

INTRODUCCIÓN A LOS MICROCONTROLADORES PIC

¿Qué es un microcontrolador?

Un microcontrolador es como un **auto completo en miniatura**.

No es solo el motor.

No es solo la computadora.

Es TODO integrado en un solo chip.

Si tomamos como ejemplo el **PIC12F1840**, tenemos un micro de 8 pines, pequeño, pero con todo lo necesario para controlar sistemas electrónicos.

Analogía 1: El microcontrolador es como un auto

Cuando compras un auto ves especificaciones como:

Motor 2.0L

4 cilindros

150 HP

Transmisión automática

Capacidad del tanque

No necesitas saber cómo funciona internamente el motor para conducirlo.

Lo mismo ocurre con un microcontrolador.

Especificaciones típicas de un microcontrolador

En lugar de cilindros y caballos de fuerza, verás cosas como:

Frecuencia de trabajo (MHz)

Memoria Flash (programa)

Memoria RAM

Número de pines de entrada/salida

Temporizadores

Convertidor ADC

Comunicación serial

Analogía 2: Es como una computadora completa

Un microcontrolador contiene dentro:

CPU (el cerebro)

Memoria (para guardar el programa)

Puertos de entrada/salida

Temporizadores

Módulos de comunicación

Es una computadora completa, pero especializada en controlar cosas físicas.

No necesitas ser mecánico

Para manejar un auto no necesitas:

Saber cómo funciona la inyección electrónica

Entender el árbol de levas

Reparar la transmisión

Solo necesitas saber

Acelerar

Frenar

Girar el volante

Con microcontroladores pasa lo mismo.

No necesitas:

Entender la arquitectura interna en detalle

Diseñar el silicio

Conocer cómo funcionan los registros internamente

Solo necesitas saber:

Configurar un pin como salida

Encenderlo o apagarlo

Leer una entrada

Diferencia clave

Un auto transporta personas.

Una computadora procesa información.

Un microcontrolador **controla el mundo físico**.

Controla:

LEDs

Motores

Sensores

Pantallas

Relés

Equipos industriales

¿Dónde se usan?

Están en:

Lavadoras

Microondas

Autos modernos

Controles remotos

Alarmas

Robots
Equipos médicos

Sin microcontroladores, la electrónica moderna no existiría.

Aprender microcontroladores no es aprender electrónica profunda.
Es aprender a dar instrucciones claras a un sistema inteligente pequeño.

Así como no necesitas ser ingeniero automotriz para manejar un auto,
no necesitas ser ingeniero en microarquitectura para programar un microcontrolador.

¿Qué tiene por dentro un microcontrolador?

Como una computadora:

CPU (cerebro)
Memoria (programa)
Puertos de entrada/salida
Temporizadores
Módulos especiales

La diferencia es que no está hecho para abrir Word.
Está hecho para controlar el mundo físico.

¿Qué hace realmente?

Ejemplo básico:

Si presiono un botón → se enciende un LED.
Si sube la temperatura → enciendo un ventilador.
Si detecto movimiento → activo una alarma.
Eso es control.

Especificaciones técnicas del PIC12F1840

Arquitectura

8 bits

Arquitectura Harvard (memoria de programa separada de datos)
Núcleo RISC

¿Qué significa 8 bits?

Que el CPU procesa datos de 8 bits a la vez.
Es como si el “ancho de carretera” fuera de 8 carriles.

Frecuencia de operación

Hasta **32 MHz** con oscilador interno

Puede usar reloj interno o externo

El reloj es el “ritmo” del sistema.

Cada pulso del reloj permite ejecutar una instrucción.

Analogía:

Es como las revoluciones por minuto (RPM) del motor.

Memoria Flash (Programa)

7 KB de memoria Flash

Aquí se guarda el programa.

Es como el disco duro donde se guarda el software.

Memoria RAM (Datos)

256 bytes de RAM

Aquí se guardan:

Variables

Registros

Datos temporales

Es como la memoria RAM de una computadora:

rápida pero volátil.

EEPROM

256 bytes de EEPROM

Memoria no volátil para guardar datos permanentes.

Pines

8 pines físicos

6 pines configurables como entrada/salida

Periféricos internos

ADC de 10 bits

Temporizadores (Timer0, Timer1, Timer2)

PWM

Comunicación I2C, SPI, UART

Comparadores

Interrupciones

Es un micro pequeño pero muy completo.

¿Cómo funciona internamente?

Aquí lo explicamos con claridad.

Cómo funciona el CPU

El CPU hace siempre este ciclo:

- Leer instrucción de memoria
- Decodificarla
- Ejecutarla
- Pasar a la siguiente instrucción

Esto ocurre millones de veces por segundo (de acuerdo a la velocidad del reloj).
Si trabaja a 32 MHz, puede ejecutar aproximadamente 8 millones de instrucciones por segundo.

Analogía simple

Es como un cocinero que:

- Lee una receta
- Hace el paso
- Lee el siguiente paso
- Lo ejecuta
- Y así sucesivamente

Nunca improvisa.
Solo ejecuta instrucciones.

Cómo funciona la memoria Flash

La Flash guarda el programa.
Cuando compilas, generas un archivo .hex.
Ese archivo se graba en la memoria Flash.
El CPU no inventa nada.
Solo ejecuta lo que está ahí.
Es Fija.
Si apagas el micro, no se borra.

3. Cómo funciona la RAM

La RAM es la zona de trabajo.
Aquí se guardan:

- Variables
- Resultados intermedios
- Registros especiales

Es temporal.
Si apagas el micro, se borra.

Los registros son parte de la RAM

Esto es fundamental.

Un registro no es algo mágico.

Es simplemente una dirección especial dentro de la RAM que controla hardware.

Ejemplo

TRISA

PORTA

LATA

Son direcciones específicas en la RAM.

Pero no son variables normales.

Cuando escribes en ellas, modificas hardware real.

Analogía potente para explicar esto

Imagina que la RAM es un edificio de oficinas.

Cada oficina tiene un número.

Algunas oficinas solo guardan papeles (variables).

Pero algunas oficinas están conectadas directamente con máquinas.

Si escribes en esa oficina, se mueve algo físico.

Eso es un registro.

RAM normal

Guarda números.

No afecta directamente el mundo físico.

Registros especiales (SFR)

Controlan hardware.

Ejemplo:

Si escribes 1 en el bit 0 de LATA

Se activa voltaje en el pin GP0.

Eso ya no es solo memoria.

Eso es control físico.

Diferencia clara

CPU → ejecuta instrucciones

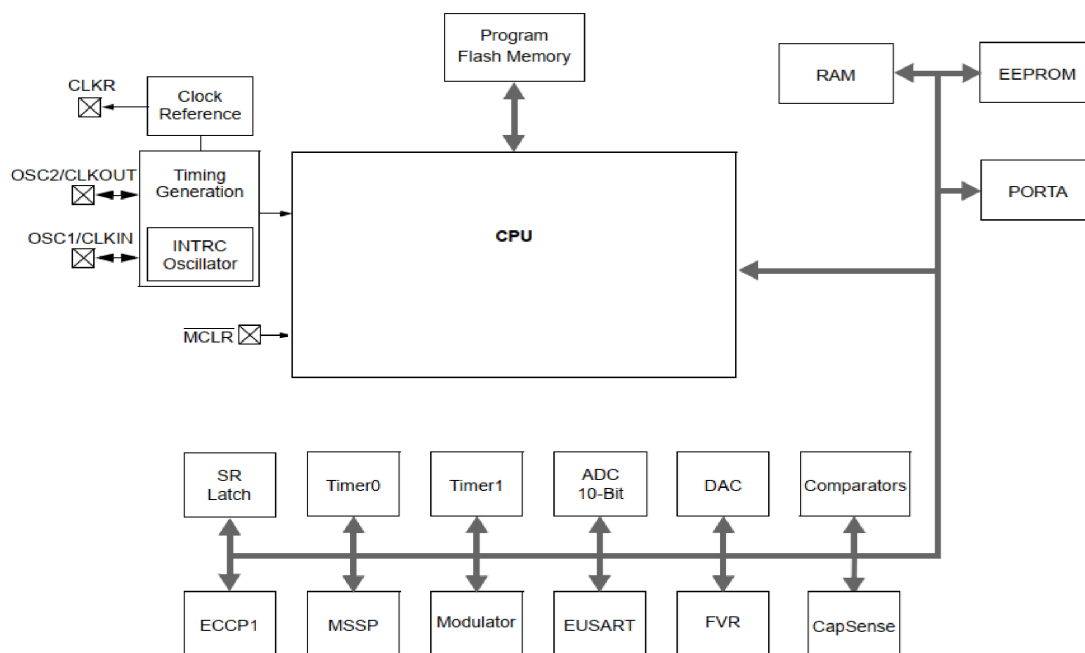
Flash → guarda el programa

RAM → guarda datos temporales

Registros → controlan el hardware

Variable normal → solo vive en memoria

A continuación se presenta un esquema gráfico de la arquitectura del microcontrolador PIC12F1840.



Se ve mas complejo de lo que es. Por el momento no entraremos en detalles del microcontrolador sino que lo iremos haciendo poco a poco. Esta figura es simplemente para que se aprecien todas las partes internas que contiene el microcontrolador.

Memoria RAM

A continuación presentaremos una tabla general de los 256 registros que integra la memoria RAM.

0x00		INDF0		INDF1		PCL		STATUS		FSR0L		FSR0H		FSR1L		FSR1H	
0x08		BSR		WREG		PCLATH		INTCON		PORTA		GPR		GPR		GPR	
0x10		TMR0		TMR1L		TMR1H		T1CON		TMR2		PR2		T2CON		GPR	
0x18		OSCCON		OSCSTAT		GPR		GPR		GPR		GPR		GPR		GPR	
0x20		PIE1		PIR1		ADCON0		ADCON1		ADRESL		ADRESH		GPR		GPR	
0x28		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0x30		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0x38		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0x40		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0x48		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0x50		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0x58		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0x60		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0x68		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0x70		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0x78		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0x80		GPR		GPR		GPR		GPR		TRISA		LATA		ANSELA		GPR	
0x88		GPR		GPR		GPR		GPR		GPR		WPUA		IOCAF		GPR	
0x90		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0x98		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0xA0		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0xA8		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0xB0		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0xB8		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0xC0		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0xC8		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0xD0		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0xD8		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0xE0		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0xE8		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0xF0		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	
0xF8		GPR		GPR		GPR		GPR		GPR		GPR		GPR		GPR	

GPR = Memoria general

Cada registro o posición de memoria esta integrado por 8 bits. Como ejemplo mostraremos algunos registros que usaremos más adelante.

Ahora los organizaremos **por función**, no por dirección.

Registros del Puerto A (Entradas y Salidas)

Estos controlan directamente los pines físicos.
TRISA, PORTA, LATA, ANSELA, WPUA, IOCAF

Registros del Sistema / CPU

Control interno del procesador.
WREG, STATUS, BSR, PCL, PCLATH, FSR0L, FSR0H, FSR1L, FSR1H, INDF0, INDF1

Registros del Oscilador (Reloj)

Definen la velocidad de operación.
OSCCON, OSCSTAT

Registros de Temporizadores

Permiten medir tiempo o generar señales periódicas.

Timer0

TMR0, OPTION_REG

Timer1

TMR1L, TMR1H, T1CON

Timer2

TMR2

Registros de Interrupciones

Permiten que el micro reaccione automáticamente a eventos.
INTCON, PIE1, PIR1

Registros del ADC (Convertidor Analógico-Digital)

Permiten leer voltajes analógicos.
ADCON0, ADCON1, ADRESL, ADRESH

Registros de Comunicación Serial (si se usan)

TXSTA, RCSTA, SPBRG, SPBRGH, TXREG, RCREG, SSPBUF, SSPCON1, SSPCON2, SSPSTAT

Registro TRISA (Dirección de pines) (0x84)

TRISA define si un pin es **entrada** o **salida**.

Bit: 7 6 5 4 3 2 1 0
- - TR5 TR4 TR3 TR2 TR1 TR0

Significado de cada bit

Bit	Nombre	Función
7	—	No usado
6	—	No usado
5	TRISA5	Controla GP5
4	TRISA4	Controla GP4
3	TRISA3	Controla GP3
2	TRISA2	Controla GP2
1	TRISA1	Controla GP1
0	TRISA0	Controla GP0

Valores posibles

1 → Entrada

0 → Salida

Ejemplo práctico

Si quieres que **GP0** sea **salida**:

TRISA0 = 0

Si TRISA = 11111110 (0xFE)

→ Solo GP0 es salida

→ Los demás son entrada

Aquí observe el valor: Si es **0**, como **Output** (salida), y si es **1**, como **Input** (entrada).

Registro LATA (Latch de salida) (0x85)

Define el **valor lógico** que queremos enviar al pin.

Bit: 7 6 5 4 3 2 1 0
- - LA5 LA4 LA3 LA2 LA1 LA0

Bit	Controla
5	GP5
4	GP4
3	GP3
2	GP2
1	GP1
0	GP0

Valores

1 → Nivel alto (5V o 3.3V según alimentación)

0 → Nivel bajo (0V)

Ejemplo

Si haces:

LATA0 = 1

GP0 se pone en alto.

Si luego:

LATA0 = 0

GP0 se pone en bajo.

Registro PORTA (Estado real del pin) (0x0C)

PORTA permite **leer el estado actual** del pin.

Bit: 7 6 5 4 3 2 1 0
- - RA5 RA4 RA3 RA2 RA1 RA0

Diferencia clave:

LATA → Lo que tú quieres enviar

PORTA → Lo que realmente está en el pin

Ejemplo

Encendido y apagado de un LED (GP0)

Objetivo

Configurar el pin **GP0** como salida digital y alternar su estado lógico para encender y apagar un LED.

Este ejemplo permite comprender:

Qué es un pin digital

Cómo se configura como salida

Cómo un bit controla un elemento físico

Cómo intervienen los registros TRISA, LATA y PORTA

Conexión física

El **LED** se conecta:

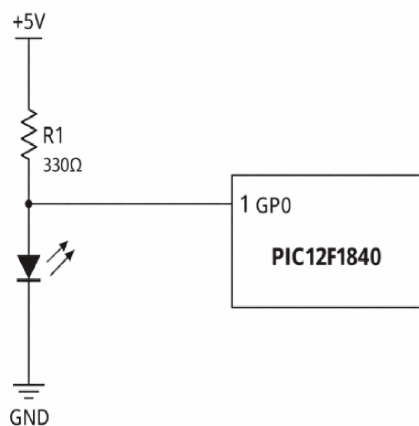
Ánodo → GP0

Cátodo → Resistencia → GND

Cuando GP0 está en nivel alto (1), el LED enciende.

Cuando GP0 está en nivel bajo (0), el LED apaga.

Diagrama eléctrico – LED en GP0



Valor típico de la resistencia

Para 5V:

LED rojo ≈ 2V caída

Corriente deseada ≈ 10 mA

Cálculo aproximado:

$$R = (5V - 2V) / 0.01A = 300\Omega$$

Valor comercial recomendado: **330Ω**

Código del ejemplo

```
#define OUTPUT 0
#define INPUT 1
#define HIGH 1
#define LOW 0

// Configurar GP0 como salida
GP0 = OUTPUT

while(1){
    GP0 = HIGH
    GP0 = LOW
}
```

Explicación paso a paso

Paso 1 – Definiciones

Se crean definiciones simbólicas:

OUTPUT → 0

INPUT → 1

HIGH → 1

LOW → 0

Esto hace el código más legible.

Paso 2 – Configurar dirección del pin

GP0 = OUTPUT

Internamente esto equivale a:

TRISA0 = 0

Recordemos:

TRISA controla dirección

0 = salida (**O**utput)

1 = entrada (**I**nput)

Si TRISA0 = 0 → GP0 es salida digital.

Paso 3 – Bucle infinito

while(1)

Significa:

Repetir para siempre.

En microcontroladores, el programa casi siempre vive dentro de un bucle infinito.

Paso 4 – Encender LED

GP0 = HIGH

Internamente equivale a:

LATA0 = 1

Esto coloca nivel alto en el pin.

Paso 5 – Apagar LED

GP0 = LOW

Internamente equivale a:

LATA0 = 0

El pin se pone en bajo.

Qué ocurre a nivel de registros

Supongamos que el sistema inicia con:

TRISA = 11111111

LATA = 00000000

PORTA = 00000000

Después de configurar GP0 como salida

TRISA = 11111110

Bit 0 pasa a 0 → salida.

Después de GP0 = HIGH

LATA = 00000001

PORTA = 00000001

LED encendido.

Después de GP0 = LOW

LATA = 00000000

PORTA = 00000000

LED apagado.

Qué aprendimos

Un solo bit controla un pin físico

TRISA define dirección

LATA define valor de salida

PORTA refleja el estado real

El micro ejecuta instrucciones en secuencia

Concepto clave

Cuando escribimos:

GP0 = HIGH

En realidad estamos modificando:

Un bit dentro de una dirección específica de memoria.

Ese bit está conectado internamente a un transistor que controla el pin físico.

Relación con el simulador

En el simulador desarrollado:
Se observa el cambio en TRISA
Se observa el cambio en LATA
Se observa el LED encender/apagar
Se ejecuta paso a paso

Esto permite visualizar cómo el software modifica el hardware.

