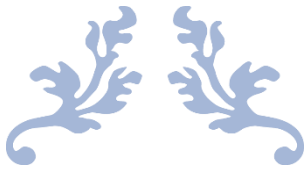


PRÁCTICA 4

Por Luis Díaz del Río

Introducción

En esta práctica se nos pide implementar los módulos Output Compare, Input Capture y SPI.

Se pide:

Programar el módulo OC y generar dos señales de PWM, con un duty cycle fijo del 40% para la primera señal de PWM y 70% para la segunda señal de PWM. El Periodo T de las señales debe ser exactamente el mismo, el cual queda a libre elección del alumno/a.

En mi caso el periodo T es de 5ms.

Además, se deben leer las dos señales de PWM anteriores mediante el módulo IC. Se debe imprimir por pantalla el ancho de pulso activo de cada una de las señales capturadas, en la escala de tiempo que mejor se ajuste.

En cuanto a las comunicaciones SPI, se debe configurar para establecer comunicación con una memoria EEPROM, en mi caso Microchip 25LC256 (recomendada en el enunciado). Se deberá cargar un vector de 10 posiciones con los valores que el alumno/a considere. Una vez cargados todos los datos, se deberá ejecutar una rutina de chequeo disparada por una tecla recibida por el puerto Rx de la UART y deberá comprobar que los valores almacenados son correctos. Se deberá imprimir cada uno de los valores almacenados y esperados para cada posición de memoria durante la ejecución de esta rutina.

Por último, se deberá encender un LED VERDE si todos los datos son correctos (Test Pass). Se deberá encender un LED ROJO si alguno de los datos es incorrecto (Test Fail) al final de la rutina de chequeo.

He decidido implementar todos los módulos en un mismo programa.

En mi caso utilizo una frecuencia de CPU de 2Mhz.

Los comandos son los siguientes:

Tecla “+”: Aumenta OC2.

Tecla “- “: Reduce OC2.

Tecla “p”: Aumenta OC1.

Tecla “m”: Reduce OC1.

Tecla “i”: Inicia la rutina de chequeo de la memoria EEPROM.

Defines

LED_GREEN y **LED_RED**: Los LED verde y rojo respectivamente, definidos así para claridad en el código.

LED_ON_STATE y LED_OFF_STATE: Estados para encender y apagar los LED respectivamente.

baud_9600: Frecuencia de transmisión para una frecuencia de 2Mhz.

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

Variables

DataCMD_ISR: Array de chars (string) para la recepción de los mensajes metidos por el Terminal Virtual.

TxBuffer: Array de chars (string) para el envío de mensajes a la Terminal Virtual.

TxBuffer_ISR: Array de chars (string) para el envío de mensajes a la Terminal Virtual (Por interrupciones, no lo he utilizado al final).

Tiempo_real_IC1 y **Tiempo_real_IC2:** De tipo double, utilizado para guardar el tiempo final tras los cálculos realizamos en el main.

NextChar: Para indexar en la interrupción de Tx.

Time_Pulso_IC1 y **Time_Pulso_IC2:** Guardamos el tiempo de pulso en las interrupciones de IC1 e IC2.

```
char dataCMD_ISR[50];               //UART ISR in
char txBuffer[200];                 //UART out
char txBuffer_ISR[200];             //UART ISR out
double tiempo_real_IC1      = 0.0; //Final time calc result
double tiempo_real_IC2      = 0.0;
unsigned int nextChar       = 0;   //Index tcBuffer_ISR;
unsigned int time_pulso_IC1 = 0;
unsigned int time_pulso_IC2 = 0;
unsigned int rise_pulso_IC1 = 0;
unsigned int rise_pulso_IC2 = 0;
unsigned int duty_OC1       = 4000;
unsigned int duty_OC2       = 4000;
unsigned int pulso1         = 0;
unsigned int pulso2         = 0;
unsigned int flag           = 0;
unsigned int flagMemory     = 0;
unsigned int testResult = 0;
unsigned char memoryBuffer[10] = {0x10, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09};
unsigned char memoryAddress[10] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A};
unsigned char badMemory_data[10] = {0x10, 0x11, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09};
unsigned char memory_data[10];
```

Rise_Pulso_IC1 y **Rise_Pulso_IC2**: Guardamos el tiempo en el flanco de subida en las interrupciones de IC1 e IC2.

Duty_OC1 y **Duty_OC2:** Almacena el ciclo de trabajo de OC1 y OC2.

Pulso1 y **Pulso**2: Para detectar una subida o bajada de flanco de reloj en las interrupciones de IC1 e IC2.

Flag: Para detectar cuando sacar por el Terminal Virtual el ancho de pulso del PWM.

FlagMemory: Para detectar cuando empezar la rutina de chequeo de la EEPROM.

TestResult: Para saber cuando debe encenderse el LED rojo o el LED verde tras la rutina de chequeo de la EEPROM.

MemoryBuffer: Array con los valores con los que se inicializará la memoria EEPROM.

MemoryAddress: Array con los espacios de memoria que serán inicializados con los valores dentro de MemoryBuffer.

badMemory_data: Array con un elemento incorrecto para comprobar si se enciende el LED rojo en caso de que haya un dato mal. Exclusivamente para pruebas.

Memory_data: Array donde se almacenarán los valores guardados en la EEPROM.

Funciones:

Uart_config: Configura los pines los pines de recepción y transmisión, los registros de la UART, la velocidad de transmisión y recepción de datos y las interrupciones de la UART.

Spi_config: Configura el modulo SPI, tanto los pines de entrada y salida como los registros.

Output_compare_config: Configura el módulo Output Compare.

Input_capture_config: Configura el módulo Input Capture

Spi_write: Función para leer y escribir datos en la EEPROM.

SendChar: Envía in carácter al Terminal Virtual.

sendString: Envía un array de chars (string) al Terminal Virtual.

