PRÁCTICA 3

Por Luis Díaz del Río Delgado

Introducción:

En esta práctica se pide programar los drivers y código necesario para realizar la comunicación entre dos microcontroladores, a través del puerto UART.

Se dispondrá de dos microcontroladores, MIC1 y MIC2 actuando como Master y Slave respectivamente.

MIC1: Encargado de recibir comandos del usuario, y transmitir los datos necesarios al microcontrolador Slave MIC2 para realizar una determinada operación.

MIC2: Encargado de recibir las tramas de datos del microcontrolador Master MIC1 y ejecutar las operaciones que se le ordenen. De esta manera, el usuario tiene control sobre MIC2, aunque no directamente.

Tareas pedidas:

- El usuario debe introducir los comandos a través del Virtual Terminal 1. El canal Tx del Virtual Terminal 1 irá conectado al puerto UART Rx del Master MIC1.
- El puerto UART Tx del Master MIC1 irá conectado al puerto UART Rx del Slave MIC2.
- El puerto UART Tx del Slave MIC2 irá conectado al canal receptor del Virtual Terminal 2.
- MIC2 dispondrá de dos LEDS de interacción o estado, de color verde y rojo. Estos LEDS podrán controlarse mediante comandos recibidos desde MIC1.

Lista de comandos:

“set MIC2_ledgreen 1” -> MIC1 deberá enviar el frame de datos correspondiente (ver tabla) para hacer que el LED verde de MIC2 se encienda.

“set MIC2_ledgreen 0” -> MIC1 deberá enviar el frame de datos correspondiente (ver tabla) para hacer que el LED verde de MIC2 se apague.

“set MIC2_ledred 1” -> MIC1 deberá enviar el frame de datos correspondiente (ver tabla) para hacer que el LED rojo de MIC2 se encienda.

“set MIC2_ledred 0” -> MIC1 deberá enviar el frame de datos correspondiente (ver tabla) para hacer que el LED rojo de MIC2 se apague.

“print MIC2_data” -> MIC1 deberá enviar el frame de datos correspondiente (ver tabla) para hacer que MIC2 imprima los datos almacenados en una variable interna, mostrados en Virtual Terminal 2. Los datos de esta variable son el conjunto de números del rango (0,…,100). La impresión debe hacerse a 5Hz.

“stop MIC2_data” -> MIC1 deberá enviar el frame de datos correspondiente (ver tabla) para hacer que MIC2 deje de imprimir los datos almacenados en una variable interna.

MIC1 y MIC2:

Los dos microcontroladores comparten las mismas librerías y tienen dos funciones que funcionan exactamente igual, por lo que las explicaré aquí antes de entrar en detalle con el código de cada uno de los microcontroladores.

Además, los dos microcontroladores van a una frecuencia de 2MHz.

Librerías:

```
#include <xc.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stddef.h>          // Defines NULL
#include <stdbool.h>         // Defines true
#include <math.h>
```

Además de la librería “xc.h” la cual viene por defecto, he utilizado las siguientes librerías:

- **Stdio.h:** Con esta librería podemos hacer uso de la función “sprintf()” que utilizaremos para escribir en el buffer de envío.
- **String.h:** Nos da acceso a las funciones “memset()” que utilizaremos para poner a NULL los buffers y la función “strlen()” que utilizaremos para ver el tamaño de los buffers.
- El resto de las librerías son las recomendadas por el profesor en la práctica, aunque no hago uso de ellas. Tienen como objetivo la definición de NULL y booleanos. Creo que la idea principal era que fueran utilizados a la hora de imprimir, o más bien al parar de imprimir, pero no he llegado a integrar esa parte de la práctica.

Funciones:

Para esta práctica solo utilizo dos funciones:

Delay_ms: Esta función es la misma que en el resto de las prácticas. Su objetivo es perder tantos ciclos de reloj como le digamos ejecutando la función en ensamblador “NOP”.

Uart_config: Esta función, utilizada también en la práctica anterior, cambia ligeramente para permitir el uso de interrupciones. Además de los pines de entrada y salida, el Baud Rate, también define las prioridades de las interrupciones y los valores de los flags.

```
void delay_ms(unsigned long time_ms){
    unsigned long i;
    for(i=0; i < time_ms*100; i++){
        asm("NOP");
    }
}
```

```c
void uart_config (unsigned int baud){
    //UART PORT/PIN COMMUNICATION ASSIGMENT
    TRISCbits.TRISC0    = 1;  //Pin RC0 input (digital).
    RPINR18bits.U1RXR   = 16; //Pin RC0 connected to recepcion port of UART1.
    RPOR8bits.RP17R     = 3;  //Pin RC1 connected to transmision pin of UART1.

    //U1MODE REGISTER CONFIG
    U1MODEbits.UARTEN   = 0; //Unable UART before config.
    U1MODEbits.USIDL    = 0; //Continue operation in idle mode.
    U1MODEbits.IREN     = 0; //Infrared.
    U1MODEbits.RTSMD    = 1; //Flow control. No using RTS/CTS. RX/TX mode.
    U1MODEbits.UEN      = 0; //Only using TX and RX pin.
    U1MODEbits.WAKE     = 0; //Awake when idle mode and recive data.
    U1MODEbits.LPBACK   = 0; //Loopback desable.
    U1MODEbits.ABAUD    = 0; //Auto baud rate desable.

    U1MODEbits.URXINV   = 0; //Idle state = '1'.
    U1MODEbits.BRGH     = 1; //High-Speed Mode (1 bit = 4 clock cycles).
    U1MODEbits.PDSEL    = 0; //8 bits data lenght and null parity.
    U1MODEbits.STSEL    = 0; //1-bit Stop at the end of data frame (8N1).

    //U1STA REGISTER CONFIG
    U1STAbits.URXISEL0 = 0;
    U1STAbits.URXISEL1 = 0;
    U1STAbits.ADDEN     = 0;
    U1STAbits.UTXBRK    = 0; //Sync frame desable.
    U1STAbits.OERR      = 0; //Buffer is empty (No Overflow problems)
    U1STAbits.PERR      = 0;
    U1STAbits.FERR      = 0;
    U1STAbits.UTXEN     = 1; //Enable transmisor

    // Configuramos la velocidad de transmisión/recepcción de los datos
    U1BRG = baud;

    // Prioridades, flags e interrupciones correspondientes a la Uart
    IPC2bits.U1RXIP = 6;     // U1RX con nivel de prioridad 6 (7 es el maximo)
    IFS0bits.U1RXIF = 0;     // Reset Rx Interrupt flag
    IEC0bits.U1RXIE = 1;     // Enable Rx interrupts

    IPC3bits.U1TXIP = 5;     // U1TX con nivel de prioridad 6 (7 es el maximo)
    IFS0bits.U1TXIF = 0;     // Reset Tx Interrupt flag
    IEC0bits.U1TXIE = 0;     // Enable Tx interrupts

    U1MODEbits.UARTEN = 1;      // Uart habilitada por completo
}
```

MIC1

Tras las librerías tenemos definimos “baud_9600” que tiene como valor 51. Tras aplicar la formula (BRGH = 1) sacamos que la velocidad de transmisión para la frecuencia de reloj elegida (2Mhz) es de 51.

Variables globales:

DataCMD_ISR: De tipo string (array de chars con una capacidad máxima de 50). Esta variable se utiliza en la recepción del mensaje metido por el Terminal Virtual.

txBuffer_ISR: También de tipo string, pero con capacidad máxima de 200. En esta variable se almacena el mensaje que se enviará, en este caso, a MIC2.

Dummy: De tipo unsigned char. Tiene como objetivo almacenar los caracteres que llegan por el Terminal Virtual cuando se ha detectado un comando. No he llegado a implementarlo.

NextChar: De tipo unsigned int. Su único objetivo es ser utilizado para indexar el buffer txBuffer_ISR.