Conclusión del ejemplo

Este ejemplo demuestra el principio fundamental de los microcontroladores:

Un programa modifica bits en memoria.
Esos bits están conectados a hardware real.
El hardware responde inmediatamente.
Un LED es solo el inicio.
El mismo principio aplica para:
Motores
Sensores
Displays
Sistemas industriales completos

Para más información acerca de las especificaciones técnicas del microcontrolador PIC12F1840 consulte el documento PDF denominado «**PIC12F1840**» el cual puede ser obtenido desde el simulador mediante el botón denominado «**PIC12F1840**».

Memoria EEPROM

¿Qué es la EEPROM?

EEPROM significa:

Memoria solo lectura electricamente borrable (Electrically Erasable Programmable Read-Only Memory, es decir

Es una memoria:

No volátil

Programable eléctricamente

Reescribible

Independiente de la memoria Flash

¿Para qué sirve?

Sirve para guardar datos que deben mantenerse aunque el micro se apague.

Ejemplos reales:

Configuración del usuario

Valores de calibración

Contador de uso

Número de serie

Último estado guardado

Características en el PIC12F1840

256 bytes de EEPROM

Direccionamiento de 0x00 a 0xFF

Escritura byte por byte

Tiempo típico de escritura $\approx 2\text{--}4\text{ ms}$

Ciclos típicos de escritura: $\sim 100,000$

Diferencia entre Flash, RAM y EEPROM

Tipo	Volátil	Uso
Flash	No	Guarda el programa
RAM	Sí	Variables temporales
EEPROM	No	Datos persistentes

¿Dónde está físicamente?

No está en la RAM normal.

Es un bloque de memoria separado dentro del chip.

Se accede a través de registros especiales.

Registros asociados a EEPROM

En el PIC12F1840, el acceso a EEPROM se hace mediante:

Registro	Función
EEADR	Dirección EEPROM
EEDATA	Dato a leer o escribir
EECON1	Control de operación
EECON2	Secuencia de seguridad

Cómo funciona la lectura

Proceso simplificado:

Cargar dirección en EEADR

Activar bit RD

El dato aparece en EEDATA

Cómo funciona la escritura

Proceso (resumido):

Colocar dirección en EEADR

Colocar dato en EEDATA

Habilitar escritura

Ejecutar secuencia de seguridad

Activar bit WR

Esperar a que termine

¿Por qué hay secuencia de seguridad?

Para evitar que una instrucción accidental borre datos.

Se requiere escribir una secuencia específica en EECON2 antes de activar WR.

Esto evita corrupción por ruido o errores.

Ejemplo práctico conceptual

Guardar un contador:

EEPROM[0x10] = 25

Apagas el micro.

Lo vuelves a encender.

El valor sigue ahí.

Limitaciones importantes

No escribir constantemente en el mismo byte.
La EEPROM tiene límite de ciclos (~100,000).
Es lenta comparada con RAM.

Buenas prácticas

Guardar datos solo cuando cambian
No escribir en cada ciclo de programa
Usar una bandera para evitar escrituras repetidas
Usar delay o esperar bandera de fin de escritura

Analogía clara

RAM es una hoja de papel borrrable.
EEPROM es un cuaderno permanente.
Flash es el libro impreso.

¿Cuándo se usa en la vida real?

Control industrial
Domótica
Equipos médicos
Control de acceso
Contadores eléctricos
Dispositivos IoT

Ejemplo didáctico potente

Imagina que el usuario ajusta el brillo de una lámpara.
Si lo guardas en RAM, al apagar se pierde.
Si lo guardas en EEPROM, al encender vuelve al último valor.
Eso es persistencia.

A continuación se presentan los registros de memoria RAM asociados al EEPROM.

Registro EECON1 (bit a bit)

EECON1 controla las operaciones de lectura y escritura de EEPROM y Flash.

Bit:	7	6	5	4	3	2	1	0
	EEPGD	CFGS	-	FREE	WRERR	WREN	WR	RD

Significado de cada bit

Bit	Nombre	Función
7	EEPGD	1 = Acceso a Flash, 0 = Acceso a EEPROM
6	CFGS	1 = Acceso a configuración, 0 = Memoria normal
5	—	No usado
4	FREE	Borrado de memoria (Flash)
3	WRERR	Indica error en escritura
2	WREN	Habilita escritura
1	WR	Inicia escritura
0	RD	Inicia lectura

Para EEPROM normal

Debemos asegurar:

EEPGD = 0

CFGS = 0

Así estamos trabajando en EEPROM de datos.

Proceso de LECTURA (simple)

1 Cargar dirección en EEADR

2 Poner RD = 1

3 El dato aparece en EEDATA

El hardware limpia RD automáticamente.

PROCESO DE ESCRITURA (paso a paso real)

Este es el procedimiento correcto.

Paso 1 — Colocar dirección

EEADR = dirección

Paso 2 — Colocar dato

EEDATA = valor

Paso 3 — Seleccionar EEPROM

EECON1bits.EEPGD = 0

EECON1bits.CFGS = 0

Paso 4 — Habilitar escritura

EECON1bits.WREN = 1

Sin esto, no se puede escribir.

Paso 5 — Secuencia de seguridad (obligatoria)

Aquí está la parte crítica.

Se debe escribir:

EECON2 = 0x55

EECON2 = 0xAA

Importante:

EECON2 no es un registro físico real.

Es un registro virtual usado solo para desbloqueo.

Paso 6 — Iniciar escritura

EECON1bits.WR = 1

Aquí empieza el proceso interno.

Paso 7 — Esperar fin de escritura

Mientras WR = 1

La escritura está en progreso.

Cuando WR vuelve a 0

La operación terminó.

Paso 8 — Deshabilitar escritura

EECON1bits.WREN = 0

Buena práctica.

Algoritmo completo en pseudocódigo

EEADR = direccion

EEDATA = dato

EEPGD = 0

CFGS = 0

WREN = 1

EECON2 = 0x55

EECON2 = 0xAA

WR = 1

Esperar hasta que WR = 0

WREN = 0

¿Por qué la secuencia 0x55 / 0xAA?

Es una protección contra:

Ruido eléctrico

Instrucciones erróneas

Fallas de alimentación

Si no se hace esa secuencia exacta, la escritura no ocurre.

Qué pasa internamente

Cuando WR = 1:

El CPU se detiene momentáneamente

Se aplica alto voltaje interno

Se programa el byte

WR vuelve a 0

Tiempo típico: 2–4 ms

Posibles errores

Si se interrumpe alimentación durante escritura:

WRERR = 1

Eso indica escritura fallida.

Buenas prácticas reales

Deshabilitar interrupciones durante secuencia

No escribir repetidamente el mismo byte

Verificar WRERR

Usar bandera para evitar múltiples escrituras

Ejemplo didáctico potente

Imagina que guardas un contador:

Si no haces la secuencia 0x55 / 0xAA

El micro ignora la escritura.

Es como una doble llave de seguridad.

Ejemplo

Guardar y recuperar un valor en EEPROM

Objetivo

Aprender a:

Escribir un dato en **EEPROM**

Leerlo posteriormente

Entender el proceso interno de escritura

Comprender la secuencia de seguridad

Este ejemplo demuestra cómo guardar un valor permanente que no se pierde al apagar el micro-controlador.

Situación práctica

Supongamos que queremos:

Guardar en EEPROM el número **25** en la dirección **0x10**.

Ese valor podría representar:

Nivel de brillo

Número de arranques

Configuración del usuario

Valor calibrado

Código conceptual (estilo didáctico)

Escritura en EEPROM

```
EEPROM_WRITE(0x10, 25)
```

Internamente esto realiza:

Cargar dirección

Cargar dato

Habilitar escritura

Ejecutar secuencia de seguridad

Iniciar escritura

Esperar finalización

Lectura desde EEPROM

valor = EEPROM_READ(0x10)

Proceso interno real paso a paso

Escritura

Paso 1 – Dirección

EEADR = 0x10

Paso 2 – Dato

EEDATA = 25

Paso 3 – Seleccionar EEPROM

EEPGD = 0

CFGS = 0

Paso 4 – Habilitar escritura

WREN = 1

Paso 5 – Secuencia de seguridad

EECON2 = 0x55

EECON2 = 0xAA

Paso 6 – Iniciar escritura

WR = 1

Paso 7 – Esperar

Mientras WR = 1

La escritura está en progreso.

Paso 8 – Deshabilitar

WREN = 0

Proceso de lectura

Paso 1

EEADR = 0x10

Paso 2

RD = 1

Paso 3

valor = EEDATA

Qué ocurre internamente?

Cuando $WR = 1$:

Se aplica alto voltaje interno

Se programa físicamente la celda EEPROM

Toma aproximadamente 2–4 ms

WR vuelve automáticamente a 0

Estado antes y después

Antes

EEPROM[0x10] = 0

Después

EEPROM[0x10] = 25

Si apagamos el micro:

El valor sigue almacenado.

Diferencia con RAM

Si guardáramos 25 en RAM:

Al apagar → se pierde.

Si lo guardamos en EEPROM:

Al apagar → se conserva.

Ejemplo práctico integrado con LED

Podríamos hacer:

Guardar estado del LED en EEPROM

Al reiniciar, leer ese estado

Encender el LED según el valor almacenado

Concepto clave del ejemplo

La EEPROM permite que el micro:

Recuerde información entre reinicios.

Eso lo convierte en un sistema inteligente.

Buenas prácticas

No escribir constantemente

No usar EEPROM como RAM

Verificar WRERR

Deshabilitar interrupciones durante escritura

Memoria Flash

Ahora entramos a la parte más importante del micro: **la memoria Flash**, donde vive el programa.

¿Qué es la memoria Flash?

Es la memoria donde se almacena el **programa**.

Cuando compilas tu código y generas el archivo .hex,

ese archivo se graba en la memoria Flash del microcontrolador.

Sin Flash, el micro no tendría instrucciones que ejecutar.

Características principales

7 KB de memoria Flash

No volátil

Programable eléctricamente

Soporta miles de ciclos de reprogramación

Organizada en palabras de programa

Arquitectura Harvard (muy importante)

El PIC12F1840 usa arquitectura Harvard:

Memoria de programa (Flash) separada

Memoria de datos (RAM) separada

Eso significa:

El CPU lee instrucciones desde Flash

Pero guarda variables en RAM

Son buses diferentes.

Cómo está organizada

La Flash no se organiza en bytes normales como la RAM.

Se organiza en **palabras de programa**.

En este micro:

Cada palabra es de 14 bits (arquitectura PIC de gama media mejorada)

Cada dirección apunta a una instrucción

¿Qué guarda exactamente?

La Flash almacena:

Instrucciones del programa

Tabla de constantes

Vector de reset

Vector de interrupciones

¿Qué ocurre al encender el micro?

Se aplica alimentación.

El contador de programa (PC) se pone en 0x0000.

El CPU lee la primera instrucción en Flash.

Comienza la ejecución secuencial.

Relación con el archivo .HEX

Cuando compilas:

Código C → Ensamblador → Código máquina → Archivo .HEX

El archivo .HEX contiene:

Direcciones

Datos binarios

Información de configuración

Ese archivo es lo que se graba en Flash.

¿Se puede modificar la Flash desde el programa?

Sí, pero con restricciones.

El micro permite:

Autoescritura de Flash

Borrado por bloques

Programación por palabras

Esto se usa en:

Bootloaders

Actualización de firmware

Sistemas reprogramables

Pero no es común en aplicaciones básicas.

Diferencia Flash vs EEPROM

Flash

Guarda programa

Escritura por bloques

Más rápida en lectura

No se usa para variables

EEPROM

Guarda datos

Escritura por byte

Más lenta en escritura

Sí se usa para datos persistentes

Ciclo de ejecución del CPU

El CPU funciona así:

PC apunta a una dirección en Flash

Se lee una instrucción

Se decodifica

Se ejecuta

PC incrementa

Esto ocurre millones de veces por segundo.

Ejemplo aplicado al LED

Cuando escribiste:

GP0 = HIGH

Eso fue compilado en instrucciones máquina que están guardadas en Flash.

Cada vez que el micro ejecuta el bucle:

Está leyendo esas instrucciones desde Flash.

Concepto clave

Flash es el “libro de instrucciones”.

RAM es la “mesa de trabajo”.

EEPROM es el “archivo permanente”.

Limitaciones

No escribir constantemente

Escritura requiere borrado previo

Se borra por bloques (no byte individual)

Número limitado de ciclos de programación

Uso profesional

La Flash se usa también para:

Tablas de calibración

Lookup tables

Firmware actualizable

Bootloaders

Mensaje pedagógico fuerte

Sin Flash, el micro no piensa.

La Flash contiene la secuencia exacta de instrucciones que el CPU ejecuta.

No interpreta.

No improvisa.

Solo ejecuta lo que está grabado ahí.

Ahora entramos al corazón real del micro: **cómo se ve una instrucción en código máquina** y cómo interviene el **contador de programa (PC)**.

¿Cómo se ve una instrucción en código máquina?

En este micro:

Cada instrucción ocupa **una palabra de 14 bits**

Cada dirección de Flash contiene una instrucción

El CPU lee 14 bits en cada ciclo de instrucción

Ejemplo simple en ensamblador

Supongamos esta instrucción:

BSF LATA, 0

Significa:

Poner en 1 el bit 0 del registro LATA.

En código máquina

Internamente esa instrucción se codifica en 14 bits, algo como:

01 01 01 ffff bbb

(donde "ffff" representa dirección del registro

y "bbb" representa el número de bit)

Por ejemplo, podría verse como:

0x140C

Ese valor hexadecimal representa una instrucción completa.

Lo que realmente guarda la Flash

La Flash no guarda texto.

Guarda algo así:

Dirección	Contenido
-----------	-----------

0x0000	0x2805
--------	--------

0x0001	0x140C
--------	--------

0x0002	0x2FFF
--------	--------

Cada valor es una instrucción máquina.

Estructura general de una instrucción PIC

Las instrucciones se dividen en campos:

[OPCODE][Dirección][Bit]

Ejemplo conceptual:

0101 0011 0000 10

Primeros bits → tipo de operación

Siguientes bits → dirección de registro

Últimos bits → número de bit

Todo está codificado dentro de 14 bits.

¿Qué hace el CPU con eso?

El CPU realiza:

Lee los 14 bits desde la dirección apuntada por el PC

Decodifica el campo OPCODE

Ejecuta la operación

Incrementa el PC

El Contador de Programa (PC)

El PC es un registro interno que contiene:

La dirección de la siguiente instrucción en Flash.

Ejemplo paso a paso

Supongamos que el programa empieza en:

PC = 0x0000

Paso 1

Lee instrucción en 0x0000

Ejecuta

PC pasa a 0x0001

Paso 2

Lee instrucción en 0x0001

Ejecuta

PC pasa a 0x0002

Y así sucesivamente.

¿Qué pasa en un salto?

Si existe:

GOTO 0x0005

El PC no incrementa normalmente.

El CPU carga:

PC = 0x0005

Y la ejecución continúa desde ahí.

¿Qué pasa en un bucle while(1)?

El compilador genera algo como:

0x0003 GOTO 0x0003

Entonces:

PC llega a 0x0003

Ejecuta GOTO 0x0003

PC vuelve a 0x0003

Eso crea el bucle infinito.

¿Qué pasa en una interrupción?

Cuando ocurre una interrupción:

PC actual se guarda en la pila

PC se carga con la dirección del vector de interrupción

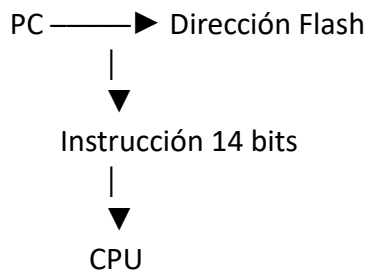
ejecuta la rutina

ejecuta RETFIE

PC vuelve al valor anterior

Relación PC – Flash – CPU

Visualmente:



El PC es el “dedo” que apunta a la siguiente línea del programa.

Analogía clara

La Flash es un libro.

Cada página es una instrucción.

El PC es el marcador de página.

El CPU lee la página señalada, la ejecuta y mueve el marcador.

Concepto clave

El micro no ejecuta líneas de código C.

Ejecuta palabras binarias de 14 bits.

Cada palabra es una instrucción máquina perfectamente definida.

El PC determina cuál se ejecuta.

Aplicado al ejemplo del LED

El programa:

GP0 = HIGH

GP0 = LOW

Se convierte en algo como:

BSF LATA,0

BCF LATA,0

GOTO ...

Cada una ocupa una dirección en Flash.

El PC las recorre secuencialmente.

Punto fuerte para el alumno

Cuando el LED parpadea:

No es magia.

Es el PC recorriendo direcciones de memoria
ejecutando instrucciones binarias
que modifican bits de registros
que activan transistores internos
que generan voltaje en un pin.

Ahora vamos a ver **una instrucción real**, con su **formato exacto de 14 bits**, tal como la ejecuta el CPU del **PIC12F1840**.