Delay_ms: Espera una cantidad de tiempo elegida por el usuario mediante la ejecución de la instrucción en ensamblador “NOP”

```
void uart_config (unsigned int baud){
    //UART PORT/PIN COMMUNICATION ASSIGMENT
    TRISBbits.TRISB2    = 1;  //Pin RB2 input (digital).
    RPINR18bits.U1RXR   = 2; //Pin RB2 connected to recepcion port of UART1.
    RPOR1bits.RP3R      = 3;  //Pin RB3 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;

    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
}
```

```
void spi_config(void){
    TRISCbits.TRISC0 = 1; //MISO asignado a RC0 (PIN 25)
    TRISCbits.TRISC1 = 0; //MOSI asignado a RC1 (PIN 26)
    TRISCbits.TRISC2 = 0; //SCK  asignado a RC2 (PIN 27)
    TRISCbits.TRISC3 = 0; //CS   asignado a RC3 (PIN 36)

    LATCbits.LATC1 = 0;

    RPINR20bits.SDI1R = 16; //RC0 trabajando como MISO (entrada);
    RPOR8bits.RP17R = 7; //(00111); RC1 es MOSI (salida);
    RPOR9bits.RP18R = 8; //(0100); RC2 es SCK (salida);

    SPI1STATbits.SPIEN = 0;
    SPI1STATbits.SPISIDL = 0;
    SPI1STATbits.SPIROV = 0;
    SPI1STATbits.SPITBF = 0;
    SPI1STATbits.SPIRBF = 0;

    SPI1CON1bits.DISSCK = 0;
    SPI1CON1bits.DISSDO = 0;
    SPI1CON1bits.MODE16 = 0;
    SPI1CON1bits.SMP = 0;

    SPI1CON1bits.CKP = 0;
    SPI1CON1bits.CKE = 0;

    SPI1CON1bits.SSEN = 0;
    SPI1CON1bits.MSTEN = 1;

    SPI1CON1bits.PPRE = 1;
    SPI1CON1bits.SPRE = 6;

    SPI1CON2bits.FRMEN = 0;
    SPI1CON2bits.SPIFSD = 0;
    SPI1CON2bits.FRMPOL = 0;
    SPI1CON2bits.FRMDLY = 0;

    IFS0bits.SPI1IF = 0;
    IFS0bits.SPI1EIF = 0;
    IEC0bits.SPI1IE = 0;
    IEC0bits.SPI1EIE = 0;
    IPC2bits.SPI1IP = 6;
    IPC2bits.SPI1EIP = 6;

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

```c
void output_compare_config(void){
    /* Configuración de pines OC */
    TRISCbits.TRISC5   = 0; //Pin configurado como salida
    TRISCbits.TRISC6   = 0;
    RPOR10bits.RP21R   = 0x12; //OC1 conectado al pin RP21(RC0)
    RPOR11bits.RP22R   = 0x13; //OC2 conectado al pin RP22 (RC1)

    /* Configurar modulo OC */
    OC1CONbits.OCM    = 0; //Deshabilitar PWM
    OC1CONbits.OCTSEL = 0; //Trabajamos con el Timer2
    //OC1CONbits.OCM    = 6; //Modo PWM sin pin de fallo
    /*DutyOC1 = (Ton/T) = (2/5)*100 = 40%*/
    OC1R  = 4000;
    OC1RS = 4000;

    OC2CONbits.OCM    = 0; //Modulo OC2 inicialmente deshabilitado
    OC2CONbits.OCTSEL = 0; //Trabajamos con el Timer2
    //OC2CONbits.OCM    = 6; //Modo PWM sin pin de fallo
    OC2R  = 7000; //Solo registro de lectura
    OC2RS = 7000; //Ciclo de trabajo a 1ms

    //Configurar timer2
    T2CONbits.TON     = 0; //Deshabilitar Timer2
    T2CONbits.T32     = 0; //Timer2 modo de 16-bits de resolución-
    T2CONbits.TGATE   = 0; //Gated Timer mode deshabilitado
    T2CONbits.TCKPS   = 0; //Prescaler = 1;
    T2CONbits.TCS     = 0; //Seleccionar clock interno

    //Registro interrupcion
    IPC1bits.T2IP = 5; //Priotidad de nivel 5
    IFS0bits.T2IF = 0; //Reset flag
    IEC0bits.T2IE = 0; //Deshabilitar interrupción

    PR2   = 10000; //Periodo de 5ms, basado en Freq = 2MHz
    /*tiempo = PRESCALER*(reg + 1)/Fbus -> reg = ((tiempo * fbus)/PRESCALER) - 1 = ((0,005 * 4*10^6) / 1) - 1 = 10000. */

    TMR2 = 0; //Valor inicial del contador del Timer
    OC1CONbits.OCM = 6; //Mod PWM
    OC2CONbits.OCM = 6; //Modo PWM
    T2CONbits.TON = 1; //Activar timer2
}
```

```c
void input_capture_config(void){
    TRISCbits.TRISC8 = 1;   //Pin configurado como entrada
    TRISCbits.TRISC9 = 1;
    RPINR7bits.IC1R = 21;   // IC1 es el pin RC8 (RP24)
    RPINR7bits.IC2R = 22;   // IC2 es el pin RC9 (RP25)

    //Configurar IC1
    IC1CONbits.ICM    = 0;  //IC deshabilitado
    IC1CONbits.ICSIDL = 0;
    IC1CONbits.ICTMR  = 0;  //Timer3 seleccionado como base de tiempos para IC
    IC1CONbits.ICI    = 0;  // salta en cada evento
    IFS0bits.IC1IF = 0;
    IEC0bits.IC1IE = 1;  // Enable IC1 interrupts

    //Configurar IC2
    IC2CONbits.ICM    = 0;
    IC2CONbits.ICSIDL = 0;  // el IC se detiene si la CPU entra en modo reposo
    IC2CONbits.ICTMR  = 0;  // Trabajamos con Timer 3
    IC2CONbits.ICI    = 0;  // salta en cada evento
    IFS0bits.IC2IF = 0;
    IEC0bits.IC2IE = 1;  // Enable IC2 interrupts

    //Configuramos Timer3
    T3CONbits.TSIDL   = 0;
    T3CONbits.TON     = 0;
    T3CONbits.TGATE   = 0;
    T3CONbits.TCKPS   = 0; //Prescaler mas bajo = 1.
    T3CONbits.TCS     = 0;
    PR3 = 10000;

    // Activar los modulos IC1, IC2 y Timer3
    T3CONbits.TON     = 1;  //Activar Timer3
    IC1CONbits.ICM    = 3;  //Activar modulo OC1 por pin RC2
    IC2CONbits.ICM    = 3;  //Activar modulo OC2 por pin RC3
}
```

```c
unsigned char spi_write(unsigned char data){
    while(SPI1STATbits.SPITBF);
    SPI1BUF = data;
    while(!SPI1STATbits.SPIRBF);
    return SPI1BUF;
}

void sendChar(char c){
    while(U1STAbits.UTXBF);
    U1TXREG = c;
}

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

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

Las funciones de configuración son prácticamente idénticas a las explicadas en clase salvo por cambios menores para adaptarlas al proyecto y a las condiciones dadas en el enunciado de la práctica.

