecución del sistema haciendo uso de la librería uartstdio.

```
void InitConsole(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);

    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    UARTStdioInit(0);
}
```

Método blinkled

Como su nombre lo dice este método sirve para ejecutar un parpadeo en el led rojo de la tarjeta stellaris, el cual sirve para notificar al usuario cuando se ejecutó alguna acción en el sistema.

```
void blinkled(void)
{
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 2);
    SysCtlDelay(SysCtlClockGet()/30);
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0);
}
```

Método InitUARTComm

Este método hace la configuración necesaria para usar el módulo UART 7 de la tarjeta, en el cual se habilitan periféricos y pines a utilizar así como la configuración para la comunicación serial y habilitación de interrupciones.

```
Void InitUARTComm(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART7);

    GPIOPinConfigure(GPIO_PE0_U7RX);
    GPIOPinConfigure(GPIO_PE1_U7TX);

    GPIOPinTypeUART(GPIO_PORTE_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    UARTConfigSetExpClk(UART7_BASE, SysCtlClockGet(), 115200,
UART_CONFIG_WLEN_8|UART_CONFIG_PAR_NONE|UART_CONFIG_STOP_ONE);

    IntEnable(INT_UART7);

    UARTIntEnable(UART7_BASE, UART_INT_RX | UART_INT_RT);
}
```

Método UARTIntHandler

El método UARTIntHandler es la función que se dispara cuando una interrupción es generada al momento de recibir datos de la PC a la tarjeta Stellaris a través de la comunicación serial, cabe mencionar que inmediatamente después de haber recibido el dato se hace el envío del mismo por la comunicación SSI maestro – esclavo.

```
Void UARTIntHandler(void)
{
    int i = 0;
    unsigned long ulStatus;
    unsigned long data2Send;
```



```

        ulStatus = UARTIntStatus(UART7_BASE, true);
        UARTIntClear(UART7_BASE, ulStatus);

        while(UARTCharsAvail(UART7_BASE))
        {
            data2Send = UARTCharGetNonBlocking(UART7_BASE);

            GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
            SysCtlDelay(SysCtlClockGet() / (1000 * 3));
            GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);

            i++;
        }

        UARTprintf("Message recived from uart: %c\n", data2Send);
        sendDataSSI(data2Send, 0);
    }

```

Como parte de la configuración del proyecto es necesario hacer la configuración del handler UARTIntHandler el cual trabajara con el modulo UART 7, esto en el archivo "lm4f120h5qr_startup_ccs.c", agregando las siguientes lineas:

```

#include <stdint.h>

//*****
//
// Forward declaration of the default fault handlers.

.
.
.

// External declaration for the reset handler that is to be called when the
// processor is started
//
//*****
extern void _c_int00(void);
extern void UARTIntHandler(void);

.
.
.

IntDefaultHandler,           // UART5 Rx and Tx
IntDefaultHandler,           // UART6 Rx and Tx
    UARTIntHandler,          // UART7 Rx and Tx

.
.
.

```

Configuración inicial del módulo SSI para la comunicación SPI

Para el ejemplo se configuro el módulo SSI de la tarjeta Stellaris LM4F120 como maestro de la siguiente manera:

El método

SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI2)

Habilita el módulo SSI2 de la tarjeta el cual hace uso de alguno pines del puerto B, por lo cual se habilita el puerto B de la tarjeta.

Los pines a utilizar son los siguientes:

- PB4 para el reloj de sincronización con el esclavo.
- PB6 es el pin para la recepción de datos esclavo - maestro.
- PB7 es el pin para la transmisión de datos maestro – esclavo.

Con el método

GPIOPinTypeSSI(GPIO_PORTB_BASE, GPIO_PIN_7 | GPIO_PIN_6 | GPIO_PIN_4)

Se indica el puerto y los pines a usar para el módulo SSI habilitado.

El método

SSIConfigSetExpClk(SSI2_BASE, **SysCtlClockGet**(), SSI_FRF_MOTO_MODE_2, SSI_MODE_MASTER, 1200000, 8)

Indica la configuración con la que trabajara el módulo SSI, dicho método se le pasan como parámetros de entrada:

- El módulo SSI a configurar.
- La frecuencia del reloj del sistema para poder definir la velocidad de sincronización maestro – esclavo.
- La polaridad y fase las cuales serán 1 y 1 respectivamente.
- Se le indica que trabajara como maestro.
- Se establece la frecuencia del reloj de sincronización maestro – esclavo de 1.2 MHz.
- Los datos a transmitir serán de 8 bits.