Data_count: De tipo volatile unsigned int. Utilizado para indexar el buffer DataCMD_ISR.

Comando_detectado: De tipo volatile char. Esta variable se pone 1 en el caso de que se haya detectado un comando.

Cmd0, cmd1, …, cmd5: Cada una de estas variables almacena un string correspondiente a cada uno de los comandos especificados en el enunciado de la práctica.

```
#define baud_9600 51

char dataCMD_ISR[50];
char txBuffer_ISR[200];
unsigned char dummy;
unsigned int nextchar = 0;
volatile unsigned int data_count = 0;
volatile unsigned char comando_detectado = 0;

const char cmd0[] = {"set MIC2_ledgreen 1"}; //MIC2 green LED ON
const char cmd1[] = {"set MIC2_ledgreen 0"}; //MIC2 green LED OFF
const char cmd2[] = {"set MIC2_ledred 1"};   //MIC2 red LED ON
const char cmd3[] = {"set MIC2_ledred 0"};   //MIC2 green LED OFF
const char cmd4[] = {"print MIC2_data"};     //Print date
const char cmd5[] = {"stop MIC2_data"};      //Stop print
```

Main:

Primero configuramos la frecuencia de reloj, el baud rate de la UART (mediante uart_config) y el timer.

En el interior de un bucle infinito, y mediante un if, comprobamos si se ha detectado un comando. En el caso de que se haya detectado, no significa que sea un comando correcto, si no que se ha pulsado la tecla “Intro”.

En el caso de que se haya detectado un comando, debemos comprobar si el comando es correcto. Para que sea correcto, debe coincidir con cualquiera de las constantes cmdx.

Para comprobar si el comando es correcto, utilizo la función “strcmp()” que compara dos strings y devuelve un valor de tipo entero. En el caso de que los strings sean iguales devuelve un 0 (!strcmp()).

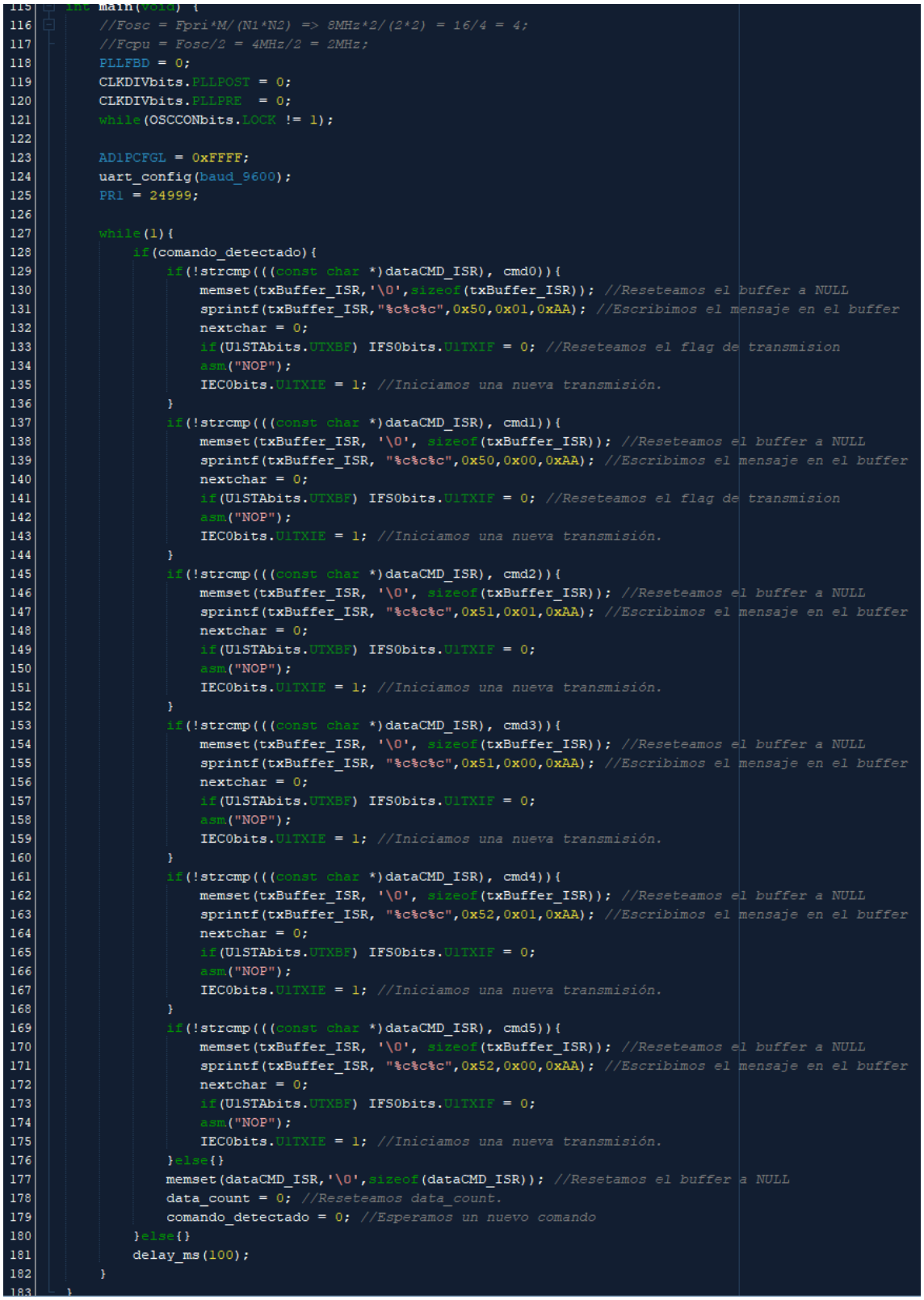
Si los dos strings son iguales, y mediante la función “memset()”, reseteamos todo el buffer txBuffer_ISR() a NULL para poder guardar un nuevo mensaje. Una vez resetado, utilizo la función “sprintf()” para guardar el buffer el mensaje a mandar.

El contenido del mensaje dependerá del comando añadido por el usuario. Todos los mensajes se envían en hexadecimal y su contenido se rige por la tabla dada en el enunciado de la práctica.

Además, se resetea la variable nextchar, reseteamos el flag de transmisión en el caso de que el buffer de transmisión esté lleno y tras esperar un ciclo de reloj iniciamos una nueva transmisión.