Main

Tras ajustar la frecuencia de la CPU, configuramos los pines A0 y A1 como pines de salida que irán conectados a los LED rojo y verde respectivamente.

Llamamos a las funciones de configuración de todos los módulos e indicamos al usuario mediante un mensaje enviado por el Terminal Virtual que la EEPROM está en proceso de inicialización.

Para inicializar la EEPROM utilizo un bucle for, en el que por cada iteración habilitamos las operaciones de escritura mediante la instrucción WREN, que tiene el formato hexadecimal 0x06. Aunque creo que esto no es necesario en cada iteración, añadirlo al bucle fue la única manera de que me funcionara correctamente.

También en cada iteración se llamada a la instrucción WRITE con formato en hexadecimal 0x02, con el que indicamos que vamos a escribir en la dirección de memoria que le especificaremos a continuación. Como la dirección de memoria debe ser de 16 bits, llamamos 2 veces a la función spi_write(), la primera vez con los 8 bits más significativos (en mi caso siempre son 0) y la segunda con los 8 bits menos significativos. En esta segunda llamada en donde utilizo el array "memoryAddress" para especificar el espacio de memoria que quiero.

Finalmente, escribo el dato que quiero dentro de la dirección de memoria especificada, sacando los datos a escribir del array "memoryBuffer".

Una vez este proceso se termina, indicamos al usuario que la EEPROM ya se ha inicializado y ponemos a 1 la variable "flag" para que al entrar al bucle infinito lo primero que salga por la Terminal Virtual sea la información de tiempos que pide el enunciado.

Dentro del bucle infinito tenemos dos if. El primero de ellos comprueba si la variable "flag" está a 1, de ser así, calcula mediante una formula el tiempo real de cada uno de los pulsos. Para hacer este cálculo, utilizo las variables "time_pulso_IC1" y "time_pulso_IC2", que tiene los valores calculados en las interrupciones "_IC1" e "_IC2" restando el tiempo entre la bajada del flanco de reloj y la subida del flanco de reloj.

Tras hacer los cálculos, sacamos por pantalla la información y ponemos la variable "flag" a 0, para que solo saque por pantalla la información de tiempos cuando pulsamos los comandos correspondientes.

El otro if comprueba que la variable "flagMemory" esté a 1. Si ese fuera el caso, ponemos la variable "flag" a 0 para evitar problemas y mediante un bucle for leemos la información almacenada en la EEPROM mediante la instrucción READ con formato hexadecimal 0x03. Le indicamos, al igual que en la escritura, los 16 bits de dirección mediante dos llamadas a la función spi_write(). Tras especificar la dirección de donde queremos leer el dato, guardo en el array "memory_data" el dato recibido de la función spi_write(), a la cual le pasamos un dato "dummy" que no es relevante. Una vez almacenado el dato en el array, sacamos por el Terminal Virtual un mensaje compuesto por el dato leído, la dirección de memoria en el que estaba el dato y el dato que debería estar en ese espacio de memoria.

Además, una vez terminado, ejecutamos la rutina de comprobación con otro bucle for, en el que comparamos los valores de "memory_data" con los de "memoryBuffer" (los datos leídos con los utilizamos para inicializar la EEPROM).

En el caso de que alguno de los valores no coincida, encendemos el LED rojo (no ha pasado el test) y ponemos a 1 la variable “testResult”. Una vez terminado el for, mediante un if se comprueba el valor de “testResult”, en el caso de que sea 0 significa que no hemos encontrado ningún error, por lo que encendemos el LED verde.

Finalmente vuelvo a poner la variable flagMemory a 0 para evitar que vuelva a ejecutarse la rutina.

Main antes del bucle infinito:

```
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);
    delay_ms(10);
    spi_config();
    delay_ms(10);
    output_compare_config();
    delay_ms(10);
    input_capture_config();
    delay_ms(10);

    sprintf(txBuffer, "Inicializando EEPROM... \r\n");
    sendString(txBuffer);

    unsigned char j;
    for(j=0; j < 10; j++){
        LATCbits.LATC3 = 0;
        delay_ms(5);
        spi_write(0x06); //Enable write operations;
        delay_ms(5);
        LATCbits.LATC3 = 1;
        LATCbits.LATC3 = 0;
        delay_ms(5);
        spi_write(0x02); //Write data instrucction;
        spi_write(0x00); //Address MSB 8 bits;
        spi_write(memoryAddress[j]); //Address LSB 8 bits;
        spi_write(memoryBuffer[j]); //Write data on Address
        delay_ms(5);
        LATCbits.LATC3 = 1;
        delay_ms(1000);
    }
    sprintf(txBuffer, "EEPROM inicializada \r\n");
    sendString(txBuffer);
    flag = 1;
```

Main dentro del bucle infinito:

```
    while(1){
        if(flag){
            tiempo_real_IC1 = 1.0*((double)time_pulso_IC1)/2000.0;
            tiempo_real_IC2 = 1.0*((double)time_pulso_IC2)/2000.0;

            sprintf(txBuffer, "TIME_IC1: %05.3fms \t TIME_IC2: %05.3fms \r\n", tiempo_real_IC1, tiempo_real_IC2);
            sendString(txBuffer);
            delay_ms(100);
            flag = 0;
        }
        if(flagMemory){
            flag = 0;
            int i;
            for(i=0; i < 10; i++){
                LATCbits.LATC3 = 0;
                delay_ms(5);
                spi_write(0x03);//Read data instruction.

                spi_write(0x00); //MSB
                spi_write(memoryAddress[i]); //LSB
                memory_data[i] = spi_write(0x00);
                sprintf(txBuffer, "Memory Value: %02X at address %04X, Expected: %02X \r\n", memory_data[i], memoryAddress[i], memoryBuffer[i]);
                sendString(txBuffer);
                delay_ms(5);
                LATCbits.LATC3 = 1;
                delay_ms(1000);
            }
            for(i=0; i < 10; i++){
                if(memory_data[i] != memoryBuffer[i]){
                    LED_RED = LED_ON_STATE;
                    testResult = 1;
                }
            }
            if(testResult == 0){
                LED_GREEN = LED_ON_STATE;
            }
            flagMemory=0;
        }
    }
    return 0;
}
```

Interrupciones

_IC1Interrupt e _IC2Interrupt: Las dos funcionan de la misma forma. Como he explicado antes en el main, detectamos y sacamos el tiempo entre la bajada y subida del flanco de reloj. De esta forma podemos más tarde mostrar los tiempos.

```
void __attribute__((__interrupt__, no_auto_psv)) _IC1Interrupt(void){
    if(pulsol == 0){
        rise_pulso_IC1 = IC1BUF;
        IC1CONbits.ICM = 2; // Capture next falling edge
        pulsol = 1;
    }else{
        time_pulso_IC1 = IC1BUF - rise_pulso_IC1;
        IC1CONbits.ICM = 3; // Capture next rising edge
        pulsol = 0;
    }
    IFS0bits.IC1IF = 0;          // Reset IC1 Interrupt
}
```

_T1Interrupt: Resetea el Timer1.

