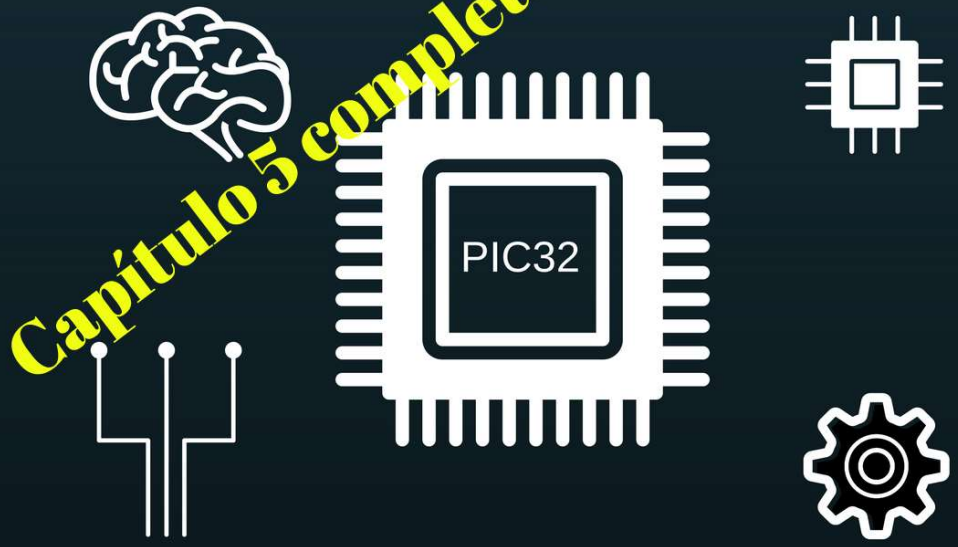
Programación Práctica del Microcontrolador PIC32

Implementando multitareas, estados de máquina e interrupciones con MPLAB® X y XC32

Capítulo 5 completo en la versión DEMO



Fabián Romo

Programación práctica del microcontrolador PIC32

Implementando multitareas, estados de máquina e interrupciones con MPLAB® X y XC32

Fabián Romo

Reconocimientos

La presente obra posee referencias a diferentes productos y herramientas y los derechos de autor de los mismos son propiedad de Microchip Technology.

Todas las imágenes relacionadas con diagramas, herramientas, etc, relacionados con productos de Microchip Technology tienen el permisos correspondiente para ser utilizados en este libro.

MPLAB®, XC32®, MPLAB® ICD 3 In-Circuit Debugger, PICkit™ 3 In-Circuit Debugger y MPLAB REAL ICE In-Circuit Emulator son marcas registradas de Microchip Tecnology Inc.

Introducción

El presente libro es una guía de cómo aprender a programar de manera práctica, firmware para administrar los periféricos del microcontrolador PIC32.

Este libro es una guía en la cual deseo compartir mis conocimientos respecto a este dispositivo.

El libro lo iré actualizando, modificando, corrigiendo errores en el mismo y agregando nuevos proyectos. También deseo que el lector aporte con ideas o sugerencias acerca del mismo para añadirlas en las siguientes versiones.

¿Para quiénes es este libro?

La presente obra está pensada en aquellas personas que ya han trabajado con algún tipo de microcontrolador y desean pulir, mejorar o tener una referencia de sus conocimientos.

¿Qué es lo que deberías conocer antes de leer este libro?

Este libro no es una introducción a programación en lenguaje C. Este libro asume que usted tiene por lo menos un conocimiento básico en ese lenguaje de programación.

También es importante conocer algo de electrónica digital y mucho mejor si usted ya ha trabajado con algún microcontrolador

¿Qué herramientas de software se utilizará?

El software que utilizaremos será el **MPLAB® X** la cual es gratis y trabaja en Windows, Linux y OS X, el compilador que utilizaremos es el **MPLAB® XC32 compiler** el cual nos permitirá crear nuestro firmware en lenguaje C. Más adelante, si el libro tiene una buena acogida, deseo utilizar periféricos avanzados para los cuales utilizaré **MPLAB® Harmony Integrated Software**

Framework que podría decir que es un conjunto de librerías en lenguaje C las cuales están escritas de manera abstracta de manear modular y flexible para el usuario.

¿Qué herramientas de hardware se utilizará?

En los primeros proyectos se utilizarán los microcontrolador PIC32MX170F256B y PIC32MX110F016B. Para las personas que desean utilizar una placa de pruebas denominada 'protoboard', la familia de esos microcontroladores (PIC32MX1XX/2XX PIN28) son ideales ya que vienen en encapsulado tipo **SPDIP**.

En futuros proyectos con periféricos avanzados podría recurrir a la utilización de alguna herramienta de desarrollo de Microchip Technology.

La herramienta de depuración y programación del MCU utilizada en los proyectos es el **MPLAB® ICD 3.**

Modelo de programación del Firmware

La forma en que está escrito el firmware será el modelo de estado funciones cooperativas lo que permite realizar varios procesos en 'paralelo'. Lo pongo entre comillas porque para procesamiento en paralelo deben al menos existir dos procesadores y el microcontrolador usado posee un solo CPU. Este modelo de programación es similar a **MPLAB® Harmony Integrated Software Framework** y más adelante se explicará claramente este proceso.

Acerca del autor

En la universidad cuando empecé a estudiar los microcontroladores, fue algo que me gustó y me apasioné tanto por esas pequeñas máquinas que hasta el día de hoy disfruto aprendiendo y diseñando algún tipo de sistema microcontrolado.

El primer microcontrolador con el cual empecé a jugar fue el PIC16F877®, en aquella época lo programaba en lenguaje ensamblador. Prácticamente el resto de mi carrera fue realizar varios proyectos de automatización en base a ese dispositivo.

Cuando terminé mis estudios pasé al siguiente nivel con el DSC dsPIC30F® pero aún mantenía la idea que programar en lenguaje ensamblador era lo mejor. Con dicho dispositivo construí un osciloscopio de 3 canales de baja frecuencia y la información adquirida se la visualizaba en una aplicación de PC. La información que adquiría el dsPIC® era enviada por comunicación USB al computador. Este proyecto solamente fue un pasatiempo.

Cuando empezó a llegar diferentes trabajos independientes, un amigo que ya se dedicaba a este tipo de negocio utilizaba el compilador en lenguaje C de CCS® con el cual conseguía resultados inmediatos en sus proyectos, así que poco a poco la idea de utilizar un lenguaje de alto nivel me empezó a gustar y empecé aprender por cuenta propia la utilización del compilador C18® de Microchip Technology para su familia de microcontroladores PIC18F®.

Luego trabajé en diferentes empresas en las cuales refiné aún más mi experiencia con microcontroladores PIC® de 8 bits y de 32 bits. Cada día aprendiendo y mejorando mis conocimientos.

Sugerencias y comentarios

Cualquier sugerencia o comentario las pueden realizar al siguiente correo electrónico: programacionpracticapic32@outlook.com

Fabián Romo

Organización de cada capítulo

Los primeros capítulos que poseerá este libro son los siguientes:

Capítulo 1. Instalación de las herramientas de software y breve explicación de las herramientas de hardware

En este capítulo se indicará los enlaces que permiten descargas las herramientas de software con las cuales se trabajan en este libro.

Capítulo 2. Una mirada rápida a la arquitectura del microcontrolador PIC32

En esta parte se tratará de una manera breve como está constituida la arquitectura del MCU. Cuando empiezas a programar proyectos con cualquier microcontrolador debes tener claro de una manera global cuál es su arquitectura, pero está se la comprende mejor a medida que desarrollas proyectos más complejos.

Capítulo 3. Primer proyecto

El primer proyecto va ser algo tan simple que es encender un diodo led con un pulsante, todo mediante el microcontrolador. Este proyecto es didáctico y permitirá al lector familiarizarse con las herramientas instaladas en el capítulo 1 y configurar de manera correcta al MCU.

Capítulo 4. Introducción a las multitareas e interrupciones en un microcontrolador

Aquí se revisará estos conceptos que nos permiten realizar procesos en ‘paralelo’ (nuevamente entre comillas) con un microcontrolador. Este capítulo es importante para aquellos lectores que desconocen de este tema ya que todos los siguientes proyectos se programarán de esa manera.

Capítulo 4. Encendido y apagado de un led de manera periódica.

Aplicando la programación de multitareas se realizará el encendido y apagado de un led periódicamente por el microcontrolador.

Capítulo 6. Cambio del modo de encendido ya apagado de un led con un pulsante mediante la interrupción externa.

Similar al proyecto del capítulo 4, mediante el pulsante se cambiará la manera de encender y apagar al led.

Capítulo 7. Envío y recepción de datos mediante comunicación RS232

Este proyecto indicará los conceptos básicos para enviar y recibir datos desde una PC mediante la clásica comunicación RS232.

Capítulo 8. Manejo de un display de 7 segmentos de 3 dígitos mediante multitareas

Al proyecto anterior añadiremos un display para mostrar diferentes números mediante el pulsante

Capítulo 1

Instalación de las herramientas de software

1.1 Herramientas de software

La primera herramienta que se utiliza en este libro es el MPLAB® X (a que se puede descargar del siguiente enlace:

http://www.microchip.com/mplab/mplab-x-ide

Una vez instalado correctamente la aplicación, su aspecto será algo similar al indicado en la Figura 1.1.

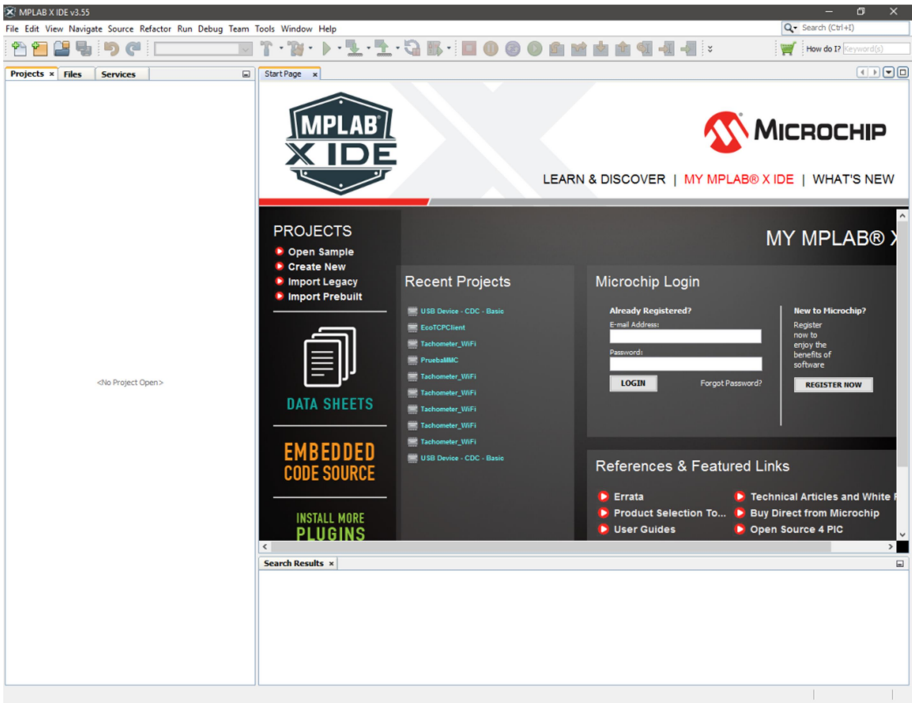


Figura 1.1. Aspecto de MPLAB®X

La segunda herramienta que instalaremos será el compilador MPLAB® XC32 cuyo enlace de descarga es el siguiente:

http://www.microchip.com/mplab/compilers en la pestaña 'Downloads'

Este compilador tiene tres tipos de licencias, una gratuita, una estándar y una profesional.

La licencia gratuita tiene la desventaja que no optimiza el código que escribimos pero es útil para aprendizaje y entrenamiento.

La licencia estándar tiene un costo que podríamos llamar medio y reduce el tamaño del código.

Finalmente si nuestro proyecto necesita mayor espacio de memoria y/o velocidad de procesamiento, la licencia profesional es ideal a utilizar, pero su costo de uso es mayor.

También podemos recurrir a una licencia demo de 60 días con la cual podemos trabajar en la mejor de las compilaciones (profesional).

Cuando instalamos cualquier compilador XC® nos pide con cuál de esas licencias vamos a trabajar. Figura 1.2

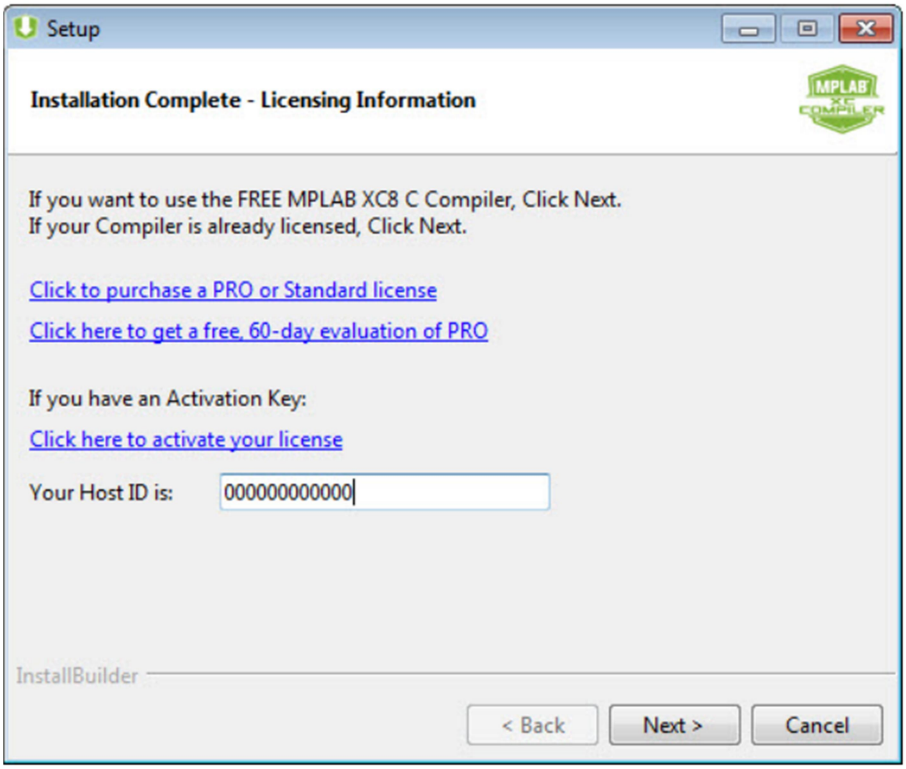


Figura 1.2. Información de las licencias para el compilador XC8®, es algo similar al XC32®

Cuando se necesita una licencia para varios desarrolladores, si el número de computadoras no es mayor a 3, se puede recurrir a una licencia de tipo estación de trabajo o 'Work Station'.

Si se necesita una licencia para varias computadoras, es mejor utilizar una licencia en un servidor o 'Network Client'.