En el caso de que el comando añadido no fuera correcto, se resetea dataCMD_ISR, data_count y comando_detectado.

```
while(1){
    if(comando_detectado){
        if(!strcmp(((const char *)dataCMD_ISR), cmd0)){
            memset(txBuffer_ISR,'\0',sizeof(txBuffer_ISR)); //Reseteamos el buffer a NULL
            sprintf(txBuffer_ISR,"%c%c%c",0x50,0x01,0xAA); //Escribimos el mensaje en el buffer
            nextchar = 0;
            if(U1STAbits.UTXBF) IFS0bits.U1TXIF = 0; //Reseteamos el flag de transmision
            asm("NOP");
            IEC0bits.U1TXIE = 1; //Iniciamos una nueva transmisión.
        }
```



```
int main(void) {
    //Fosc = Fpri*M/(N1*N2) => 8MHz*2/(2*2) = 16/4 = 4;
    //Fcpu = Fosc/2 = 4MHz/2 = 2MHz;
    PLLFBD = 0;
    CLKDIVbits.PLLPOST = 0;
    CLKDIVbits.PLLPRE  = 0;
    while(OSCCONbits.LOCK != 1);

    AD1PCFGL = 0xFFFF;
    uart_config(baud_9600);
    PR1 = 24999;

    while(1){
        if(comando_detectado){
            if(!strcmp(((const char *)dataCMD_ISR), cmd0)){
                memset(txBuffer_ISR,'\0',sizeof(txBuffer_ISR)); //Reseteamos el buffer a NULL
                sprintf(txBuffer_ISR,"%c%c%c",0x50,0x01,0xAA); //Escribimos el mensaje en el buffer
                nextchar = 0;
                if(U1STAbits.UTXBF) IFS0bits.U1TXIF = 0; //Reseteamos el flag de transmision
                asm("NOP");
                IEC0bits.U1TXIE = 1; //Iniciamos una nueva transmisión.
            }
            if(!strcmp(((const char *)dataCMD_ISR), cmd1)){
                memset(txBuffer_ISR, '\0', sizeof(txBuffer_ISR)); //Reseteamos el buffer a NULL
                sprintf(txBuffer_ISR, "%c%c%c",0x50,0x00,0xAA); //Escribimos el mensaje en el buffer
                nextchar = 0;
                if(U1STAbits.UTXBF) IFS0bits.U1TXIF = 0; //Reseteamos el flag de transmision
                asm("NOP");
                IEC0bits.U1TXIE = 1; //Iniciamos una nueva transmisión.
            }
            if(!strcmp(((const char *)dataCMD_ISR), cmd2)){
                memset(txBuffer_ISR, '\0', sizeof(txBuffer_ISR)); //Reseteamos el buffer a NULL
                sprintf(txBuffer_ISR, "%c%c%c",0x51,0x01,0xAA); //Escribimos el mensaje en el buffer
                nextchar = 0;
                if(U1STAbits.UTXBF) IFS0bits.U1TXIF = 0;
                asm("NOP");
                IEC0bits.U1TXIE = 1; //Iniciamos una nueva transmisión.
            }
            if(!strcmp(((const char *)dataCMD_ISR), cmd3)){
                memset(txBuffer_ISR, '\0', sizeof(txBuffer_ISR)); //Reseteamos el buffer a NULL
                sprintf(txBuffer_ISR, "%c%c%c",0x51,0x00,0xAA); //Escribimos el mensaje en el buffer
                nextchar = 0;
                if(U1STAbits.UTXBF) IFS0bits.U1TXIF = 0;
                asm("NOP");
                IEC0bits.U1TXIE = 1; //Iniciamos una nueva transmisión.
            }
            if(!strcmp(((const char *)dataCMD_ISR), cmd4)){
                memset(txBuffer_ISR, '\0', sizeof(txBuffer_ISR)); //Reseteamos el buffer a NULL
                sprintf(txBuffer_ISR, "%c%c%c",0x52,0x01,0xAA); //Escribimos el mensaje en el buffer
                nextchar = 0;
                if(U1STAbits.UTXBF) IFS0bits.U1TXIF = 0;
                asm("NOP");
                IEC0bits.U1TXIE = 1; //Iniciamos una nueva transmisión.
            }
            if(!strcmp(((const char *)dataCMD_ISR), cmd5)){
                memset(txBuffer_ISR, '\0', sizeof(txBuffer_ISR)); //Reseteamos el buffer a NULL
                sprintf(txBuffer_ISR, "%c%c%c",0x52,0x00,0xAA); //Escribimos el mensaje en el buffer
                nextchar = 0;
                if(U1STAbits.UTXBF) IFS0bits.U1TXIF = 0;
                asm("NOP");
                IEC0bits.U1TXIE = 1; //Iniciamos una nueva transmisión.
            }else{}
            memset(dataCMD_ISR,'\0',sizeof(dataCMD_ISR)); //Reseteamos el buffer a NULL
            data_count = 0; //Reseteamos data_count.
            comando_detectado = 0; //Esperamos un nuevo comando
        }else{}
        delay_ms(100);
    }
}
```

Interrupciones:

_T1Interrupt: Resetea el timer.

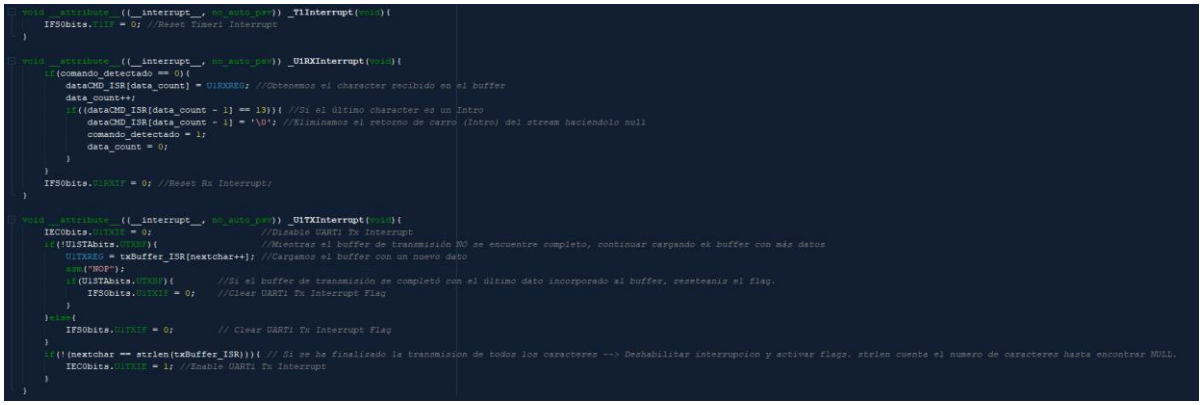
_U1RXInterrupt: Lo primero que hacemos al entrar en esta interrupción es que la variable comando_detectado sea igual a 0 para dejar de guardar caracteres en el caso de que el usuario pulse la tecla Intro y salir de la interrupción. En el caso de que sea 0, vamos rellenando el

buffer dataCMD_ISR con los caracteres que vamos recibiendo. Cada vez que se añade un carácter se comprueba mediante un if si el último valor en dataCMD_ISR es igual a 13, que es el número ascii equivalente al retorno de carro-Intro. Si efectivamente el usuario ha pulsado Intro, eliminamos ese valor para que no afecte a la hora de comprobar si el comando es correcto o no, ponemos la variable comando_detectado a 1 y reseteamos data_count junto al flag de recepción.

_U1TXInterrupt: Lo primero, tras desactivar la interrupción de Tx, comprobamos el buffer de transmisión, en el caso de que no se encuentre lleno, cargamos uno de los datos almacenados en txBuffer_ISR. Cuando el buffer de transmisión se queda vacío, reseteamos el flag de Tx.

SI el buffer está vacío según se entra, desactivamos el flag.

Compruebo si se ha terminado la transmisión comparando la variable nextchar con el tamaño de txBuffer_ISR mediante la función strlen.



MIC2:

Defines:

LED_GREEN: Corresponde a uno de los LED, específicamente el LED verde.

LED_RED: Corresponde a otro de los LED, en este caso el rojo.

LED_ON_STATE y LED_OFF_STATE: Definiciones con el estado de los LEDs, siendo 1 (encendido) y 0 (apagado) respectivamente.

Baud_9600: Igual que en MIC1.

```
#define LED_GREEN LATAbits.LATA0
#define LED_RED   LATAbits.LATA1
#define LED_ON_STATE 1
#define LED_OFF_STATE 0
#define baud_9600 51
```

Variables globales:

dataCMD_ISR: Esta variable, que también podemos encontrar en MIC1, cumple la misma función que en el primer microcontrolador, ser el buffer de Rx.

txBuffer_ISR: Al igual que en MIC1, este va a ser el buffer de Tx.

BufferLoadDone: Esta variable, de tipo unsigned char, se activará cuando se haya terminado la transmisión.

AllowPrint: Esta variable, de tipo unsigned char, se activará o desactivará en función del comando que se añada. En el caso de ser “print MIC2_data” se pondrá a 1 y en caso de ser “stop MIC2_data” se pondrá a 0, habilitando la impresión por el Terminal Virtual.