Por último se habilita el módulo SSI2, habilita las interrupciones para el modulo y se indican las banderas que dispararan el handler de interrupción:

- SSI_RXFF.
- SSI_RXTO.
- SSI_RXOR.

A continuación se muestra el código completo para la configuración del módulo SSI a utilizar.

```
void initSSI(void){
    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI2);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

    GPIOPinConfigure(GPIO_PB4_SSI2CLK);
    GPIOPinConfigure(GPIO_PB6_SSI2RX);
    GPIOPinConfigure(GPIO_PB7_SSI2TX);

    GPIOPinTypeSSI(GPIO_PORTB_BASE, GPIO_PIN_7 | GPIO_PIN_6 | GPIO_PIN_4);

    SSIConfigSetExpClk(SSI2_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_2,
        SSI_MODE_MASTER, 1200000, 8);

    SSIEnable(SSI2_BASE);

    IntEnable(INT_SSI2);
    SSIIntEnable(SSI2_BASE, SSI_RXFF | SSI_RXTO | SSI_RXOR);
}
```

Transmisión de datos por SPI.

La librería para el uso del módulo SSI de la tarjeta Stellaris LM4F120 cuenta con varios métodos para la transmisión de datos en este ejemplo se implementa el método “**SSIDataPut**(SSI2_BASE, data2ssi*2)” el cual hace el envío de un dato a la vez, como parámetros recibe el módulo SSI a utilizar y el dato a enviar, cabe mencionar que por conflicto de datos que se tuvieron durante las pruebas se optó por multiplicar el dato a enviar por 2.

Antes de hacer el envío se verifica que no se esté recibiendo datos con el método “**SSIDataGetNonBlocking**(SSI2_BASE, &ulDataRx[0])”, las últimas tres líneas son logs de información.

```
void sendDataSSI (unsigned long data2ssi){
    while(SSIDataGetNonBlocking(SSI2_BASE, &ulDataRx[0]))
    {
    }
}
```

```

    SSIDataPut(SSI2_BASE, data2ssi*2);

    UARTprintf("Dato enviado:\n ");
    UARTprintf("%c" \n ", data2ssi);
    UARTprintf("Data transfer complete!!! \n");
}

```

Recepción de datos por SSI.

La recepción de datos enviados por el esclavo se hace mediante el handler de interrupciones ssirxhandler, el cual es disparado cada que se cumplan las condiciones configuradas en la etapa de inicialización del módulo SSI.

Primeramente al dispararse una interrupción se limpia la interrupción con el método “**SSIIntClear**(SSI2_BASE, SSI_RXFF | SSI_RXT0 | SSI_RXOR)” indicando el módulo SSI y las banderas a limpiar.

Para obtener el dato enviado desde el esclavo se utiliza el método “**SSIDataGet**(SSI2_BASE, &ulDataRx[ulindex2])”, el cual lee del buffer de entrada y lo almacena en el arreglo ulDataRx en el índice ulindex2, cabe mencionar que se hace una operación con el dato recibida y una máscara para quedarse con el byte más significativo pues originalmente el módulo SSI trabaja con datos de 16 bits, el resto del código son logs de información.

A continuación se muestra el código del handler par interrupciones de entrada del módulo SSI.

```

void ssirxhandler(){

    SSIIntClear(SSI2_BASE, SSI_RXFF | SSI_RXT0 | SSI_RXOR);

    SSIDataGet(SSI2_BASE, &ulDataRx[ulindex2]);

    ulDataRx[ulindex2] &= 0x00FF;

    UARTprintf("Recibido:\n ");
    UARTprintf("%c" \n", ulDataRx[ulindex2]);

    UARTprintf("Data receive complete!!!! \n");

}

```

Como parte de la configuración del proyecto es necesario hacer la configuración del handler UARTIntHandler el cual trabajara con el módulo SSI 2, esto en el archivo “lm4f120h5qr_startup_ccs.c”, agregando las siguientes líneas:

```

#include <stdint.h>

//*****
//
// Forward declaration of the default fault handlers.

```

```

.
.
.

// External declaration for the reset handler that is to be called when the
// processor is started
//
//*****
static void FaultISR(void);
static void IntDefaultHandler(void);
static void ssirxhandler(void);
.
.
.

IntDefaultHandler,           // GPIO Port K
IntDefaultHandler,           // GPIO Port L
    ssirxhandler,             // SSI2 Rx and Tx

.
.
.