Para más detalles acerca de estas características de los compiladores XC® por favor refiérase al documento 50002059 de Microchip Technology denominado **Installing and Licensing MPLAB® XC C Compilers** que puede ser descargado del siguiente enlace:

http://www.microchip.com/mplab/compilers

En la pestaña 'Documentation', la opción 'Installing and Licensing MPLAB® XC C Compilers'.

2.2 Herramientas de hardware

En internet usted puede encontrar diferentes clases de 'programadores' que permiten descargar nuestro programa hacia el microcontrolador, pero lo mejor es utilizar un programador y depurador que no solamente permite descargar nuestro firmware, sino que nos permite interactuar con el hardware para depurar el código que hemos desarrollado.

Yo utilizo el "**MPLAB® ICD 3 In-Circuit Debugger**" Figura 1.3

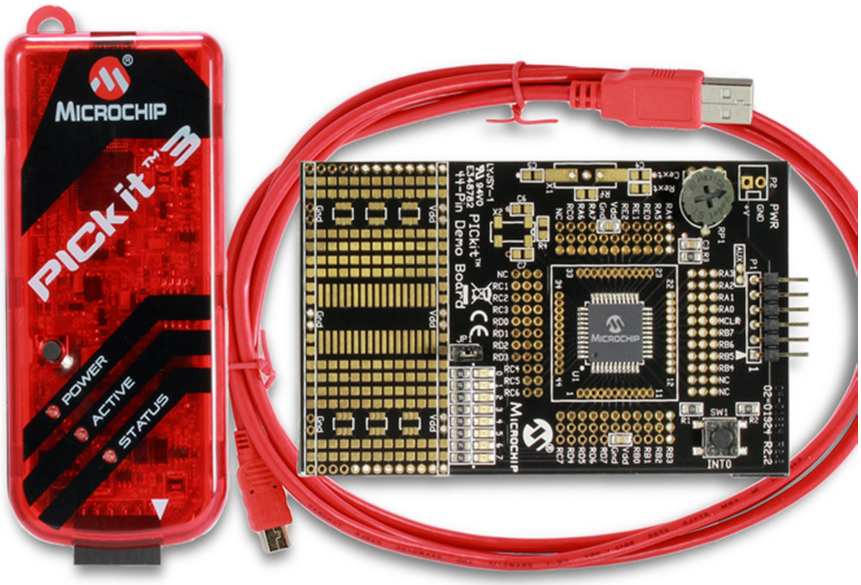
Una opción más económica puede ser el "**PICkit™ 3 In-Circuit Debugger**" el cuál posee algunas limitantes respecto a la anterior herramientas como la velocidad de comunicación con el computador, el número de puntos de ruptura o 'breakpoints' que se puede añadir en el código al momento de la depuración, etc. Figura 1.4

También existe una herramienta de mejores características que se llama "**MPLAB® REAL ICE In-Circuit Emulator"** Figura 1.5

O la última versión del ICD, el MPLAB® ICD4. Figura 1.6.

MPLAB® ICD 3 and Explorer 16 Kit
(Part # DV164037)

Figura 1.3. MPLAB® ICD3 con tarjeta de desarrollo Explorer 16 Kit.



PICkit™ 3 Debug Express
(Part # DV164131)

Figura 1.4, PICkit™ Debug Express

Figura 1.5 MPLAB® REAL ICE.

Figura 1.6. MPLAB® ICD 4 In-Circuit Debugger

También debo indicar que muchas de las herramientas de desarrollo con kits de tarjetas demo poseen un circuito programador/depurador embebido de tal manera que no es necesario comprar alguna de los dispositivos anteriores, si deseas en base a esas herramientas de desarrollo construir tu propio hardware deberás recurrir a un dispositivo externo para descargar el firmware diseñado en el MCU.

Capítulo 2

Una mirada rápida a la arquitectura del microcontrolador PIC32

2.1 Procesador MIPS

El CPU del microcontrolador PIC32 es un procesador RISC de tipo MIPS (Microprocessor without Interlocked Pipeline Stages) desarrollado por la empresa MIPS Technologies.

Una característica que he observado de Microchip Technology es que intenta no ser igual a sus competidores (antes de la adquisición de Atmel). La mayoría de fabricantes de microcontroladores de 32 bits apuestan por el procesador ARM mientras que Microchip Technology se arriesgó a algo diferente.

Hay mucho debate en foros y blogs acerca de las ventajas y desventajas de dicha tecnología, pero lo que realmente mueve el desarrollo de un producto es el costo de su diseño, su mantenimiento, etc. y en algún lado leí que las licencias del núcleo MIPS son más flexibles que el núcleo ARM que posee muchas más restricciones de uso.

En fin, sin importar las razones por las que se tomó esa decisión, la ventaja de utilizar un microcontrolador de 32 bits son las siguientes:

- Desempeño del MCU
- Manejo de excepciones
- Mejor procesamiento aritmético
- Interrupciones vectorizadas (respecto a microcontroladores de 8 y 16 bits de MCHP)
- Mayor memoria RAM y de programa

Respecto al desempeño es claro que muchas instrucciones que necesitaban realizarse en varios pasos en un microcontrolador de 8 bits se las puede realizar en menos y a una velocidad de reloj superior, así que si un diseño necesita un mejor desempeño con respecto a velocidad, un PIC32 puede ser una mejor elección.

El manejo de excepciones es algo muy útil cuando se realiza alguna operación ilógica como apuntar y escribir a una región que no existe en la memoria, dividir un número para cero, etc. Nos permite analizar qué fue lo que causó la excepción y evitar que el MCU se reinicie.

El procesamiento aritmético me parece que debería estar dentro de la categoría 'Desempeño del MCU' ya que al ser de 32 bits pueden realizar dichas operaciones de manera más eficiente y rápida.

Las interrupciones vectorizadas era algo que no poseían los microcontroladores de 8 bits de microchip. Por ejemplo para el PIC18F® existe un vector de interrupción de alta prioridad localizado en la dirección 0x08h de la memoria de programa y uno de baja prioridad localizada en la dirección 0x18h de dicha memoria. Cuando se deseaba que un periférico genere una interrupción, esta se asociaba con alguno de esos vectores o prioridades.

El problema era cuando se utilizaba varios periféricos para generar una interrupción en el mismo vector, lo cual implicaba realizar un discernimiento mediante las banderas de los registros asociados. Si por ejemplo quien generó la interrupción fue el último periférico de la lista donde se están realizando dicha comprobación, esto implica que la interrupción tiene una latencia mucho mayor que aquella que se analiza al principio.

Si la velocidad del sistema que controla el MCU no es tan crítica o no es tan importante, dichas latencias en las interrupciones son desapercibidas. Actualmente Microchip lanzó al mercado los microcontroladores de 8 bits denominada PIC18F® “K42” con interrupciones vectorizadas.

Finalmente al ser el PIC32 de 32 bits puede administrar un bus de datos mayor con lo cual puede direccionar una mayor memoria de programa o de datos.

En este libro en el inicio vamos a trabajar con el microcontrolador PIC32MX cuyo procesador MIPS es de tipo M4K. Figura 2.1

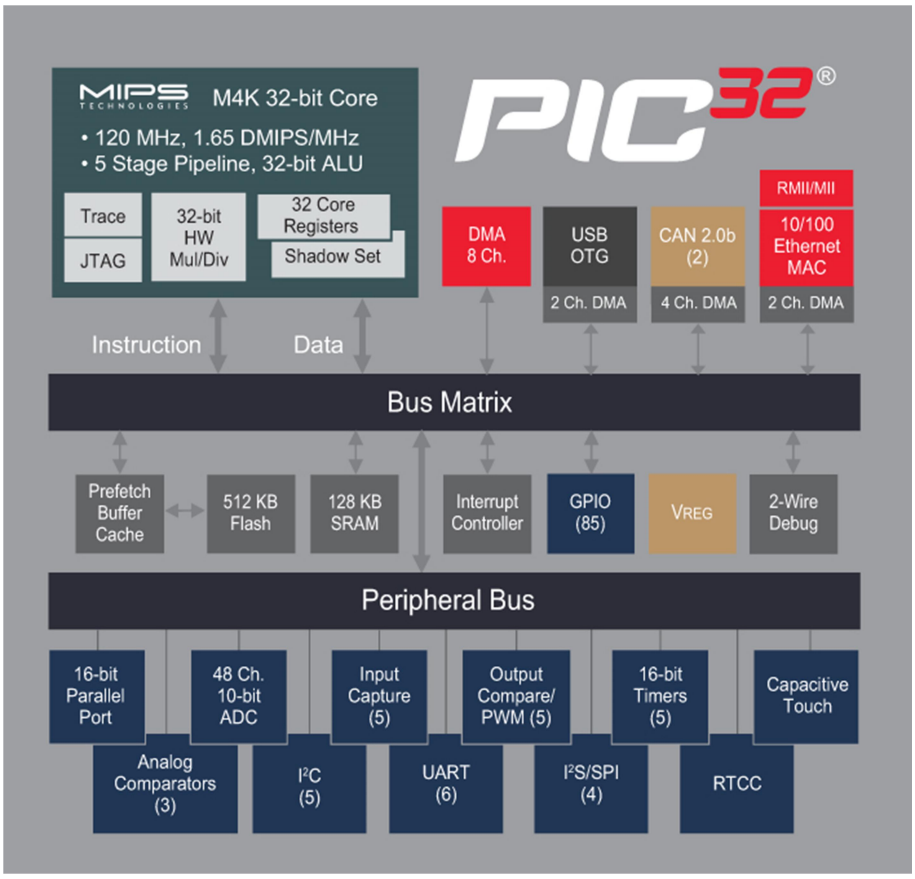


Figura 2.1. Diagrama del núcleo M4K

2.2 El mapa de memoria

A diferencia de los microcontroladores de 8 bits de MCHP cuyos procesadores tenían una arquitectura denominada Harvard, es decir la memoria RAM y de programa estaban separadas, las memorias del microcontrolador están en un mismo mapa de memoria (arquitectura Von Neumann).

Esta arquitectura elimina la manera que el contador de programa tenía los para acceder a dichas memorias en los dispositivos de 8 bits y 16 bits. Aquellos que han trabajado con esos microcontroladores y sobre todo si han programado en lenguaje ensamblador, había que tener en cuenta en que banco de memoria se encontraba una variable. También la memoria de programa se dividía en varios bancos. Con un lenguaje de alto nivel no había que pensar mucho en ese problema, ya que el compilador realizaba los pasos adicionales para acceder a las variables o al programa en otras localidades, lo que implicaba generar más código de programación.

También existe la posibilidad de ejecutar código de programa en la memoria RAM del microcontrolador. En el foro de microchip alguna vez leí que alguien había almacenado funciones en una memoria flash externa cuya capacidad era mucho mayor que la memoria de programa. Cada vez que se necesitaba realizar algún proceso con una función específica, se leía la función necesaria de la memoria externa y la almacenaba en RAM para ejecutarla.

Pero también hay algo especial de este microcontrolador y es que el procesador MIPS posee un módulo de caché de memoria el cual tiene dos buses de datos de 32 bits, uno para instrucciones y otro para datos lo que permite al procesador obtener instrucciones y datos simultáneamente con lo que se podría decir que se comporta como la arquitectura von Neumann. Al habilitar esta característica el desempeño en el procesamiento de datos mejora aún más.

El procesador MIPS posee un módulo denominado 'unidad de administración de memoria' o MMU que permite operara en dos diferentes modos, el de usuario y *kernel.*

EL modo *kernel* es utilizado cuando se desea que el procesador ejecute un sistema operativo, mientras que el modo de usuario permite ejecutar un firmware de control común y corriente.

El microcontrolador PIC32 está orientado a ese tipo de aplicaciones que no necesitan ser tan complejas por lo que no existe el MMU sino un traductor de mapeo fijo FMT o *fixed mapping translation* con un mecanismo de control mediante un bus denominado *bus matrix* BMX.

De esa manera el FMT regulariza al CPU para que utilice direcciones de memoria estándar de acuerdo al modelo de programación. Esto es una ventaja ya que de esa manera el tamaño del CPU es más pequeño.

En cambio el bus BMX permite al CPU comunicarse de manera flexible con las memorias, el DMA, la unidad de depuración, etc.

En la Figura 2.2 se muestra el mapa de memoria para la familia del microcontrolador PIC32MX170F256B (imagen de sus hojas de datos). En dicho mapa se puede ver dónde está la memoria RAM, la memoria de programa, sus periféricos (SFR), la sección de memoria flash para *boot*, etc.

En realidad la imagen de la Figura 2.2 muestra dos mapas de memoria, uno virtual y el otro físico. Utilizar un sistema operativo implica que un área de memoria RAM es destinada para los datos o variables del usuario mientras que otra parte es el núcleo o *kernel* del sistema operativo. El usuario no debe tener la posibilidad de sobrescribir esta área por voluntad o por error. Mediante registros especiales, el diseñador del firmware puede dividir las regiones en las secciones mencionadas. Al realizar estas divisiones, el FMT convierte las direcciones físicas en direcciones virtuales,

de esa manera el usuario no podrá alterar la región que corresponde al sistema operativo, ni alterar a los periféricos, etc.

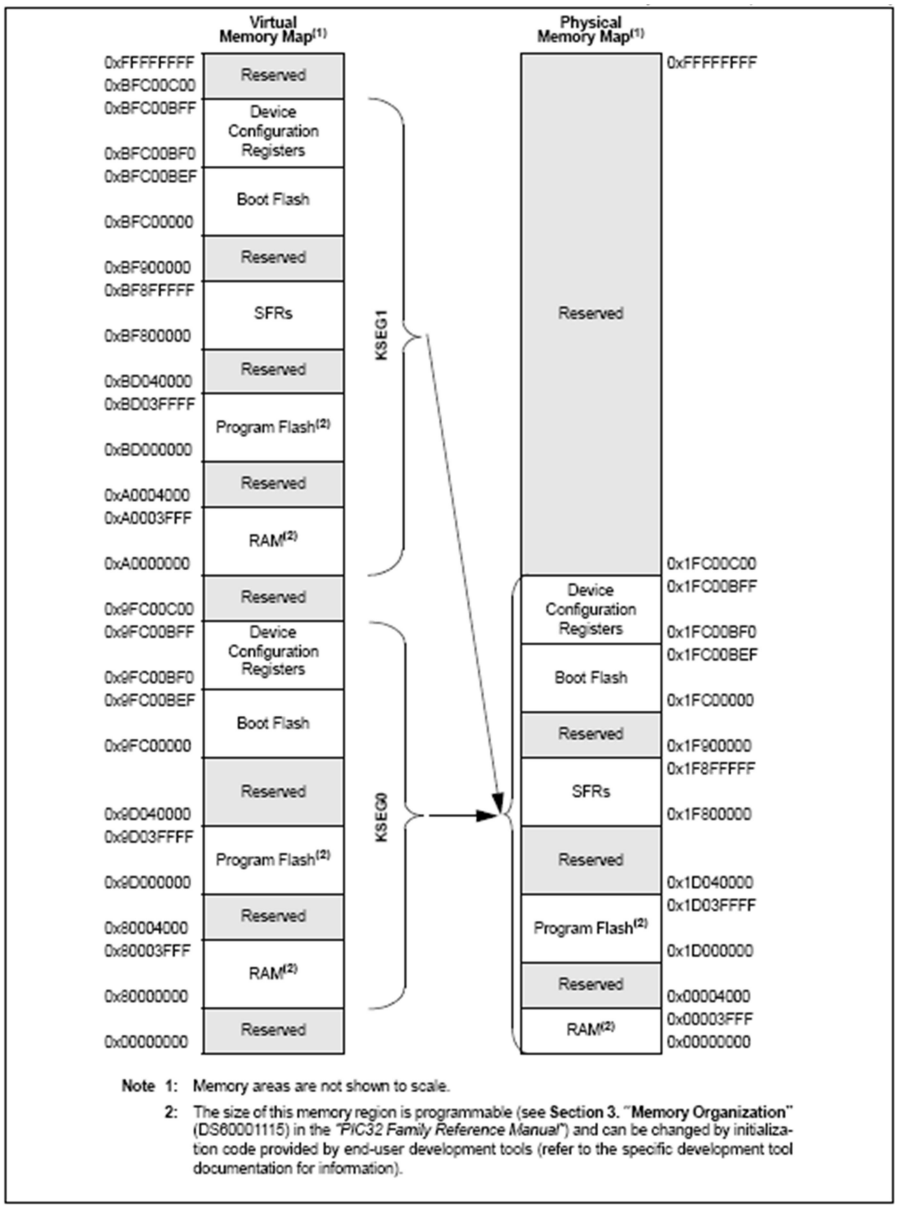


Figura 2.2. Mapa de memoria del microcontrolador PIC32MX170F256B.