Arquitectura de 14 bits (Enhanced Mid-Range)

En esta familia:

Cada instrucción ocupa **14 bits**

Cada dirección de programa contiene una instrucción

El CPU trabaja directamente con esos 14 bits

Ejemplo real: BSF f, b

Instrucción:

BSF f, b

Significa:

Bit Set File

Poner en 1 el bit **b** del registro **f**

Formato exacto en 14 bits

Las instrucciones tipo “bit-oriented” tienen este formato:

01 01 bbb ffffff

Distribución exacta:

Campo	Bits	Significado
01 01	4 bits	Opcode base para BSF
bbb	3 bits	Número de bit (0–7)
ffffff	6 bits	Dirección del registro

Total = 4 + 3 + 6 = **13 bits**

Pero realmente el opcode completo ocupa:

0101 bbb ffffff

En total 14 bits.

Ejemplo concreto

Queremos ejecutar:

BSF LATA, 0

Supongamos que:

LATA está en dirección 0x0C (ejemplo simplificado para entender el formato).

En binario:

Bit = 000 (porque queremos el bit 0)

f = 001100 (0x0C en 6 bits)

Entonces la instrucción queda:

0101 000 001100

En binario completo:

01010000001100

En hexadecimal:

0x140C (ejemplo ilustrativo)

Eso es exactamente lo que la Flash guarda.

No guarda texto.

Guarda ese número.

Otro ejemplo: BCF f, b

BCF LATA,0

Formato:

0100 bbb ffffff

Si bit = 000 y f = 001100:

0100 000 001100

Hex aproximado:

0x100C

Otro tipo de instrucción: GOTO k

Formato distinto.

Para salto:

10 kkkkkkkkkkk

Donde:

10 = opcode

k = dirección destino

Ejemplo:

GOTO 0x0003

Se convierte en algo como:

10000000000011

El CPU carga ese valor en el PC.

Qué ocurre en hardware

Cuando el PC apunta a una dirección:
Se leen 14 bits desde Flash
Se cargan en el registro de instrucción
El decodificador interpreta los bits
Se activa el bloque interno correspondiente
Para BSF:
Se activa la unidad lógica
Se selecciona registro f
Se activa línea del bit b
Se escribe el nuevo valor

Todo ocurre en un ciclo de instrucción.

Flujo real

Flash → Registro de instrucción → Decodificador → ALU → Registro f

Concepto clave para el alumno

Cuando escribes:
GP0 = HIGH
El compilador genera:
BSF LATA,0
Y lo que realmente vive en la Flash es:
01010000001100
Eso es lo que el CPU entiende.

Mensaje potente

El micro no entiende:
GP0
HIGH
While
C

Entiende:
Catorce bits perfectamente ordenados.
Eso es todo.

TEMPORIZADORES (TIMERS)

Aquí entramos a uno de los temas más importantes en microcontroladores: **los temporizadores (Timers)**.

¿Por qué necesitamos un Timer?

En el ejemplo anterior del LED:

```
while(1){  
    GPIO = HIGH  
    GPIO = LOW  
}
```

El LED **no parpadea visiblemente**.

¿Por qué?

Porque el micro trabaja a millones de ciclos por segundo.

Si el reloj interno está en 4 MHz:

El CPU ejecuta ~1 millón de instrucciones por segundo.

El LED cambia de estado cientos de miles de veces por segundo.

El ojo humano solo detecta cambios menores a ~60 Hz.

Resultado:

El LED parece encendido fijo.

En realidad está conmutando extremadamente rápido.

¿Qué es un Timer?

Un Timer es:

Un contador automático que incrementa su valor con cada pulso de reloj.

Es hardware dedicado.

Funciona aunque el programa esté haciendo otra cosa.

¿Para qué sirve?

Permite:

Generar retardos precisos

Medir tiempo

Generar señales periódicas

Activar interrupciones

Crear PWM

Contar eventos externos

Timers en el PIC12F1840

Este micro tiene:

Timer	Tamaño
Timer0	8 bits
Timer1	16 bits
Timer2	8 bits + comparador

Cómo funciona un Timer

Ejemplo: Timer0 (8 bits)

Tiene un registro llamado **TMRO**.

Cuenta desde 0 hasta 255.

Cuando llega a 255 y vuelve a 0 → ocurre un desbordamiento (overflow).

Ese evento puede:

Activar una bandera

Generar una interrupción

Ejemplo conceptual

Si el reloj es 4 MHz:

Frecuencia de instrucción = 1 MHz

(En PIC clásicos, cada instrucción tarda 4 ciclos de reloj)

Eso significa:

1 incremento cada 1 microsegundo.

Si Timer0 cuenta hasta 256:

$256 \mu s \approx 0.000256$ segundos.

Eso aún es muy rápido.

¿Cómo lo hacemos más lento?

Usando un **prescaler**.

El prescaler divide la frecuencia antes de entrar al timer.

Ejemplo:

Prescaler 1:256

Ahora el timer incrementa cada $256 \mu s$.

Entonces:

$256 \text{ cuentas} \times 256 \mu s \approx 65 \text{ ms}$

Ahora ya estamos en tiempos visibles.

Aplicado al parpadeo del LED

Queremos que el LED cambie cada 500 ms ($\frac{1}{2}$ segundo).

Proceso:

Configuramos Timer0

Esperamos overflow

Cada cierto número de overflows, cambiamos el LED

Ahora sí el parpadeo será visible.

Concepto clave

Sin timer:

El retardo depende de velocidad del CPU.

Con timer:

El tiempo depende del hardware dedicado.

Eso da precisión.

Analogía didáctica

Sin timer:

Es como contar segundos diciendo:

"uno, dos, tres..."

(depende de qué tan rápido hables)

Con timer:

Es como usar un reloj digital interno.

Flujo interno real

Oscilador → Prescaler → Timer → Overflow → Bandera → CPU

El CPU solo revisa la bandera.

El hardware hace el conteo.

Por qué esto es fundamental

Sin timers no podrías:

Generar PWM

Medir frecuencia

Hacer reloj digital

Crear comunicación serial precisa

Controlar motores

Conclusión para el alumno

El LED no parpadea porque el micro es demasiado rápido.

El Timer es la herramienta que permite controlar el tiempo de forma precisa.

El tiempo en microcontroladores no se "espera".

Se cuenta.

Ejemplo

Parpadeo visible usando Timer0

Objetivo

Hacer que el LED conectado a GP0 parpadee aproximadamente cada **500 ms**, usando hardware de temporización (Timer0).

Datos del sistema

Usaremos:

Oscilador interno = 4 MHz

Frecuencia de instrucción = $F_{osc} / 4 = 1 \text{ MHz}$

1 ciclo de instrucción = $1 \mu s$

Paso 1 – Configuración general

Configurar GP0 como salida

TRISA0 = 0

Paso 2 – Configurar Timer0

Timer0 es un contador de 8 bits:

Cuenta de 0 a 255

Cuando pasa de 255 a 0 → overflow

Prescaler

Usaremos prescaler 1:256

Esto significa:

El Timer incrementa cada:

$256 \times 1 \mu s = 256 \mu s$

Tiempo hasta overflow

Timer cuenta 256 incrementos:

$256 \times 256 \mu s = 65.536 \text{ ms}$

Un overflow $\approx 65 \text{ ms}$

Paso 3 – Obtener 500 ms

$500 \text{ ms} / 65 \text{ ms} \approx 7.6$

Necesitamos aproximadamente 8 overflows.

Código ejemplo en C

```
void main() {  
  
    TRISA0 = 0;    // GP0 como salida  
  
    OPTION_REG = 0b00000111;  
    // Prescaler 1:256 asignado a Timer0  
  
    TMR0 = 0;    // Reiniciar contador  
  
    unsigned char contador = 0;  
  
    while(1){  
  
        if(TMR0 == 0){    // Detectar overflow  
  
            contador++;  
  
            if(contador >= 8){  
                LATA0 = !LATA0; // Cambiar estado del LED  
                contador = 0;  
            }  
        }  
    }  
}
```

¿Qué hace este programa?

Timer0 comienza a contar.
Cada vez que desborda (cada ~65 ms),
Incrementa una variable contador.
Cuando llega a 8 desbordamientos (~524 ms),
Cambia el estado del LED.
El LED ahora parpadea visible.

Comparación clara

Sin Timer

El LED cambia millones de veces por segundo.
No se ve.

Con Timer

El cambio ocurre cada ~0.5 segundos.
Visible al ojo humano.

¿Qué ocurre internamente?

Flujo real

Oscilador → Prescaler → Timer0 → Overflow → CPU detecta → Cambia LATA0
El CPU no cuenta el tiempo.
El hardware lo hace.

Forma profesional (usando interrupción)

Una versión más correcta sería usar la interrupción por overflow:

```
void __interrupt() ISR(){  
  
    if(TMR0IF){  
  
        TMR0IF = 0; // Limpiar bandera  
  
        contador++;  
  
        if(contador >= 8){  
            LATA0 = !LATA0;  
            contador = 0;  
        }  
    }  
}
```

Esto es mejor porque:
El CPU no está preguntando constantemente
El hardware avisa cuando ocurre el evento

Concepto clave

Timer0 funciona independientemente del programa.
El tiempo ya no depende de velocidad del CPU.
Depende del hardware interno.

Resultado visual esperado

LED:

ON → 0.5 s

OFF → 0.5 s

ON → 0.5 s

OFF → 0.5 s

Parpadeo claro y visible.

Un microcontrolador no "espera".

Cuenta tiempo usando hardware dedicado.

Eso es control temporal real.

Cálculo EXACTO de 500 ms con Timer0

Datos base

Oscilador interno = **4 MHz**

En PIC:

Frecuencia de instrucción = $F_{osc} / 4$

Entonces:

$4 \text{ MHz} / 4 = 1 \text{ MHz}$

Cada ciclo de instrucción dura:

$1 / 1 \text{ MHz} = 1 \mu\text{s}$

Timer0 con prescaler 1:256

El timer incrementa cada:

$256 \times 1 \mu\text{s} = 256 \mu\text{s}$

¿Cuántos incrementos necesitamos para 500 ms?

$500 \text{ ms} = 500,000 \mu\text{s}$

Dividimos:

$500,000 / 256 = 1953.125$ incrementos

Necesitamos exactamente **1953 incrementos**.

Timer0 es de 8 bits

Cuenta de 0 a 255 $\rightarrow$ 256 incrementos por overflow.

Entonces:

$1953 / 256 = 7.63$ overflows

No es entero.

Por lo tanto, no podemos usar solo overflows completos.

Solución exacta

Usamos:

7 overflows completos

Más un conteo parcial

7 overflows completos

$7 \times 256 = 1792$ incrementos

Faltan:

$1953 - 1792 = 161$ incrementos

Ajuste final con precarga

Queremos que el último conteo sea solo 161 incrementos.

Timer0 desborda después de 256 incrementos.

Entonces lo precargamos con:

$$256 - 161 = 95$$

Es decir:

$$TMR0 = 95$$

Resultado exacto

Secuencia:

Contar 7 overflows completos

En el último ciclo cargar TMR0 = 95

Esperar overflow final

Tiempo total:

$$(7 \times 256 + 161) \times 256 \mu s = 1953 \times 256 \mu s$$

$$= 500,000 \mu s$$

$$= 500 \text{ ms EXACTOS}$$

Sin aproximación.

Registro OPTION_REG (bit a bit)

Formato:

Bit:	7	6	5	4	3	2	1	0
	nWPUEN	INTEDG	TMR0CS	TMR0SE	PSA	PS2	PS1	PS0

Explicación de cada bit

Bit	Nombre	Función
7	nWPUEN	Habilita pull-ups
6	INTEDG	Flanco de interrupción externa
5	TMR0CS	Fuente de reloj de Timer0
4	TMR0SE	Selección de flanco externo
3	PSA	Asignación del prescaler
2	PS2	Prescaler bit 2
1	PS1	Prescaler bit 1
0	PS0	Prescaler bit 0

Para nuestro ejemplo

Queremos:

Fuente interna → TMR0CS = 0

Prescaler asignado a Timer0 → PSA = 0

Prescaler 1:256 → PS2:PS0 = 111

Entonces:

PS2 PS1 PS0 = 111

OPTION_REG sería:

00000111

En binario:

0b00000111

Registro TMR0IF

TMR0IF es la bandera de interrupción por overflow.

Está dentro del registro INTCON.

INTCON (parcial)

Bit: 7 6 5 4 3 2 1 0

GIE PEIE TMR0IE INTE IOCIE TMR0IF INTF IOCIF

TMR0IF (bit 2)

Se pone en 1 cuando ocurre overflow.

Debe limpiarse manualmente.

Si TMR0IE y GIE están activados → genera interrupción.

¿Qué pasa en un overflow?

Cuando TMR0 pasa de 255 a 0:

TMR0IF = 1

Si interrupciones activadas:

Se ejecuta ISR

El programa debe limpiar:

TMR0IF = 0

Si no se limpia, la interrupción se repite.

Flujo completo profesional

Oscilador



Prescaler



TMR0 incrementa



Overflow



TMR0IF = 1



CPU detecta



Cambiar LED



Limpiar bandera

Conclusión técnica

Podemos calcular tiempos exactos

Timer0 es hardware dedicado

OPTION_REG controla el prescaler

TMR0IF indica evento de tiempo

El control temporal ya no depende del software

Ahora vamos a escribir el **código completo**, con cálculo exacto de **500 ms**, usando:

Oscilador interno = 4 MHz

Timer0

Prescaler 1:256

Precarga en el último ciclo

Sin aproximaciones

Recordatorio del cálculo exacto

Frecuencia de instrucción:

$F_{osc} / 4 = 4 \text{ MHz} / 4 = 1 \text{ MHz}$

Cada ciclo = $1 \mu s$

Con prescaler 1:256:

Incremento Timer0 cada $256 \mu s$

Para 500 ms:

$500,000 \mu s / 256 \mu s = 1953 \text{ incrementos}$

Descomposición:

7 overflows completos $\rightarrow 7 \times 256 = 1792$

Faltan 161 incrementos

Precarga final:

$256 - 161 = 95$

Código completo en C (XC8)

```
#include <xc.h>

// CONFIG bits (ejemplo básico)
#pragma config FOSC = INTOSC
#pragma config WDTE = OFF
#pragma config PWRTE = OFF
#pragma config MCLRE = OFF
#pragma config CP = OFF
#pragma config BOREN = OFF
#pragma config CLKOUTEN = OFF

#define _XTAL_FREQ 4000000

volatile unsigned char overflow_count = 0;

void __interrupt() ISR(void)
{
    if (TMR0IF) // ¿Hubo overflow?
    {
        TMR0IF = 0; // Limpiar bandera

        overflow_count++;

        if (overflow_count < 7)
        {
            // Overflows normales
            TMR0 = 0;
        }
        else
        {
            // Último ciclo con precarga
            TMR0 = 95; // Precarga exacta
            overflow_count = 0;

            // Cambiar estado del LED
            LATA ^= 0x01; // Toggle bit 0
        }
    }
}

void main(void)
{
    // Oscilador interno 4 MHz
    OSCCON = 0b01101010;

    // GP0 como salida
    TRISA = 0b11111110;
```

```

// Inicializar LED apagado
LATA = 0x00;

// Configurar Timer0
OPTION_REG = 0b00000111;
// PSA = 0 (prescaler asignado a TMR0)
// PS2:PS0 = 111 (1:256)
// TMR0CS = 0 (reloj interno)

TMR0 = 0;

// Habilitar interrupción de Timer0
TMR0IE = 1;
PEIE = 1;
GIE = 1;

while(1)
{
    // El micro queda libre.
    // El parpadeo ocurre por hardware + interrupción.
}
}

```

Qué hace exactamente este programa

Timer0 incrementa cada 256 μ s
 Cada overflow genera interrupción
 Se cuentan 7 overflows completos
 En el último ciclo se carga TMR0 = 95
 Se completa exactamente 500 ms
 Se conmuta el LED
 El proceso se repite

Resultado real

LED:
 ON → 500 ms
 OFF → 500 ms
 ON → 500 ms
 OFF → 500 ms
 Tiempo exacto de 1 segundo por ciclo completo.

Por qué esta versión es profesional

No usa retardos por software
 No bloquea el CPU
 Usa interrupciones
 Usa cálculo exacto
 Control temporal basado en hardware

Concepto fuerte para el alumno

Esto ya no es un “delay”.

Es control temporal basado en contadores digitales internos.

El tiempo lo genera el hardware.

El software solo responde al evento.

Ahora vamos a hacer **el mismo parpadeo exacto de 500 ms**, pero usando **Timer1 (16 bits)**.
Verás que es más simple y elegante que con Timer0.

¿Por qué Timer1 es más simple?

Timer1 es de **16 bits**:

Cuenta de:

0 → 65535

Eso permite generar tiempos largos sin acumular múltiples overflows.

Con Timer0 tuvimos que contar 7 overflows + precarga.

Con Timer1 solo necesitamos **una sola precarga**.

Datos del sistema

Oscilador interno = 4 MHz

Frecuencia de instrucción:

$F_{osc} / 4 = 1 \text{ MHz}$

Cada tick = 1 μs

Queremos 500 ms

500 ms = 500,000 μs

Timer1 incrementa cada 1 μs (sin prescaler).

Entonces necesitamos:

500,000 incrementos

Pero Timer1 máximo cuenta 65,536.

No alcanza.