Dummy: Igual que en MIC1, sigue sin estar implementado.

NextChar: Igual que en MIC1.

U1_PrintRate_ISR: Esta variable, de tipo unsigned int, sirve de contador para la frecuencia de impresión de MIC2.

Counter: Este contador, de tipo unsigned int, será la que salga e incremente en el mensaje mostrado por el Terminal Virtual.

Data_count: Misma función que MIC1, indexar el buffer de Rx (dataCMD_ISR).

Comando_detectado: Igual que en MIC1.

Cmd0, cmd1, …, cmd5: Prácticamente igual que en MIC1, salvo que ahora guarda la información con los valores hexadecimales que van junto a cada comando usando la tabla dada en el enunciado de la práctica.

```
char dataCMD_ISR[50];                          //Rx buffer
char txBuffer_ISR[200];                        //TX buffer
unsigned char BufferLoadDone = 1;
unsigned char AllowPrint = 0;                  //Ver de donde sale esto
unsigned char dummy;
unsigned int nextchar = 0;                     //Size/index variable (Used txBuffer)
unsigned int U1_PrintRate_ISR = 0;             //Aux counter for Print Rate
unsigned int counter = 0;                      //Tx print numeric counter;
volatile unsigned int data_count = 0;          //Size/index variable (Used Rx Buffer)
volatile unsigned char comando_detectado = 0;  //Detect new command

const char cmd0[50] = {0x50, 0x01, 0xAA};
const char cmd1[50] = {0x50, 0x00, 0xAA};
const char cmd2[50] = {0x51, 0x01, 0xAA};
const char cmd3[50] = {0x51, 0x00, 0xAA};
const char cmd4[50] = {0x52, 0x01, 0xAA};
const char cmd5[50] = {0x52, 0x00, 0xAA};
```

Main:

La primera parte del main, antes de bucle infinito, con el añadido de los pines de salida de los LED verde y rojo. En mi caso he decidido utilizar los pines A0 para el verde y A1 para el rojo.

Dentro del bucle infinito tampoco hay mucha diferencia respecto a MIC1, se comprueba de la misma manera que el mensaje recibido sea igual a alguno de los elementos guardados en las constantes cmdx.

Si dataCMD_ISR es igual a:

- Cmd0: El LED verde pasa a estar encendido.
- Cmd1: El LED verde pasa a estar apagado.
- Cmd2: El LED rojo pasa estar encendido.
- Cmd3: El LED rojo pasa estar apagado.
- Cmd4: Se activa la variable que nos permite imprimir por el Terminal Virtual (AllowPrint).
- Cmd5: Se desactiva la variable que nos permite imprimir por el Terminal Virtual (AllowPrint).

Tras comprobar los comandos, la variable dataCMD_ISR se resetea. También se resetea la variable data_count.

Ahora queda imprimir el mensaje en el caso de que realmente se cumplan los requisitos. Para poder imprimir, AllowPrint debe estar activado (el comando “print MIC2_data” se ha añadido correctamente por la terminal), la variable BufferLoadDone debe estar a 1 y el tanto el buffer como el Shift Register están vacíos.

Si se cumplen debemos imprimir a 5Hz. Utilizando la formula T = 1 / F, siendo F = 5Hz, sacamos que cada mensaje debe imprimirse cada 200ms. Utilizo un delay_ms de 10ms por lo que sacamos que U1_PrintRate_ISR debe ser >= 20 (10ms * 20 veces = 200ms).

Cuando U1_PrintRate_ISR cumple ese requisito, comprobamos el valor de counter para poder resetearlo en el caso de que llegue a 100. Además reseteo el buffer de Tx (txBuffer_ISR) mediante la función “memset()”, cargo el nuevo mensaje, compuesto por el string “IMPRIMIENDO DATOS: counter” mediante la función “sprintf()”.

Reseteamos las variables nextchar, BufferLoadDone y U1_PrintRate_ISR para preparar un nuevo mensaje. Además reseteo el flag de Tx interrupt en el caso de que el buffer esté lleno y tras esperar un ciclo de reloj, activamos la interrupción de Tx.

```
int main(void) {
    //Fosc = Fpri*M/(N1*N2) => 8MHz*2/(2*2) = 16/4 = 4;
    //Fcpu = Fosc/2 = 4MHz/2 = 2MHz;
    PLLFBD = 0;
    CLKDIVbits.PLLPOST = 0;
    CLKDIVbits.PLLPRE  = 0;
    while(OSCCONbits.LOCK != 1);

    AD1PCFGL = 0xFFFF;
    TRISAbits.TRISA0 = 0;    //Pin A0 -> D1 output (Green LED)
    TRISAbits.TRISA1 = 0;    //Pin A1 -> D2 output (Red LED)
    uart_config(baud_9600);
    PR1 = 24999;

    while(1){
        if(!strcmp(((const char *)dataCMD_ISR), cmd0)) LED_GREEN = LED_ON_STATE;
        if(!strcmp(((const char *)dataCMD_ISR), cmd1)) LED_GREEN = LED_OFF_STATE;
        if(!strcmp(((const char *)dataCMD_ISR), cmd2)) LED_RED = LED_ON_STATE;
        if(!strcmp(((const char *)dataCMD_ISR), cmd3)) LED_RED = LED_OFF_STATE;
        if(!strcmp(((const char *)dataCMD_ISR), cmd4)) AllowPrint = 1;
        if(!strcmp(((const char *)dataCMD_ISR), cmd5)) AllowPrint = 0;
        else{}
        memset(dataCMD_ISR,'\0',sizeof(dataCMD_ISR)); //Resetamos el buffer a NULL
        data_count = 0; //Reseteamos data_count.

        if((AllowPrint)&&(BufferLoadDone)&&(U1STAbits.TRMT)){
            //T = 1 / 5Hz = 0,2 segundos = 200ms.
            //delay_ms = 10ms; U1_PrintRate_ISR = 20; 10ms * 20 times = 200ms.
            if(U1_PrintRate_ISR++ >= 20){ // 5Hz printing
                if(counter == 100){
                    counter = 0;
                }
                memset(txBuffer_ISR, '\0',sizeof(txBuffer_ISR));
                sprintf(txBuffer_ISR, "IMPRIMIENDO DATOS: %d \r\n", counter++);
                nextchar = 0;
                BufferLoadDone = 0;
                U1_PrintRate_ISR = 0;
                if(U1STAbits.UTXBF){
                    IFS0bits.U1TXIF = 0;
                }
                asm("NOP");
                IEC0bits.U1TXIE = 1;
            }
        }
        delay_ms(10);
    }
    return 0;
}
```

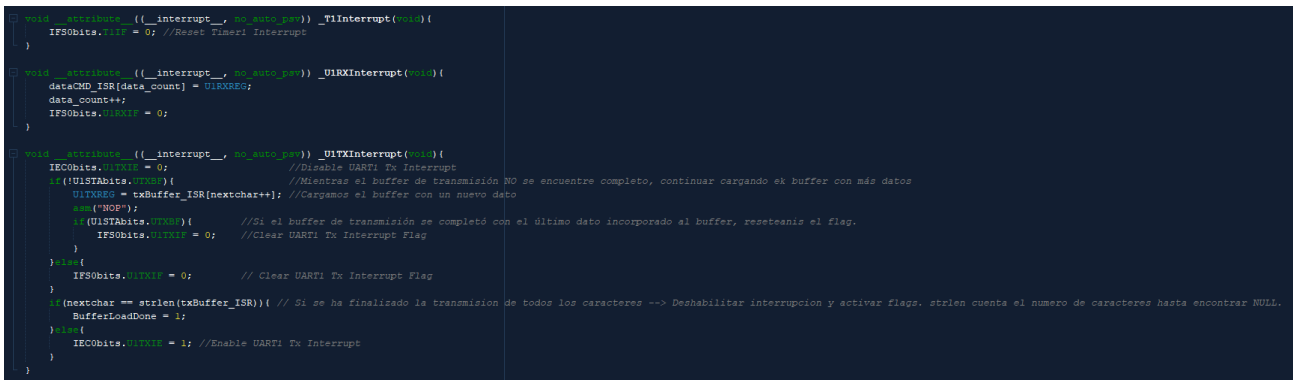
Interrupciones:

Las interrupciones son prácticamente iguales a la de MIC1:

_T1Interrupt: No cambia.