Entrar en detalles respecto a estos particulares sería necesario si deseamos utilizar un OS en nuestro dispositivo, pero generalmente no se utiliza todas estas funciones, el compilador crea el código de tal manera que se ejecute en modo *kernel*.

Capítulo 3

Primer proyecto

3.1 Breve descripción del 'Primer proyecto'

Tal cuál mencioné anteriormente el primer proyecto es algo muy sencillo como encender un led con un pulsante y el microcontrolador PIC32MX170F256B es un intermediario en dicha tarea. Figura 3.1

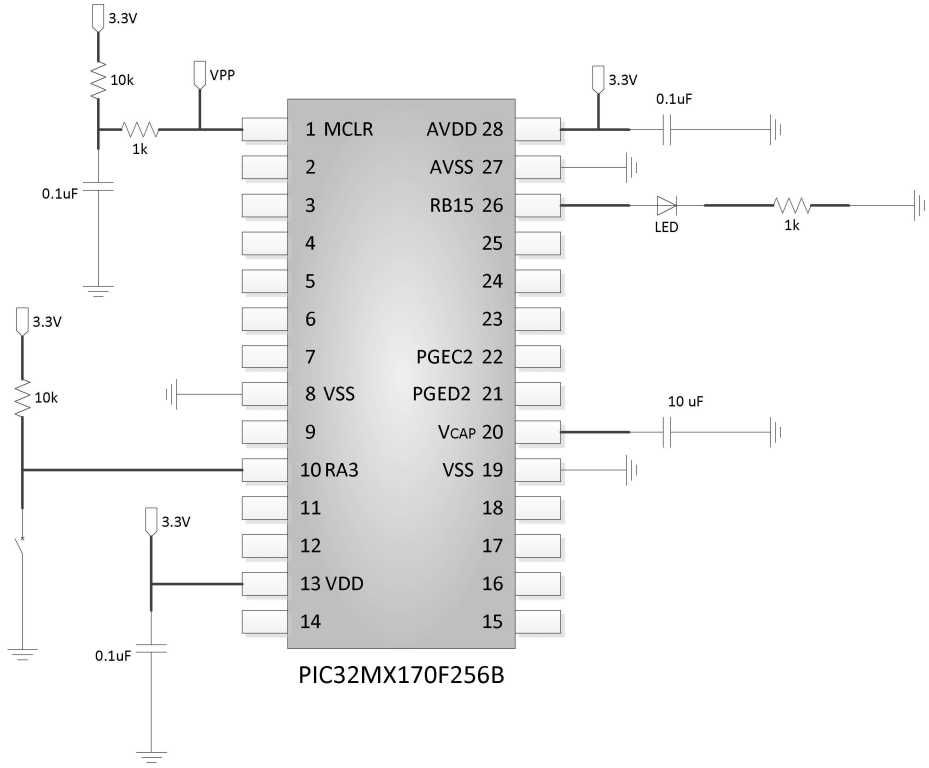


Figura 3.1. Diagrama esquemático del Primer proyecto.

Este proyecto es didáctico ya que permite relacionar al lector con la herramienta MPLAB® X y el compilador, ya que para encender un led simplemente había que conectar en serie al pulsante con el led y una resistencia.

3.2 Análisis del hardware para el primer proyecto.

El microcontrolador utilizará el oscilador interno RC, por el momento no utilizaremos algún cristal externo. Además de la ventaja económica de utilizar el oscilador interno, también se gana algo de espacio físico en el circuito PCB. La desventaja es evidente en un sistema altamente preciso, como comunicaciones de alta velocidad, sistemas que trabajan en ambientes de altas temperaturas, en esas condiciones posiblemente no es muy buena idea usarlo.

También es recomendable para un diseño que no sea didáctico utilizar los condensadores de filtro que recomienda el fabricante en sus hojas de datos. Figura 3.2

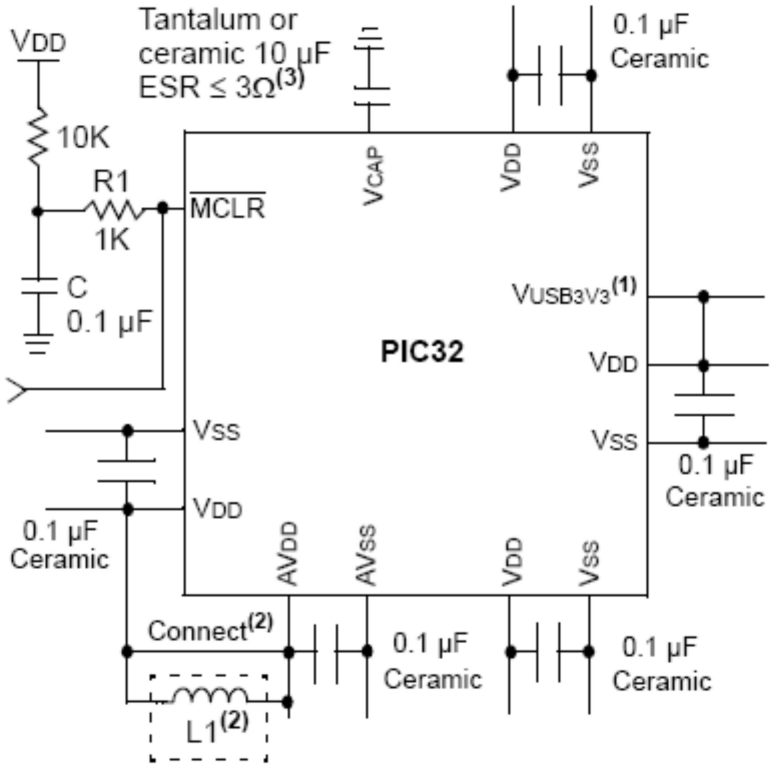


Figura 3.2 Condensadores de filtro recomendados en los terminales de polarización del microcontrolador

Nótese que el terminal denominado V_{CAP} tiene un capacitor cerámico de 10 uF, este capacitor es necesario debido a que el microcontrolador utiliza un regulador interno de 1.8V para el CPU, si en dicho terminal el voltaje medido respecto a V_{SS} no es el mencionado, el microcontrolador no funcionará correctamente. Esta falla es debido a un daño en el capacitor o en la conexión eléctrica al mismo.

Personalmente yo he usado un capacitor electrolítico el cual no cumple con un valor bajo ESR, en un microcontrolador PIC32MX795F512H, podría decir que funcionaba bien a velocidades bajas (20 MHz para los periféricos y 40 MHz para el CPU) pero intentando alcanzar los 80 MHz que es la velocidad máxima en ese MCU, el microcontrolador simplemente no funcionaba.

Así que lo recomendable es hacer caso a lo que el fabricante indica en sus hojas de datos, fuera de los rangos recomendados, el fabricante no garantiza el funcionamiento correcto del mismo.

También en la Figura 3.1 se ve que utilizo el canal número 2 para la programación y depuración del hardware (PGEC2 y PGED2).

3.3 Crear un nuevo proyecto en MPLAB® X

Una vez que abrimos MPLAB® X damos clic en la opción '*File*' de la barra de menú y escogemos la opción '*New Project*'. Figura 3.3.

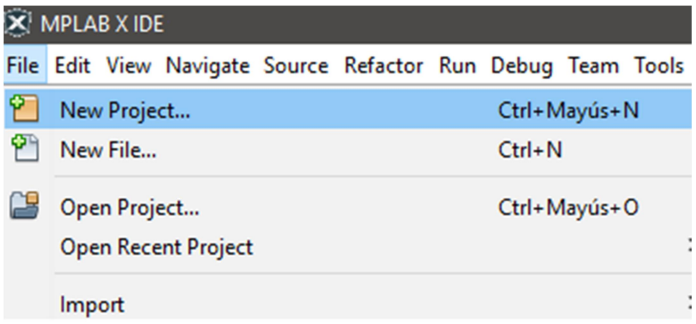


Figura 3.3. Creación de un nuevo proyecto.

Al realizar dicha acción se abrirá una ventana para un nuevo proyecto en la cual hay dos listas verticales, una para categorías '*Categories*' y otra para proyectos '*Projects*'. Seleccionamos '*Microchip Embebded*'. Figura 3.4.

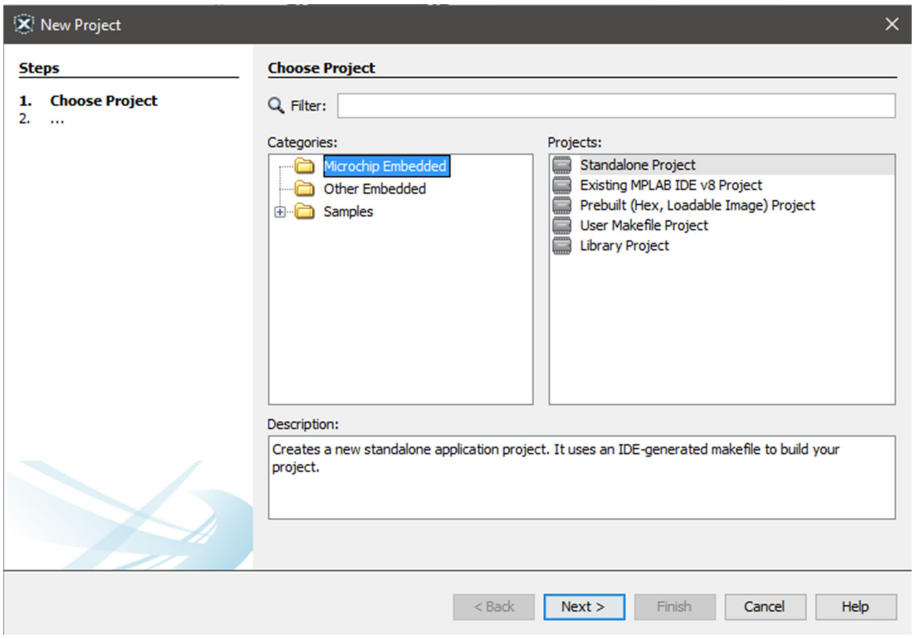


Figura 3.4. Seleccionado el tipo de nuevo proyecto a realizar.

Presionamos el botón denominado '*Next >*' y la siguiente ventana en presentarse nos permitirá escoger el tipo de dispositivos con el cual trabajaremos. Posee dos listas desplegables, la primera que es la superior nos permite reducir la búsqueda del dispositivo mediante la familia a la que pertenece, la segunda nos permite buscar al dispositivo específicamente. También podemos en esta segunda lista desplegable simplemente escribir el nombre del MCU a utilizar y una vez seleccionado damos clic en '*Next >*'. Figura 3.5

Luego de presionar el botón 'Next >' la siguiente ventana nos preguntará que tipo de herramienta de depuración vamos a utilizar. En mi caso selecciono al ICD3®, al final del proyecto utilizaremos el simulador para su demostración. Figura 3.4.

Cuando una herramienta no es útil para el dispositivo seleccionado, se marca en color rojo. Por ejemplo en la Figura 3.4, el PICkit® 2 no es útil para programar el MCU seleccionado o la versión de MPLAB® ya no soporta a dicha herramienta.

Cuando es de color amarillo indica que parcialmente puede trabajar con el dispositivo, como por ejemplo, no puede depurar o emular a algunos los periféricos del microcontrolador. Generalmente esto sucede cuando el dispositivo es nuevo y aún no se han implementado todas las funcionalidades para su simulación o depuración.

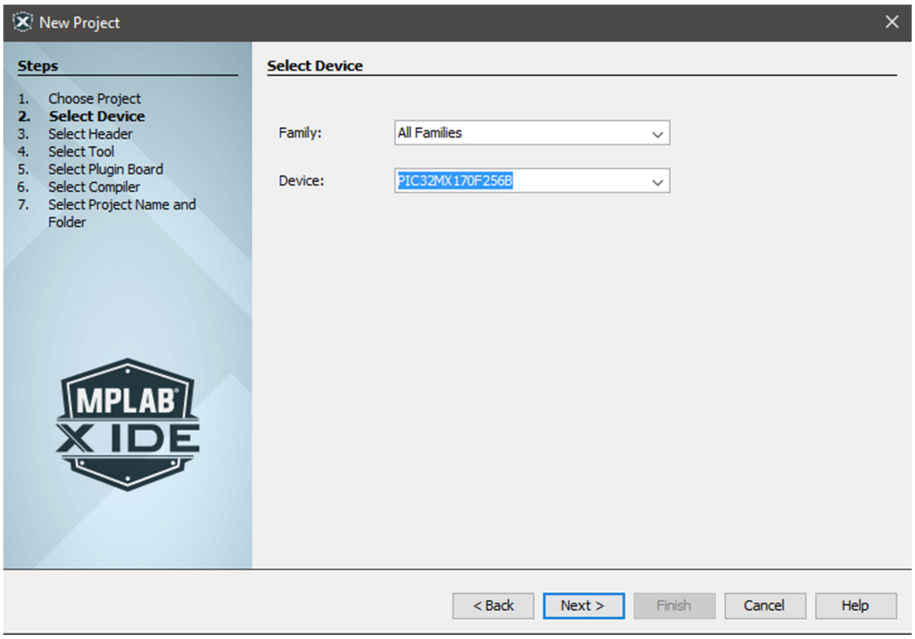


Figura 3.5. Seleccionando al MCU PIC32MX170F256B

Luego de seleccionar '*Next >*' la siguiente ventana en aparecer nos pide ingresar el compilador, yo selecciono XC32 v1.43. Figura 3.5

Nuevamente damos un clic en '*Next >*' y la última venta en aparecer nos permite seleccionar la ruta del proyecto y el nombre del mismo. Figura 3.6.