_U1RXInterrupt: Como nuestros comandos son únicamente un bit, metemos en la posición 0 del array “dataCMD_ISR” el símbolo que nos llegue de la Terminal Virtual.

Comandos:

‘+’ -> Incrementamos en 150 el ciclo de trabajo de OC2

‘-‘ -> Reducimos en 150 el ciclo de trabajo de OC2

‘p’ -> Incrementamos en 150 el ciclo de trabajo de OC1

‘m’ -> Reducimos en 150 el ciclo de trabajo de OC1

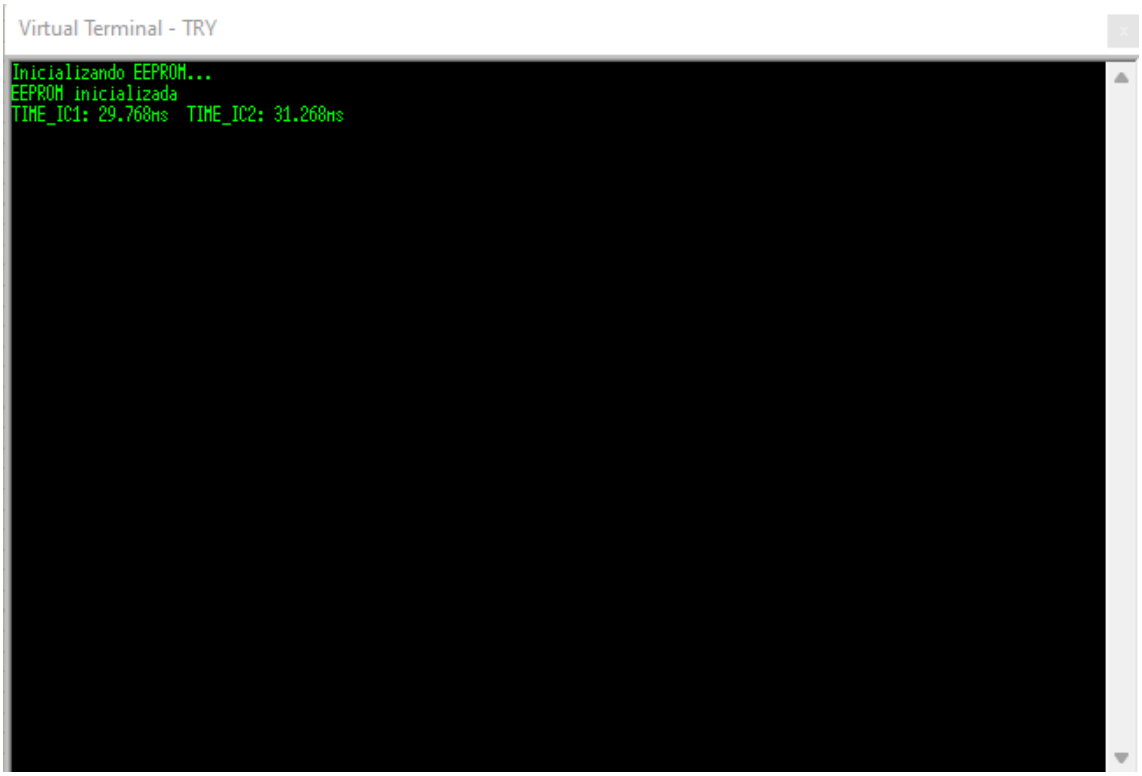
‘i’ -> Ponemos la variable “flag” a 0 y la variable “flagMemory” a 1, que inicia la rutina de chequeo.

Si alguno de los 4 primeros comandos (los que controlan OC1 y OC2), la variable “flag” se pone a 1 para imprimir los nuevos valores por el Terminal Virtual.

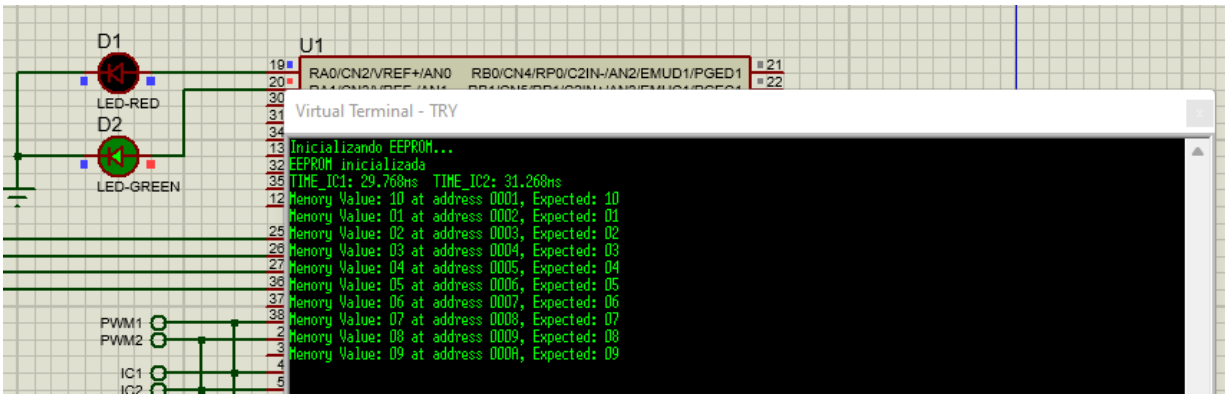
```
void __attribute__((__interrupt__, no_auto_psv)) _U1RXInterrupt(void){
    dataCMD_ISR[0] = U1RXREG;
    if(dataCMD_ISR[0] == '+') duty_OC2 = duty_OC2 + 150;
    if(dataCMD_ISR[0] == '-') duty_OC2 = duty_OC2 - 150;
    if(dataCMD_ISR[0] == 'p') duty_OC1 = duty_OC1 + 150;
    if(dataCMD_ISR[0] == 'm') duty_OC1 = duty_OC1 - 150;
    OC2RS = duty_OC2;
    OC1RS = duty_OC1;
    flag=1;
    if(dataCMD_ISR[0] == 'i'){
        flag = 0;
        flagMemory = 1;
    }

    IFS0bits.U1RXIF = 0;        // Reset Rx Interrupt
}
```