Usamos prescaler 1:8

Timer1 tiene prescaler:

1:1

1:2

1:4

1:8

Si usamos 1:8:
Incremento cada 8 μ s
Entonces:
 $500,000 / 8 = 62,500$ incrementos
Eso sí cabe en 16 bits.

Cálculo de precarga exacta

Timer1 desborda en 65,536.
Queremos que desborde después de 62,500 incrementos.
Entonces:
Precarga = 65536 - 62500
Precarga = 3036
En hexadecimal:
3036 decimal = 0x0BDC
Entonces:
TMR1H = 0x0B
TMR1L = 0xDC
Tiempo exacto:
 $62500 \times 8 \mu s = 500,000 \mu s$
= 500 ms exactos

Código completo con Timer1 (XC8)

```
#include <xc.h>

// CONFIG bits (ejemplo básico)
#pragma config FOSC = INTOSC
#pragma config WDTE = OFF
#pragma config PWRTE = OFF
#pragma config MCLRE = OFF
#pragma config CP = OFF
#pragma config BOREN = OFF
#pragma config CLKOUTEN = OFF

#define _XTAL_FREQ 4000000
```

```

void __interrupt() ISR(void)
{
    if (TMR1IF)
    {
        TMR1IF = 0; // Limpiar bandera

        // Recargar para 500 ms exactos
        TMR1H = 0x0B;
        TMR1L = 0xDC;

        // Cambiar estado del LED
        LATA ^= 0x01;
    }
}

```

```

void main(void)
{
    // Oscilador interno 4 MHz
    OSCCON = 0b01101010;

    // GP0 como salida
    TRISA = 0b11111110;

    LATA = 0x00;

    // Configurar Timer1
    T1CON = 0b00110001;
    /*
        Bit 7-6: no usados
        Bit 5-4: Prescaler 1:8 (11)
        Bit 3: Oscilador desactivado
        Bit 2: Reloj interno (Fosc/4)
        Bit 0: Timer1 ON
    */

    // Precarga inicial
    TMR1H = 0x0B;
    TMR1L = 0xDC;

    // Habilitar interrupción
    TMR1IE = 1;
    PEIE = 1;
    GIE = 1;

    while(1)
    {
        // El micro queda libre.
    }
}

```

¿Qué ocurre ahora?

Timer1 cuenta internamente
Después de 62,500 incrementos (500 ms)
Se genera overflow
Se ejecuta la interrupción
Se recarga el timer
Se conmuta el LED
Y se repite.

Comparación directa

Timer0	Timer1
8 bits	16 bits
Múltiples overflows	Solo uno
Más cálculos	Más simple
Menor rango	Mayor rango

Resultado

LED:
ON → 500 ms
OFF → 500 ms
Periodo completo: 1 segundo exacto.

Timer1 es más limpio para retardos largos.
Timer0 es útil para eventos rápidos o divisiones.
En diseño real, elegir el timer correcto simplifica el sistema.

Ahora vamos a analizar **T1CON bit a bit** y luego veremos qué ocurre si cambiamos el reloj a **8 MHz** en el **PIC12F1840**.

Registro T1CON (Timer1 Control)

Formato típico:

Bit:	7	6	5	4	3	2	1	0
	—	—	T1CKPS1	T1CKPS0	T1OSCEN	TMR1CS	—	TMR1ON

(Algunos bits pueden variar ligeramente según configuración, pero conceptualmente funcionan así).

Bit 0 — TMR1ON

1 → Timer1 encendido
0 → Timer1 apagado

Si este bit está en 0, el timer no cuenta.

Bit 1 — (No usado o sincronización)

En algunos modos controla sincronización cuando se usa reloj externo.
En modo interno generalmente no afecta.

Bit 2 — TMR1CS (Clock Source)

Selecciona la fuente de reloj:
0 → Reloj interno ($F_{osc}/4$)
1 → Reloj externo (pin T1CKI)

Para temporización precisa normalmente usamos:
TMR1CS = 0

Bit 3 — T1OSCEN

1 → Activa oscilador secundario para Timer1
0 → Desactivado

Solo se usa cuando se conecta cristal externo para reloj independiente.

Bits 5-4 — T1CKPS1:T1CKPS0 (Prescaler)

Determinan el prescaler:

T1CKPS1	T1CKPS0	Prescaler
0	0	1:1
0	1	1:2
1	0	1:4
1	1	1:8

En nuestro ejemplo usamos:
1 1 → 1:8

Resumen de nuestra configuración anterior

T1CON = 0b00110001;
Desglose:
Prescaler 1:8
Fuente interna
Timer encendido

¿Qué pasa si cambiamos el reloj a 8 MHz?

Antes:

$$F_{osc} = 4 \text{ MHz}$$

$$F_{inst} = F_{osc} / 4 = 1 \text{ MHz}$$

$$\text{Tick} = 1 \mu\text{s}$$

Ahora:

$$F_{osc} = 8 \text{ MHz}$$

$$F_{inst} = 8 / 4 = 2 \text{ MHz}$$

$$\text{Tick} = 0.5 \mu\text{s}$$

El micro ejecuta el doble de rápido.

Impacto en Timer1

Si mantenemos prescaler 1:8:

Incremento ahora será:

$$0.5 \mu\text{s} \times 8 = 4 \mu\text{s}$$

Antes era:

$$1 \mu\text{s} \times 8 = 8 \mu\text{s}$$

Ahora el timer cuenta el doble de rápido.

Recalculando 500 ms con 8 MHz

Queremos 500,000 μs .

Cada incremento = 4 μs .

Entonces:

$$500,000 / 4 = 125,000 \text{ incrementos}$$

Pero Timer1 solo puede contar hasta 65,536.

No cabe.

Solución con 8 MHz

Debemos usar prescaler menor impacto o dividir en dos eventos.

Probemos prescaler 1:8 ya es máximo.

Entonces necesitamos usar **dos overflows**.

Cálculo correcto

Máximo conteo por overflow:

$$65536 \times 4 \mu\text{s} = 262,144 \mu\text{s}$$

$$\approx 262 \text{ ms}$$

Entonces:

500 ms requiere aproximadamente 2 overflows.

Precarga para exactitud

Queremos exactamente 125,000 incrementos.

Dividimos:

$$125,000 / 65536 \approx 1.9$$

Usamos:

1 overflow completo

Más conteo parcial

Primer overflow:

65536 incrementos

Faltan:

$$125,000 - 65,536 = 59,464 \text{ incrementos}$$

Precarga:

$$65536 - 59464 = 6,072$$

En hexadecimal:

$$6072 = 0x17B8$$

Conclusión técnica

Si duplicas la frecuencia:

El timer cuenta más rápido

Los tiempos se reducen a la mitad

Debes recalcular siempre

Regla de oro

El tiempo depende de:

$$\text{Tiempo} = (\text{Precarga y conteo}) \times (\text{Prescaler}) \times (\text{Periodo instrucción})$$

Si cambia el reloj, cambia todo.

Mensaje clave para el alumno

El micro no sabe lo que es un segundo.

Solo cuenta pulsos de reloj.

Si el reloj cambia, el tiempo cambia.

El Módulo ADC (Convertidor Analógico-Digital)

Este módulo es clave porque conecta el mundo **analógico** con el mundo **digital**.

¿Por qué necesitamos un ADC?

Un microcontrolador trabaja con:

Bits

0 y 1

Niveles lógicos

Pero el mundo real tiene:

Voltajes variables

Temperatura

Luz

Sonido

Potenciómetros

Sensores analógicos

El ADC convierte un voltaje analógico en un número digital.

¿Qué hace exactamente el ADC?

Convierte un voltaje entre:

0V → Vref

En un número binario de 10 bits.

10 bits significa:

$2^{10} = 1024$ niveles

Entonces:

Voltaje	Valor digital
0 V	0
Vref	1023
Mitad de Vref	~512

Resolución

Si Vref = 5V:

$5V / 1024 \approx 4.88$ mV por paso

Cada incremento digital representa aproximadamente 4.88 mV.

Registros principales del ADC

Registro	Función
ADCON0	Control principal
ADCON1	Configuración
ADRESL	Resultado bajo
ADRESH	Resultado alto

ADCON0 bit a bit (conceptual)

Formato simplificado:

Bit:	7	6	5	4	3	2	1	0
	—	CHS2	CHS1	CHS0	GO	—	—	ADON

GO/DONE

1 → Inicia conversión

0 → Conversión terminada

Cuando termina la conversión, el hardware pone GO en 0 automáticamente.

Flujo de conversión

Seleccionar canal

Encender ADC

Esperar tiempo de adquisición

Poner GO = 1

Esperar que GO vuelva a 0

Leer ADRESH:ADRESL

Ejemplo práctico

Supongamos:

Potenciómetro conectado a AN0

Queremos leer su valor

Código conceptual:

```
ADCON0bits.CHS = 0; // Canal AN0
```

```
ADCON0bits.ADON = 1; // Encender ADC
```

```
__delay_us(5); // Tiempo adquisición
```

```
GO_nDONE = 1; // Iniciar conversión
```

```
while(GO_nDONE); // Esperar
```

```
unsigned int valor = (ADRESH << 8) | ADRESL;
```

Ahora valor tendrá un número entre 0 y 1023.

Aplicación práctica

Con ese valor puedes:

Regular brillo de LED (PWM)

Medir temperatura

Leer sensores

Ajustar velocidad de motor

Detectar nivel de batería

Analogía didáctica

El ADC es como una regla digital.

Mide cuánto voltaje hay.

Y lo convierte en un número que el micro entiende.

Flujo interno real

Voltaje en pin



Muestreo y retención (Sample & Hold)



Conversor interno



Registro ADRES



CPU lee valor

Concepto clave

El micro no mide voltajes directamente.

Solo recibe números.

El ADC es el traductor entre el mundo físico y el mundo digital.

Ejemplo didáctico potente

Podemos hacer:

Potenciómetro controla velocidad de parpadeo del LED.

ADC lee valor.

Timer usa ese valor como periodo.

Eso ya es sistema interactivo real.

Ahora entramos al análisis fino del módulo ADC del **PIC12F1840**.

Registro ADCON1 – Bit a Bit

El registro **ADCON1** controla:

Fuente de reloj del ADC

Referencias de voltaje

Justificación del resultado

Formato típico (simplificado):

Bit:	7	6	5	4	3	2	1	0
	ADFM	—	ADCS2	ADCS1	ADCS0	—	—	—

(Algunos bits pueden variar ligeramente según configuración, pero funcionalmente operan así.)

Bit 7 — ADFM (A/D Result Format)

Define cómo se almacenan los 10 bits del resultado en ADRESH:ADRESL.

1 → Justificado a la derecha

0 → Justificado a la izquierda

Recomendación

Usar:

ADFM = 1

Así el resultado se lee fácilmente como entero de 10 bits.

Bits 5-3 — ADCS2:ADCS0 (ADC Clock Select)

Seleccionan el reloj del ADC.

Opciones típicas:

ADCS2	ADCS1	ADCS0	Fuente
0	0	0	Fosc/2
0	0	1	Fosc/8
0	1	0	Fosc/32
1	0	0	FRC (reloj interno RC)

El ADC necesita un reloj específico llamado **TAD**.

¿Qué es TAD?

TAD es el periodo de reloj interno del ADC.

Cada conversión requiere:

11 TAD

(10 bits + tiempo interno adicional)

Requisito técnico importante

El datasheet establece que:

TAD mínimo $\approx 1.6 \mu s$

Si TAD es muy pequeño, la conversión es incorrecta.

Cálculo de TAD con Fosc = 4 MHz

Recordemos:

$F_{inst} = F_{osc} / 4 = 1 \text{ MHz}$

Periodo instrucción = $1 \mu s$

Pero el ADC usa Fosc directamente.

Caso 1: ADCS = Fosc/2

$4 \text{ MHz} / 2 = 2 \text{ MHz}$

Periodo = $0.5 \mu s$

TAD = $0.5 \mu s \rightarrow$ Demasiado pequeño

Caso 2: ADCS = Fosc/8

$4 \text{ MHz} / 8 = 500 \text{ kHz}$

Periodo = $2 \mu s$

TAD = $2 \mu s \rightarrow$ Correcto

Caso 3: ADCS = Fosc/32

$4 \text{ MHz} / 32 = 125 \text{ kHz}$

Periodo = $8 \mu s$

TAD = $8 \mu s \rightarrow$ También correcto

Tiempo exacto de conversión (4 MHz)

Si usamos $F_{osc}/8$:

$$TAD = 2 \mu s$$

Conversión requiere 11 TAD:

$$11 \times 2 \mu s = 22 \mu s$$

$$\text{Tiempo total} \approx 22 \mu s$$

Ahora con $F_{osc} = 8 \text{ MHz}$

$$F_{osc} = 8 \text{ MHz}$$

Si usamos $F_{osc}/8$

$$8 \text{ MHz} / 8 = 1 \text{ MHz}$$

$$\text{Periodo} = 1 \mu s$$

$$TAD = 1 \mu s \rightarrow \text{Demasiado pequeño}$$

Si usamos $F_{osc}/32$

$$8 \text{ MHz} / 32 = 250 \text{ kHz}$$

$$\text{Periodo} = 4 \mu s$$

$$TAD = 4 \mu s \rightarrow \text{Correcto}$$

Tiempo de conversión a 8 MHz

$$11 \times 4 \mu s = 44 \mu s$$

La conversión tarda más porque elegimos un divisor mayor.

Comparación directa

Fosc	ADCS	TAD	Tiempo conversión
4 MHz	$F_{osc}/8$	$2 \mu s$	$22 \mu s$
8 MHz	$F_{osc}/32$	$4 \mu s$	$44 \mu s$

Flujo completo real

Seleccionar canal

↓

Tiempo de adquisición (T_{acq})

↓

Iniciar conversión

↓

11 TAD

↓

Resultado en ADRES

Tiempo de adquisición

Antes de iniciar conversión se debe esperar típicamente:

$$T_{acq} \approx 5 \mu s$$

Entonces tiempo total real $\approx$

Para 4 MHz:

$$5 \mu s + 22 \mu s = 27 \mu s$$

Mensaje clave para el alumno

El ADC no es instantáneo.

Necesita:

Tiempo de adquisición

Tiempo de conversión

Y depende del reloj del sistema.

Si aumentas Fosc sin ajustar ADCS, el ADC deja de ser preciso.

Concepto fuerte

El ADC no mide voltaje en tiempo real continuo.

Toma una muestra, la convierte, y entrega un número.

Es un proceso discreto.

Ahora analizamos el registro **ANSELA** bit a bit en el **PIC12F1840**.

Este registro es fundamental cuando usamos el módulo ADC.

¿Qué hace ANSELA?

ANSELA significa:

Analog Select Register A

Define si cada pin del puerto A funciona como:

Entrada analógica (ADC)

Entrada/salida digital normal

Si no configuras correctamente ANSELA, el ADC no funciona bien.

Formato de ANSELA

Bit: 7 6 5 4 3 2 1 0
— — ANSA5 ANSA4 ANSA3 ANSA2 ANSA1 ANSA0

Significado de cada bit

Bit	Pin	Función
5	ANSA5	Controla modo analógico en GP5
4	ANSA4	Controla modo analógico en GP4
3	ANSA3	Controla modo analógico en GP3
2	ANSA2	Controla modo analógico en GP2
1	ANSA1	Controla modo analógico en GP1
0	ANSA0	Controla modo analógico en GP0

Valor de cada bit

1 → Pin configurado como entrada analógica

0 → Pin configurado como digital

Ejemplo práctico

Si queremos usar AN0 (GP0) como entrada analógica:

ANSELA = 0b00000001;

Esto significa:

GP0 → Analógico

Los demás → Digital

Muy importante

Si un pin está configurado como analógico:

El módulo digital se deshabilita automáticamente.

Eso significa:

No puedes usarlo como salida digital.

No puedes leerlo como digital.

Solo funciona como entrada analógica.

Ejemplo común de error

Si intentas hacer:

```
ANSELA = 0b00000001;
```

```
TRISA0 = 0; // Intentar usar como salida
```

No funcionará correctamente.

Porque:

Un pin analógico no puede funcionar como salida digital.

Configuración típica para ADC

Supongamos:

Potenciómetro en GP0

LED en GP1

Configuración correcta:

```
ANSELA = 0b00000001; // GP0 analógico
```

```
TRISA = 0b00000001; // GP0 entrada, GP1 salida
```

Flujo correcto para usar ADC

Activar modo analógico en ANSELA

Configurar TRISA como entrada

Seleccionar canal en ADCON0

Iniciar conversión

Estado por defecto al encender

Muchos pines arrancan en modo analógico.

Por eso, si el puerto no responde digitalmente, muchas veces el problema es ANSELA.

Concepto clave

ANSELA no convierte voltajes.

Solo conecta internamente el pin al módulo ADC.

Es como un interruptor que decide:

Ruta digital

Ruta analógica

Analogía didáctica

Imagina que el pin tiene dos cables internos:

Uno hacia el mundo digital

Uno hacia el ADC

ANSELA decide cuál se conecta.

Ejemplo completo típico

```
ANSELA = 0b00000001; // Solo AN0 analógico  
ADCON0bits.CHS = 0; // Canal AN0  
ADCON0bits.ADON = 1; // Encender ADC
```

Mensaje fuerte para el alumno

Si el ADC no funciona:
Primero revisa ANSELA.
Es uno de los errores más comunes en laboratorio.