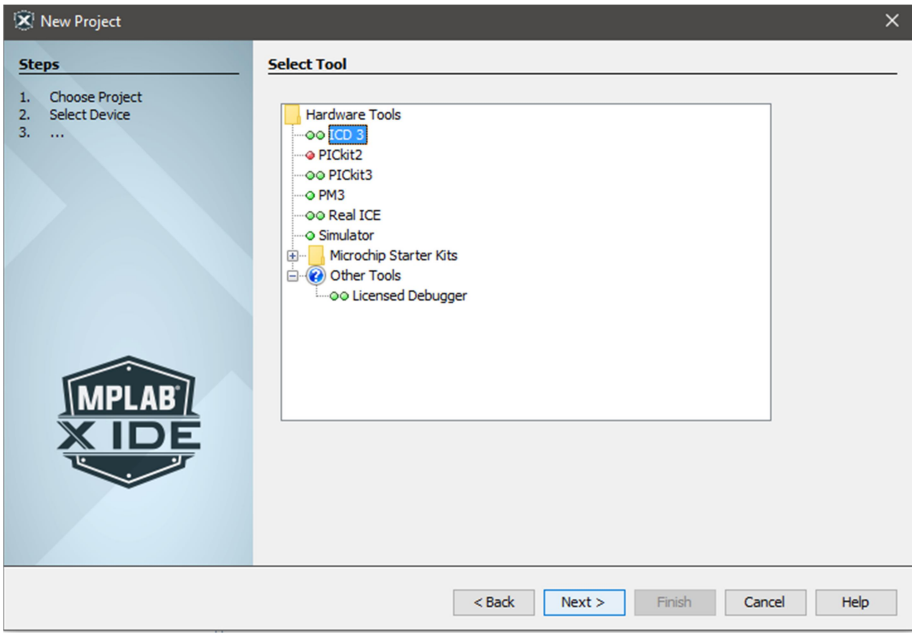


Figura 3.4. Selección de la herramienta para depuración y programación del dispositivo.

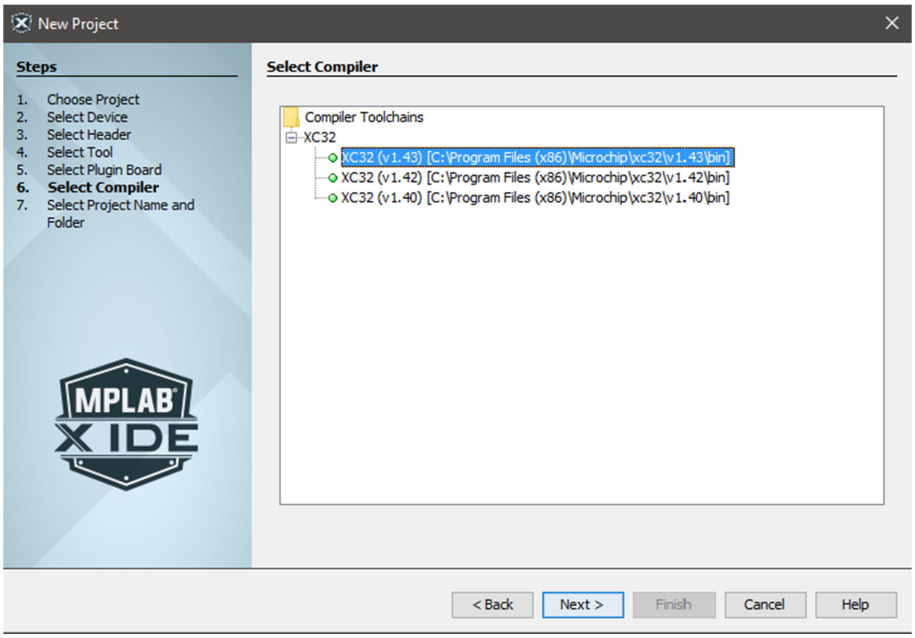


Figura 3.5. Seleccionando el compilador.

Presionamos el botón '*Finish*' y el proyecto tendrá un aspecto similar al indicado en la Figura 3.7.

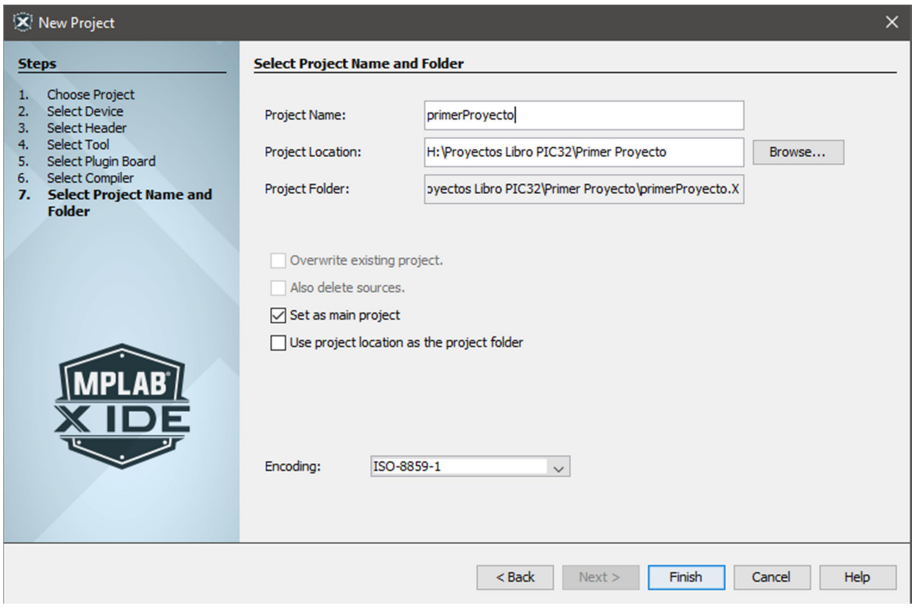


Figura 3.6. Seleccionando el nombre del proyecto y la ubicación del mismo.

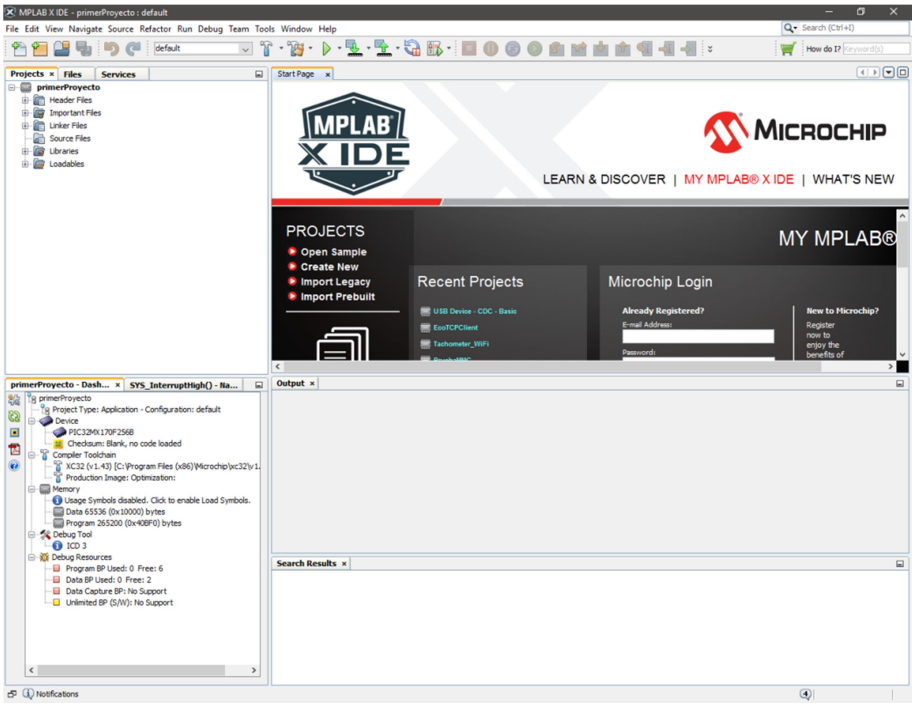


Figura 3.8. Proyecto en blanco y listo.

Aquellos lectores que han trabajado con el IDE NetBeans notaran que es casi exactamente igual a ese software ya que está construido con dicho núcleo.

A medida que avancemos en los diferentes proyectos iré explicando las diferentes partes del software.

Si alguna ventana se cierra y no conocen como volver abrirla pueden restaurar la presentación de MPLAB® X en la opción '*Windows*' de la barra de menú y escogemos la opción '*Reset Windows*'.

La siguiente acción que se procederá a realizar será la creación el archivo principal o *main* del proyecto. Para eso vamos a la ventana de proyectos (Figura 3.9) y en la carpeta *Souce Files* damos clic derecho sobre dicha carpeta.

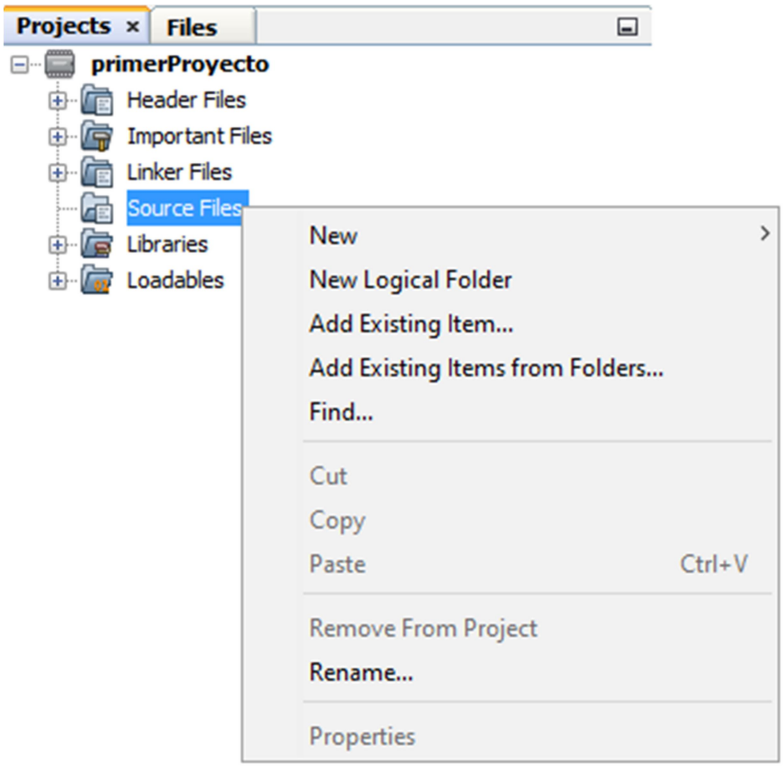


Figura 3.9

Seleccionamos la opción *New* y escogemos *C Main File* Figura 3.10

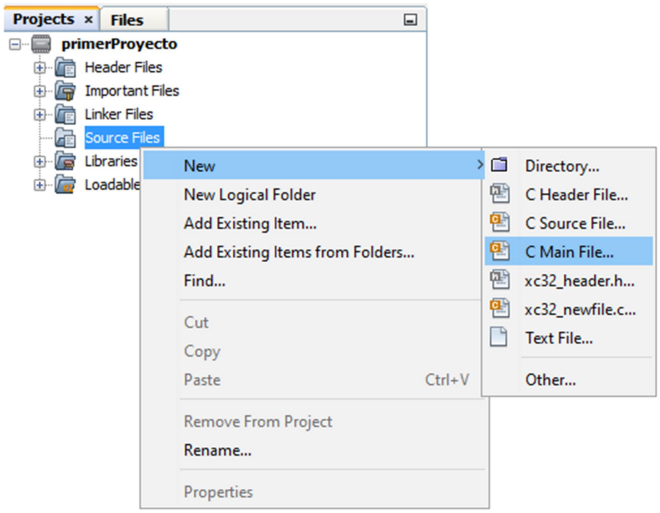


Figura 3.10. Creación de un nuevo archivo principal.

A continuación aparece una nueva ventana que nos permite ingresar el nombre para el archivo principal del proyecto, Figura 3.11. Se puede poner cualquier nombre, pero yo prefiero llamarlo *main* (de extensión c).

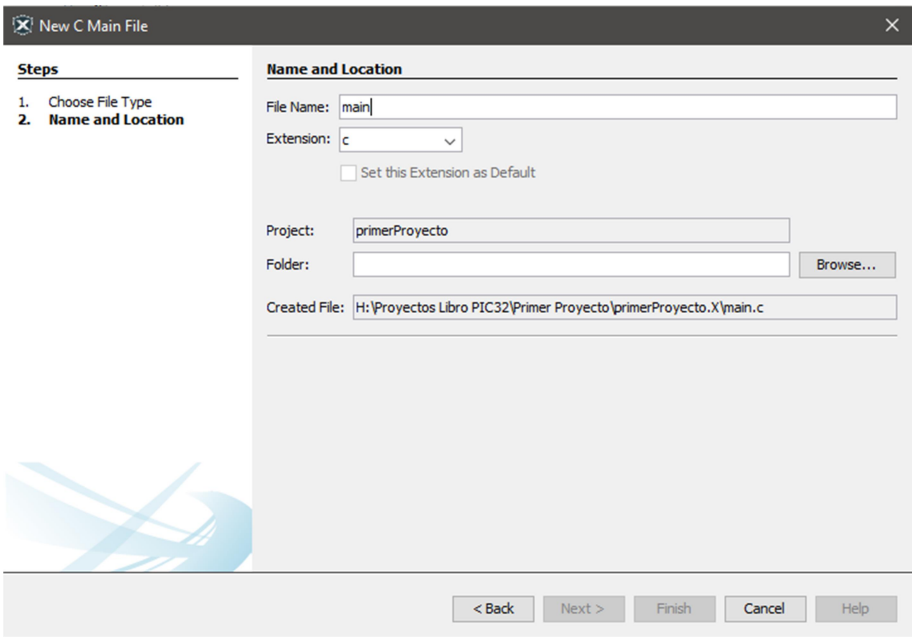


Figura 3.11 Creación del archivo principal del proyecto.

Al presionar el botón 'Finish' el proyecto tendrá un aspecto similar al indicado en la Figura 3.12

El siguiente paso es compilar el proyecto que nos permitirá determinar si el proyecto se ha creado sin errores, para lo cual damos un clic en el botón *Clean and Build Main Project* en la barra de herramientas. Figura 3.13

Si el proyecto se compila sin problemas, en la ventana de salida de mensajes *Output* nos mostrará algunos mensajes similares a los indicados en la Figura 3.14 informando que el proyecto se ha compilado correctamente.