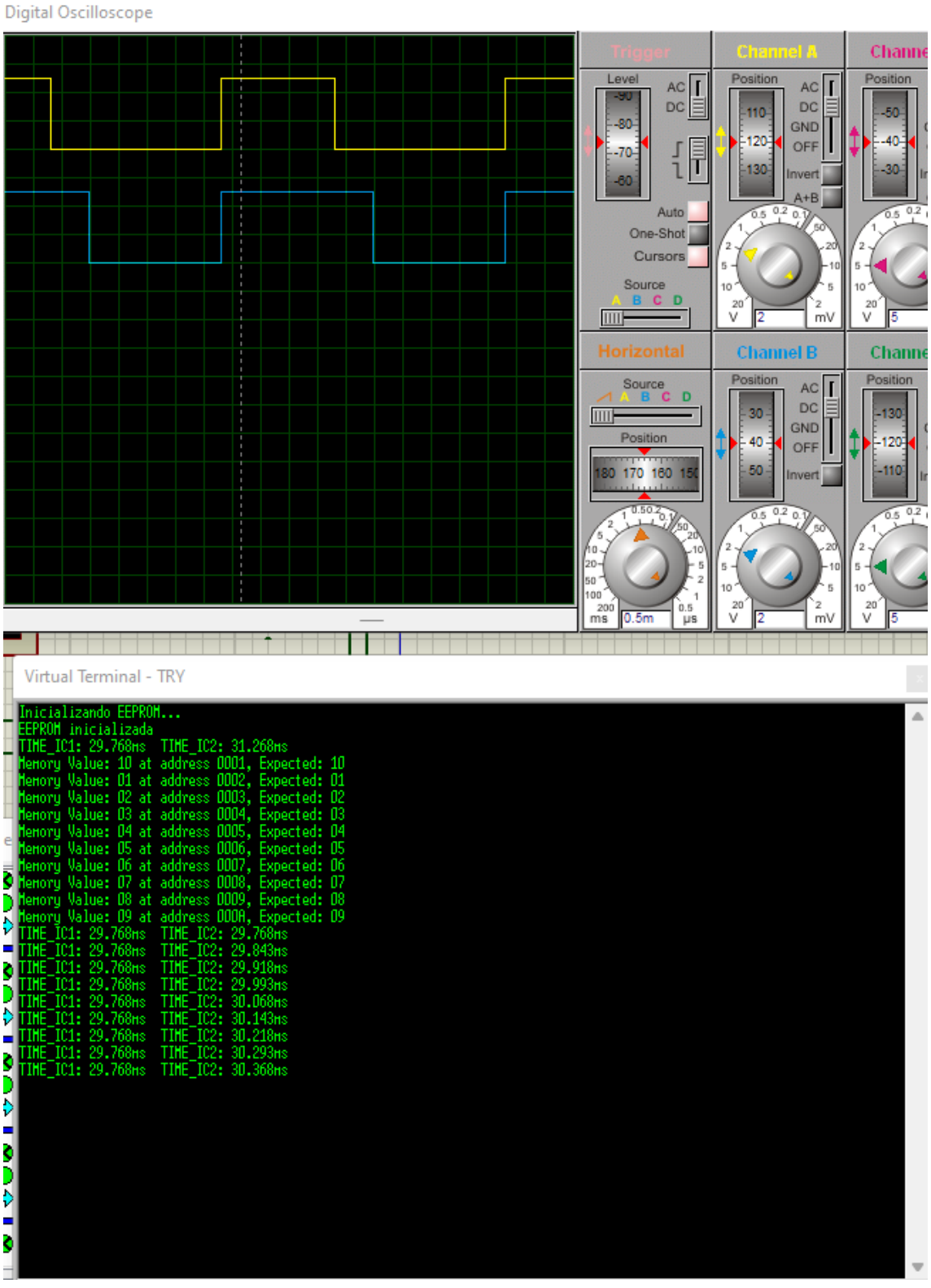
Resultados



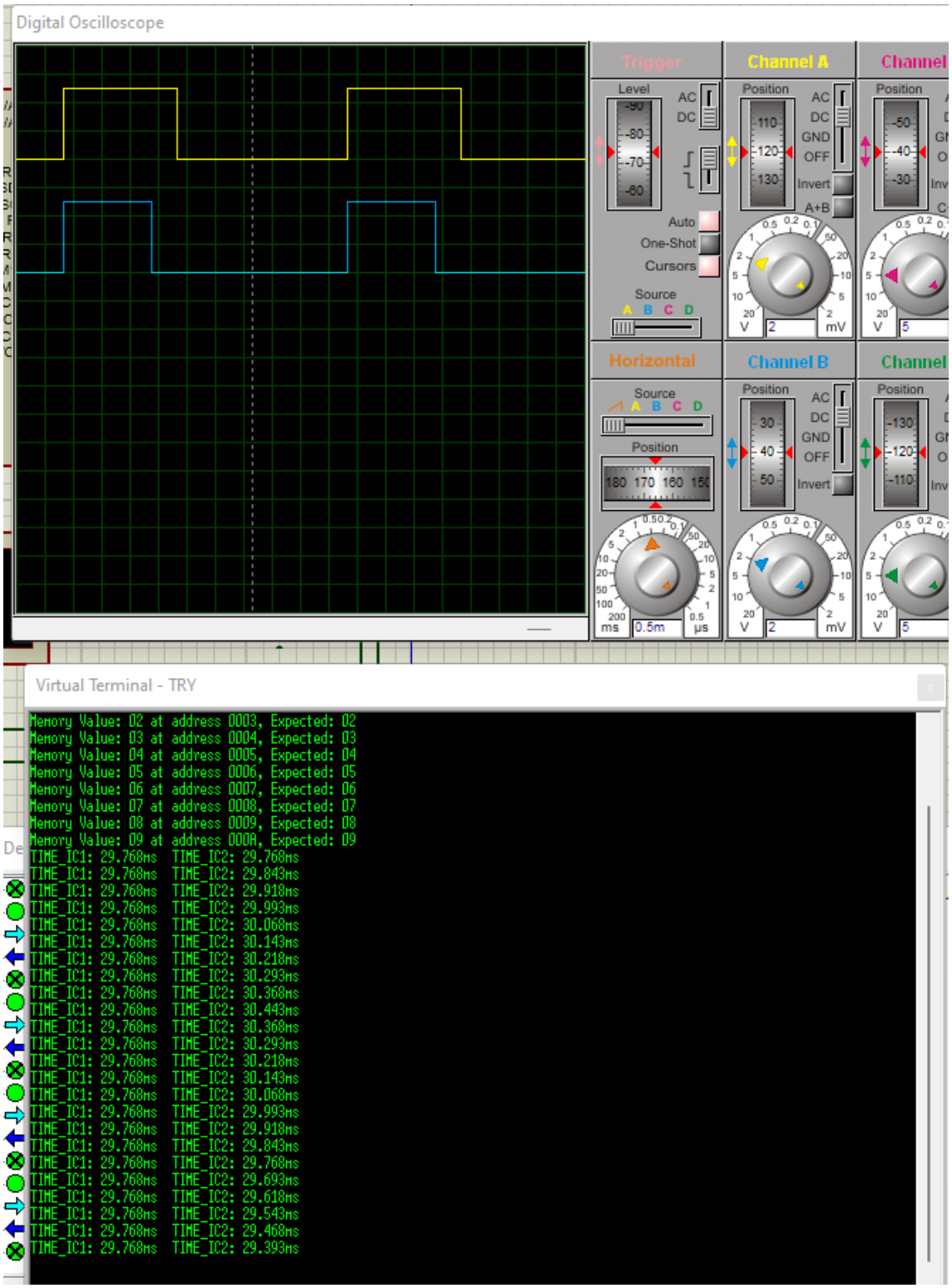
Nada más ejecutar el código nos sale la secuencia descrita antes en durante la explicación del main. Vamos a empezar con la rutina de chequeo.