Módulo PWM (Pulse Width Modulation)

Basado en el **PIC12F1840**.
Este módulo permite controlar:
Brillo de LED
Velocidad de motor
Potencia entregada
Señales moduladas

¿Qué es PWM?

PWM significa:
Modulación por Ancho de Pulso
No cambia el voltaje.
Cambia el tiempo que la señal permanece en alto dentro de un periodo fijo.

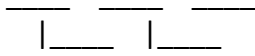
Concepto fundamental

Una señal PWM tiene:
Periodo fijo
Tiempo en alto variable

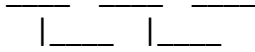
Se define por:
$$\text{Duty Cycle} = (\text{Tiempo en alto} / \text{Periodo total}) \times 100\%$$

Ejemplo visual

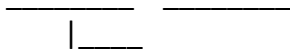
25% de duty cycle



50% de duty cycle



75% de duty cycle



¿Por qué el LED cambia de brillo?

Si el PWM es lo suficientemente rápido:

El ojo no ve el parpadeo

Percibe el promedio de energía

Ejemplo:

50% duty → LED parece a mitad de brillo.

Cómo funciona internamente en el PIC?

El PWM usa:

Timer2

Registro PR2 (periodo)

Registro CCPR1L (duty)

Comparador interno

Registros principales

Registro	Función
T2CON	Control de Timer2
PR2	Periodo PWM
CCP1CON	Configuración PWM
CCPR1L	Parte alta del duty
CCP1CON<5:4>	Parte baja del duty

Frecuencia del PWM

La frecuencia está determinada por:

$$F_{\text{pwm}} = F_{\text{osc}} / (4 \times \text{Prescaler} \times (\text{PR2} + 1))$$

Ejemplo real

Supongamos:

Fosc = 4 MHz

Prescaler = 4

PR2 = 249

Entonces:

$F_{pwm} = 4,000,000 / (4 \times 4 \times 250)$

$F_{pwm} = 4,000,000 / 4000$

$F_{pwm} = 1000 \text{ Hz}$

PWM a 1 kHz.

Perfecto para LED.

Duty Cycle (10 bits)

El duty cycle usa 10 bits:

0 a 1023

Fórmula aproximada:

$\text{Duty} = (\text{CCPR1L:DC1B}) / 4(\text{PR2}+1)$

En práctica:

0 → 0%

512 → 50%

1023 → 100%

Código básico PWM

```
TRISA2 = 0;    // Pin CCP1 como salida
```

```
PR2 = 249;    // Periodo
```

```
CCPR1L = 125; // 50% aproximado
```

```
CCP1CON = 0b00001100; // Modo PWM
```

```
T2CON = 0b00000101; // Prescaler 4, Timer2 ON
```

¿Qué ocurre internamente?

Flujo real:

Oscilador

↓

Timer2 cuenta

↓

Comparador compara con duty

↓

Salida cambia automáticamente

El CPU no interviene constantemente.

Es hardware automático.

Aplicaciones reales

Control de brillo
Control de ventilador
Regulación de potencia
Control de servomotor
Fuentes conmutadas

Diferencia entre PWM y cambiar voltaje

El micro no genera voltaje analógico real.
Genera pulsos digitales.
El promedio produce efecto equivalente.

Concepto clave

El PWM es control de potencia digital.
No cambia el voltaje.
Cambia el tiempo.

Mensaje fuerte

El ADC permite medir el mundo real.
El PWM permite actuar sobre el mundo real.
Uno mide.
El otro controla.

Ahora vamos a unir **ADC + PWM** para hacer algo profesional:

Ejemplo completo

Potenciómetro controla el brillo del LED

Basado en el **PIC12F1840**.

Aquí el sistema:

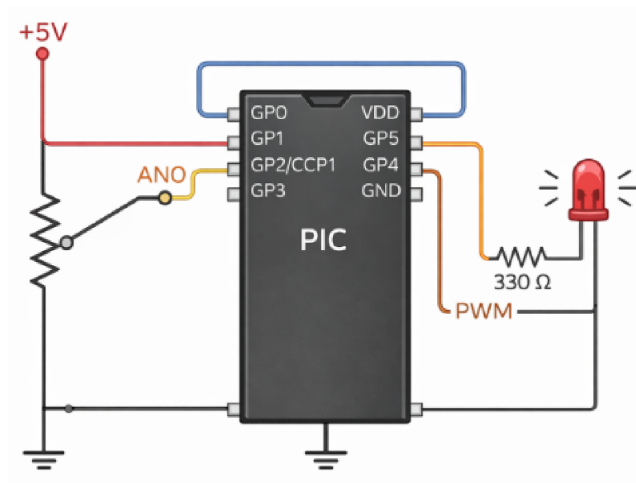
Potenciómetro → Entrada analógica (AN0)

ADC mide voltaje (0–1023)

Ese valor controla el duty cycle del PWM

PWM regula el brillo del LED

Conexión eléctrica



Idea general del programa

Leer ADC (0–1023)



Asignar valor a PWM



Cambiar brillo automáticamente

Sin cálculos complejos.

El ADC ya entrega un valor de 10 bits.

El PWM también usa 10 bits.

Encajan perfectamente.

Configuración paso a paso

Paso 1 – Configurar ANSELA

ANSELA = 0b00000001; // AN0 analógico

Paso 2 – TRISA

TRISA = 0b00000001; // GP0 entrada, resto salida

Paso 3 – Configurar ADC

ADCON1 = 0b10010000; // Justificado derecha, Fosc/32

ADCON0 = 0b00000001; // Canal AN0, ADC ON

Paso 4 – Configurar PWM

Frecuencia PWM = 1 kHz

PR2 = 249; // Periodo

CCP1CON = 0b00001100; // Modo PWM

T2CON = 0b00000101; // Prescaler 4, Timer2 ON

Código completo

```
#include <xc.h>

#define _XTAL_FREQ 4000000

void main(void)
{
    OSCCON = 0b01101010; // 4 MHz

    // --- ADC ---
    ANSELA = 0b00000001;
    TRISA = 0b00000001;

    ADCON1 = 0b10010000;
    ADCON0 = 0b00000001;

    // --- PWM ---
    PR2 = 249;
    CCP1CON = 0b00001100;
    T2CON = 0b00000101;

    while(1)
    {
        // Iniciar conversión
        GO_nDONE = 1;
        while(GO_nDONE);

        unsigned int valor = ((unsigned int)ADRESH << 8) | ADRESL;

        // Asignar 10 bits al PWM
        CCPR1L = valor >> 2;
        CCP1CONbits.DC1B = valor & 0x03;
    }
}
```

¿Qué ocurre ahora?

Si el potenciómetro está en:

Posición	ADC	Duty	Brillo
0%	0	0%	Apagado
50%	512	50%	Medio
100%	1023	100%	Máximo

Movimiento suave y continuo.

Análisis de CCP1CON bit a bit

Formato típico:

Bit:	7	6	5	4	3	2	1	0
	—	—	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0

Bits 3-0 — CCP1M3:CCP1M0

Seleccionan modo del módulo.

Para PWM:

1100

Es decir:

CCP1CON = 0b00001100;

Bits 5-4 — DC1B1:DC1B0

Son los 2 bits menos significativos del duty cycle.

El duty total es de 10 bits:

CCPR1L (8 bits altos)

DC1B1:DC1B0 (2 bits bajos)

Por eso hacemos:

CCPR1L = valor >> 2;

CCP1CONbits.DC1B = valor & 0x03;

Flujo interno real

Potenciómetro



ADC (10 bits)



CCPR1L + DC1B



Comparador PWM



Salida modulada

Todo automático en hardware.

Concepto clave

El ADC mide.

El PWM actúa.

Juntos permiten control proporcional real.

Este ejemplo ya es control analógico-digital completo.

Entrada analógica → Procesamiento digital → Salida controlada.

Eso es ingeniería embebida real.

Ahora vamos a analizar el PWM de forma profesional:

Basado en el módulo PWM del **PIC12F1840**.

Diagrama temporal del PWM

Supongamos:

Frecuencia PWM = 1 kHz

Periodo = 1 ms

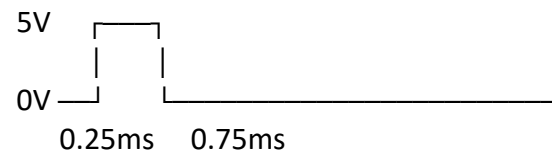
Duty = 25%, 50% y 75%

Caso 1 – 25%

Periodo = 1 ms

Tiempo alto = 0.25 ms

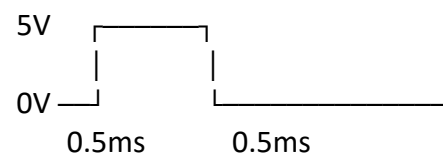
Nivel



Caso 2 – 50%

Tiempo alto = 0.5 ms

Nivel



Caso 3 – 75%

Tiempo alto = 0.75 ms

Nivel



¿Qué hace el hardware internamente?

El flujo real es:

Timer2 cuenta de 0 → PR2



Comparador compara con valor duty



Salida cambia automáticamente

El CPU no está generando los pulsos.

El hardware lo hace solo.

Fórmula completa de frecuencia

La frecuencia PWM es:

$$F_{\text{pwm}} = F_{\text{osc}} / (4 \times \text{Prescaler} \times (\text{PR2} + 1))$$

Aquí aparece PR2.

PR2 define el periodo.

¿Qué pasa si cambiamos PR2?

PR2 controla la duración del ciclo completo.

Caso 1 – PR2 pequeño

Ejemplo:

$$\text{PR2} = 124$$

La frecuencia aumenta.

El PWM es más rápido.

Resultado:

LED funciona bien

Motor más silencioso

Mejor respuesta

Caso 2 – PR2 grande

Ejemplo:

$$\text{PR2} = 249$$

La frecuencia disminuye.

Resultado:

LED puede comenzar a parpadear visible

Motor puede vibrar

Más ruido audible

Ejemplo numérico real

Supongamos:

Fosc = 4 MHz

Prescaler = 4

PR2 = 249

$F_{pwm} = 4,000,000 / (4 \times 4 \times 250)$

$F_{pwm} = 1000 \text{ Hz}$

Periodo = 1 ms

PR2 = 124

$F_{pwm} = 4,000,000 / (4 \times 4 \times 125)$

$F_{pwm} = 2000 \text{ Hz}$

Periodo = 0.5 ms

Más rápido.

Impacto en el Duty Cycle

Aquí viene algo importante.

El PWM usa 10 bits de resolución.

Pero la resolución real depende de PR2.

Resolución efectiva:

$\text{Resolución} \approx \log_2(4 \times (\text{PR2} + 1))$

PR2 grande → mejor resolución

Más pasos posibles en el duty.

Transiciones suaves.

PR2 pequeño → menor resolución

Menos niveles posibles.

Brillo menos suave.

Compromiso real

PR2 controla:

Frecuencia

Resolución

Calidad visual

Ruido audible

No puedes optimizar todo al mismo tiempo.

Concepto pedagógico fuerte

PR2 no cambia el voltaje.

Cambia el tiempo del ciclo.

Cambiar PR2 cambia:

Qué tan rápido ocurre el ciclo

Cuántos niveles finos puedes tener

Ejemplo comparativo

PR2	Frecuencia	Resolución	Uso típico
Pequeño	Alta	Baja	Comunicación
Medio	Media	Media	LED
Grande	Baja	Alta	Control fino

Conclusión clara

PR2 es el corazón del PWM.

Define periodo

Afecta resolución

Afecta calidad

Ahora vamos a entender el PWM “por dentro”:

Basado en el módulo PWM del **PIC12F1840**.

Diagrama temporal con comparador interno

El PWM funciona así:

Timer2 cuenta desde 0 hasta PR2

Cuando empieza el ciclo → salida = 1

Cuando el contador alcanza el valor del duty → salida = 0

Cuando llega a PR2 → el ciclo se reinicia

Ejemplo

Supongamos:

PR2 = 9

Duty = 4

Entonces:

Timer2 cuenta:

0 1 2 3 4 5 6 7 8 9 → vuelve a 0

Comparación interna

Mientras:

$TMR2 < Duty \rightarrow salida = 1$

$TMR2 \geq Duty \rightarrow salida = 0$

Diagrama temporal completo

Tiempo →

TMR2: 0 1 2 3 4 5 6 7 8 9 | 0 1 2 3 4 5 6 7 8 9

Salida: 
}

Donde:

 = Nivel alto

 = Nivel bajo

Visualización profesional

Nivel

5V

0V

← Duty → ← Restante → ← Duty →

Flujo interno real del hardware

Oscilador



Prescaler



Timer2 (contador)



Comparador digital



Flip-Flop salida PWM

El comparador decide el momento exacto de apagar la salida.

El CPU no interviene.

Registro T2CON – Bit a Bit

Timer2 es la base del PWM.

Formato típico:

Bit: 7 6 5 4 3 2 1 0
— TOUTPS3 TOUTPS2 TOUTPS1 TOUTPS0 TMR2ON T2CKPS1 T2CKPS0

Bit 2 — TMR2ON

- 1 → Timer2 encendido
-
- 0 → Timer2 apagado
-

Si está en 0, no hay PWM.

Bits 6-3 — TOUTPS3:TOUTPS0 (Postscaler)

Dividen la señal de overflow.

Importante:

El postscaler NO afecta la frecuencia del PWM.

Solo afecta la interrupción del Timer2.

Muchos estudiantes creen que cambia la frecuencia del PWM.

No es así.

Ejemplo real

T2CON = 0b00000101;

Desglose:

Prescaler = 1:4

TMR2ON = 1

Postscaler = 1:1

Relación entre Timer2 y PWM

Frecuencia PWM:

$$F_{\text{pwm}} = F_{\text{osc}} / (4 \times \text{Prescaler} \times (\text{PR2} + 1))$$

Timer2 controla:

El periodo completo

La base del PWM

Concepto clave

Timer2 define:

Cuándo empieza un nuevo ciclo

Cuánto dura el ciclo

El comparador define:

Cuánto tiempo está en alto

Analogía clara

Imagina:

Timer2 es un reloj que da vueltas de 0 a PR2.

El duty es una marca dentro del círculo.

Cuando el reloj pasa esa marca, se apaga la salida.

El PWM no es un delay.

Es:

Un contador

Un comparador

Un flip-flop digital

Todo funcionando en hardware.

Cambiemos a un módulo fundamental que todavía no hemos analizado a fondo y que es clave en sistemas reales:

Módulo de Interrupciones

Basado en el **PIC12F1840**.

Este módulo cambia completamente la forma de programar.

¿Qué es una interrupción?

Una interrupción es:

Un evento que detiene temporalmente la ejecución normal del programa para atender algo urgente.

En lugar de preguntar constantemente:

```
if(evento_ocurrio) { ... }
```

El hardware avisa automáticamente al CPU.

¿Por qué son importantes?

Sin interrupciones:

El programa debe estar revisando todo el tiempo.

Se desperdicia tiempo de CPU.

Con interrupciones:

El CPU hace su trabajo normal.

Solo reacciona cuando algo sucede.

Flujo interno real

Evento ocurre



Bandera se activa



CPU termina instrucción actual



Guarda PC en pila



Salta a dirección de interrupción



Ejecuta rutina ISR



RETFIE



PC restaurado



Continúa programa

Registros principales

Registro	Función
INTCON	Control global
PIE1	Habilitación periféricos
PIR1	Banderas periféricos

Registro INTCON bit a bit

Formato típico:

Bit: 7 6 5 4 3 2 1 0
GIE PEIE TMR0IE INTE IOCIE TMR0IF INTF IOCIF

Bit 7 — GIE (Global Interrupt Enable)

1 → Interrupciones activadas

0 → Interrupciones desactivadas

Si este bit está en 0, ninguna interrupción ocurre.

Bit 6 — PEIE

Habilita interrupciones de periféricos:

Timer1

ADC

PWM

etc.

Bit 5 — TMR0IE

Habilita interrupción de Timer0.

Bit 2 — TMR0IF

Bandera que indica overflow de Timer0.

Ejemplo práctico – Interrupción por botón

Supongamos:

Botón en GP2

Queremos que encienda LED cuando se presione

Configuración conceptual:

```
INTCONbits.INTE = 1;
INTCONbits.GIE = 1;
ISR:
void __interrupt() ISR()
{
    if (INTF)
    {
        INTF = 0;
        LATA0 = 1;
    }
}
```

¿Qué ocurre físicamente?

Cuando presionas el botón:
Cambia nivel lógico
Hardware detecta flanco
NTF se activa
CPU salta automáticamente a ISR

Pila de retorno

Cuando ocurre interrupción:
El PC se guarda en una pila interna.
El micro ejecuta la ISR.
RETFIE recupera el PC.

El programa continúa donde estaba.

Analogía didáctica

Imagina que estás leyendo un libro (programa principal).
Suena el teléfono (interrupción).
Marcas la página (PC en pila).
Atiendes la llamada (ISR).
Regresas exactamente donde estabas.