En la ventana *DashBoard* podemos apreciar cuanta memoria de programa y memoria RAM ocupará el proyecto. Figura 3.15

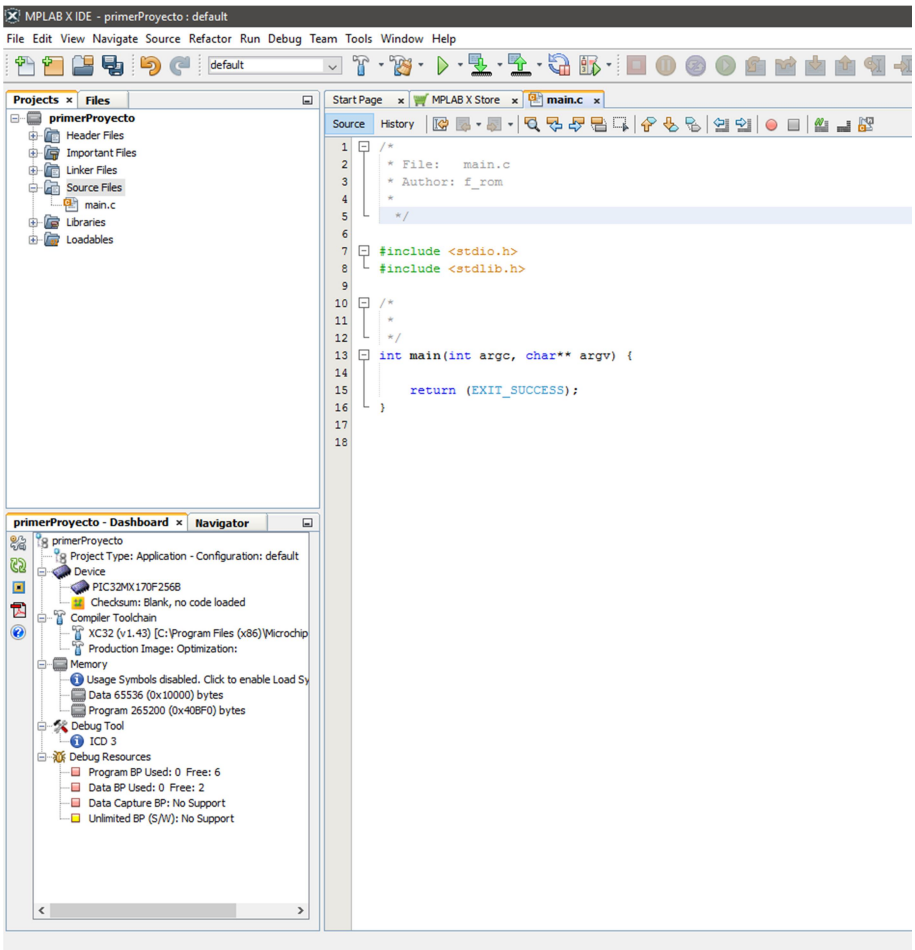


Figura 3.12. Proyecto con la función main.

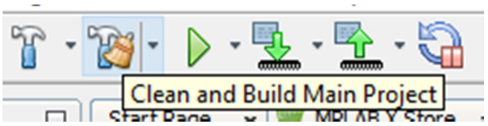


Figura 3.13 Limpiar y compilar el proyecto principal

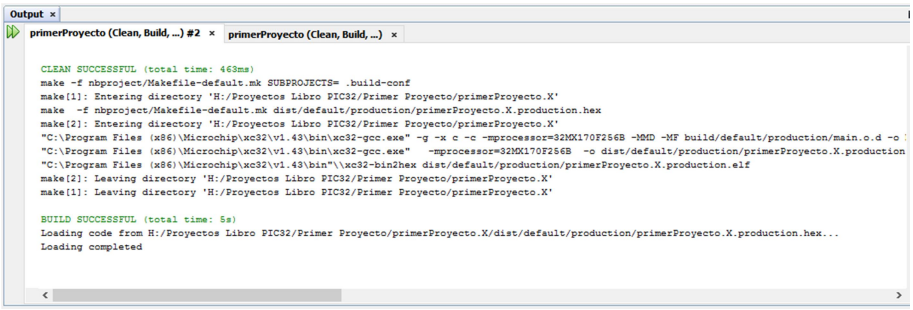


Figura 3.14. Ventada de salida del MPLAB® indicando que la compilación se ha realizado sin problemas.

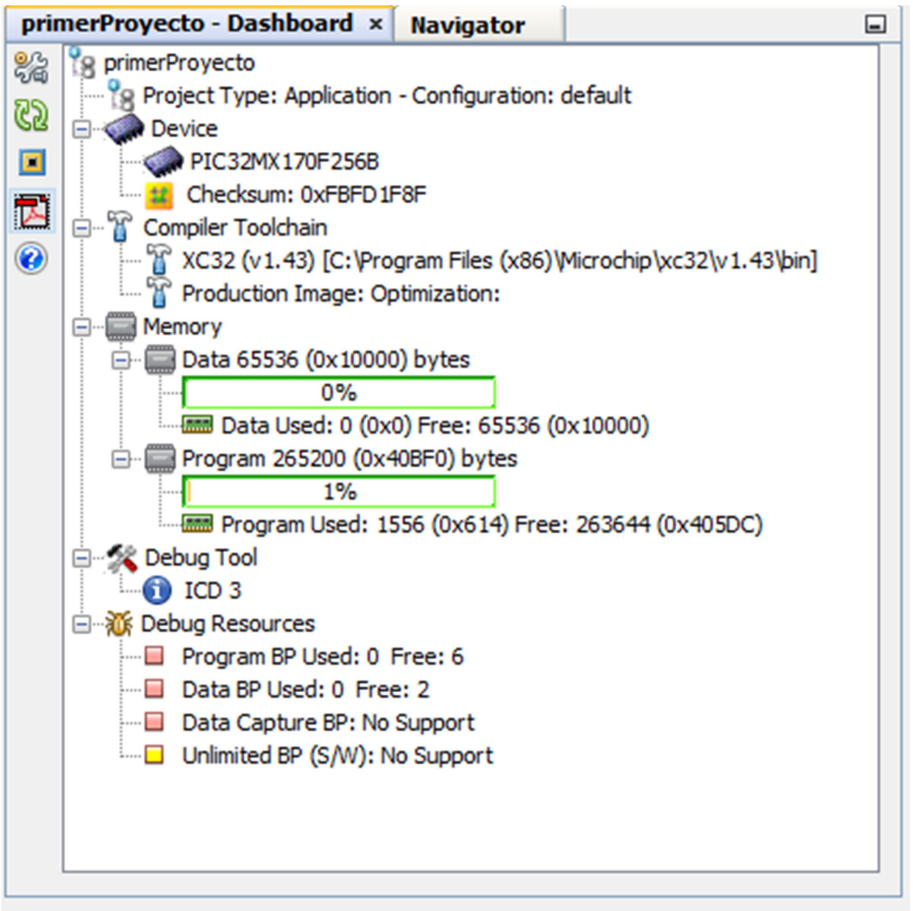


Figura 3.15. Dashboard indicando algunas características del proyecto.

La *DashBoard* también nos indica algunas cosas como cuantos puntos de ruptura de hardware hay disponibles (Program BP), puntos de ruptura por software (Data BP), etc. Esas características se detallarán a medida que se vaya desarrollando el proyecto.

El proyecto por defecto arranca con la configuración de mínima optimización del compilador, para cambiar dicha opción damos un clic en Propiedades del Proyecto o *Project Properties*. Figura 3.17 y aparece una ventana que nos permite ver y modificar las propiedades del proyecto. Figura 3.18

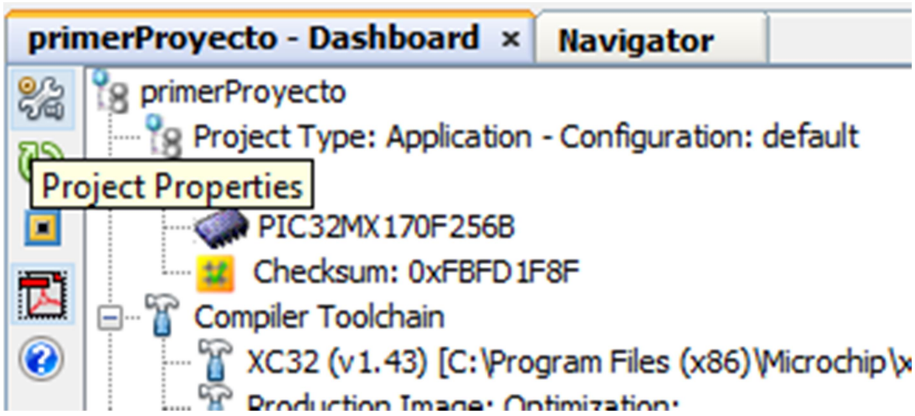


Figura 3.17. Accediendo a las propiedades del proyecto.

En la ventana de propiedades del proyecto, en la lista denominada categorías '*Categories*', en la opción *XC32(global Options)* escogemos la opción *xc32-gcc*. En la lista desplegable denominada *Option categories* escogemos optimización *Optimization.* Figura 3.18

Existen cuatro rangos de optimización del código, digo cuatro, ya que la primera opción es 0 y es igual a ninguna optimización.

La opción 1 y 2 reducen el código generado, pero la mejor reducción del código es la opción 3.

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

Capítulo 4

Introducción a las multitareas e interrupciones en un microcontrolador

4.1 ¿Qué es un sistema?

Un sistema es algo complejo de describir pero se puede afirmar que es un objeto real o conceptual que soluciona algún problema. La descripción puede ser mucho más complicada pero para comprender este capítulo propongo un ejemplo.

Imaginemos que tenemos una sencilla fuente de voltaje DC variable y programable como se indica en la Figura 4.1.

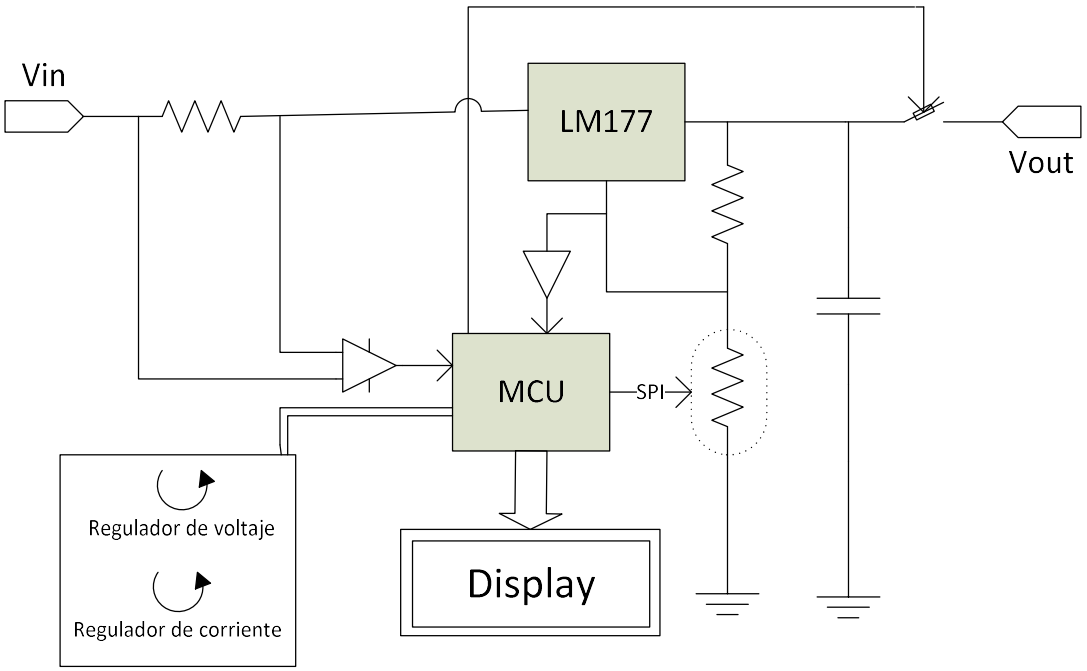


Figura 4.1. Diagrama de una fuente de voltaje controlada por un microcontrolador.

Las partes fundamentales de este sistema son:

- Entrada y salida de voltaje.
- Control de la resistencia digital (vía SPI)
- Un display para mostrar información
- Potenciómetros analógicos para controlar la corriente y voltajes de salida
- Medición de corriente del sistema.
- Medición del voltaje de salida
- Relé para desconexión rápida de la carga

En esta fuente existen varios procesos que deberá realizar el microcontrolador, por ejemplo el MCU deberá incrementar el valor de la resistencia digital hasta que el voltaje de salida sea el deseado por el usuario y obviamente sin descuidar la corriente que circula por el sistema.

Todo sistema que se diseña debe funcionar como se espera y la respuesta debe estar dentro de los tiempos establecidos. El firmware escrito para un microcontrolador debe ser lo más legible posible, que pueda ser fácilmente expandible, modificable o corregible. Además debe estar dividido en módulos de tal manera que puedan más de un programador encargarse del desarrollo del mismo si es necesario.

Por ejemplo un programador se encargaría de escribir el código que corresponde al control de la resistencia digital para obtener el voltaje de salida, mientras otro podría realizar el código para mostrar el voltaje de salida medido en el display.

Estas dos partes de código deberán funcionar de tal manera que no se interpongan entre ellas o sí existe intervención se a lo mínimo posible, estos objetivos se alcanzan mediante un **planificador de tareas cooperativas** (*Schuleder*) y **procesamiento de multitarea apropiativa** (*Preemptive Multitasking*).

4.2 Tarea concurrente

Un proceso concurrente en el ejemplo es modificar el valor de la resistencia digital hasta alcanzar el voltaje deseado. Supongamos que dicho proceso es de la manera sencilla sin cosas complicadas como un control PID.

El proceso de manera sencilla es primero medir el voltaje en el divisor de tensión entre la resistencia común y la resistencia digital que están conectadas a la salida de voltaje. En función de dicho valor medido, se enviará un valor a la resistencia para incrementar o decrementar dicho valor vía SPI. A continuación se volverá a medir el voltaje en el divisor para tomar la decisión de nuevamente modifica o mantener el valor de la resistencia digital. Este proceso se repetirá hasta que voltaje en el divisor sea proporcional al deseado. Se puede afirmar que es relativamente determinístico y puede existir algún tipo de variación que puede afectar a este proceso concurrente.

Pero existen otras tareas que el MCU debe realizar, como se mencionó anteriormente hay otra tarea para mostrar el valor del voltaje de salida medido (que es un valor proporcional al del divisor de tensión) en el display.

Por lo tanto el CPU del microcontrolador debe realizar dos tareas a la vez (aún no considero las otras) y el tiempo que toma realizar cada una podría ser mayor que si hiciera una sola tarea, ya que el CPU sólo puede completar una tarea concurrente a la vez.

Entonces lo que debe existir es algo que regule el acceso al CPU y las tareas se dividirán en pequeñas subtareas para que se turnen entre sí, en pocas palabras cada tarea realiza una porción de lo que debe hacer, para dar paso a otra tarea que también realice una porción de su código. El

proceso debe ser de tal manera que ante el usuario parecería que el MCU hace varias cosas al mismo tiempo, esto se denomina sincronización.