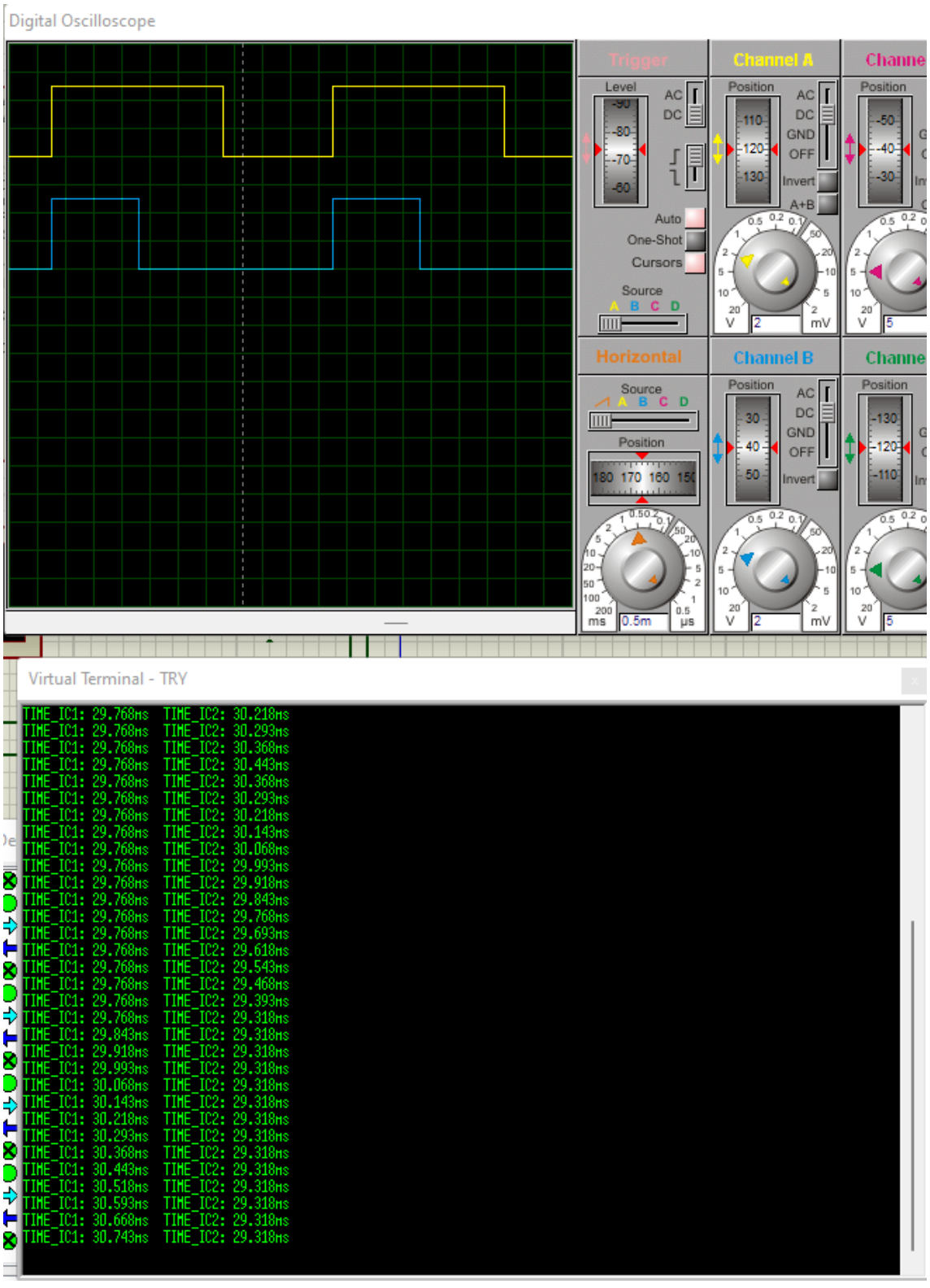
Como podemos ver en la captura, tras ejecutar la rutina de chequeo comprueba que los datos leídos son correctos, por lo que se enciende el LED verde.



