UNIVERSIDAD NEBRIJA

MEMORIA PRACTICA 4
SISTEMAS EMPOTRADOS

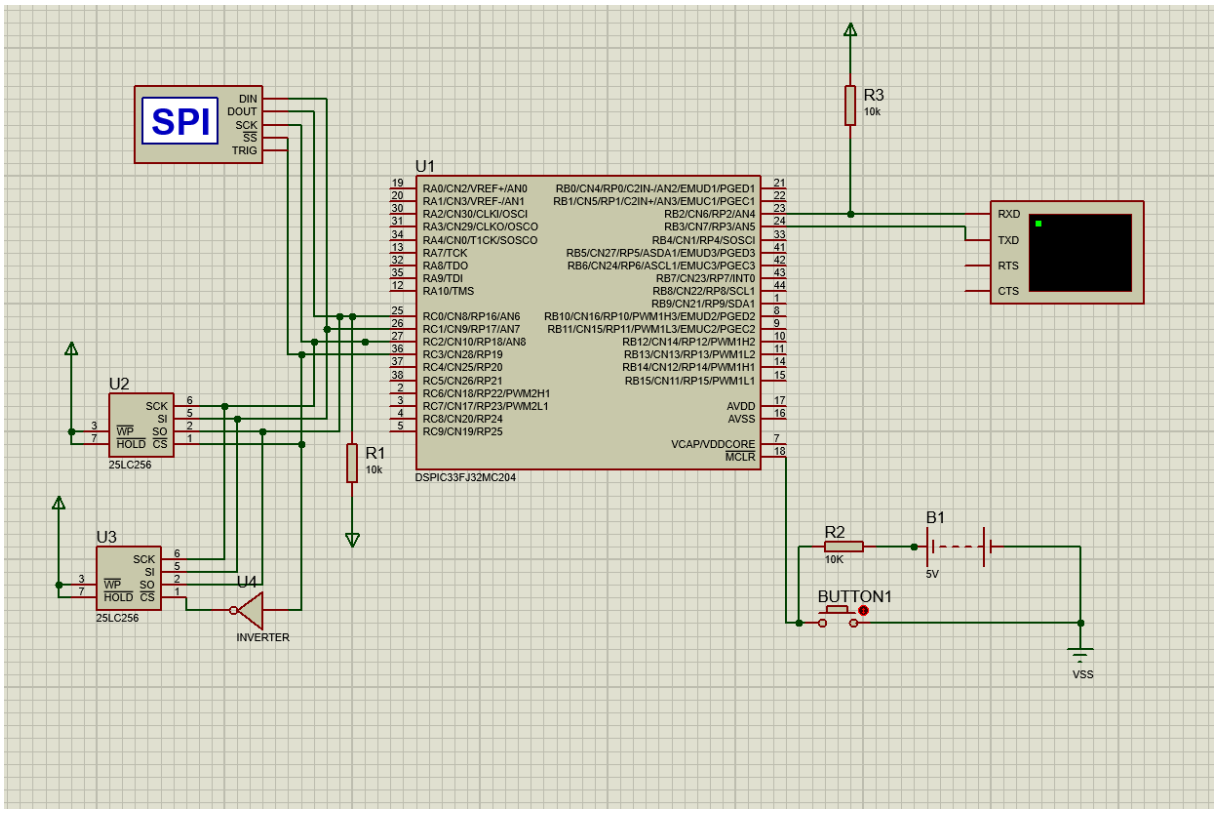
José María Fernández Gómez
Pablo Rayón Zapater
Fernando Pérez Ballesteros

OBJETIVOS

Se pretende diseñar un sistema que escriba en dos memorias EEPROM por medio del protocolo SPI, de cara a la compresión de protocolos de comunicación síncronos. Por otro lado, se trata de paralelizar la escritura de estas memorias con un reporte en texto claro de cada operación, esto a través de protocolo de comunicaciones UART.

CONFIGURACION HARDWARE

A continuación, se muestra la configuración hardware empleada por el sistema.