_U1RXInterrupt: No necesitamos comprobar si el usuario ha pulsado Intro ya que el mensaje llega de MIC1, no del usuario.

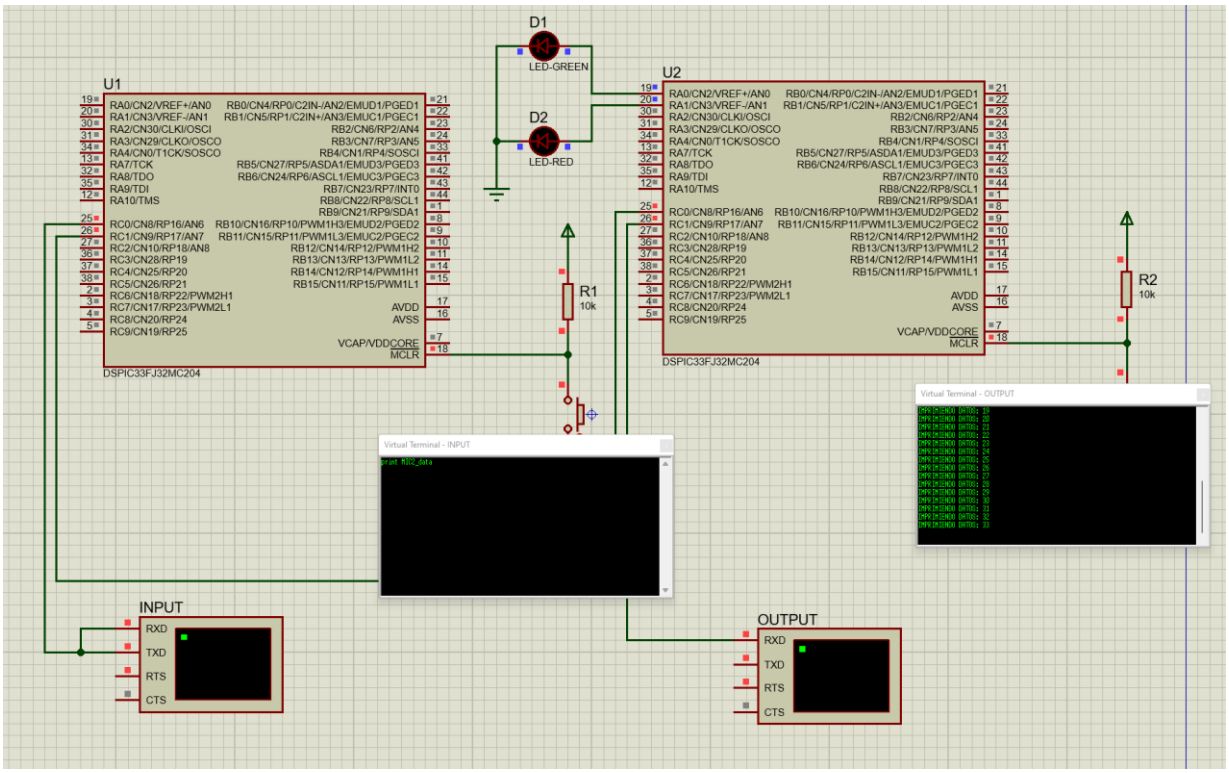
_U1TXInterrupt: El if donde se compara nextchar con el tamaó de txBuffer_ISR cambia, esta vez activamos BufferLoadDone para poder imprimir si se ha terminado la transmisión. Activamos la interrupción de Tx sino se ha terminado la transmisión.



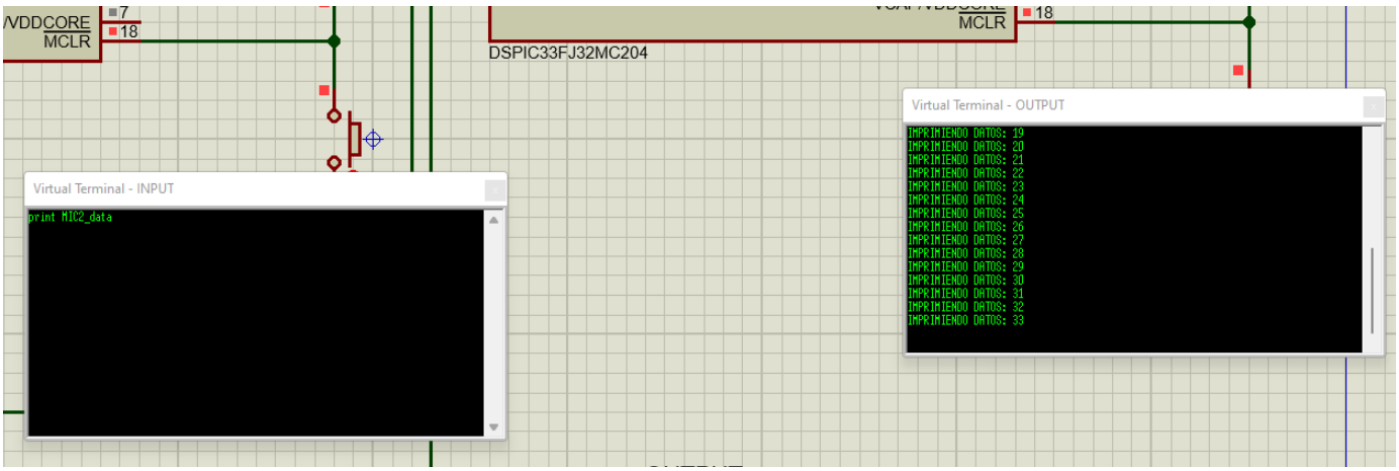
```
void __attribute__((__interrupt__, no_auto_psv)) _T1Interrupt(void){
    IFS0bits.T1IF = 0; //Reset Timer1 Interrupt
}

void __attribute__((__interrupt__, no_auto_psv)) _U1RXInterrupt(void){
    dataCMD_ISR[data_count] = U1RXREG;
    data_count++;
    IFS0bits.U1RXIF = 0;
}

void __attribute__((__interrupt__, no_auto_psv)) _U1TXInterrupt(void){
    IEC0bits.U1TXIE = 0;                    //Disable UART1 Tx Interrupt
    if(!U1STAbits.UTXBF){                   //Mientras el buffer de transmisión NO se encuentre completo, continuar cargando ek buffer con más datos
        U1TXREG = txBuffer_ISR[nextchar++]; //Cargamos el buffer con un nuevo dato
        asm("NOP");
        if(U1STAbits.UTXBF){        //Si el buffer de transmisión se completó con el último dato incorporado al buffer, reseteanis el flag.
            IFS0bits.U1TXIF = 0;    //Clear UART1 Tx Interrupt Flag
        }
    }else{
        IFS0bits.U1TXIF = 0;        // Clear UART1 Tx Interrupt Flag
    }
    if(nextchar == strlen(txBuffer_ISR)){ // Si se ha finalizado la transmision de todos los caracteres --> Deshabilitar interrupcion y activar flags. strlen cuenta el numero de caracteres hasta encontrar NULL.
        BufferLoadDone = 1;
    }else{
        IEC0bits.U1TXIE = 1; //Enable UART1 Tx Interrupt
    }
}
```

Resultados

Lo primero que pruebo es el comando “print MIC2_data”:

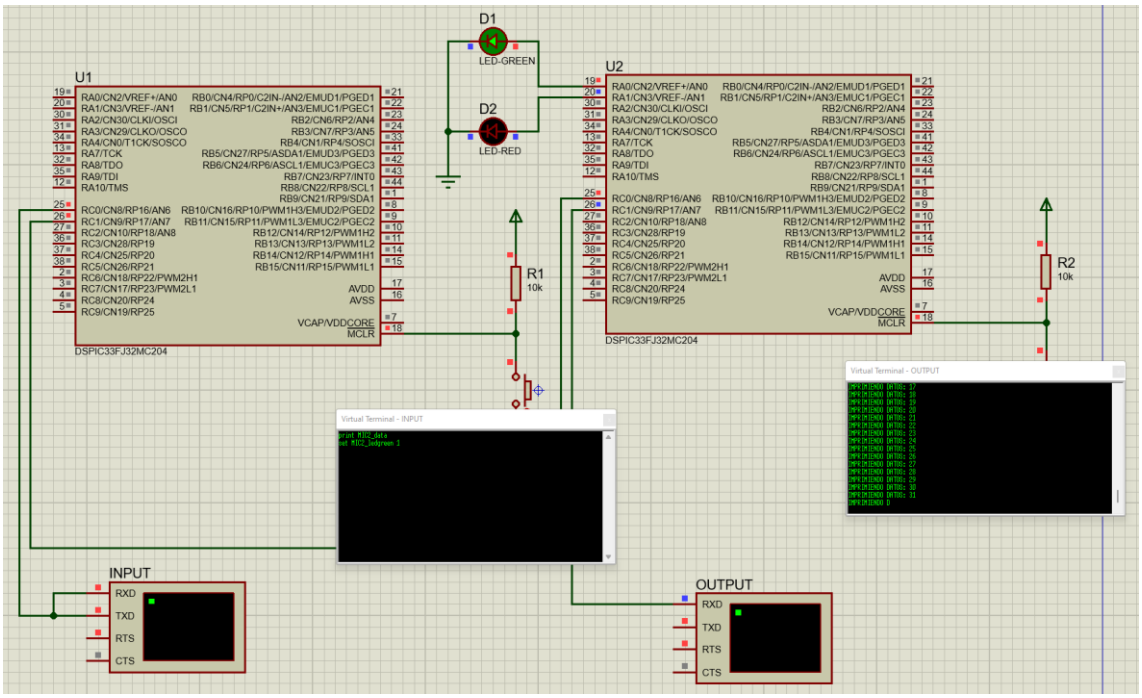


En las capturas no se ven bien las letras de los terminales, en todos los resultados dejaré el esquemático completo junto a una ampliación de los terminales.

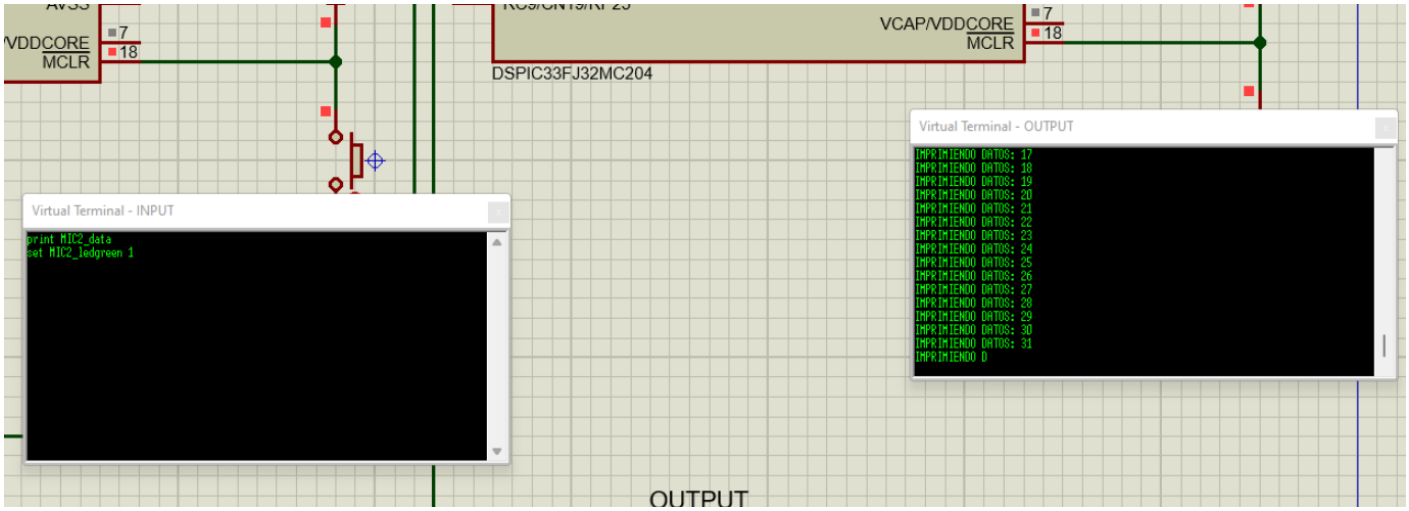


Podemos ver que el comando, al ser correcto, funciona según lo pedido.

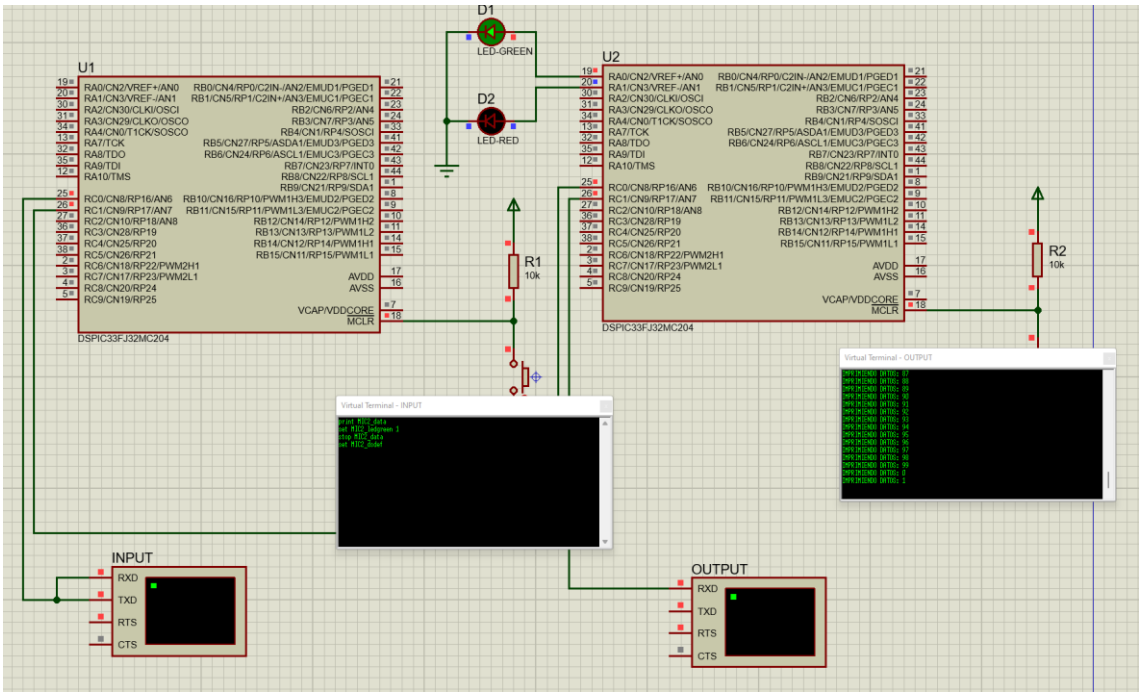
El siguiente comando que compruebo es “set MIC2_ledgreen 1” para encender el led:



En el esquemático podemos ver que el led verde se enciende correctamente. Si nos acercamos a las terminales podemos ver que el comando es el pedido. Como nota adicional, parece que al hacer la captura uno de los mensajes estaba en medio de la impresión.

