Universidad Simón Bolívar

Departamento de Electrónica y Circuitos

Sistemas Embebidos I, EP5801

Prof. Oswaldo Monasterios

Enero-Marzo 2025

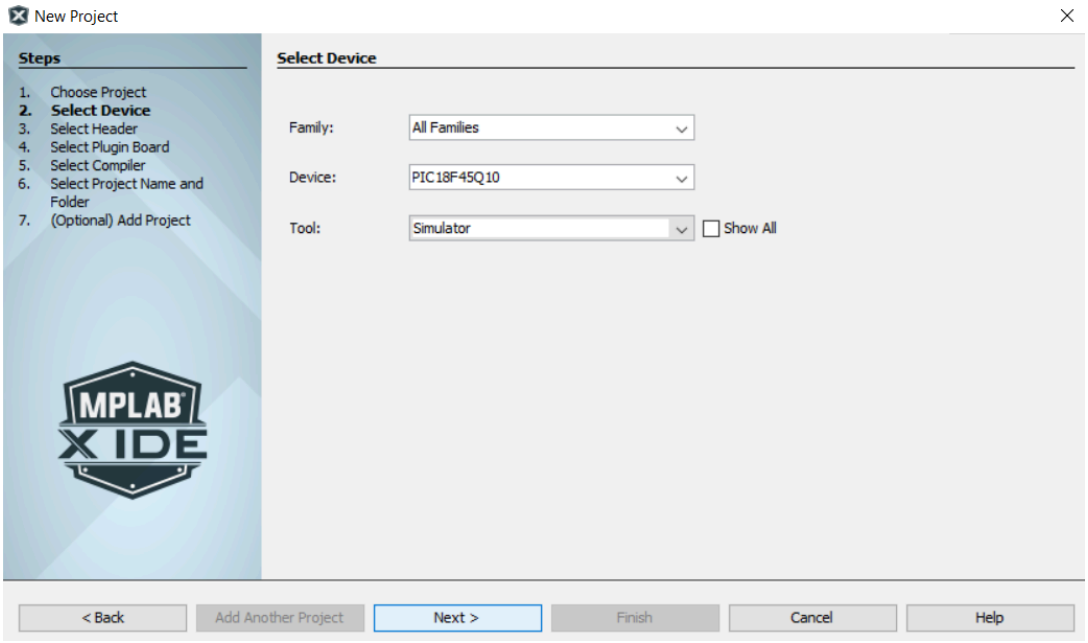
INFORME

LABORATORIO 3

Estudiante:
Shantal Infante
15-10724

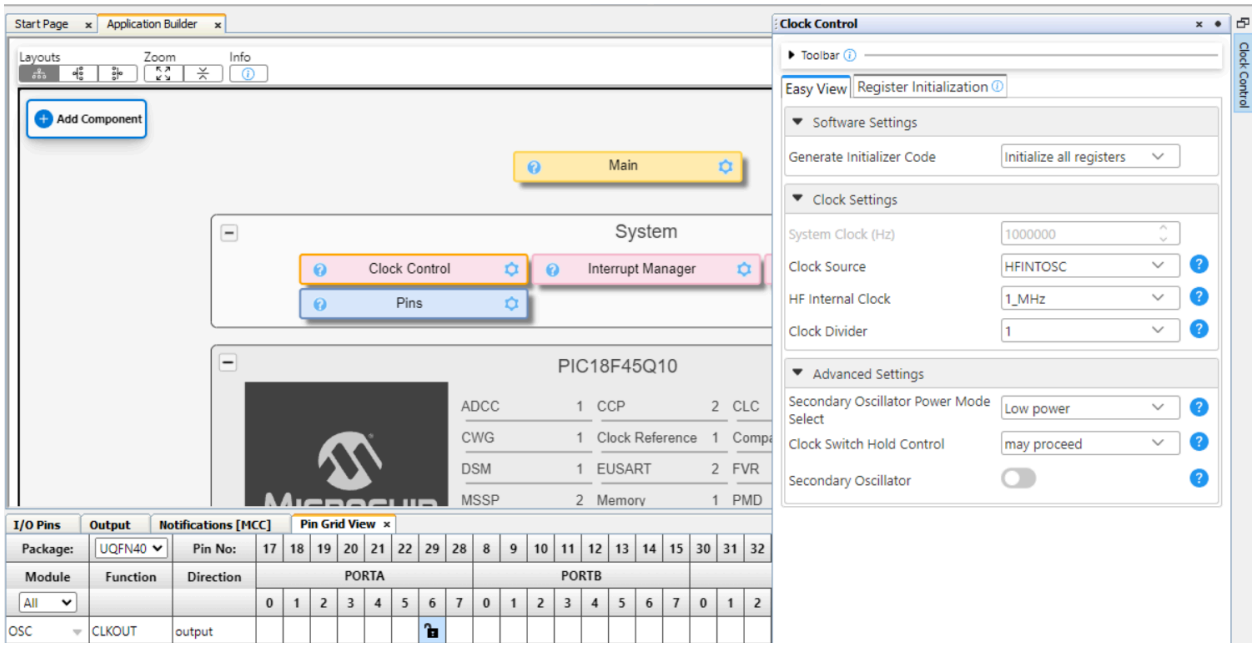
Introducción

En este proyecto de laboratorio, se enfocará en conocer y utilizar el microcontrolador PIC18F45Q10, con énfasis en la configuración y uso de sus timers. Además, se explorará la interfaz y las opciones disponibles en la última versión de MPLAB, específicamente utilizando MCC Melody, una herramienta de Microchip que