Se dispone de un microcontrolador dspic33fj32mc204, dos memorias EEPROM 25lc256 y dos dispositivos para poder comprobar el input/output de este primero y ser capaz de visualizar los datos y procedimientos que se están llevando a cabo, un terminal virtual, el cual irá reportando en texto claro la operación que se está realizando sobre qué memoria e introduciendo x dato en dicha memoria, también irá reportando la lectura de cada memoria y el estado de la comprobación de carga y lectura de estas. Por otro lado, tenemos un SPI debugger, el cual toma mediciones sobre todos los puertos y señales involucrados en el protocolo SPI, por el medio del cual trataremos de escribir y leer en las memorias.

Los pines 23 y 24 están conectados al puerto de recepción y de transmisión del terminal virtual, estos actúan como transmisor y receptor del protocolo UART respectivamente, a través del cual se envían los datos de reporte de las otras operaciones realizadas por el protocolo SPI.

Los pines 25 y 26 son los pines de datos del protocolo SPI, el primero de estos corresponde el puerto MOSI, el cual se encargará de sacra datos del microcontrolador (Master)

Y enviarlos a la/las memorias (Slave/s), el otro pin realizará las funciones de MISO, que representan la entrada de datos al microcontrolador (Master) y la salida o lectura de datos de la /las memorias (Slave/s).

CONFIGURACION SOFTWARE

```
#include <xc.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdbool.h>
#include <math.h>

#define baud_9600  1041
char txbuffer[200];
unsigned char memRead;
int bytesCargadosM1[64];
int bytesLeidosM1[64];
int bytesCargadosM2[64];
int bytesLeidosM2[64];
```

Primero se muestran las librerías requeridas y las variables globales declaradas para todo el programa. Se define baud_9600 como la velocidad de transmisión del protocolo UART, txbuffer como buffer de transmisión para este mismo protocolo y el resto de las variables para procedimientos de lectura/escritura del protocolo SPI. Se declaran 4 arrays de carga/lectura para poder comprobar la inserción de datos en las memorias EEPROM como es requerido en el enunciado.

```
void uart_config(unsigned int baud) {
    //Interface uart (tx/rx)
```

Aquí se muestra de manera representativa la función de configuración de la UART, que como ya fue tratada en practicas anteriores se deja indicado el uso de esta, pero sin entrar en detalles funcionales.

Configuración del SPI:

```
void spi_config(void) {

    //Configuración de pines SPI
    TRISCbits.TRISC0 = 1; //MISO asi
    //el resto de pines como son gen
    TRISCbits.TRISC1 = 0; //MOSI asi
    TRISCbits.TRISC2 = 0; //SCK asig
    TRISCbits.TRISC3 = 0; //CS asign

    LATCbits.LATC1 = 0;

    //aqui se remapea (los valores q
    RPINR20bits.SDI1R = 16; //RC0 tr
    RPOR8bits.RP17R = 7; //(00111);
    RPOR9bits.RP18R = 8; //(01000);

    //Configuración SPISTAT
    SPI1STATbits.SPIEN = 0;
    SPI1STATbits.SPISIDL = 0;
    SPI1STATbits.SPIROV = 0;
    SPI1STATbits.SPITBF = 0;
    SPI1STATbits.SPIRBF = 0;

    //Configuración SPICON1
    SPI1CON1bits.DISSCK = 0;
    SPI1CON1bits.DISSDO = 0;
    SPI1CON1bits.MODE16 = 0; //el sp
    SPI1CON1bits.SMP = 0;
    SPI1CON1bits.CKP = 0; //SPI MO
    SPI1CON1bits.CKE = 0;

    SPI1CON1bits.SSEN = 0; //Hardw
    SPI1CON1bits.MSTEN = 1; //Mast

    SPI1CON1bits.PPRE = 1; //1:1
    SPI1CON1bits.SPRE = 6; //2:1

    //Configuración SPICON2 Framed
    SPI1CON2bits.FRMEN = 0;
    SPI1CON2bits.SPIFSD = 0;
    SPI1CON2bits.FRMPOL = 0;
    SPI1CON2bits.FRMDLY = 0;

    //SPI Interrupciones
    IFS0bits.SPI1IF = 0;
    IFS0bits.SPI1EIF = 0;
    IEC0bits.SPI1IE = 0;
    IEC0bits.SPI1EIE = 0;
    IPC2bits.SPI1EIP = 6;
    IPC2bits.SPI1EIP = 6;

    LATCbits.LATC3 = 1;
    SPI1STATbits.SPIEN = 1;
}
```