```

Programa principal del sistema.

En el “main” del programa se hace la inicialización de variables, módulos y recursos necesarios para el funcionamiento del sistema, a continuación se listan las partes que conforman el trozo de código del main:

- Configura el reloj para que sea ejecutado directamente desde el cristal/oscilador externo, el cual tiene una frecuencia de 16 MHz.
- Indica el puerto y pines de salida para usar el led RGB de la tarjeta para indicar ciertas acciones del sistema.
- Inicializa los pines 1 y 2 del led RGB de la tarjeta.
- Configura la consola serial para desplegar mensajes de información durante la ejecución del sistema.
- Configura el módulo UART para la comunicación serial.
- Inicializa el módulo SSI.
- Habilita el interruptor maestro.
- Inicializa los pines 1 (led rojo) y 2 (led azul) del led RGB de la tarjeta.
- Enciende y apaga el led Azul como indicador de que el sistema se ha terminado de configurar correctamente.
- Por último se envía un valor dommie al esclavo para que detecte que la conexión SPI.

```

Int main(void)
{
    ulindex1 = 0;
    ulindex2 = 0;

```

```

SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                SYSCTL_XTAL_16MHZ);

SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_2|GPIO_PIN_1);

InitConsole();
UARTprintf("UARTPrintf inicializada.\n");

InitUARTComm();
UARTprintf("Módulo UART inicializado.\n");

initSSI();
UARTprintf("Módulo SSI inicializado.\n");

IntMasterEnable();

GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0);

GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
SysCtlDelay(SysCtlClockGet() / (1000 * 3));
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);
UARTprintf("Led indicador en la tarjeta LM4F120 configurado.\n");

UARTprintf("  SSI ->\n");
UARTprintf("  Modo: SPI\n");
UARTprintf("  Dato: 8-bit\n\n");

UARTprintf("Sistema iniciado con éxito.\n");

sendDataSSI(0, 0);

while (1){
}

```

Código en el esclavo

A continuación se muestra el código correspondiente al esclavo, el cual cuenta con una función de interrupción la cual es disparada al recibir un dato desde el maestro y retorna el dato enviado.

```

#include "msp430g2553.h"
#include <string.h>

unsigned long cmdbuf[3];
unsigned int cmd_index=0;
char respbuff[2];

/** Delay function. */
void delay(unsigned int d) {
    int i;
    for (i = 0; i<d; i++) {
        //nop();
    }
}

```



```

}

void flash_spi_detected(void) {
    int i=0;
    P1OUT = 0;
    for (i=0; i < 6; ++i) {
        P1OUT = ~P1OUT;
        delay(0x4fff);
        delay(0x4fff);
    }
}

void main(void)
{
    respbuff[0] = 'o';
    respbuff[1] = 'k';

    WDTCTL = WDTPW + WDTHOLD;           // Stop watchdog timer
    /* led */
    //P1DIR |= BIT0 + BIT5;
    P1DIR |= BIT0;
    while (P1IN & BIT4);                 // If clock sig from mstr stays low,
                                         // it is not yet in SPI mode

    flash_spi_detected();                // Blink 3 times

    P1SEL = BIT1 + BIT2 + BIT4;
    P1SEL2 = BIT1 + BIT2 + BIT4;
    UCA0CTL1 = UCSWRST;                  // **Put state machine in reset**
    UCA0CTL0 |= UCMSB + UCMODE_2 + UCSYNC; // 3-pin, 8-bit SPI
slave
    UCA0CTL1 &= ~UCSWRST;                // **Initialize USCI state
machine**

    IE2 |= UCA0RXIE;                    // Enable USCI0 RX interrupt

    __bis_SR_register(LPM4_bits + GIE); // Enter LPM4, enable interrupts

    while(1){
    }
}

__attribute__((interrupt(USCIAB0RX_VECTOR))) void USCI0RX_ISR (void)
{
    unsigned long dataReceived;

    if (cmd_index == 3) {/(strcmp(cmdbuf, "spi", 3) == 0) {
        P1OUT |= BIT0;
        cmd_index = 0;
    } else {
        P1OUT &= ~BIT0;

        cmdbuf[cmd_index] = UCA0RXBUF;
        dataReceived = UCA0RXBUF;

        UCA0TXBUF = dataReceived;
        cmd_index++;
    }
}

```