Suponiendo que existe una variación que afecte el proceso y una tarea debe tomar otro camino más largo para cumplir su objetivo, su **latencia** aumentará así como el aumento del **volumen de trabajo** (***Throughput***) del sistema y en el mejor de los casos tomará un mayor tiempo en cumplir su objetivo, este aumento de tiempo en la latencia se denomina ***jitter*** y en el peor de los casos se producirá un **bloqueo mutuo** (***deadlock***), en el cual una parte del sistema o todo permanece sin responder (colgado).

En nuestro ejemplo supongamos que la tarea que varía el voltaje a la salida de la fuente envía un valor a la resistencia digital para que actualice su valor, pero esta no responde y el CPU permanece en espera a la respuesta al comando enviado. El código escrito debe responder rápidamente hasta perturbación crítica y tomar una decisión.

Para completar el análisis del sistema ejemplo, las tareas que tiene serían las siguientes:

- Medición de la corriente de entrada y control del relé de desconexión del voltaje de salida (Tarea 1)
- Medición del voltaje de salida y control de la resistencia digital (Tarea 2)
- Lectura del potenciómetro para calibrar la máxima corriente de salida del sistema. (Tarea 3)
- Lectura del potenciómetro para calibrar el voltaje deseado a la salida. (Tarea 4)
- Administración del display (Tarea 5)

La primera tarea de la lista anterior, denominada medición de la corriente de entrada y control del relé de desconexión del voltaje de salida tendrá la **prioridad** más alta en el sistema.

Una tarea de alta prioridad puede poner en pausa a una de baja prioridad. En nuestro ejemplo, si la Tarea 2 va cambiando el valor de la resistencia digital y el voltaje de salida va aumentando cada vez más y más. La Tarea 1 mide la corriente del sistema y si determina que la corriente alcanzó el máximo de corriente establecido, pone en pausa a la Tarea 2 para que no siga aumentando el voltaje. Si dicha corriente sigue aumentando a pesar de estar en pausa la Tarea 2, activará al relé de salida para proteger al sistema.

La asignación de prioridades a las tareas y el control de las mismas mediante un planificador también se conoce como **algoritmo de exclusión mutua** (*Mutex implementation*)

En resumen una tarea concurrente tiene las siguientes propiedades:

Actores, que son las tareas, hilos, procesos, etc.

Recursos compartidos, que son el CPU, la memoria, los registros, etc.

Reglas de acceso son la sincronización, prioridades, etc.

Resultados determinísticos, tareas finalizadas, resultados correctos o esperados.

4.3 Posible problema con las tareas concurrentes.

4.3.1 Condición de Secuencia

La **Condición de Secuencia** es un problema típico cuando dos o más tareas acceden a un recurso compartido y la salida o estado de un proceso es dependiente de una secuencia de eventos que se ejecutan en orden arbitrario. Lo mejor es explicar con un ejemplo para comprender claramente que significan todos esos términos.

Supongamos que un vendedor ofrece productos desde una página web y a través de la misma, los vende.

Supongamos también que tiene un producto X del cual sólo dispone 5 unidades y en la página web en un mismo instante hay dos compradores, comprador A y comprador B, ambos que están viendo dicha la página interesados en el producto X. También asumamos que el vendedor no podrá tener más productos X dentro de las próximas semanas.

Los dos compradores ven que hay en existencias 5 unidades, el comprador A decide adquirir 4 unidades y el comprador B adquiere 2 unidades, ambos al mismo tiempo. Los dos compradores asumen que hay 5 unidades, pero el software que realiza la transacción, suponiendo que no está correctamente escrito, va a tener un balance de menos una unidad. Conjeturando que la primera transacción la hizo el comprador A, va estar tranquilo porque sus 4 productos han sido despachados, mientras el comprador B a pesar que compró 2 unidades, sólo podrán enviarle 1 unidad. Mientras quien administra el sistema va a observar un balance negativo en su stock de productos o un error en el sistema. Además de esto habrá un problema ya que tendrá que devolver parte de lo pagado al usuario además de explicarle que sólo se pudo enviar una unidad a pesar que en la página web se indicaba que había 5 unidades.

Un ejemplo clásico de condición de secuencia es el problema con temporizadores de mayor número de bits que el CPU en un MCU, por ejemplo en un microcontrolador de 16 bits podría existir un temporizador de 32 bits, eso implica que se utilizan dos variables de 16 bits para representar a la variable completa, supongamos que el temporizador tiene el valor de 0x0000FFFF. Figura 4.2.

IMAGEN NO DISPONIBLE EN LA VERSIÓN DEMO

Figura 4.2. Variable de 32 bits representada en dos variables de 16 bits.

Supongamos que el temporizador se incrementa a razón de cada ciclo de máquina y cuyo valor ha alcanzado el valor mencionado anteriormente.

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

Si se hubiese empezado leyendo el resultado bajo, no se soluciona el error, se obtendría el valor de 0x0001FFFF, cuando en realidad el valor deseado es 0x0000FFFF o a lo mucho un error de una unidad, es decir 0x00010000.

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

Si por alguna razón no se debe detener al temporizador para no generar errores en la medición de tiempo, existe una solución más compleja que implica más código para realizarla.

Esta solución consiste en almacenar primero la parte alta y luego la baja del temporizador en otras dos variables, luego verificar que la parte alta del temporizador no haya cambiado respeto a la variable que tiene su valor anterior. Si existiera algún cambio repetiría nuevamente el ciclo de lectura, este método se llama de ***revisión redundante***. Figura 4.3.

IMAGEN NO DISPONIBLE EN LA VERSIÓN DEMO

Figura 4.3. Manera de obtener el valor del temporizador sin detenerlo para evitar la condición de secuencia.

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

4.3.2 Métodos de sincronización

En el problema anterior respecto al temporizador, se indicó que detenerlo es una solución al problema, y puede considerarse como una forma de **Exclusión Mutua** abreviada como ***mutex.***

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

4.3.3 Exclusión Mutua

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

4.3.4 Semáforos

Los **Semáforos** son variables o un tipo de dato abstracto el cual es usado para controlar el acceso mediante múltiples procesos a un recurso común en un sistema concurrente. Estas variables pueden ser banderas o variables condicionales.

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

4.3.5 Inanición

La **Inanición** o ***Starvation*** es cuando un proceso se le rechaza siempre el acceso a un recurso compartido. Sin este recurso, la tarea a ejecutar no puede ser nunca finalizada.

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

4.3.6 Punto Muerto

Un **Punto Muerto** o **Deadlock** es cuando una tarea no puede liberar el bloqueo de un proceso o dos o más tareas compiten para alcanzar un proceso y se bloquean entre sí mutuamente.

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

4.3.7 Escalabilidad

La escalabilidad también significa que el sistema se adapta o reacciona ante un imprevisto sin perder la calidad.

La **Escalabilidad Horizontal** o ***Scale Out/In*** es cuando las tareas poseen un conjunto amplio de reglas ya que existe gran cantidad de recursos disponibles como memoria como una computadora con su sistema operativo. Las tareas pueden tener muchas definiciones y conceptos sin afectar al rendimiento del sistema.

La **Escalabilidad Vertical** o ***Scale Up/Down*** es cuando la tarea es refinada y bien definida para un propósito estricto para sólo una aplicación, la cual es muy especializada y realiza un trabajo cerrado o único. Es útil en sistemas con pocos recursos como lo son los microcontroladores.

4.4 Recursos que dispone un microcontrolador

4.4.1 Interrupciones

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

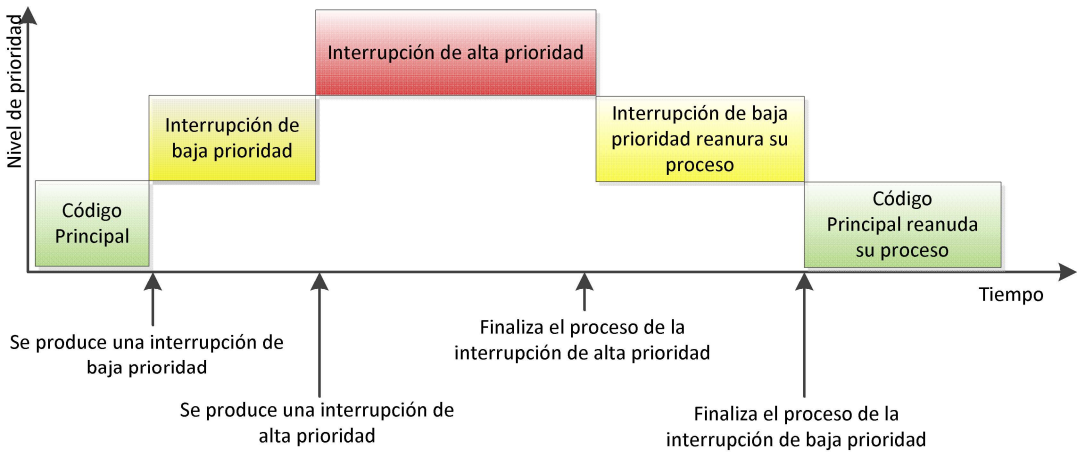


Figura 4.7. Una interrupción que sea de alta prioridad respecto a otra puede interrumpirla

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

La interrupción 7 por defecto utiliza el **Conjunto de Registros Sombra** o ***Shadow Register Set*** (**SRS**) para almacenar y recuperar el contexto. Utilizar dichos registros implica que la interrupción 7 tiene una latencia muy baja respecto a las otras. El resto de interrupciones recurren al **Interruptor de Software de Contexto** o ***Software Context Switch*** para salvar los registros importantes del CPU. Para cambiar la asignación de los SRS a una prioridad se recurre a la palabra de configuración **DEVCFG3**. Figura 4.8

IMAGEN NO DISPONIBLE EN LA VERSIÓN DEMO

Figura 4.8. Asignación de los registros sombra a la interrupción de prioridad 7

Se debe indicar que la familia de microcontroladores que se utiliza para los primeros proyectos no posee SRS. (PIC32MX1XX/2XX).

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

IMAGEN NO DISPONIBLE EN LA VERSIÓN DEMO

Figura 4.9. Fuentes de interrupción de la familia del microcontrolador PIC32MX1XX/2XX

Los posibles problemas que pueden suceder con las interrupciones si no se manejan correctamente son la condición de secuencia, latencia excesiva, variación de la latencia (*jitter*), bloqueo de interrupciones, etc.

En nuestro sistema ejemplo podemos mejorar su eficiencia utilizando una interrupción para el conversor ADC. En la Figura 4.10 se indica un pseudocódigo del sistema analizado hasta el momento.

IMAGEN NO DISPONIBLE EN LA VERSIÓN DEMO

Figura 4.10. Pseudocódigo del sistema analizado.

Suponiendo que el contador de programa ha finalizado la tarea 1 y existe una sobre corriente en el sistema, se deberá esperar hasta que las tareas desde la número 2 hasta la 5 finalicen, luego hay que esperar a que la tarea 0 adquiera los valores del conversor ADC y finalmente la tarea 1 procesará la información. (Obviamente este sistema es sólo un ejemplo, también deberá existir otros sistemas de auxilio y no sólo depender que el MCU sea el responsable de proteger al sistema).

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

IMAGEN NO DISPONIBLE EN LA VERSIÓN DEMO

Figura 4.11. Pseudocódigo para la rutina de interrupción del conversor ADC en el sistema analizado.

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

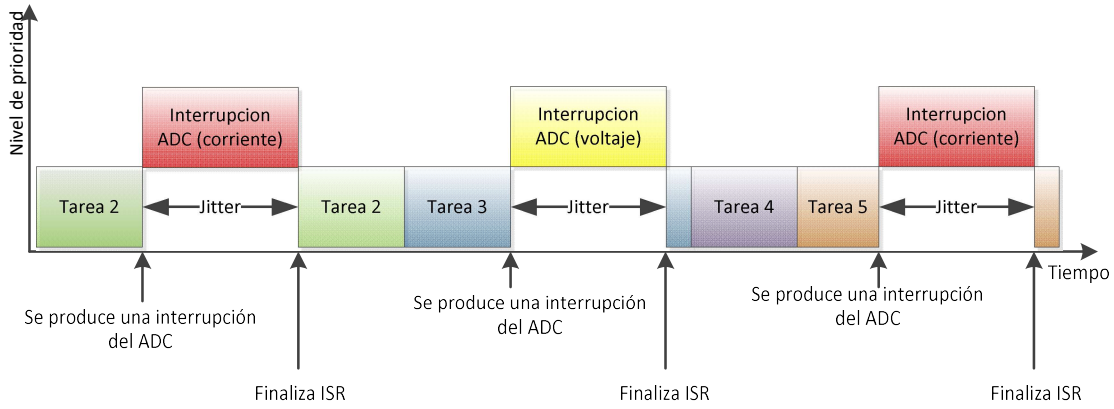


Figura 4.13. La ISR introduce *jitter* en el código principal que está en la función *main.*

Para disminuir el aumento de la latencia (*jitter*) cada tarea debe realizar una parte pequeña de sí misma (más adelante se explica caramente a que se refiere), los procesos dentro de ISR deben ser lo más cortos posibles, dentro de las ISR se debe evitar esperar a algún proceso, no manipular variables globales entre el código principal y la ISR ya que pueden ser alterada en la interrupción y al retornar al código principal no es el valor que se estaba procesando (Condición de Secuencia)

4.4.2 Paralelismo

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

4.5 Planificador de tareas cooperativas

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

4.5.1 Procesos del Planificador de Tareas.

Como se indicó anteriormente hay dos tipos de procesamiento, cooperativo y de tarea apropiativa o procesamiento preventivo. La mezcla de los dos se denomina **procesamiento híbrido.**

4.5.1.1 Procesamiento Cooperativo.

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

4.5.1.2 Procesamiento de Tarea Apropiativa

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

4.5.1.3 Procesamiento Híbrido

En un microcontrolador se logra combinar los dos métodos de procesamiento, procesamiento cooperativo en el código principal y procesamiento de tarea apropiativa mediante las interrupciones de los periféricos que posee el MCU.

4.6 Conceptos de un RTOS

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

4.6 Tipos de Planificadores

4.6.1 Planificador de Tareas Secuencial

Es la forma más común de escribir varias tareas en el hilo principal del MCU. Como su nombre lo indica, secuencial implica que se ejecuta una tarea para pasar a otra, y luego a otra, hasta concluir con todas y volver nuevamente a la tarea inicial, todo esto dentro de un lazo infinito. Figura 4.15

IMAGEN NO DISPONIBLE EN LA VERSIÓN DEMO

Figura 4.15. Planificador de Tareas Secuencial

Esta forma de planificador es simple y permite seguir o analizarlo de manera lineal. Tiene la desventaja que una Tarea puede consumir mucho tiempo para realizar su proceso lo que se traduce en una disminución del rendimiento del sistema, generalmente esto sucede porque no se utiliza otras técnicas para mejorar su rendimiento. Además la latencia puede variar enormemente en esta clase de programación.