Tipos de interrupciones en el PIC12F1840

Timer0
Timer1
Timer2
ADC
Cambio en pines (IOC)
Comparador
EEPROM

Ventajas reales

Respuesta inmediata
CPU más eficiente
Diseño más profesional
Permite multitarea básica

Concepto clave

Las interrupciones convierten el sistema en:
Orientado a eventos.
No a encuestas (polling).

Ahora entramos a un nivel más interno del **PIC12F1840**.
Vamos por partes, claro y estructurado.

IOC (Interrupt-On-Change) Bit a Bit

El módulo **IOC** permite generar interrupción cuando un pin cambia de estado.
No necesitas estar leyendo el pin constantemente.

Registros principales del IOC

Registro	Función
IOCAP	Flanco positivo
IOCAN	Flanco negativo
IOCAF	Bandera de evento

IOCAP – Interrupt On Change Positive Edge

Bit: 5 4 3 2 1 0
GP5 GP4 GP3 GP2 GP1 GP0

Si un bit = 1 → Interrupción cuando el pin pasa de 0 → 1

Ejemplo:

IOCAP = 0b00000100; // GP2 flanco ascendente

IOCAN – Interrupt On Change Negative Edge

Igual estructura.

Si un bit = 1 → Interrupción cuando pasa de 1 → 0

IOCAN = 0b00000100; // GP2 flanco descendente

IOCAF – Interrupt On Change Flag

Este registro indica qué pin generó la interrupción.

Bit:	5	4	3	2	1	0
	GP5	GP4	GP3	GP2	GP1	GP0

Si GP2 cambió:

IOCAFbits.IOCAF2 = 1

Debe limpiarse manualmente:

IOCAFbits.IOCAF2 = 0;

Habilitación global

INTCONbits.IOCIE = 1; // Habilita IOC

INTCONbits.GIE = 1; // Global

La pila interna del micro (Stack)

El PIC12F1840 tiene una pila de hardware.

No es RAM.

Es una pila especial para guardar direcciones de retorno.

¿Qué guarda la pila?

Guarda el valor del **PC (Program Counter)** cuando ocurre:

CALL

Interrupción

Funcionamiento

Cuando ocurre interrupción:

PC actual → Guardado en pila

PC ← Dirección ISR

Al ejecutar:

RETFIE

El PC vuelve al valor anterior.

Profundidad

El PIC12F1840 tiene pila de 16 niveles.

Eso significa:

Puedes tener hasta 16 llamadas anidadas/interrupciones antes de desbordar.

Analogía clara

La pila es como un marcador automático de página.

Cada vez que ocurre algo importante:

Guarda dónde estabas.

Ejemplo completo Timer + ADC con interrupciones

Objetivo:

Timer1 genera evento cada 500 ms

ADC convierte automáticamente

LED cambia según lectura

Idea del sistema

Timer1 overflow



Inicia conversión ADC



ADC termina



Se actualiza LED

Código estructurado

```
void main(void)
{
    OSCCON = 0b01101010;

    // LED
    TRISA0 = 0;

    // ADC
    ANSELA = 0b00000001;
    TRISA1 = 1;

    ADCON1 = 0b10010000;
    ADCON0 = 0b00000001;

    ADIE = 1;

    // TIMER1
    T1CON = 0b00110001;
    TMR1H = 0x0B;
    TMR1L = 0xDC;
    TMR1IE = 1;

    PEIE = 1;
    GIE = 1;

    while(1)
    {
        // Programa principal libre
    }
}
```

Simulación del flujo del PC durante interrupción

Supongamos:

Programa principal en dirección:

0x120

Paso 1

Timer1 overflow ocurre.

Paso 2

Hardware hace:

Guardar 0x120 en pila

PC $\leftarrow$ Dirección ISR (0x004)

Paso 3

ISR ejecuta código.

Paso 4

RETFIE ejecutado.

PC $\leftarrow$ 0x120

Programa continúa exactamente donde estaba.

Diagrama del flujo del PC

MAIN (0x120)



[Evento]



PC guardado en Stack



ISR (0x004)



RETFIE



PC restaurado



MAIN continúa

Concepto clave final

Las interrupciones:

No destruyen el programa.

No reinician el micro.

Solo desvían temporalmente la ejecución.

El PC es el que se mueve.

La pila es la que permite regresar.

Ahora entramos a un tema interno y poco explicado en cursos básicos:

Desbordamiento de Pila (Stack Overflow)

En el **PIC12F1840** la pila:

Es de hardware

Tiene profundidad fija

No está en la RAM

No la puedes dimensionar

¿Qué es la pila en este PIC?

La pila almacena:

Direcciones de retorno del PC

Cada vez que ocurre:

CALL

Interrupción

Profundidad

El PIC12F1840 tiene:

16 niveles de pila

Eso significa:

Puedes tener máximo 16 direcciones guardadas simultáneamente.

¿Cuándo se usa la pila?

Cada vez que ocurre:

CALL

Interrupción

Se hace:

PC → Stack

Cuando ocurre:

RETURN

RETFIE

Se recupera el PC.

¿Qué es un desbordamiento?

Ocurre cuando:

Intentas guardar una dirección número 17.

Pero la pila solo tiene 16 espacios.

¿Qué pasa realmente?

En estos PIC:

La pila es circular.

Eso significa:

Si se llena y se agrega otra dirección:

Se sobrescribe la más antigua.

No hay advertencia.

No hay excepción.

No hay error visible.

Y el programa puede saltar a una dirección inesperada.

Ejemplo peligroso

Supongamos:

```
void funcionA() {  
    funcionA();  
}
```

Eso es recursión infinita.

Cada llamada hace:

CALL funcionA

La pila se llena rápidamente.

En la llamada 17:

Se sobrescribe la dirección más antigua.

El retorno será incorrecto.

El programa:

Puede reiniciarse

Puede saltar a memoria aleatoria

Puede comportarse erráticamente

Otro caso real

Interrupciones anidadas mal controladas.

Ejemplo:

ISR llama función.

Esa función llama otra.

Mientras tanto ocurre otra interrupción.

Cada evento empuja PC.

Si no se controla:

Desbordamiento.

Simulación conceptual

Estado inicial:

Nivel 0

Después de 16 CALL:

Nivel 16 (llena)

Intentas un CALL más:

Nivel 17 → sobrescribe nivel 0

Ahora la dirección original se perdió.

¿Cómo evitarlo?

No usar recursión

Mantener funciones poco anidadas

ISR pequeñas

No llamar funciones largas dentro de ISR

Desactivar interrupciones dentro de ISR si es necesario

Diferencia con microcontroladores grandes

En microcontroladores más avanzados:

La pila está en RAM

Puede crecer dinámicamente

Puede generar excepción de overflow

En el PIC12F1840:

Es fija

Es silenciosa

Es peligrosa si no se entiende

Señales típicas de desbordamiento

Salto inesperados

Reset espontáneo

Código que “funciona a veces”

Comportamiento aleatorio

Analogía didáctica

Imagina una torre con 16 espacios.

Si colocas una caja número 17:

La caja más antigua cae al suelo sin que te des cuenta.

Luego cuando intentas volver por ella...

Ya no está.

Concepto clave

La pila no es RAM.

No la puedes limpiar manualmente.

No la puedes ver fácilmente.

Funciona sola.

Y si la llenas... el programa pierde coherencia.

Ahora entramos a nivel arquitectura interna real del **PIC12F1840**.

Vamos en orden.

Ejemplo controlado de Stack Overflow

Vamos a diseñar un caso didáctico que provoque desbordamiento intencional.

Idea

Crear una cadena de llamadas profundas sin retorno inmediato.

Recordemos:

La pila tiene 16 niveles.

Si hacemos más de 16 CALL anidados → overflow.

Ejemplo en C (no recomendado en práctica)

```
void f16(void){ while(1); }
void f15(void){ f16(); }
void f14(void){ f15(); }
void f13(void){ f14(); }
void f12(void){ f13(); }
void f11(void){ f12(); }
void f10(void){ f11(); }
void f9(void){ f10(); }
void f8(void){ f9(); }
void f7(void){ f8(); }
void f6(void){ f7(); }
void f5(void){ f6(); }
void f4(void){ f5(); }
void f3(void){ f4(); }
void f2(void){ f3(); }
void f1(void){ f2(); }
```

```
void main(void)
{
    f1();
}
```

Aquí tenemos:

main → f1 (1)

f1 → f2 (2)

...

f15 → f16 (16)

Ya estamos al límite.

Si agregas una función más → nivel 17.

¿Qué pasará?

Cuando ocurra el retorno:

La dirección más antigua fue sobrescrita.

El programa regresará a una dirección incorrecta.

Comportamiento típico:

Salto inesperado

Reinicio

Bloqueo

¿Qué pasa con interrupciones múltiples?

Ahora algo más interesante.

Caso 1: Interrupción simple

Flujo normal:

MAIN

↓

[Evento]

↓

PC guardado

↓

ISR

↓

RETFIE

↓

MAIN continúa

Consume 1 nivel de pila.

Caso 2: ISR llama función

MAIN



[Evento]



ISR



CALL funcion()

Ahora tenemos:

1 nivel por interrupción

1 nivel por CALL

Total: 2 niveles usados.

Caso 3: Interrupción ocurre dentro de ISR

En este PIC:

Mientras se ejecuta ISR:

GIE se desactiva automáticamente.

No puede entrar otra interrupción hasta que termine la ISR.

Eso evita interrupciones anidadas automáticas.

Pero...

Si dentro de ISR vuelves a habilitar GIE manualmente:

GIE = 1;

Ahora sí permites interrupciones anidadas.

Eso puede:

Consumir pila rápidamente

Provocar overflow

Escenario peligroso

ISR1 activa

Dentro de ISR1 → ocurre evento ISR2

Dentro de ISR2 → CALL función

Ya tienes:

Nivel 1 (ISR1)

Nivel 2 (ISR2)

Nivel 3 (CALL)

Y así sucesivamente.

Diferencia entre CALL y GOTO

Este punto es clave.

CALL

Hace dos cosas:

1º Guarda PC en pila

2º

3º Salta a la función

4º

Cuando ocurre RETURN:

Recupera la dirección guardada.

Consume pila.

GOTO

Hace solo una cosa:

Salta a otra dirección.

No guarda nada.

No usa pila.

No hay retorno automático.

Comparación directa

Instrucción	Guarda PC	Usa pila	Regresa automáticamente
CALL	Sí	Sí	Sí (RETURN)
GOTO	No	No	No

Ejemplo ilustrativo

Con CALL

CALL rutina

Pila:

Nivel 1 → Dirección siguiente

Con GOTO

GOTO rutina

Pila:

No cambia

Pero ahora no hay retorno implícito.

Concepto arquitectónico importante

El PIC tiene arquitectura Harvard.

El PC apunta a memoria de programa.

CALL y RET manipulan el PC usando la pila interna.

GOTO solo cambia el PC.

Caso didáctico comparativo

Si reemplazas llamadas profundas por GOTO encadenados:

No hay overflow.

Pero pierdes estructura modular.

Conclusión fuerte

CALL = seguro pero consume pila

GOTO = simple pero sin retorno

ISR también consume pila

16 niveles es límite físico real

Concepto final

El stack no es infinito.

En sistemas embebidos pequeños:

La arquitectura impone límites físicos.

Diseñar bien el flujo evita fallas misteriosas.

Cambiemos a un módulo muy importante en aplicaciones reales y que aún no hemos analizado a fondo:

Módulo WDT (Watchdog Timer)

Este módulo no hace cálculos, no mide señales...

Su función es proteger el sistema.

¿Qué es el Watchdog Timer?

El WDT es:

Un temporizador independiente que reinicia el microcontrolador si el programa se “queda colgado”.

Es un sistema de seguridad.

¿Por qué es necesario?

En sistemas reales pueden ocurrir:

Ruido eléctrico

Fallos lógicos

Errores de programación

Bloqueos por software

Si el programa se detiene en un bucle infinito, el sistema deja de funcionar.

El WDT evita eso.

¿Cómo funciona?

Internamente:

Oscilador interno independiente



Contador WDT



Si no se reinicia a tiempo



Reset del micro

El WDT tiene su propio oscilador RC interno.

No depende del reloj principal.

Concepto clave

El programa debe “alimentar” al perro guardián.

Si no lo hace → reset.

Instrucción CLRWD

Para evitar el reset:

`CLRWD();`

Eso reinicia el contador del WDT.

Tiempo típico del WDT

El tiempo base es aproximadamente:

≈ 1 ms a varios segundos

Depende del postscaler configurado.

Configuración del WDT

Se configura mediante:

Bits de configuración (CONFIG)

Registro WDTCN

WDTCN (conceptual)

Contiene bits para:

Habilitar/deshabilitar WDT

Ajustar postscaler

Ejemplo conceptual:

WDTCNbits.SWDTEN = 1; // Habilitar por software

Ejemplo práctico

Supongamos:

Sistema que debe ejecutar continuamente.

Si algo falla → reiniciar.

```
void main(void)
```

```
{
```

```
    WDTCNbits.SWDTEN = 1;
```

```
    while(1)
```

```
    {
```

```
        // Tareas normales
```

```
        CLRWDT(); // Alimentar WDT
```

```
    }
```

```
}
```

Si el programa entra en:

```
while(1); // Bucle infinito accidental
```

No se ejecuta CLRWDT().

El WDT expira.

El micro se reinicia.

¿Cómo saber si el reset fue por WDT?

Existe un bit en registro STATUS o PCON (según modelo) que indica la causa del reset.

Permite detectar:

Reset por encendido

Reset externo

Reset por WDT

Analogía didáctica

Imagina que el micro tiene un supervisor.

Cada cierto tiempo pregunta:

“¿Sigues vivo?”

Si no respondes:

Reinicia todo.

Ventajas reales

Sistemas industriales

Sistemas automotrices

Equipos médicos

Dispositivos remotos

Sin WDT, un fallo puede dejar el sistema muerto indefinidamente.

Precauciones

Si el tiempo del WDT es muy corto:

Puede reiniciar el sistema constantemente.

Debe configurarse acorde al tiempo máximo que tarda el programa en ejecutar su ciclo.

Concepto fuerte

El WDT no mejora el programa.

Lo protege contra fallos.

Es un mecanismo de resiliencia.

Vamos a analizar el **Watchdog Timer (WDT)** en profundidad en el **PIC12F1840**.

Registro WDTCON — Bit a Bit

Formato típico:

Bit:	7	6	5	4	3	2	1	0
	—	—	WDTPS4	WDTPS3	WDTPS2	WDTPS1	WDTPS0	SWDTEN

Bit 0 — SWDTEN

Software Watchdog Enable

1 → WDT habilitado

0 → WDT deshabilitado

(Si el WDT está habilitado por configuración, este bit puede o no tener efecto según el modo.)

Bits 4–1 — WDTPS (Postscaler)

Determinan cuánto tiempo tarda el WDT en provocar reset.

El WDT tiene un periodo base interno (~1 ms típico, depende del dispositivo).

El postscaler multiplica ese tiempo.

Tiempo exacto según postscaler

Tiempo aproximado:

$T_w = \text{Periodo_base} \times \text{Postscaler}$

Periodo base típico ≈ 1 ms (oscilador RC interno).

Valores comunes del postscaler

WDTPS	Multiplicador	Tiempo aproximado
00000	1:32	~32 ms
00001	1:64	~64 ms
00010	1:128	~128 ms
00011	1:256	~256 ms
00100	1:512	~512 ms
00101	1:1024	~1 s
...	...	...
10100	1:65536	~65 s
11111	1:1048576	~1048 s (~17 min)

(Valores aproximados, pueden variar por tolerancia del RC interno.)

Ejemplo real

Si configuramos:

WDTCN = 0b00010101;

Eso podría significar:

WDTPS = 00101 $\rightarrow$ 1:1024

SWDTEN = 1

Tiempo ≈ 1 segundo.

Ejemplo donde el WDT salva el sistema

Escenario

Sistema que:

Lee sensor

Controla motor

Puede quedarse bloqueado por ruido

Código sin protección

```
while(1)
{
    if(sensor_falla)
        while(1); // Error grave

    controlar_motor();
}
```

Si ocurre falla:
El sistema muere.

Código con WDT

```
void main(void)
{
    WDTCONbits.SWDTEN = 1;

    while(1)
    {
        if(sensor_falla)
            while(1); // Se cuelga

        controlar_motor();

        CLRWDT(); // Alimentar perro
    }
}
```

Si se cuelga:
No se ejecuta CLRWDT().
Después de 1 segundo:
El WDT provoca reset.
Sistema vuelve a arrancar.

Diferencia entre reset por WDT y reset por MCLR

Ambos reinician el micro, pero no son lo mismo.

Reset por WDT

Origen: Interno (software bloqueado).
Causa:
No se ejecutó CLRWDT.
Timeout del WDT.

Características:
Automático
Independiente del reloj principal
Detecta bloqueos

Reset por MCLR

Origen: Externo (pin físico MCLR).

Causa:

Botón de reset

Supervisor externo

Circuito de encendido

Características:

Control físico

Reset manual

Reset por caída de tensión

¿Cómo saber qué tipo de reset ocurrió?

Existe un bit en registros como PCON que indica causa del reset.

Permite distinguir:

Power-on Reset

Brown-out Reset

WDT Reset

MCLR Reset

Esto es clave en sistemas críticos.