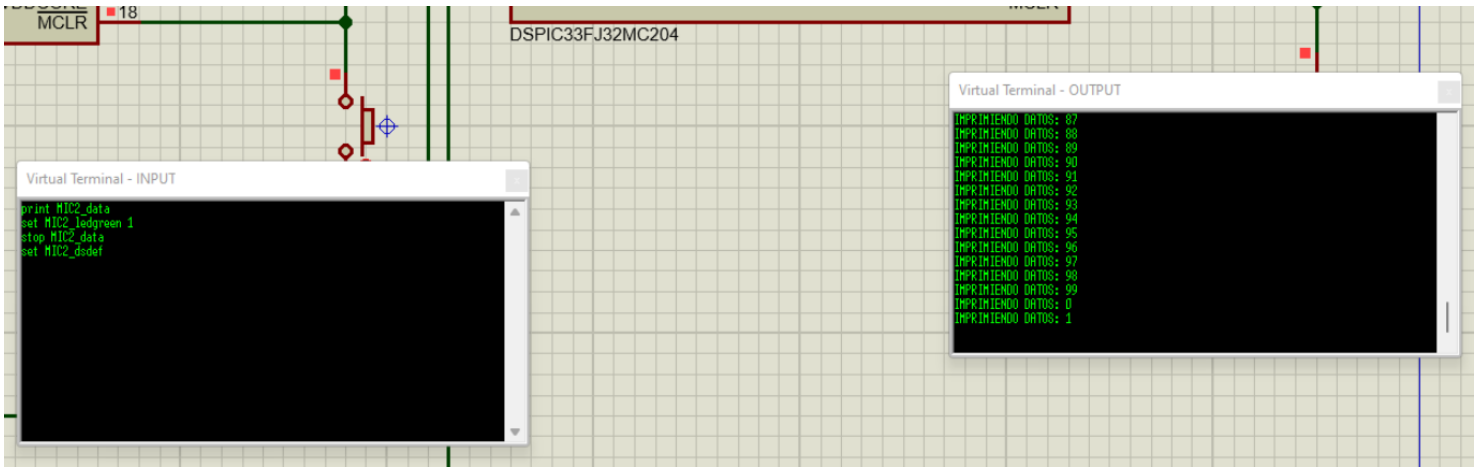


A continuación quería probar el comando “stop MIC2_data”:



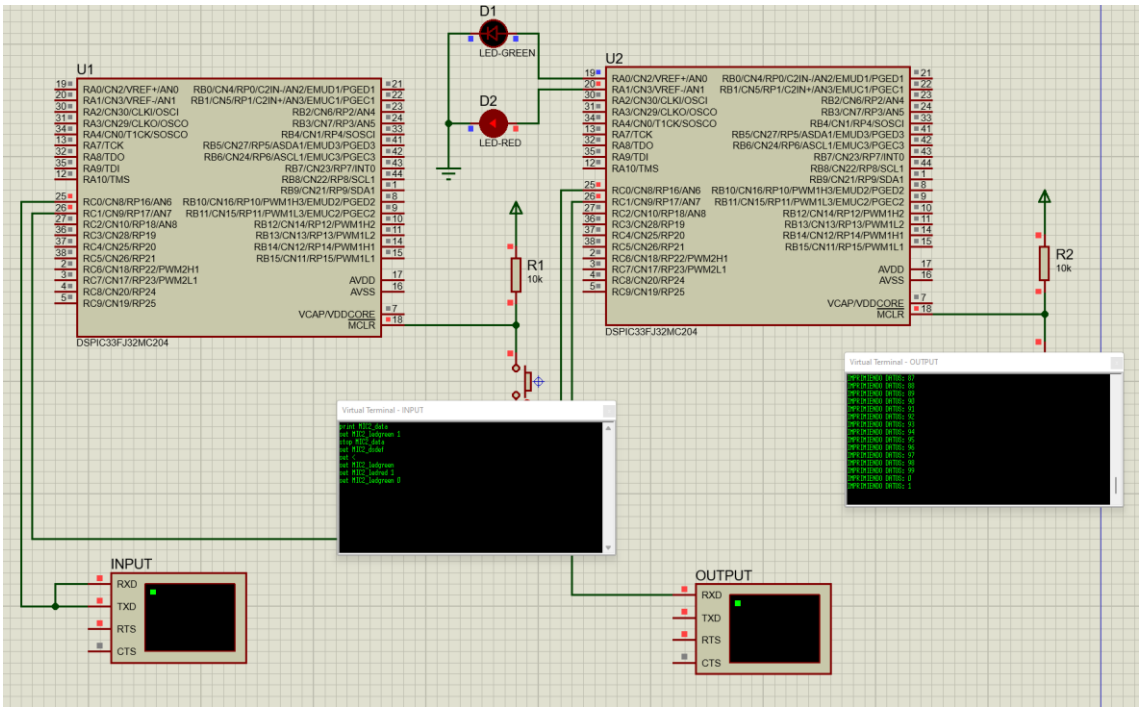
Aquí se pueden ver varías cosas, la primera de ellas es que el comando de stop funciona correctamente. Además, he esperado para esperar el momento justo en el que contador resetea al llegar a 99.

Iba a comprobar que el LED rojo se encendiera, pero se me había olvidado hacer la captura, por lo que lo cancelo. En la siguiente entraremos más en detalle ya que me equivoco varias veces.



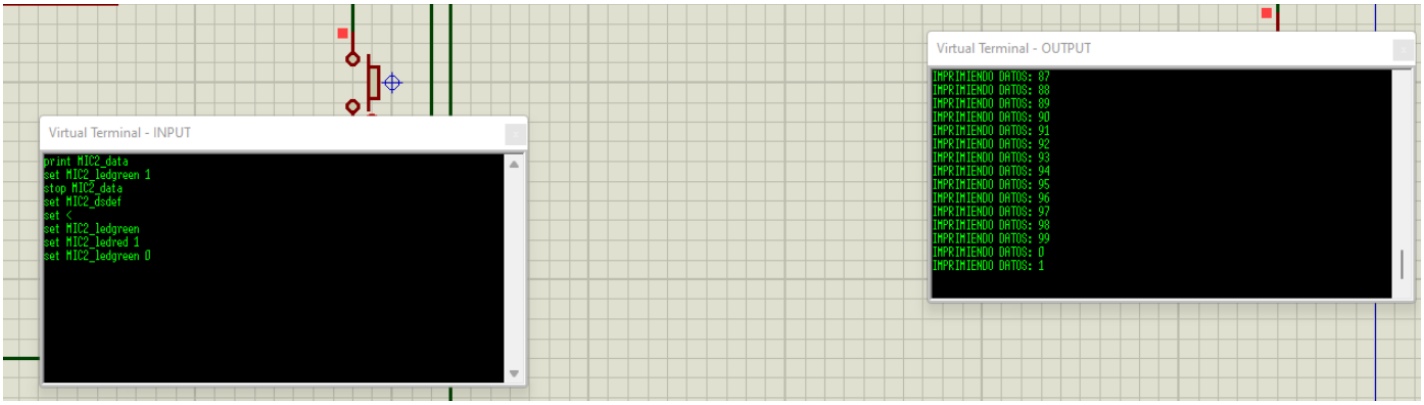
Se puede ver más de cerca que efectivamente funciona el comando de stop y que resetea.