Esta función es análoga de la función uart config, pero para el protocolo SPI, esta a diferencia de la anterior no requiere ningún parámetro de entrada de cara a la configuración de la velocidad, ya que se configura por frecuencias a través del bit SPI1CON.PPRE Y SPI1CON.SPRE, que son preescalers que modifican la función de reloj del controlador.

CONFIGURACION SOFTWARE

Funciones útiles:

```
void uart_send_byte(char c) {
    while (U1STAbits.UTXBF); //mientras el buffer no este lleno
    U1TXREG = c;
}

void uart_send_string(char *s) {
    while ((*s) != '\0') {
        uart_send_byte(*(s++));
    }
}

void delay_ms(unsigned long time_ms) {
    unsigned long u;
    for (u = 0; u < time_ms * 200; u++) {
        asm("NOP");
    }
}
```

Aquí tenemos tres funciones que ya se han usado previamente, estas funciones son útiles a la hora de hacer el reporte de estado a través del protocolo UART y de cara a la gestión de temporización y sincronización.

Funciones útiles 2:

```
void sort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int arrays_equal(int arr1[], int arr2[], int n, int m) {
    sort(arr1, n);
    sort(arr2, m);
    int i, res;
    for (i = 0; i < n; i++) {
        if (arr1[i] != arr2[i]) {
            res = 0;
        } else res = 1;
    }
    return res;

}
```

Estas dos funciones se han empleado para poder comparar dos arrays, de cara a la última parte requerida en la practica de comparar que la escritura y lectura en las memorias EEPROM haya sido correcta. Primero la función sort se encarga de ordenar los arrays por valor en orden ascendente, esto reduce la complejidad de la funcione siguiente y facilita la comparación final, en el caso base que se presenta la práctica no sería necesario el empleo de esta función ya que los datos se insertan secuencialmente ordenados en las memorias, pero se ha dejado en el código por si se quisiera probar otro caso de lectura/escritura con otros datos. La segunda función recorre ambos arrays comparando los datos y devolviendo 1 si la comparación ha sido satisfactoria o 0 si se ha encontrado alguna discrepancia.

CONFIGURACION SOFTWARE

Escritura/Lectura de SPI:

```
unsigned char spi_write(unsigned char data) {
    while (SPI1STATbits.SPITBF);
    SPI1BUF = data;
    while (!SPI1STATbits.SPIRBF); //Esperamos a recibir el sitio de vuelta
    return SPI1BUF;
}
```

Con esta función conseguimos tanto cargar como leer datos del buffer del SPI, al este constar de una estructura circular que involucra múltiples registros, de los cuales solo tenemos acceso al SPI1BUF, es posible por medio de una única función extraer y cargar datos. Lo gramos esta dualidad por medio de dos condicionales persistentes que monitorizan constantemente dos flags, uno que indica si existe hueco en la parte de transmisión (SPITBF) y otro que indica si existe algo en la parte de recepción (SPIRBF).

Por lo que mientras que el primer flag este desactivado, sería posible enviar la cadena que deseamos enviar, y en caso de que haya llegado algo a la parte de recepción esta función devolverá el buffer del SPI, en los casos cuando cualquiera de estas condiciones no se cumpla se enviaran y se devolverán “dummies” o bits no deseados, por lo que es importante tratar bien los valores de retorno de esta función.