se integra con el entorno de desarrollo MPLAB XIDE y facilita la configuración de periféricos y la generación de código para sus microcontroladores.
Se realizará un ejercicio práctico que aborda diferentes formas de implementar un Led que parpadea (blinking led), discutiendo las fortalezas y debilidades de cada una.

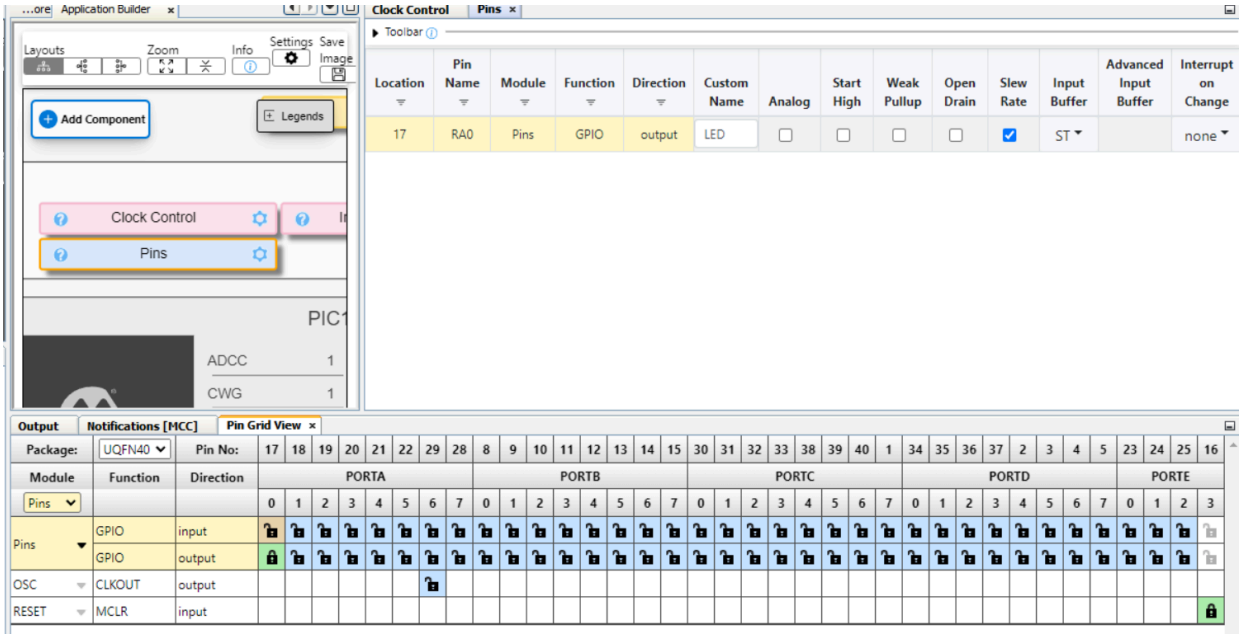


1. Blinking Led usando delay

En el MCC se realizó la configuración del reloj interno del microcontrolador, en el apartado de Clock Control y se seleccionó el oscilador interno de alta frecuencia (HFINTOSC) y se configuró a una frecuencia de 1 MHz.