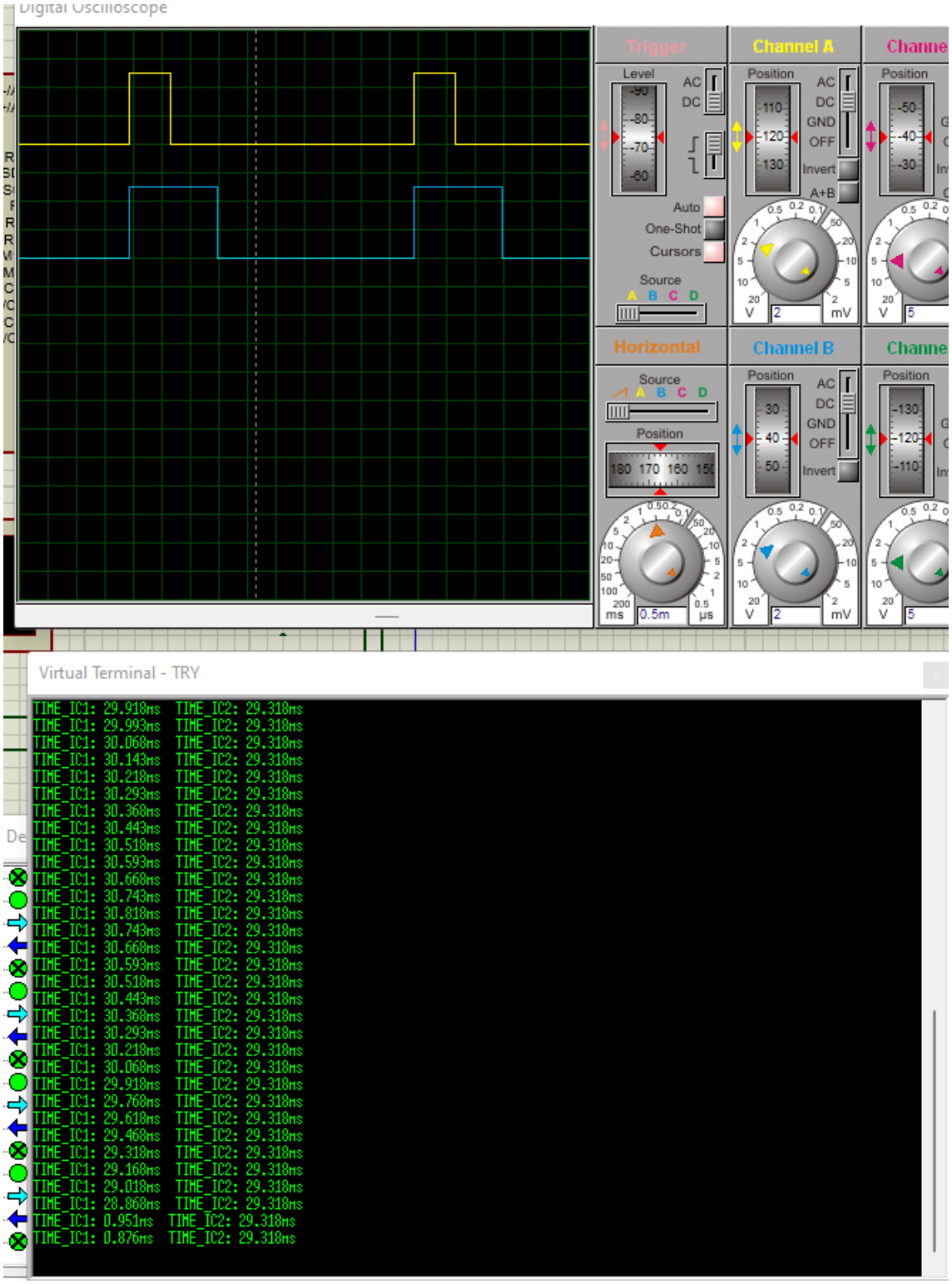
Como podemos ver, tras pulsar repetidas veces la tecla “+” no solo se ve en el Terminal Virtual que incrementa el tiempo, además podemos ver en el osciloscopio como el tiempo es mucho mayor para IC2.



Ahora hemos presionado repetidas veces la tecla “-“, se puede ver que el tiempo disminuye cada vez que pulsas la tecla. En el oscilador observamos también como el tiempo ha disminuido.

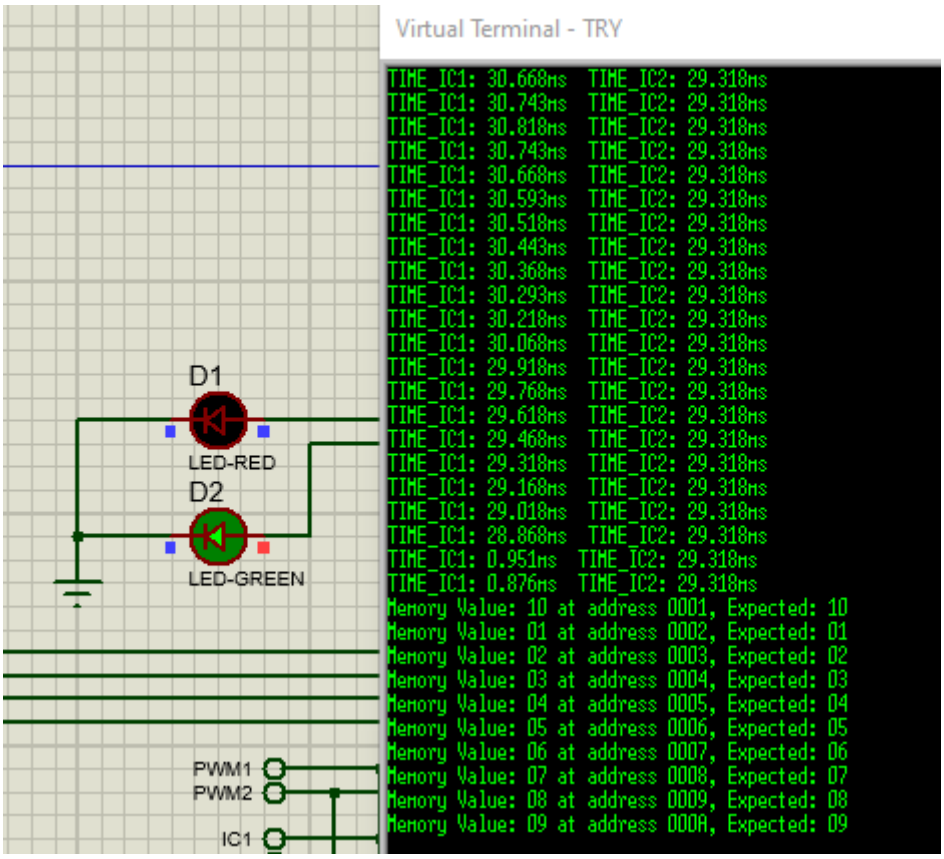


Tras pulsar repetidas veces la tecla “p”, vemos como aumenta el tiempo.