Comparación directa

Característica	WDT	MCLR
Interno		
Externo		
Detecta cuelgue		
Activación manual		
Usa pin físico		

Concepto clave

El WDT es un mecanismo de recuperación automática.

MCLR es un mecanismo de reinicio externo.

Uno protege contra fallos lógicos.

El otro permite reinicio físico.

Concepto fuerte para el alumno

Un sistema profesional:

Siempre usa Watchdog

Siempre detecta causa del reset

Siempre diseña pensando en fallos

En sistemas embebidos reales, no importa si fallará.

Importa qué pasa cuando falle.

Ahora entramos en diseño **real de ingeniería embebida** con el **PIC12F1840**.

Vamos a construir un sistema robusto completo.

Sistema robusto con WDT + detección de reset

Objetivo

Diseñar un sistema que:

Detecte si el reset fue por WDT

Detecte si fue por caída de tensión

Registre el evento

Arranque en modo seguro si hubo falla

Paso 1 – Detectar causa del reset

El PIC posee bits en registros como:

PCON

STATUS

Estos indican:

POR (Power On Reset)

BOR (Brown-Out Reset)

WDT Reset

MCLR Reset

Ejemplo conceptual

```
if (!PCONbits.nWDT)
{
    // Reset causado por WDT
}
```

```
if (!PCONbits.nBOR)
{
    // Reset causado por Brown-Out
}
```

Paso 2 – Sistema robusto ejemplo

Supongamos:

Controlador de bomba de agua

Sensor analógico

Motor

Si ocurre fallo:

El sistema se reinicia

Entra en modo seguro

Espera estabilización

Código estructurado

```
#include <xc.h>
#define _XTAL_FREQ 4000000

void modo_seguro(void)
{
    LATA0 = 0; // Apagar motor
    __delay_ms(2000); // Esperar estabilización
}

void main(void)
{
    unsigned char reset_wdt = 0;
    unsigned char reset_bor = 0;

    if (!IPCONbits.nWDT)
        reset_wdt = 1;

    if (!IPCONbits.nBOR)
        reset_bor = 1;

    if (reset_wdt || reset_bor)
        modo_seguro();

    WDTCONbits.SWDTEN = 1;

    while(1)
    {
        // Operación normal
        CLRWDT();
    }
}
```

Simulación de fallo eléctrico real

Escenario real

Sistema alimentado con batería.

Ocurre:

Motor arranca

La tensión cae momentáneamente

Vdd baja por debajo del mínimo

Sin protección

El micro:

Ejecuta instrucciones erráticas

Puede escribir datos corruptos en EEPROM

Puede activar salidas incorrectas

Comportamiento impredecible.

Con Brown-Out Reset (BOR)

Cuando Vdd baja por debajo del umbral:

$V_{dd} < V_{bor} \rightarrow \text{Reset inmediato}$

El micro se mantiene en reset hasta que Vdd vuelva a nivel seguro.

Brown-Out Reset (BOR)

El BOR es un detector de baja tensión interno.

Funciona automáticamente si está habilitado en los bits de configuración.

¿Qué hace?

Supervisa Vdd constantemente.

Si Vdd cae debajo del umbral (por ejemplo 2.1V o 4V según configuración):

Genera reset

Detiene ejecución

Evita corrupción de memoria

Flujo interno

Vdd normal



Caída de tensión



Detector BOR activa reset



Micro detenido



Vdd estable



Reinicio limpio

Diferencia entre WDT y BOR

Característica	WDT	BOR
Detecta fallo software		
Detecta baja tensión		
Usa temporizador		
Protege contra ruido eléctrico		
Requiere CLRWDT		

Escenario combinado real

Sistema industrial:

Ruido eléctrico → caída momentánea → BOR reset

Software bloqueado → WDT reset

Ambos detectados en arranque

El sistema:

Entra en modo seguro

Registra causa

Arranca nuevamente

Diseño robusto profesional

Un sistema serio debe:

Habilitar BOR

Habilitar WDT

Detectar causa del reset

Iniciar en estado seguro

Evitar salidas activas al arrancar

Concepto fuerte de ingeniería

En diseño embebido real:

No se diseña para que nunca falle.

Se diseña para que, cuando falle, se recupere correctamente.

Arquitectura mental correcta

WDT protege contra:

Errores lógicos.

BOR protege contra:

Errores eléctricos.

Ambos juntos:

Sistema resiliente.

Ahora analizamos otro mecanismo fundamental del **PIC12F1840**:

Power-On Reset (POR)

El **POR** es el sistema que garantiza que el micro arranque correctamente cuando se aplica alimentación.

¿Qué es el Power-On Reset?

Es un circuito interno que detecta cuando:

Vdd pasa de 0V → nivel operativo

Cuando eso ocurre:

El micro se mantiene en reset

Espera que la tensión sea estable

Luego inicia ejecución desde dirección 0x0000

¿Por qué es necesario?

Cuando se aplica energía:

La tensión no sube instantáneamente

Puede haber rebotes

Puede haber ruido

El oscilador aún no está estable

Sin POR:

El CPU podría comenzar a ejecutar instrucciones con voltaje inestable.

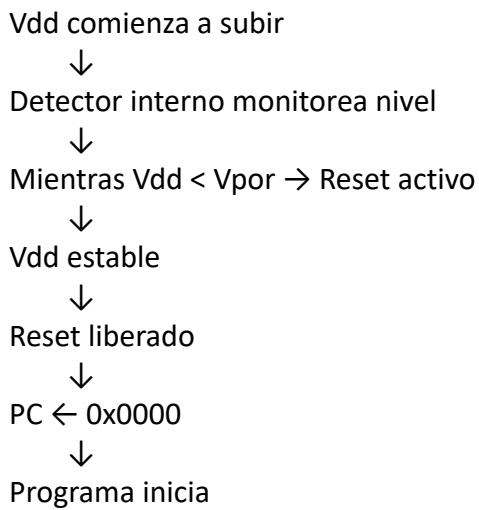
Eso generaría:

Comportamiento errático

Escrituras corruptas

Salto aleatorios

Flujo interno del POR



¿Qué ocurre exactamente al liberar el reset?

Cuando el reset termina:
 $PC = 0x0000$
Registros vuelven a estado inicial
Config bits ya están cargados
Periféricos en estado por defecto

Es un arranque limpio.

Diferencia entre POR y BOR

Característica	POR	BOR
Ocurre al encender		
Detecta caída posterior		
Actúa solo al inicio		
Supervisa continuamente		

Registro de causa de reset

Después del arranque, el micro conserva información sobre qué causó el reset.

En registros como:

PCON

STATUS

Permite distinguir:

Power-On Reset

Brown-Out Reset

WDT Reset

MCLR Reset

¿Qué pasa si la subida de tensión es muy lenta?

Si la fuente es débil:

Vdd puede subir lentamente.

El POR mantiene el micro detenido hasta cruzar el umbral mínimo.

Eso evita:

Ejecución parcial o inestable.

Interacción con PWRT (Power-Up Timer)

Muchos PIC incluyen además:

Power-Up Timer (PWRT).

Este agrega un pequeño retardo adicional después del POR.

Flujo:

Vdd estable



POR libera reset



PWRT espera (ej. 64 ms)



Inicio real del programa

Esto permite:

Estabilización del oscilador

Estabilización de fuentes externas

Escenario real

Sistema alimentado con batería.

Al conectar batería:

Vdd sube lentamente

POR mantiene el micro en reset

Cuando alcanza nivel correcto

El micro arranca correctamente

Sin POR:

Podría iniciar en estado indeterminado.

Diferencia clave con MCLR

POR

Automático por energía

No requiere hardware externo

Solo al aplicar alimentación

MCLR

Activado por pin externo

Requiere botón o circuito

Puede usarse en cualquier momento

Concepto fuerte

POR garantiza que el sistema comience desde un estado coherente.

Es el “inicio seguro” del microcontrolador.

Secuencia completa de arranque

Aplicar energía



POR activo



(Opcional) PWRT



Cargar configuración



Inicializar registros



PC = 0x0000



main()

Mensaje clave

Un microcontrolador no empieza a ejecutar inmediatamente al recibir energía.

Existe una secuencia interna de validación y estabilización.

Eso es lo que hace al sistema confiable.

Ahora vamos a cerrar el ciclo de arranque interno del micro y luego pasamos a comunicación serial.

Secuencia completa de arranque paso a paso

Cuando aplicas alimentación, no empieza a ejecutarse el main() inmediatamente.
Existe una secuencia interna muy precisa.

Aplicación de Vdd

Vdd = 0V → empieza a subir

En este momento:

CPU inactivo

Oscilador detenido

Registros indefinidos

Activación del Power-On Reset (POR)

Cuando Vdd supera el umbral mínimo:

El detector POR mantiene el micro en estado de reset.

El PC aún no ejecuta código.

$V_{dd} < V_{por} \rightarrow$ Reset activo

Estabilización del oscilador

El oscilador interno necesita estabilizarse.

Dependiendo de configuración:

INTOSC

Oscilador externo

PLL

El micro espera que el reloj sea válido.

Power-Up Timer (PWRT) (si está habilitado)

Agrega un retardo adicional (por ejemplo ~64 ms).

Esto permite:

Estabilizar fuente

Estabilizar sensores

Estabilizar referencias

Flujo:

POR libera → PWRT cuenta → CPU aún detenido

Brown-Out Reset (si aplica)

Si durante el arranque Vdd cae:

BOR mantiene el reset activo

El micro no ejecuta código inestable

Liberación del Reset

Cuando:

Vdd estable

Oscilador estable

PWRT finaliza

Entonces:

PC $\leftarrow$ 0x0000

Ejecución desde 0x0000

En la dirección 0x0000 normalmente hay:

GOTO main

O código inicial del compilador.

Inicialización del compilador

Si usas XC8:

Antes de entrar a main() se ejecuta:

Inicialización de variables globales

Inicialización de memoria estática

Configuración del stack

Luego entra a:

main();

Aplicar energía

↓

POR activo

↓

Oscilador estabiliza

↓

PWRT (si habilitado)

↓

BOR verifica nivel

↓

Reset liberado

↓

PC = 0x0000

↓

Inicialización C

↓

main()

Sistema listo

Ahora el programa comienza ejecución normal.

Secuencia completa resumida

Concepto fuerte

El arranque es una coreografía de seguridad eléctrica.

No es inmediato.

Está diseñado para evitar comportamiento errático.

Módulo UART (EUSART)

Ahora pasamos a comunicación.

UART permite que el micro:

Envíe datos a PC

Reciba datos

Se comunique con otros micros

Use módulos Bluetooth, WiFi, GPS, etc.

¿Qué es UART?

UART = Universal Asynchronous Receiver Transmitter

Comunicación serial asíncrona.

Solo necesita:

TX

RX

GND

Formato típico de transmisión

Cada byte se envía así:

Start bit (0)

8 bits de datos

Stop bit (1)

Ejemplo enviando 'A' (0x41):

0 10000010 1

(Se transmite LSB primero.)

Velocidad – Baud Rate

Velocidad en bits por segundo.

Ejemplo común:

9600 bps

19200 bps

115200 bps

Registros principales UART

Registro	Función
TXSTA	Control transmisión
RCSTA	Control recepción
SPBRG	Baud rate generator
BAUDCON	Configuración adicional
TXREG	Registro de transmisión
RCREG	Registro de recepción

Configuración básica a 9600 bps (4 MHz)

Fórmula modo asíncrono:

$$\text{Baud} = F_{\text{osc}} / (64 \times (\text{SPBRG} + 1))$$

Para 4 MHz y 9600:

$$\text{SPBRG} \approx 6$$

Código básico UART

```
void UART_Init(void)
{
    TRISA1 = 1; // RX
    TRISA0 = 0; // TX

    SPBRG = 6;

    TXSTA = 0b00100100; // TX enable, BRGH=1
    RCSTA = 0b10010000; // Serial enable, RX enable
}
```

Enviar un byte

```
void UART_Write(char data)
{
    while(!TXIF);
    TXREG = data;
}
```

Recibir un byte

```
char UART_Read(void)
{
    while(!RCIF);
    return RCREG;
}
```

Flujo interno UART

CPU escribe TXREG
↓
Shift register transmite bit a bit
↓
Pin TX envía señal serial
En recepción:
Pin RX detecta start bit
↓
Muestra bits en tiempo correcto
↓
Carga RCREG
↓
RCIF = 1

Aplicaciones reales

Comunicación con PC
Debug por terminal
Módulos Bluetooth HC-05
GPS
Comunicación entre micros

Concepto fuerte

UART convierte bytes paralelos internos en flujo serial externo.
Es la puerta de comunicación del microcontrolador.

Ahora entramos al uso profesional de la **EUSART (UART)** del **PIC12F1840**.

Cálculo de Baud Rate exacto y error porcentual

Fórmulas (Modo Asíncrono)

BRGH = 0 (Low Speed)

$$\text{Baud} = \text{Fosc} / (64 \times (\text{SPBRG} + 1))$$

BRGH = 1 (High Speed)

$$\text{Baud} = \text{Fosc} / (16 \times (\text{SPBRG} + 1))$$

Caso: Fosc = 4 MHz, Baud deseado = 9600

Usamos modo High Speed (mejor precisión).

Cálculo de SPBRG

$$\text{SPBRG} = (\text{Fosc} / (16 \times \text{Baud})) - 1$$

Sustituyendo:

$$\text{SPBRG} = (4,000,000 / (16 \times 9600)) - 1$$

$$\text{SPBRG} = (4,000,000 / 153600) - 1$$

$$\text{SPBRG} \approx 26.04 - 1$$

$$\text{SPBRG} \approx 25$$

Baud real obtenido

$$\text{Baud_real} = 4,000,000 / (16 \times (25 + 1))$$

$$\text{Baud_real} = 4,000,000 / 416$$

$$\text{Baud_real} \approx 9615.38 \text{ bps}$$

Error porcentual

$$\text{Error \%} = ((\text{Baud_real} - \text{Baud_deseado}) / \text{Baud_deseado}) \times 100$$

$$\text{Error \%} \approx (15.38 / 9600) \times 100$$

$$\text{Error} \approx 0.16\%$$

Excelente precisión

(Generalmente menos de 2% es aceptable.)

Ejemplo completo

ADC → UART (Enviar lectura por serial)

Objetivo

Leer AN0

Convertir a número decimal

Enviar por UART cada 500 ms

Código completo

```
#include <xc.h>
#define _XTAL_FREQ 4000000

void UART_Init(void)
{
    TRISA0 = 0; // TX
    TRISA1 = 1; // RX

    SPBRG = 25;    // 9600 bps @ 4MHz
    TXSTA = 0b00100100; // BRGH=1, TXEN=1
    RCSTA = 0b10010000; // SPEN=1, CREN=1
}

void UART_Write(char data)
{
    while(!TXIF);
    TXREG = data;
}

void UART_Write_Text(char *text)
{
    while(*text)
        UART_Write(*text++);
}

void ADC_Init(void)
{
    ANSELA = 0b000000001;
    TRISA0 = 1;

    ADCON1 = 0b10010000; // Justificado derecha
    ADCON0 = 0b000000001; // AN0, ADC ON
}

unsigned int ADC_Read(void)
{
    GO_nDONE = 1;
    while(GO_nDONE);
    return ((unsigned int)ADRESH << 8) | ADRESL;
}
```

```

void main(void)
{
    OSCCON = 0b01101010;

    UART_Init();
    ADC_Init();

    while(1)
    {
        unsigned int valor = ADC_Read();

        char buffer[10];
        sprintf(buffer, "%u\r\n", valor);

        UART_Write_Text(buffer);

        __delay_ms(500);
    }
}

```

Interrupciones en UART

UART puede generar interrupciones por:

Transmisión completa (TXIF)

Recepción completa (RCIF)

Habilitar interrupción RX

RCIE = 1;

PEIE = 1;

GIE = 1;

ISR ejemplo

```

void __interrupt() ISR(void)
{
    if (RCIF)
    {
        char dato = RCREG; // Leer limpia bandera
        LATA0 = 1;        // Acción ejemplo
    }
}

```

Ventaja

No necesitas:

while(!RCIF);

El micro responde automáticamente cuando llega un dato.

Framing Error (FERR)

¿Qué es?

Ocurre cuando:

El bit de STOP no es detectado correctamente.

Posibles causas:

Baud incorrecto

Ruido

Desincronización

Detección

if (FERR)

```
{  
    char basura = RCREG; // Leer para limpiar  
}
```

Overrun Error (OERR)

¿Qué es?

Ocurre cuando:

Llegan datos

No se leen a tiempo

El buffer interno se llena

La recepción se bloquea.

Solución

if (OERR)

```
{  
    CREN = 0;  
    CREN = 1;  
}  
Reinicia el receptor.
```

Diferencia FERR vs OERR

Error	Causa	Solución
FERR	Bit STOP incorrecto	Leer RCREG
OERR	Buffer lleno	Resetear CREN

Flujo interno de recepción

Start bit detectado



Sincronización



Shift register recibe 8 bits



Verifica Stop bit



Carga RCREG



RCIF = 1

Concepto fuerte final

UART es extremadamente sensible al reloj.