4.6.2 Planificador de Tareas Round Robin (RR)

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

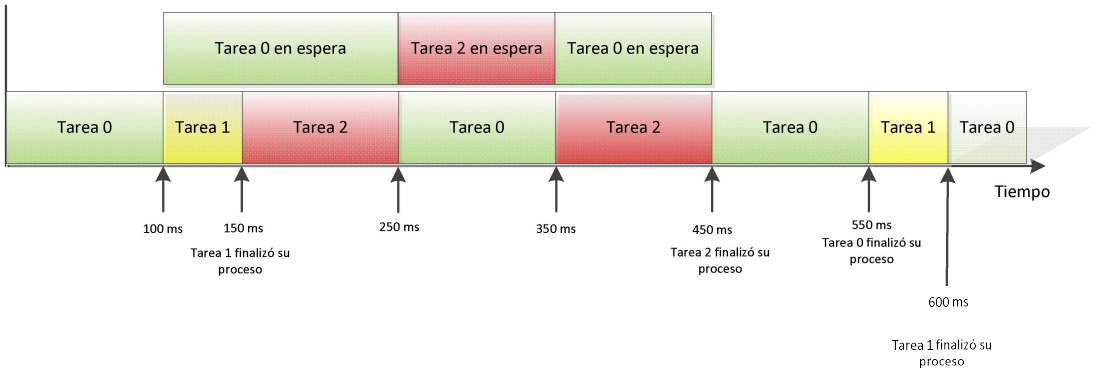


Figura 4.16. Tareas administradas por un planificador Round Robin

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

4.6.3 Planificador de Tareas de Prioridad Básica.

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

4.6.4 Planificador de Tareas de Prioridad Avanzada.

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

4.6.5 Planificador de Tareas en Estados

Este método se utiliza cuando las tareas a realizarse son muy largas de ejecutar. Las tareas son divididas en etapas o estados.

Es lo que se ha mencionado previamente al indicar que el CPU realiza pequeñas porciones de una tarea y ante los ojos del usuario parece que el CPU realiza varios procesos a la vez. Este es el método que se utilizará en los proyectos de este libro.

La Figura 4.17 muestra una imagen conceptual de este método de programación.

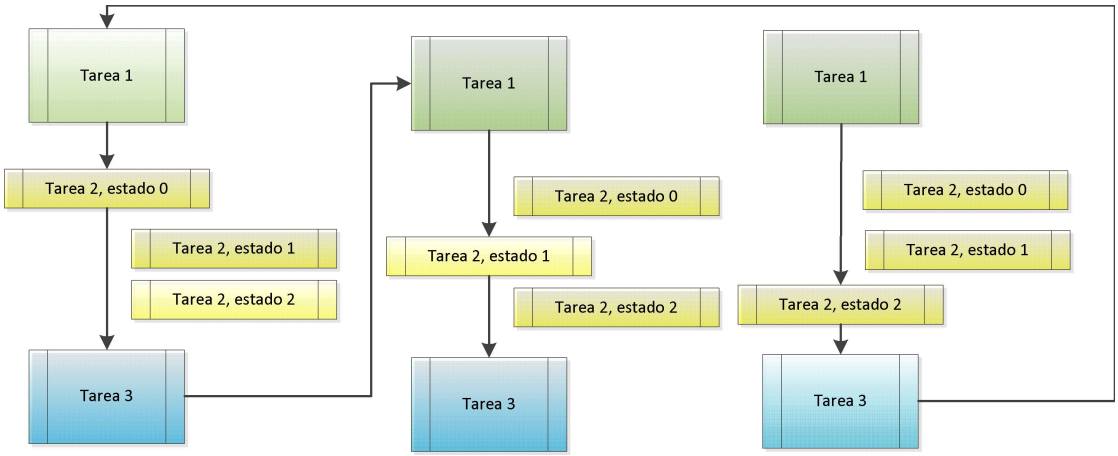


Figura 4.17. La Tarea 2 ha sido dividida en 3 estados.

Las tareas que son divididas en porciones son más fáciles de analizar, administrar y corregir. Evita que una tarea se apodere del control total del CPU. También el volumen de trabajo se reduce enormemente y mediante este método se crea **estados de máquina** para tareas que requieren un retardo en un proceso específico.

Los inconvenientes de utilizar está técnica es que el rendimiento de una tarea se aumenta al dividirla en varias partes, se requiere una variable dedicada por cada tarea para almacenar el estado.

4.6.6 Planificador de Tareas Programado (*Scheduled*)

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

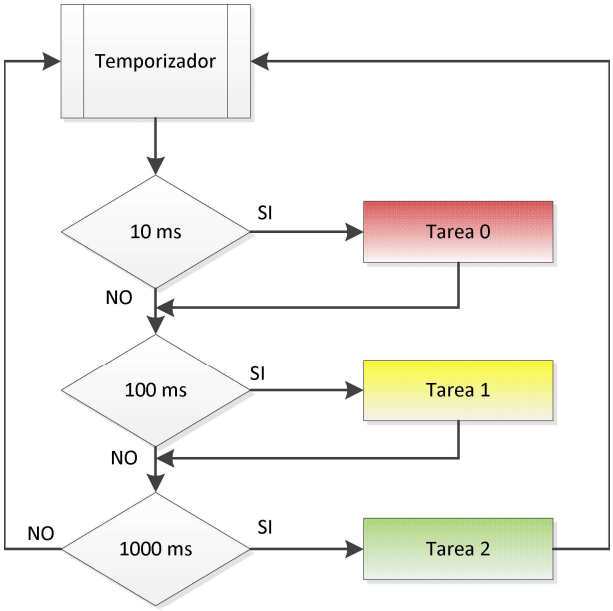


Figura 4.18. Tareas ejecutadas por un planificador programado.

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

4.6.6 Planificador de Tareas en Fila

TEXTO NO DISPONIBLE EN LA VERSIÓN DEMO

Capítulo 5

Encendido y apagado de un led de manera periódica

5.1 Estructura de los proyectos que se realizarán en este libro.

Como se mencionó en el Capítulo 4, el método para escribir todos los proyectos en este libro será utilizando un planificador de tareas por estados.

Cada tarea a la vez se podrá subdividir en estados como se indica en la Figura 4.17 del Capítulo 4.

Las tareas de ahora en adelante se las denominará aplicaciones y tendrán de sufijo la palabra **App**. Cada tarea estará involucrada en el control de un periférico. Para lo cual luego del sufijo App seguirá el nombre de periférico asociado:

App[*Nombre del Periférico*]

En la Figura 5.1 se muestra que el código principal maneja tres aplicaciones, AppLed para controlar el encendido y apagado de un led, AppUart1 que maneja todo lo relacionado con el Uart1 del MCU y AppLCD que debe contener todo lo relacionado para el control de un módulo de visualización.

```
1  void main (void){
2      ...
3      while(1){
4          AppLed();
5          AppUart1();
6          AppLCD();
7      }
8  }
9
```

Figura 5.1. Lazo principal administrando tres aplicaciones o tareas.

Antes de ejecutarse el lazo principal o superlazo, se debe configurar correctamente a los periféricos y al CPU. El código antes de ingresar al lazo principal es similar al de la Figura 5.2.

```
void main (void){
    /** Configuraciones de terminales del MCU **/
    ConfigurarIOs();
    ConfigurarPPS();

    /** Configuración de periféricos **/
    ConfigurarPerifericо0();
    ConfigurarPerifericо1();
    ConfigurarPerifericо2();
    ...
    ConfigurarPerifericoN();

    /** Inicializar Aplicaciones **/
    InicializarApp0();
    InicializarApp1();
    InicializarApp2();
    ...
    InicializarAppN();

    /** Inicializar temporizador del Sistema **/
    InicializarTemporizadorSistema();

    /** Habilitar Interrupciones **/
    HabilitarInterrupciones();

    while(1){
        App0();
        App1();
        App2();
        ...
        AppN();
    }
}
```

Figura 5.2. Estructura de la función *main* para la mayoría de los proyectos presentados en este libro.

En la Figura 5.2 se puede apreciar varias sub-funciones en *main*, ConfigurarIOs configurará los terminales del MCU como entradas y salidas según sea lo necesario, así como la activación de otros parámetros como resistencias de *pull-up*, *pull-down*, salidas de drenador abiertos, etc.

La función ConfigurarPPS asignará los terminales de entrada y salida de un periférico a los correspondientes terminales externos del MCU.

Las funciones ConfigurarPeriferico establecerán los valores correctos para el funcionamiento de un periférico, por ejemplo para un módulo UART se asignará la velocidad de comunicación, si la comunicación es de 8 o 9 bits, el número de bits de parada, etc. **Configurar un periférico no necesariamente es activarlo**, generalmente la mayoría de periféricos poseen un bit en un registro que al ponerlo en 1, activan al periférico para que empiece a funcionar. El programador deberá ser intuitivo para determinar si dentro de estas funciones se activará al periférico o si es necesario que sea más adelante.

Las funciones HabilitarApp generalmente son utilizadas para poner el valor inicial de la variable de estado de cada *app*, también aquí muchas veces se puede inicializar algún otro tipo de variables que necesitan las tareas.

InicializarTemporizadorSistema activa a un temporizador que es útil para generar bases de tiempo definidas, más adelante y con los ejemplos se entenderá mucho mejor.

La función HabilitarInterrupciones como su nombre lo indica habilita las interrupciones de los periféricos (si se las ha configurado anteriormente) para que interrumpan al CPU para atender un evento.

5.2 Temporizador de un Sistema con Microcontrolador.

Como se indicó anteriormente se necesita un temporizador que nos permita generar bases de tiempo definidas.

Para el objetivo de los dos proyectos de este capítulo, se necesita una base de tiempo para encender y apagar el led del circuito de la Figura 3.1 del Capítulo 3.

El primer proyecto encenderá y apagará al LED del terminal RB15 a razón de 500 ms. Figura 5.3

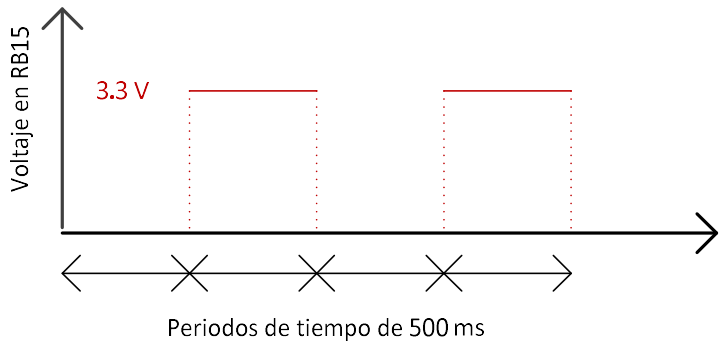


Figura 5.3. Voltaje de onda cuadrada de 500 ms de duración en el terminal RB15 del microcontrolador.

Las bases de tiempo se las generará con la ayuda de un temporizador, la utilización de ese temporizador debe ser de tal manera que pueda usar cualquier otra tarea o app. La Figura 5.4 muestra un diagrama de flujo de la manera incorrecta de como suelen utilizar un temporizador para generar retardos o *delays*.

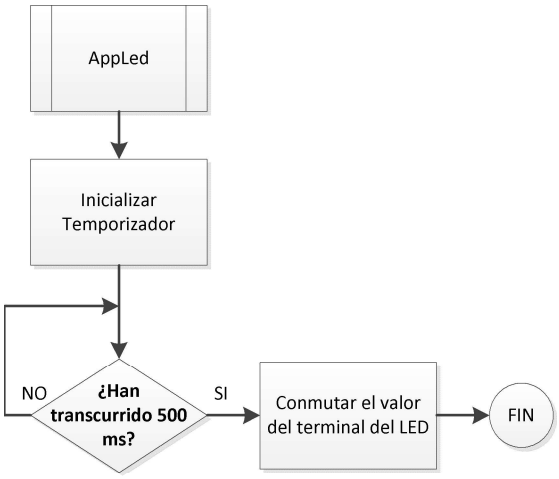


Figura 5.4. Manera incorrecta de utilizar un temporizador para generar retardos

En el código de la Figura 5.4 se ve claramente que el CPU permanece en un lazo de 500 ms en el cual pueden suceder varios eventos o realizar una tarea útil (al menos que encender y apagar un led sea el único objetivo del MCU).

En la Figura 5.5 se muestra el diagrama de flujo de una manera correcta de utilizar un temporizador a generar un retardo.

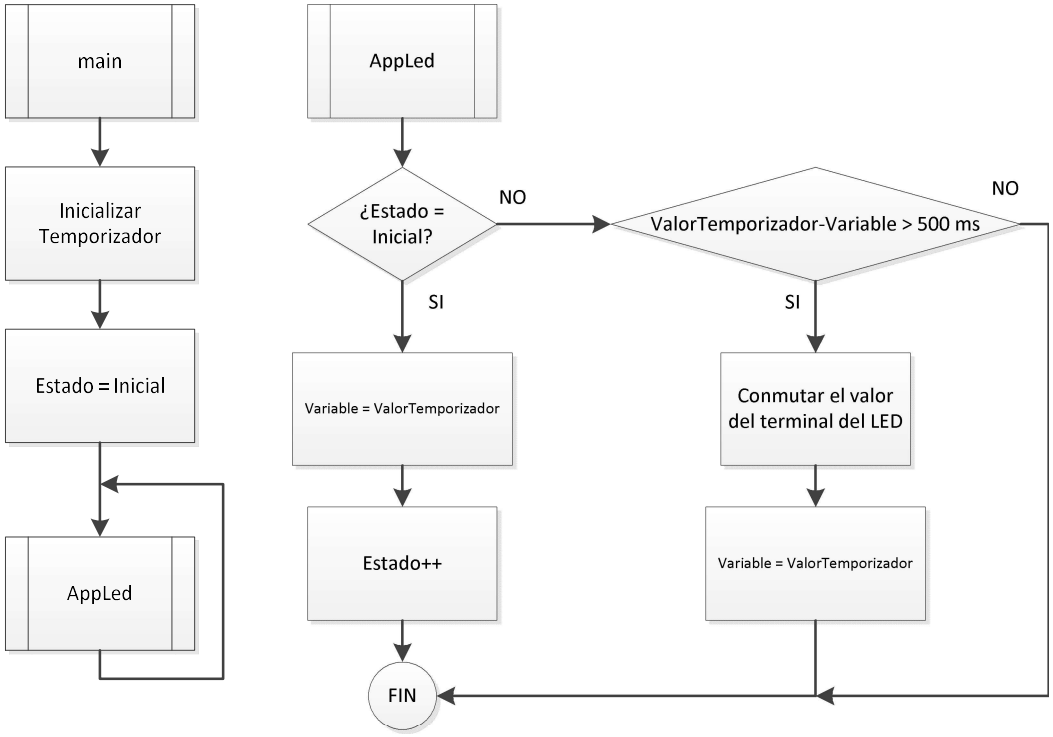


Figura 5.5. Temporizador utilizado para generar retardos de tal manera que su proceso no se apropie completamente del CPU.

En el diagrama de flujo de la Figura 5.5 en la función *main* se puede apreciar que el temporizador se inicia y permanece siempre activado contando ciclos de máquina. También el estado de la *AppLed* es puesto en un valor inicial (generalmente el valor inicial del estado de una *app* es cero).