El siguiente comando que compruebo es “set MIC2_ledgreen 0 y set MIC2_ledred 1”:



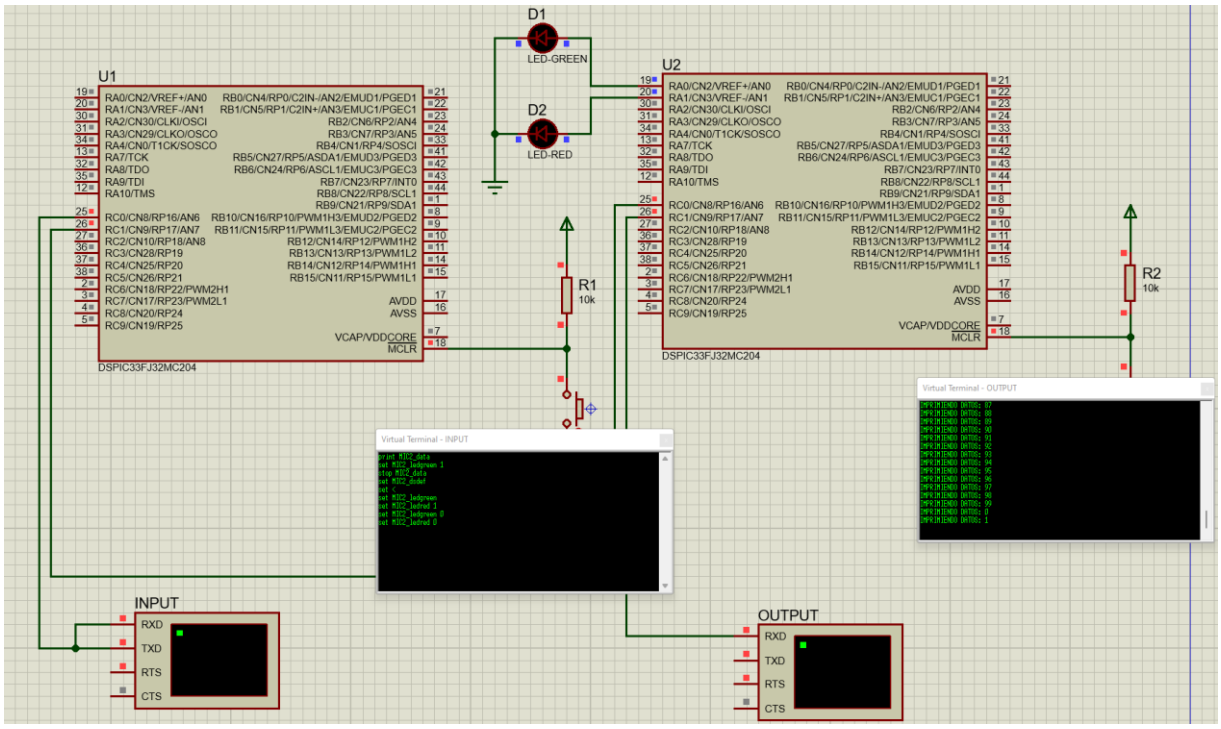
Se me olvidó hacer captura de los dos LED encendidos juntos. En la terminal de comandos podemos ver que me equivoco varias veces. Todas las veces sin querer, primero pongo un símbolo que no quería, luego se me olvida dar un valor al comando de “set MIC2_ledgreen 0” y después me confundo de LED, enciendo el LED rojo en vez de apagar el verde.

Tras todo esto al final consigo apagar el LED verde y a la vez encender el LED rojo.

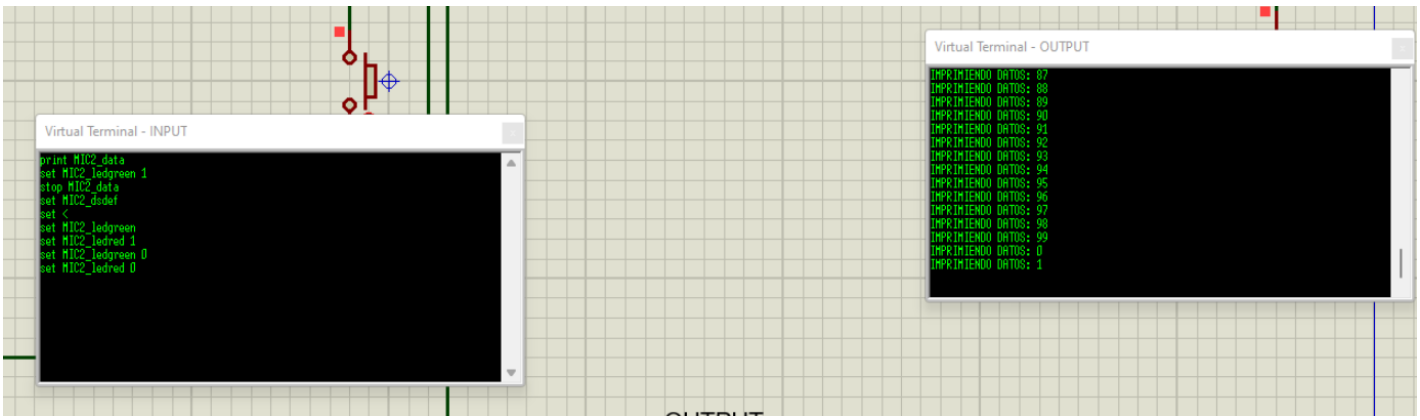


Podemos ver que la impresión de datos sigue parada.

El último comando para probar es “set MIC2_ledred 0”:



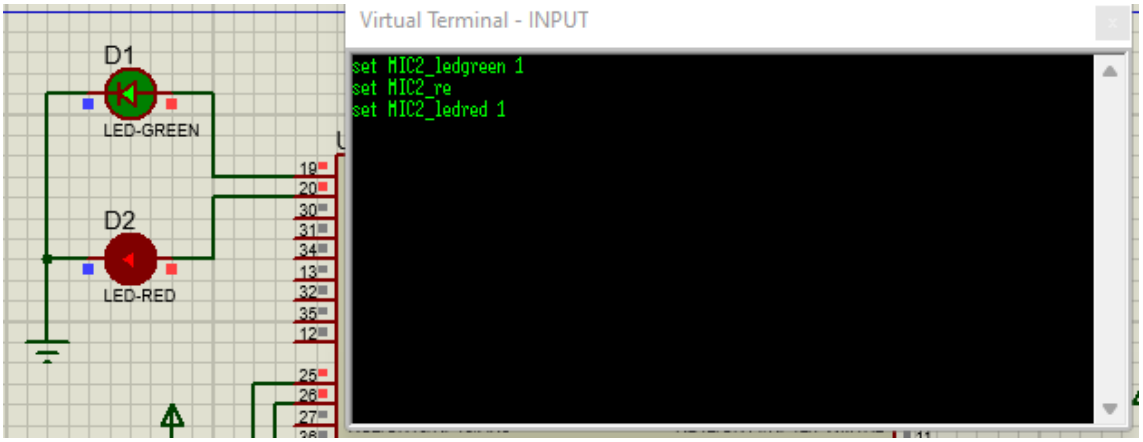
Se puede ver como el LED rojo está apagado.



Los resultados son los esperados.

LED rojo y LED verde encendidos a la vez:

Como paso adicional, ya que en las capturas no se ve por que me equivoqué, añado una captura con los dos LED encendidos.



Como podemos ver no solo me equivoco al poner uno de los comandos, si no que además los LED están encendidos a la vez.

Conclusión:

Estoy muy contento con los resultados, ha sido muy satisfactorio sacar la práctica y ver como funcionaban las cosas. A su vez creo haber adquirido algunos de los conceptos en profundidad y he aprendido nuevos conceptos.

El mayor impedimento que encontré fue añadir los ficheros .hex de la build del código. Al final tuve que hacer MIC1 y MIC2 proyectos distintos para hacer la build. Entregaré un único proyecto comentando el código de uno de los ficheros.