Luego, se abrió el Pin Manager para seleccionar una salida digital, adecuado para controlar un LED. En este caso se usó el pin RA0 (del puerto A) y se configuró como se muestra en la imagen a continuación:



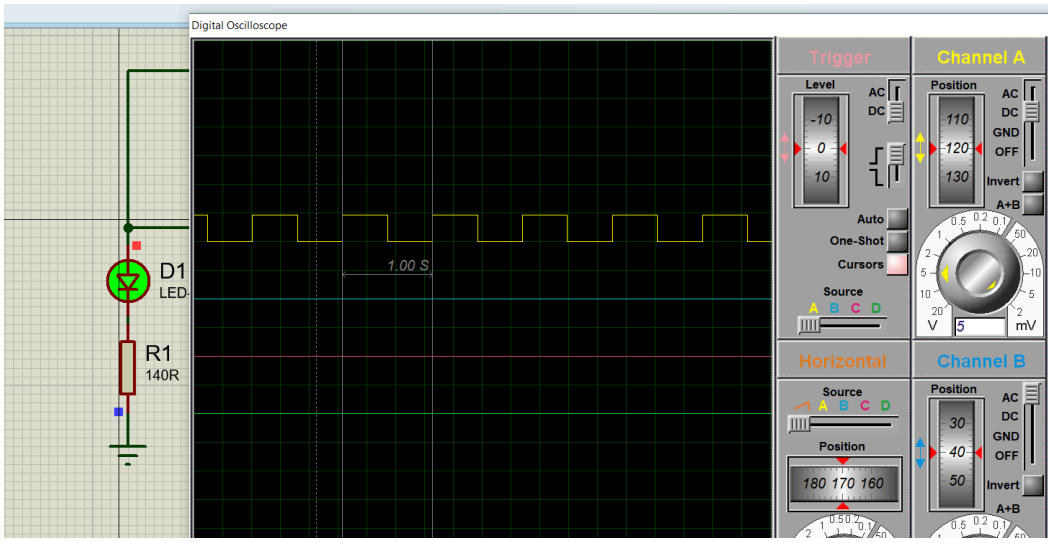
Con estas configuraciones se generaron los archivos, para a continuación añadir las siguientes líneas en el archivo main.c:

```
while(1)
{
    LED_Toggle();
    __delay_ms(500);
}
```

Donde:

- **while(1):** Es un bucle infinito que garantiza que el LED continuará parpadeando indefinidamente.
- **LED_Toggle()**: Ésta función, generada por MCC y definida en el archivo header pin.h, la cual alterna el estado del pin RA0 (encender y apagar el LED).
- **__delay_ms(500):** Genera un retardo de 500 milisegundos entre cada cambio de estado del LED, creando el efecto de parpadeo.

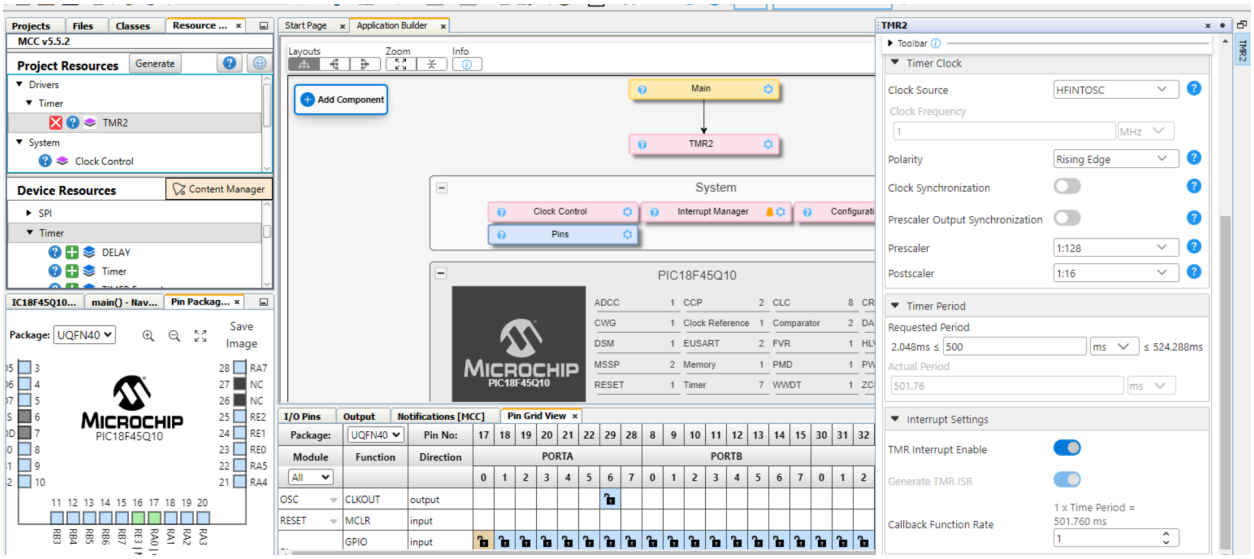
Por último se generó el archivo .hex para cargarlo en un proyecto de Proteus con el PIC18F45Q10. Se conectó un led y su respectiva resistencia de protección para poder observar cómo este enciende y apaga correctamente, además de medir el periódo y el voltaje obtenido con ayuda del osciloscopio, observando que T = 1s, lo que es consistente con programado un delay de 500 ms entre cada toggle del led.