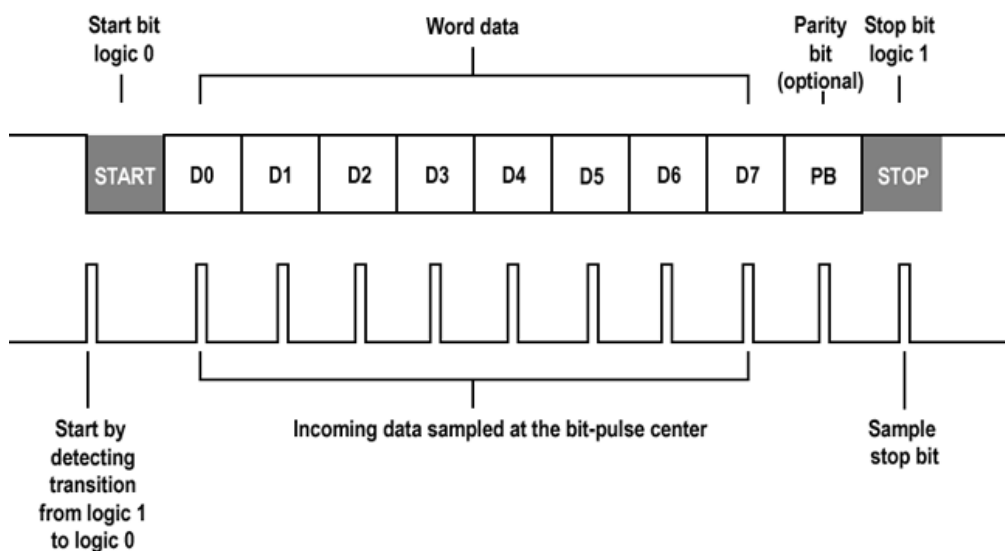
Pequeños errores en baud rate generan:

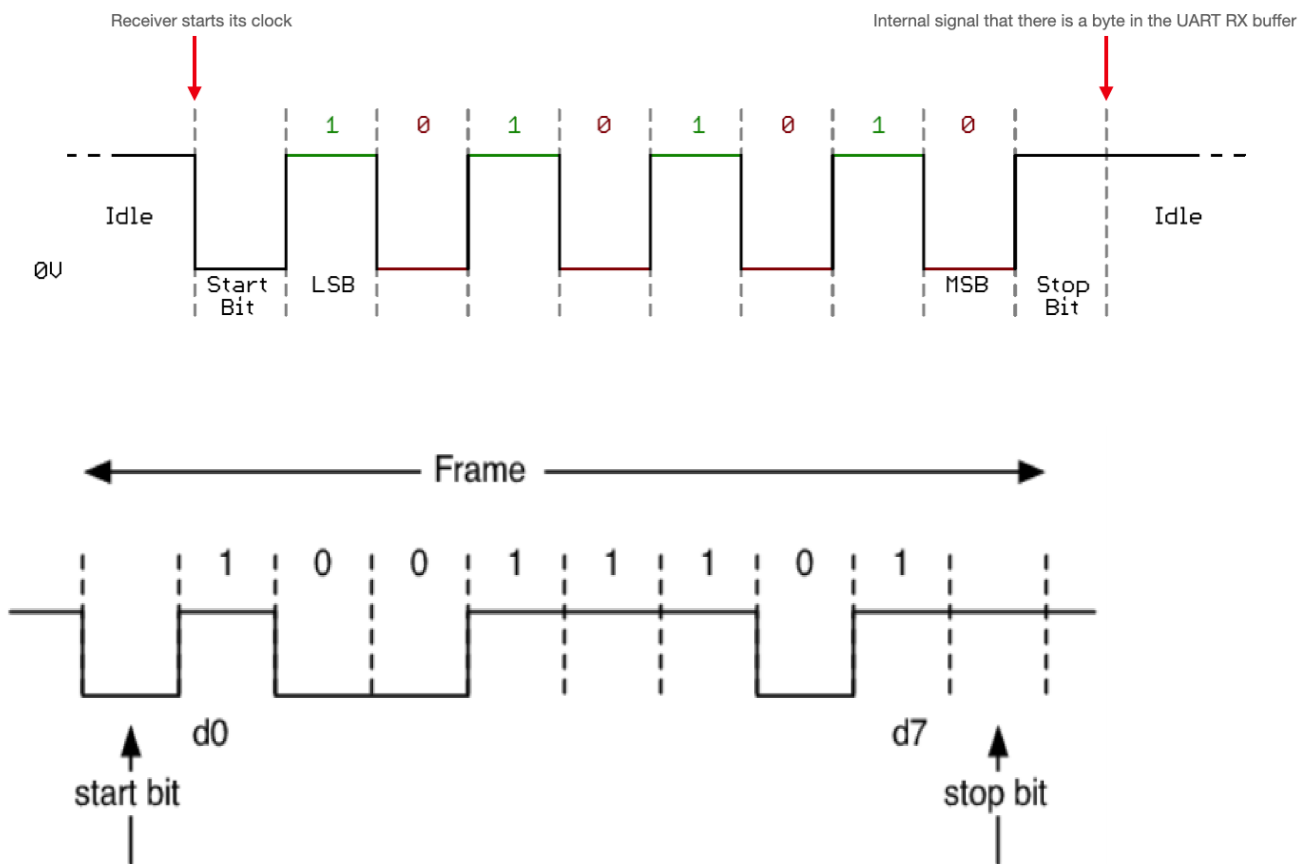
Framing errors

Comunicación inestable

Por eso el cálculo correcto es fundamental.

Formato de comunicación UART (trama serial)





La comunicación **UART** transmite los datos **bit a bit en serie**. Cada byte se envía dentro de una **trama (frame)** con una estructura fija.

Estructura básica de una trama UART

El formato más común es **8N1**, que significa:

8 bits de datos

N = sin paridad

1 bit de stop

La trama se ve así:

Idle	Start	D0	D1	D2	D3	D4	D5	D6	D7	Stop
1	0	b0	b1	b2	b3	b4	b5	b6	b7	1

Explicación de cada parte

Estado Idle (reposo)

La línea UART normalmente está en:

Nivel lógico = 1

Esto indica que **no se está transmitiendo nada**.

Start Bit

Cuando comienza la transmisión:

Start bit = 0

Este bit permite que el receptor **se sincronice** con la trama.

Duración: **1 periodo de baud rate**

Ejemplo a 9600 baud:

Tiempo de bit $\approx 104 \mu s$

Bits de datos

Se transmiten **8 bits**, comenzando por el **LSB (Least Significant Bit)**.

Ejemplo enviando el carácter 'A':

ASCII:

'A' = 0x41

Binario:

01000001

Pero en UART se envía **LSB primero**:

10000010

Bit de Paridad (opcional)

Algunos sistemas usan un bit extra para **detección de errores**.

Tipos:

Tipo	Función
Even	número de bits 1 debe ser par
Odd	número de bits 1 debe ser impar

En **8N1 no hay paridad**.

Stop Bit

Final de la trama.

Stop bit = 1

Indica al receptor que la transmisión terminó.

Puede haber:

1 stop bit

1.5 stop bits

2 stop bits

Ejemplo completo (8N1)

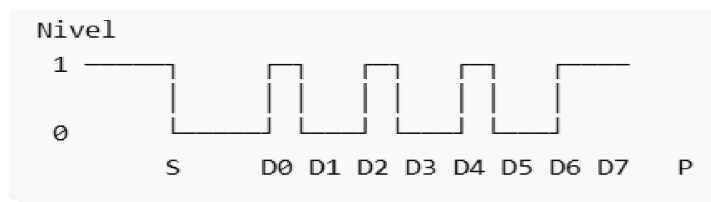
Transmitiendo **0x55**:

0x55 = 01010101

Trama:

Idle Start 1 0 1 0 1 0 1 0 Stop
1 0 1 0 1 0 1 0 1

Diagrama temporal simplificado:



Tiempo total de transmisión

Para **8N1**:

1 Start

8 Data

1 Stop

10 bits totales

Entonces:

Baud rate

9600

19200

115200

Tiempo por byte

≈ 1.04 ms

≈ 520 μs

≈ 87 μs

Cómo lo interpreta el receptor

El receptor:

Detecta el **Start bit**

Espera **1.5 tiempos de bit**

Empieza a muestrear cada bit

Reconstruye el byte

Verifica el **Stop bit**

Idea clave

UART **no envía bytes directamente.**

Envía **tramas seriales temporizadas:**

Start + Datos + Stop

Todo depende del **baud rate** y de la sincronización temporal.

Ahora vamos a diseñar **una comunicación UART bidireccional completa**, que es la base de muchos sistemas embebidos reales en el **PIC12F1840**.

Aquí el micro:

envía datos

recibe comandos

actúa según el comando recibido

Esto es lo que se usa en:

terminal serial

módulos Bluetooth

comunicación PC ↔ micro

control remoto por comandos

Arquitectura del sistema

Sistema:

PC / Terminal

↕ UART

PIC12F1840

El micro:

recibe un comando

ejecuta una acción

responde con un mensaje

Protocolo simple de comandos

Definimos comandos de un solo carácter.

Comando	Acción
'L'	Encender LED
'A'	Apagar LED
'R'	Enviar lectura ADC

Ejemplo desde terminal:

L

Respuesta:

LED ON

Configuración UART

Baud rate: **9600**

Para **Fosc = 4 MHz**

SPBRG = 25

Error $\approx$ **0.16%**

Código completo

```
#include <xc.h>
#define _XTAL_FREQ 4000000

// ----- UART -----

void UART_Init(void)
{
    TRISA0 = 0; // TX
    TRISA1 = 1; // RX

    SPBRG = 25;

    TXSTA = 0b00100100; // BRGH=1, TXEN=1
    RCSTA = 0b10010000; // SPEN=1, CREN=1
}

void UART_Write(char data)
{
    while(!TXIF);
    TXREG = data;
}

void UART_Write_Text(const char *text)
{
    while(*text)
        UART_Write(*text++);
}

char UART_Read(void)
{
    while(!RCIF);
    return RCREG;
}

// ----- ADC -----

void ADC_Init(void)
{
    ANSELA = 0b000000001;
    TRISA0 = 1;

    ADCON1 = 0b10010000;
    ADCON0 = 0b000000001;
}

unsigned int ADC_Read(void)
{
    GO_nDONE = 1;
    while(GO_nDONE);

    return ((unsigned int)ADRESH << 8) | ADRESL;
}
```

```
// ----- MAIN -----

void main(void)
{
    OSCCON = 0b01101010;

    TRISA2 = 0; // LED

    UART_Init();
    ADC_Init();

    UART_Write_Text("Sistema listo\r\n");

    while(1)
    {
        char comando = UART_Read();

        switch(comando)
        {
            case 'L':

                LATA2 = 1;
                UART_Write_Text("LED ON\r\n");

                break;

            case 'A':

                LATA2 = 0;
                UART_Write_Text("LED OFF\r\n");

                break;

            case 'R':

                {
                    unsigned int valor = ADC_Read();

                    char buffer[10];
                    sprintf(buffer,"%u\r\n",valor);

                    UART_Write_Text(buffer);
                }

                break;

            default:

                UART_Write_Text("Comando desconocido\r\n");

                break;

        }
    }
}
```

Flujo de comunicación

Ejemplo real usando terminal serial:

Sistema listo

Usuario escribe:

L

Micro responde:

LED ON

Luego:

R

Micro responde:

523

(valor ADC)

Diagrama de flujo del programa

Inicializar UART

Inicializar ADC



Esperar comando



Leer carácter UART



Evaluar comando



Ejecutar acción



Enviar respuesta

Ventajas del sistema

comunicación en tiempo real

control remoto del micro

fácil depuración

base para protocolos más complejos

Aplicaciones reales

Este esquema es exactamente el usado para:

módulos Bluetooth HC-05

comunicación con Raspberry Pi

interfaces de diagnóstico

configuración remota de dispositivos

Concepto fuerte

UART convierte al micro en **un dispositivo interactivo**.

Ya no ejecuta solo un programa.

Ahora **dialoga con el exterior**.

¿Cómo implementar I²C?

Vamos a explicar **cómo implementar comunicación I²C** en el **PIC12F1840** paso a paso.

Este micro no tiene módulo MSSP completo como otros PIC grandes, pero **sí puede implementar I²C usando el módulo MSSP en modo I²C o por software (bit-banging)** según la configuración.

Aquí veremos **la forma típica usando el hardware MSSP**.

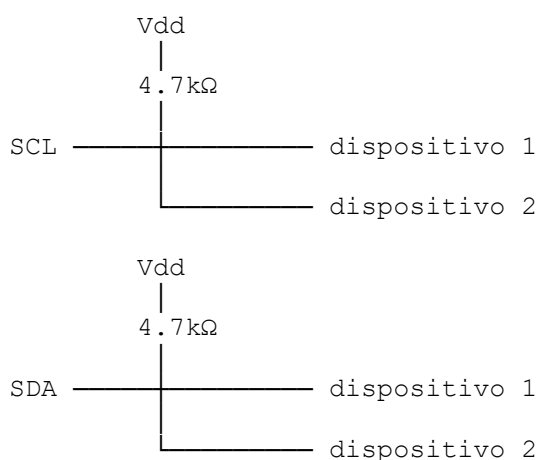
¿Qué es I²C?

I²C (Inter-Integrated Circuit) es un bus serial que usa **solo dos líneas**:

Línea	Función
SCL	reloj
SDA	datos

Ambas líneas son **open-drain**, por lo que necesitan **resistencias pull-up**.

Conexión eléctrica básica



Las resistencias pull-up son necesarias porque los dispositivos **solo pueden llevar la línea a 0**.

Estructura de comunicación I²C

Una transmisión típica se ve así:

START

Dirección dispositivo

Bit R/W

ACK

Datos

ACK

STOP

Registros principales I²C

En el PIC se utilizan:

Registro	Función
SSP1CON1	Control del módulo I ² C
SSP1CON2	Control de eventos
SSP1STAT	Estado del bus
SSP1BUF	Registro de datos
SSP1ADD	Baud rate I ² C

Configuración del bus I²C

Frecuencia típica del bus:

100 kHz

Fórmula:

$$SSPADD = (F_{osc} / (4 \times F_{scl})) - 1$$

Ejemplo:

$F_{osc} = 4 \text{ MHz}$

$F_{scl} = 100 \text{ kHz}$

$$SSPADD = (4,000,000 / (4 \times 100,000)) - 1$$

$$SSPADD = 10 - 1$$

$$SSPADD = 9$$

Inicialización I²C

```
void I2C_Init(void)
{
    TRISA1 = 1; // SCL
    TRISA2 = 1; // SDA

    SSP1ADD = 9; // 100kHz
    SSP1STAT = 0x80;
    SSP1CON1 = 0x28; // I2C Master mode
}
```

Generar condición START

```
void I2C_Start(void)
{
    SSP1CON2bits.SEN = 1;
    while(SSP1CON2bits.SEN);
}
Esto genera la transición:
SDA ↓ mientras SCL = 1
```

Generar condición STOP

```
void I2C_Stop(void)
{
    SSP1CON2bits.PEN = 1;
    while(SSP1CON2bits.PEN);
}
Transición:
SDA ↑ mientras SCL = 1
```

Generar condición START

```
void I2C_Start(void)
{
    SSP1CON2bits.SEN = 1;
    while(SSP1CON2bits.SEN);
}
Esto genera la transición:
SDA ↓ mientras SCL = 1
```

Generar condición STOP

```
void I2C_Stop(void)
{
    SSP1CON2bits.PEN = 1;
    while(SSP1CON2bits.PEN);
}
```

Transición:

SDA ↑ mientras SCL = 1

Ejemplo: escribir a un dispositivo

Supongamos un sensor con dirección **0x50**.

```
I2C_Start();
```

```
I2C_Write(0x50 << 1); // dirección + escritura
```

```
I2C_Write(0x00);      // registro
```

```
I2C_Write(0xAA);      // dato
```

```
I2C_Stop();
```

Flujo real en el bus

En un osciloscopio se vería así:

SCL _ _ _ _ _

SDA __ ADDRESS __ DATA __/
START STOP

Concepto clave

I²C funciona con **maestro y esclavos**.

Maestro genera reloj

Esclavos responden cuando su dirección coincide

Esto permite conectar **muchos dispositivos en el mismo bus**.

Aplicaciones típicas I²C

Sensores:

temperatura

Presión

acelerómetros

Memorias:

EEPROM

RTC (DS1307)

Pantallas:

OLED

LCD con adaptador I²C

Concepto fuerte

I

²C es como **una red compartida**.

Todos los dispositivos escuchan el mismo bus.

Solo responde el que reconoce su dirección.

Módulo 1 – Introducción y arquitectura básica

1. ¿Qué es un microcontrolador?

Un microcontrolador es un sistema electrónico integrado en un solo circuito que contiene:

- Unidad de procesamiento (CPU)
-
- Memoria de programa
-
- Memoria de datos
-
- Puertos de entrada y salida
-
- Módulos periféricos internos
-

A diferencia de una computadora de propósito general, un microcontrolador está diseñado para **controlar dispositivos físicos** como motores, sensores, luces o sistemas automáticos.

En este curso trabajaremos con el **PIC12F1840**, un microcontrolador compacto de 8 pines que, a pesar de su tamaño, integra múltiples recursos internos.