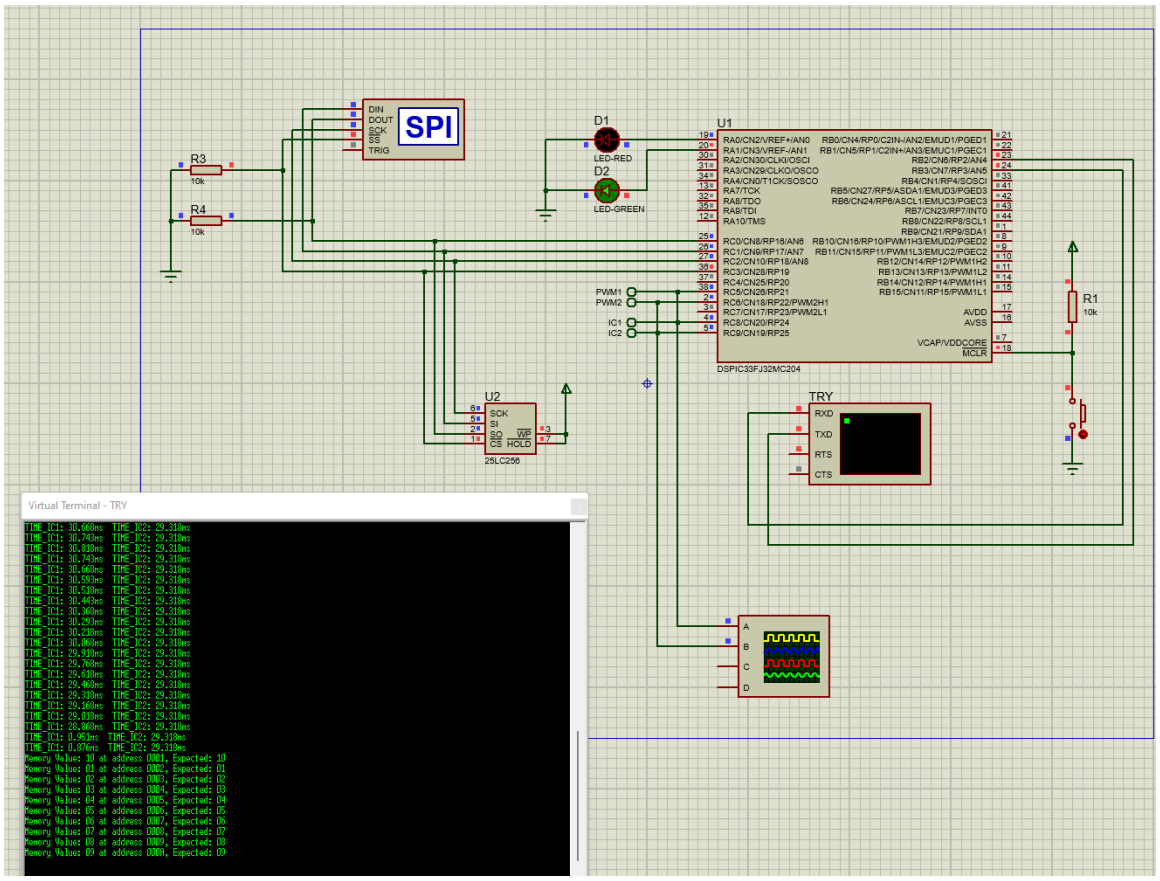


Aquí he pulsado varías veces, muy rápidamente, la tecla “m”. Se puede ver como el tiempo ha reducido, aunque no ha podido imprimir a la velocidad a la que pulsaba (creo).

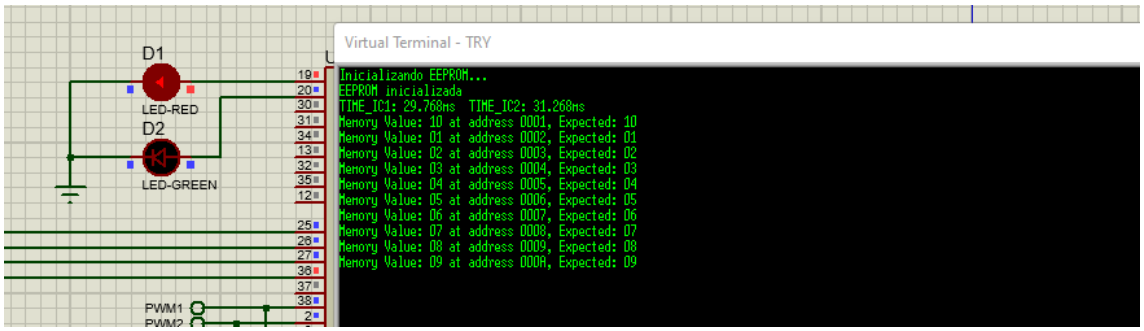
Para comprobar que podemos ejecutar la rutina de chequeo siempre que queramos:



Aquí muestro el circuito completo:



Para comprobar que pasaría si leyéramos un dato incorrecto, debo modificar el código cambiando el array con los datos leídos de la EEPROM por el array “badMemory_data” que tiene un dato incorrecto.



Como se muestra en la imagen, lee correctamente los datos pero se enciende solo el LED rojo. Esto es porque solo modifico la condición.

```
for(i=0; i < 10; i++){
    if(badMemory_data[i] != memoryBuffer[i]){
        LED_RED = LED_ON_STATE;
        testResult = 1;
    }
}
```

Este es el fragmento modificado.

Conclusiones

Estoy contento con los resultados obtenidos. Ha sido bastante difícil, pero me ha servido sobre todo para manejarme mejor por la documentación. Remapear los pines ha sido lo más difícil, y en realidad no tiene mucha complejidad. Además, me ha ayudado a entender mejor el funcionamiento de estos módulos.

En general creo que el trabajo realizado es correcto.