Desventaja: Usar delay no permite que el microcontrolador pueda ejecutar otras tareas durante ese tiempo, lo cual puede ser contraproducente en ciertos casos

2. Blinking Led usando un timer e interrupciones:

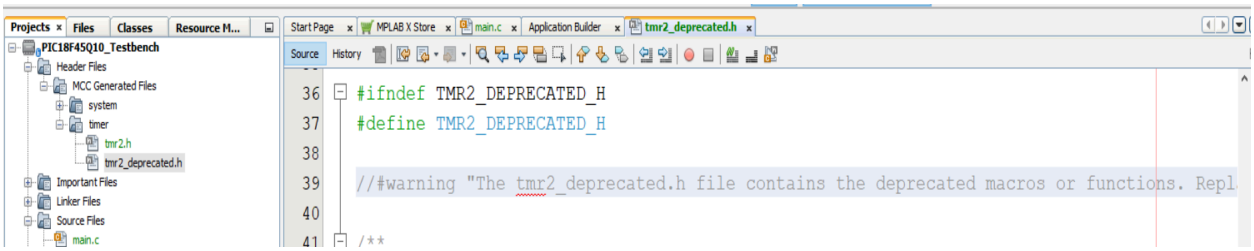
En el MCC, en el panel Device Resources se seleccionó el Timer 2 y se configuró como se muestra a continuación



- **Clock Source:** HFINTOSC.
- **Clock Frequency:** 1 MHz.
- **Prescaler**: 1:128 (divide la frecuencia del reloj por 128).
- **Postscaler:** 1:16 (divide la salida del temporizador por 16).
- **Requested Period**: 500 ms. (Y se obtuvo 501.76 ms).
- **TMR Interrupt Enable:** Habilitado.

Generamos los archivos

Y ahora en n la carpeta tim2 deprecated.h se comentó la línea //#warning "The tmr2_deprecated.h file



Luego en el main.c:

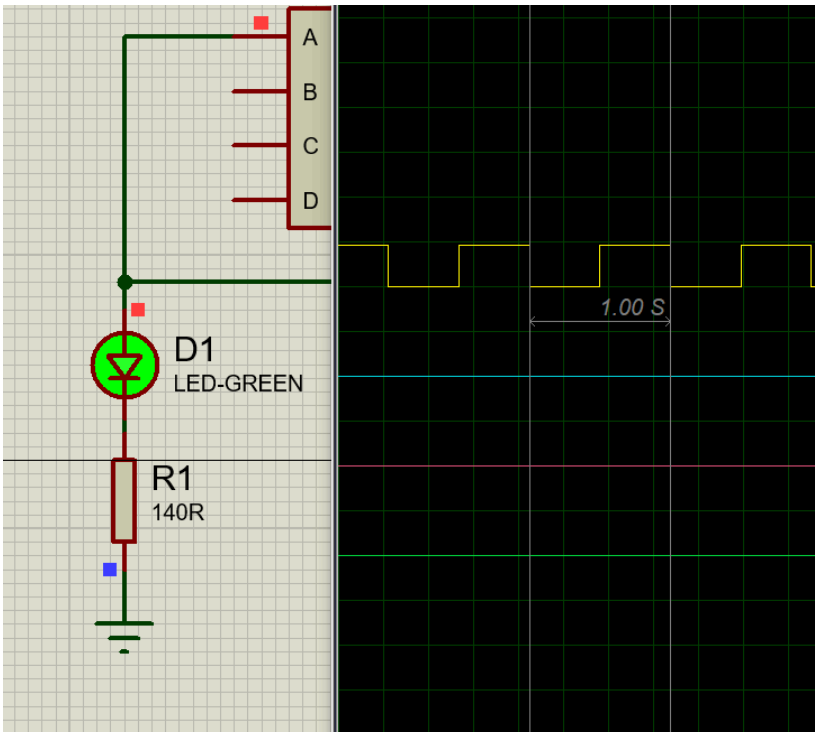
- Se habilitaron interrupciones globales y de periféricos
- Se agregó la línea “extern const struct TIMER_INTERFACE Timer2”, que pide poder usar el timer2

- Se usó TMR2_Start(void) para iniciar el temporizador TMR2, ya que cuando se llama a esta función el temporizador comienza a contar según la configuración establecida.
- También se usó la función TMR2_PeriodMatchCallbackRegister() para registrar una función de callback que se ejecutará cada vez que TMR2 alcance el período configurado y genere una interrupción
  Esta función establece que se debe pasar un puntero a una función callbackHandler, por lo que se tuvo que definir nuestro callbackHandler (se llamó timer_callback en este caso) y cuando se llame va a ejecutar la función LED_Toggle.
- Además, no usamos TMR2_Initialize(void) ya que SYSTEM_Initialize() del main.c ya inicializa todo.

```
extern const struct TIMER_INTERFACE Timer2;
/*
    Main application
*/

//Definimos el Callback Handler del timer
void timer_callback(void)
{
    LED_Toggle();
}

int main(void)
{
    SYSTEM_Initialize(); // No usamos TMR4_Initialize(void) porque 

    // Enable the Global Interrupts
    INTERRUPT_GlobalInterruptEnable();

    // Enable the Peripheral Interrupts X
    INTERRUPT_PeripheralInterruptEnable();
```

```
        TMR2_PeriodMatchCallbackRegister(timer_callback);
        TMR2_Start();
        while(1)
        {

        }
}
```

Luego de compilar y generar el archivo .Hex se observa que también para este caso la simulación se comporta a lo esperado según las configuraciones establecidas, obteniendo un T = 1s. Sin embargo, los timers y las interrupciones permiten que el microcontrolador continúe ejecutando otras instrucciones mientras espera el evento de la interrupción para reaccionar, al contrario de la primera implementación que verifica constante y repetitivamente del estado de un periférico, lo que consume recursos del microprocesador de forma ineficiente.



Desventaja: A pesar de que ahora se tiene una implementación mejor que la primera, se debe tener en cuenta que el uso de Callbacks para casos como este no son la mejor práctica ya que las funciones de callback pueden interferir con el funcionamiento de los periféricos, porque estos requieren una sincronización precisa. Además, si un callback es demasiado largo o complejo, puede retrasar la ejecución de otras tareas importantes.

3. Blinking Led usando máquina de estados:

En esta ocasión se utilizó la misma configuración del timer 2, pero se tiene una alternativa para solventar las desventajas del caso anterior, que consiste en implementar una máquina de estados.

Esta utiliza 3 estados: INIT_STATE, RUNNING_STATE e INTERRUPT_STATE. Y ahora, para la función callback usamos una variable que indique se está en el estado INTERRUPT_STATE.