FUNCION DE ESCRITURA EN EEPROM

Para una mejor legibilidad y comprensión del código se ha dividido las funciones de escritura, de lectura y de comprobación, también se ha decidido codificarlo de esta manera por optimizar el numero de líneas de código, ya que así es posible únicamente poseer de un prototipo de función para ambas memorias y puede alternarse el destino u origen de los datos por medio de una señal de selección entre ambas memorias.

```
void write_mem(int sel) {
    sprintf(txbuffer, "============ ESCRIBIENDO EN MEMORIA %d ==============\r\n", sel + 1);
    uart_send_string(txbuffer);
    delay_ms(500);
    for (int i = 0; i <64 ; i++) {
        LATCbits.LATC3 = sel;
        delay_ms(5);
        spi_write(0x06);
        delay_ms(5);
        LATCbits.LATC3 = !sel;
        LATCbits.LATC3 = sel;
        delay_ms(5);
        spi_write(0x02); //Write
        spi_write(0x00); //MSB MEM
        spi_write(i); //LSB MEM
        spi_write(i); //DATA
        if (sel == 0)bytesCargadosM1[i] = i;
        else bytesCargadosM2[i] = i;
        delay_ms(5);
        LATCbits.LATC3 = !sel;
        sprintf(txbuffer, "Escrito en memoria %d 0x%02X en direccion = 0x%02X \r\n", sel + 1, i, i);
        uart_send_string(txbuffer);
        LATCbits.LATC3 = sel;
        delay_ms(5);
        spi_write(0x04); //DISABLE WRITING
        delay_ms(5);
        LATCbits.LATC3 = !sel;
        delay_ms(100);
    }
    sprintf(txbuffer, "============ Pagina completa escrita en memoria %d ==============\r\n", sel + 1);
    uart_send_string(txbuffer);
```

En la imagen superior se muestra la función que se encarga de escribir en las memorias

EEPROM, como se puede observar tiene un parámetro de entrada “sel” el cual se encarga de seleccionar la memoria a la que va orientada la función en cada llamada, esta señal de selección estará incluida en todas las funciones que se comentarán a partir de ahora.

Los conjuntos de código que constan de un “sprintf” y “uart_send_string” forman parte de la monitorización del proceso que se va a llevar a cabo por el protocolo SPI inmediatamente después.

Esta función se compone de tres secuencias diferentes contenidas dentro de un bucle for que itera sobre 64 numero, numero de posiciones de memoria a escribir. La primera secuencia se encarga de enviar un 0x06, comando que la EEPROM reconoce como una habilitación para poder ser escrita, después de cada envío de secuencia por SPI se restablecen los flags de Chip Select, por si es necesario escribir en otra memoria. La segunda secuencia se encarga de enviar un 0x02, el cual indica a la memoria que se va a escribir sobre ella, a continuación se le envía la dirección de memoria sobre la que va a ser escrita en dos paquetes, la primera indicando la parte más significativa de dicha dirección, que a no superar la primera página nunca aumentará de 0x00, y la segunda que indica la parte menos significativa, esta es la que irá aumentando por cada iteración del bucle; por último se envía el dato que se va a escribir en la dirección mencionada previamente, en este caso el dato corresponde con la dirección de memoria sobre la que se escribe por motivos de comodidad, pero el programa seguiría funcionando perfectamente si se le introdujera arbitrariamente datos en la posición que se desee. En esta parte del código también se almacena en un vector global los datos que se están almacenando en la memoria con el propósito de ser checkeados posteriormente. La última secuencia en el proceso de escritura se encarga de deshabilitar el modo escritura de la EEPROM para sellarla y que no sea posible modificarla a menos que se desee explícitamente.

FUNCION DE LECTURA DE LA EEPROM

```
void read_mem(int sel) {
    //---------------------- Lectura -----------------------------
    sprintf(txbuffer, "============ LEYENDO MEMORIA %d ==============\r\n", sel + 1);
    delay_ms(500);
    uart_send_string(txbuffer);
    for (int i = 0; i < 64; i++) {
        LATCbits.LATC3 = sel;
        delay_ms(5);
        spi_write(0x03); //Read
        spi_write(0x00); //MSB MEB
        spi_write(i); //LSB MEM
        memRead = spi_write(0x00);
        delay_ms(5);
        LATCbits.LATC3 = !sel;
        delay_ms(5);
        if (sel == 0)bytesLeidosM1[i] = memRead;
        else bytesLeidosM2[i] = memRead;
        sprintf(txbuffer, "Dato en memoria %d 0x%02X, Adress = 0x%02X \r\n", sel + 1, memRead, i);
        uart_send_string(txbuffer);
        delay_ms(100);
    }
    sprintf(txbuffer, "============ Pagina completa leida de memoria %d ==============\r\n", sel + 1);
    uart_send_string(txbuffer);
    delay_ms(2000);

}
```