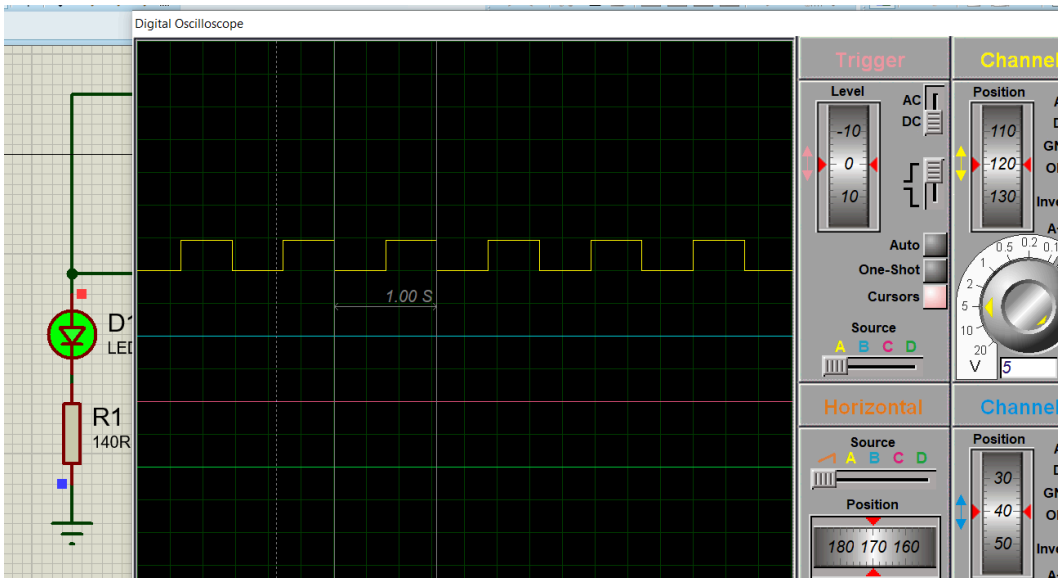
Para manejar los estados se utilizó un switch case en el while del main.c, siguiendo la siguiente lógica

- En INIT_STATE: hacemos la rutina de inicialización
- En RUNNING_STATE: ndica que está corriendo el código y no se realiza ninguna acción
- En INTERRUPT_STATE: Se ejecuta LED_Toggle() y volvemos al RUNNING_STATE

```
//Máquina de estados
typedef enum{
    INIT_STATE,
    RUNNING_STATE,
    INTERRUPT_STATE
}STATE_MACHINE;

//Variable para saber el estado de la máquina
STATE_MACHINE state = INIT_STATE; //La Inicalizo en estado 0

//Definimos el Callback Handler del timer
void timer_callback(void)
{
    state = INTERRUPT_STATE; //En callback cambiamos el estado a interrupcion
}

 int main(void)
 {
     SYSTEM_Initialize(); // No usamos TMR4_Initialize(void) porque SYSTEM

     // Enable the Global Interrupts
     INTERRUPT_GlobalInterruptEnable();

     // Enable the Peripheral Interrupts X
     INTERRUPT_PeripheralInterruptEnable();
```

```c
while(1)
{
    switch(state){
        case INIT_STATE:
            //Ejecutamos acá la rutina de inicialización
            TMR2_PeriodMatchCallbackRegister(timer_callback);
            TMR2_Start();

            state = RUNNING_STATE; //pasamos al siguiente estado
            break;

        case RUNNING_STATE:
            //No toca hacer nada acá
            break;

        case INTERRUPT_STATE:
            LED_Toggle();
            state = RUNNING_STATE;
            break;
    }
```

Finalmente se compiló y se generó el .hex y para probarlo en en proteus. Nuevamente se obtuvo el funcionamiento y periodo esperado, pero vemos que en este caso se evitan problemas de sincronización y latencia al no manipular periféricos directamente en las callbacks, sino ajustando la variable “state”



Conclusiones

A lo largo del proyecto, se destacó la utilidad de la herramienta MCC Melody para simplificar la configuración de periféricos y generar código base eficiente sobre el cual implementar un proyecto.

Por otra parte, la práctica de utilizar diferentes enfoques permitió conocer algunas funciones y características útiles del PIC18F45Q10 y comprender mejor las ventajas y limitaciones de cada una.

Finalmente, se observó que la implementación con máquina de estados demostró ser la más adecuada para manejar de manera efectiva las interrupciones y el parpadeo del LED, proporcionando un equilibrio entre precisión y eficiencia, ya que se obtuvo el funcionamiento deseado pero evitando un consumo innecesario de recursos del microcontrolador como en el caso de la implementación con delay o sin que se esté propenso a que se generen problemas por una mala sincronización de los periféricos al controlarlos mediante callbacks, como en la segunda implementación.