Esta función es muy similar a la anterior, con diferencias puntuales, la primera de estas es que solo posee dos secuencias diferenciables en lugar de tres, ya que para la lectura de datos no es necesario deshabilitar nada una vez se haya finalizado este proceso.

Por otro lado, pese a que se emplee la misma función que en la anterior subrutina, en esta ocasión se extrae un valor de esta, se le indica a la función spi_write un valor de salida y como parámetro de entrada se le introducen los “dummies” mencionados en anteriores explicaciones, por otro lado los valores leídos de la memoria se almacenan en arrays análogos a los que se declararon para la escritura, los cuales serán comprados entre ellos para comprobar la integridad de los datos escritos y leídos en memoria.

FUNCION DE COMPROBACION DE LA EEPROM

```
void check_mem(int sel) {
    sprintf(txbuffer, "============ Comprobacion de valores en memoria %d ==============\r\n", sel + 1);
    uart_send_string(txbuffer);
    if (sel == 0) {
        if (arrays_equal(bytesLeidosM1, bytesCargadosM1, 64, 64) == 1) {
            sprintf(txbuffer, "============ Bytes cargados correctamente en memoria 1 ==============\r\n");
        } else {
            sprintf(txbuffer, "============ !ERROR EN LA CARGA DE DATOS EN MEMORIA 1 ==============\r\n");
        }
    } else {
        if (arrays_equal(bytesLeidosM2, bytesCargadosM2, 64, 64) == 1) {
            sprintf(txbuffer, "============ Bytes cargados correctamente en memoria 2 ==============\r\n");
        } else {
            sprintf(txbuffer, "============ !ERROR EN LA CARGA DE DATOS EN MEMORIA 2 ==============\r\n");
        }
    }
    uart_send_string(txbuffer);
    delay_ms(2000);

}
```

Esta función comprueba la integridad de los datos leídos y escritos en ambas memorias por medio de las dos funciones auxiliares que se han explicado arriba, realiza una comparación ordenada de los datos que se han introducido en el array de carga, rellenado a la par que se escribían los datos en memoria, y el de lectura, rellenado una vez se han extraído los datos de una de estas. Vuelve a mostrar por terminal a través de la UART el estado de dichas comprobaciones.

FUNCION MAIN

```
int main(void) {
    PLLFBD = 38; //M  = PLLFBD
    CLKDIVbits.PLLPOST = 0; //N1 = PLLPOST + 2
    CLKDIVbits.PLLPRE = 0; //N1 = PLLPOST + 2
    while (OSCCONbits.LOCK != 1);
    AD1PCFGL = 0xFFFF; // Todos los pines configurados como pines digitales
    uart_config(baud_9600);
    spi_config();
    INTCON1bits.NSTDIS = 0; //Interrupt nesting enable
    SRbits.IPL = 0; //enable global interrupts
    delay_ms(1000);

    while (1) {
        write_mem(sel);
        read_mem(sel);
        check_mem(sel);
        sel = !sel;
    }
}
```

Esta función main se limita a llamar a las funciones anteriores recurrentemente en un bucle infinito, a parte de inicializar la configuración básica del microcontrolador que ya hemos explicado en tareas anteriores.

Se ha declarado una variable global llamada sel que empieza inicializada a 0, esta variable va alternando su valor entre 1 y 0 en cada iteración de este while infinito de manera que en cada una de estas iteraciones, las llamadas que posee dentro son aplicadas sobre una memoria diferente, así conseguimos un efecto toggle y una única declaración y llamada explicita a dichas funciones.