Luego dentro del lazo principal se ejecuta la función *AppLed.* Dentro de dicha función se verifica el estado de la *app* que tiene un valor inicial y se

utiliza Variable para capturar el valor que el temporizador tenga en ese momento, a continuación se incrementa para que la *app* siempre realice la siguiente acción.

Se calcula la diferencia del valor del temporizador y el valor obtenido en el estado inicial, si dicha diferencia produce un valor que corresponda a un poco más del tiempo deseado, se procede a realizar la acción de conmutar el estado del led y nuevamente se almacena en Variable el valor del temporizador para nuevamente ejecutar un retardo.

Aquí debo explicar cómo funciona la diferencia de los dos valores. Supongamos que el temporizador es de 16 bits y con el valor de 0x3E80 produce 500 ms. Es decir que si el temporizador empieza desde un valor de 0x0000, al alcanzar 0x3E80 han transcurrido 500 ms. Suponiendo que en la *app* en el estado inicial, Variable por coincidencia captura el valor del temporizador con un valor igual a 0, en la siguiente etapa de la tarea realiza la diferencia y la condición mayor que:

$$(ValorTemporizador - Variable) > 0x3E80$$

Cuando el temporizador haya alcanzado un valor de 0x3E81 o superior la condición es verdadera y se produce la acción necesaria (conmutar el valor del terminal del led).

Pero el valor capturado para realizar la diferencia es poco probable que sea igual a cero y aún es más curioso analizar la diferencia cuando el valor del temporizador es menor que Variable.

Por ejemplo supongamos que el valor capturado en Variable es 0xFFF0 que es un valor muy próximo al desborde o reinicio del temporizador ¿Qué sucede cuando el temporizador es cero o menor de 0x3E80?

Supongamos que el temporizador es 0x0000, entonces la diferencia sería:

0x0000 – 0xFFF0

El resultado de dicha diferencia sería 0x0010 **si seleccionamos trabajar con variables sin signo**, la diferencia entre variables sin signo (***unsigned***) siempre nos da el valor absoluto.

Supongamos que el temporizador alcanza un valor de 0x3E71, entonces la diferencia sería:

0x3E71 – 0xFFF0

El resultado sería 0x3E81 que cumple con la condición para conmutar el terminal que está asignado el led.

Por lo tanto no importa qué valor se capture del temporizador, lo importante es que la diferencia sea la correcta.

La condición es mayor que, debido a que el tiempo o retardo necesario no es exacto debido a que el mismo hecho de que el CPU realiza los cálculos agrega un error (la diferencia, la condición, etc.). Además suponiendo que existieran otras tareas a parte de AppLed también influyen en el valor calculado.

Recuerde que las tareas trabajando en modo de cooperación son útiles si son de prioridad media, baja o tareas que necesitan permanecer en pausa o reposo. Por lo tanto si el tiempo que el led permanece encendido (o apagado) puede ser un poco mayor a 500 ms, entonces dicho error no afecta al sistema y el retardo puede ser un poco más grande del deseado.

Si realmente se necesita precisión en la generación de retardos, se debería recurrir a interrupciones y eso se verá en los capítulos correspondientes a ese tema.

También se deberá calcular cuál es el valor mínimo que puede generar el temporizador, generalmente lo mínimo puede ser 1 ms y lo máximo se puede alcanzar cualquier valor con variables auxiliares.

5.3 Estructura de todos los proyectos en MPLAB X

Todos los proyectos que se muestran en este libro tendrán una distribución de sus archivos de tal manera que estén ordenados. Por supuesto que el usuario puede escoger la mejor manera que crea conveniente para crear dicha distribución.

La distribución mencionada está en el panel de archivos (*File Pane*), en la pestaña proyectos. Figura 5.6.

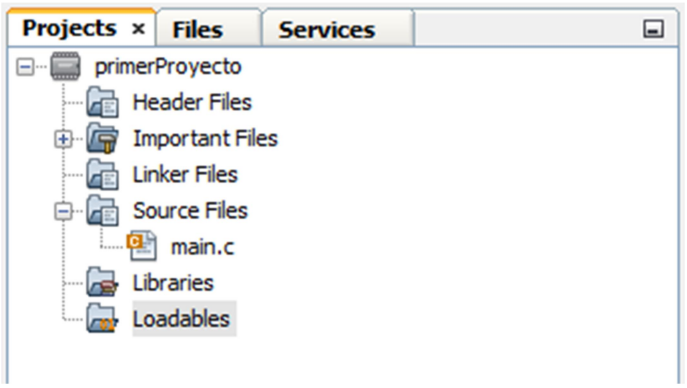


Figura 5.6. Panel de archivos de MPLAB® X

Dentro de la carpeta *Source Files* de la pestaña *Projects* del panel de archivos están todos los archivos de extensión .c que se utilicen o se necesiten.

Cada *app* tendrá su propio archivo .c que a la vez estarán dentro de una carpeta lógica. Esto también se realizará para los periféricos que también tienen sus propios archivos .c y su carpeta contenedora. La Figura 5.7

muestra un ejemplo de cómo los archivos .c del proyecto están organizados en el panel de archivos.

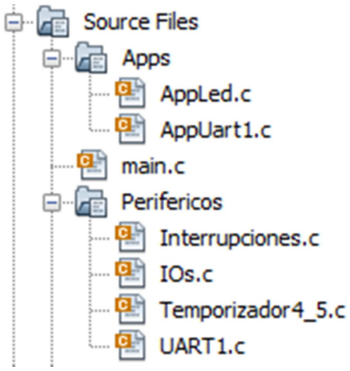


Figura 5.7. Ejemplo de distribución de los archivos .c en los proyectos de este libro.

De igual manera los archivos cabecera o .h estarán distribuidos de la misma manera. Figura 5.8.

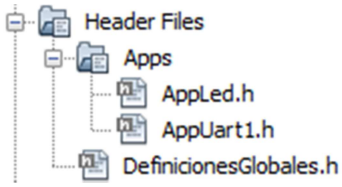


Figura 5.8. Distribución de los archivos .h

Más adelante se explicará cómo crear las carpetas lógicas y colocar sus archivos correspondientes.

5.4 Creación del proyecto de encendido y apagado de un led de manera periódica.

Similar al primer proyecto que se realizó en el capítulo 3, creamos un proyecto cuyo nombre será LedParpadeo, Figura 5.9.

Cuando el proyecto esté listo, creamos el archivo *main*, ponemos los mismos bits de configuración del primer proyecto y compilamos para verificar que el proyecto esté correctamente creado (yo utilizaré compilación tipo *s*). Figura 5.10

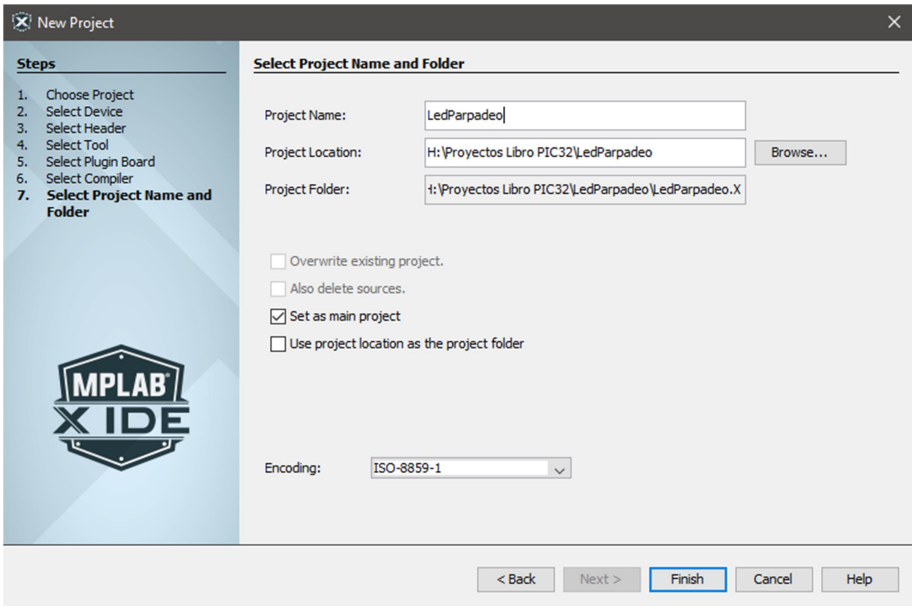


Figura 5.9. Creación del segundo proyecto de este libro.

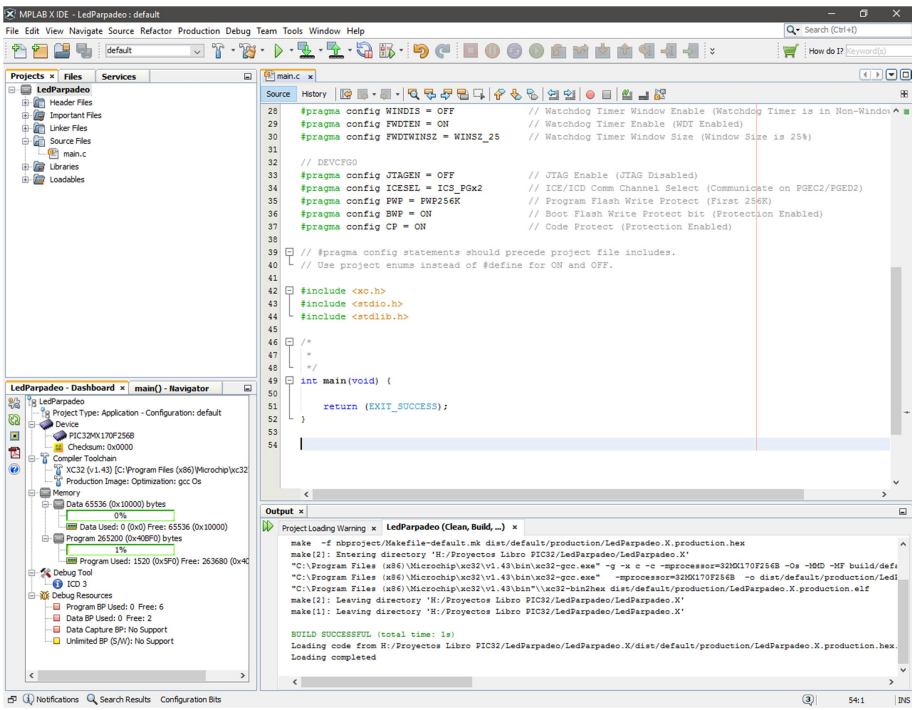


Figura 5.10 Segundo proyecto listo.

5.5 Creando la función IOs

En la función *main* se escribe la función ConfigurarIOs como se indica en la Figura 5.11.

```
49  int main(void) {
        ConfigurarIOs();
51      return (EXIT_SUCCESS);
52  }
```

Figura 5.11 Función ConfigurarIOs dentro de *main*

Debido a que aún no existe dicha función, MPLAB® X nos indica con una línea roja subrayada que existe un problema con dicha frase.

El siguiente paso es crear el archivo que contiene dicha función, ConfigurarIOs es parte de la configuración de los terminales del MCU, las funciones relacionadas con los terminales las escribo en un archivo denominado IOs.c.

Los terminales los considero como periféricos, por lo que se crea una carpeta lógica con el nombre Periféricos en el panel de archivos del proyecto. Para lo cual se da clic derecho sobre la carpeta *Source Files* y se selecciona *New Logical Folder*, Figura 5.12

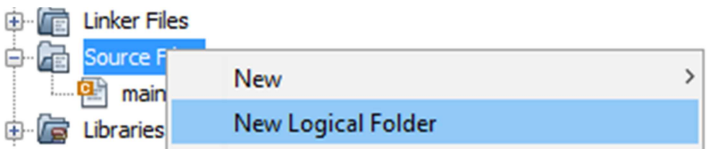


Figura 5.12. Creando una nueva carpeta lógica dentro de *Source Files*

Cuando se da clic sobre *New Logical Folder*, una nueva carpeta lógica aparece con nombre *New Folder 1*, Figura 5.13

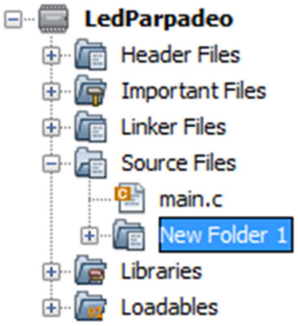


Figura 5.13. Nueva carpeta lógica creada dentro de *Source Files*.

A continuación procedemos a cambiar el nombre de la carpeta nueva a Perifericos, para lo cual damos clic derecho sobre la nueva carpeta y escogemos *Rename…* Figura 5.14.

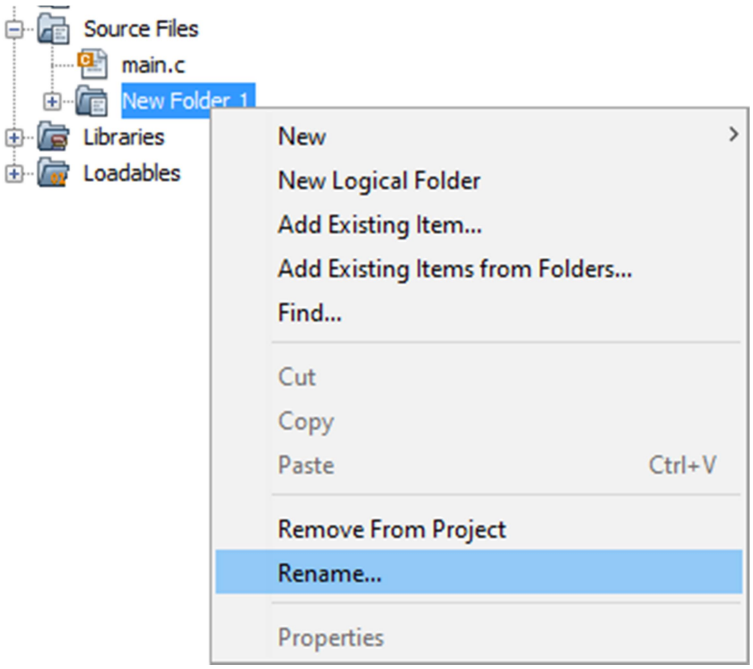


Figura 5.14. Renombrar a la carpeta lógica.

Al dar clic en *Rename…* aparece una ventana que nos permite cambiar el nombre de la carpeta lógica, la cambiamos a Perifericos (funciona con tilde, pero para evitar un posible error no se debería utilizar caracteres especiales). Figura 5.15

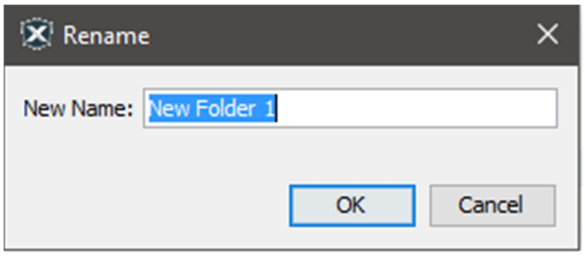


Figura 5.15 Ventana que permite cambiar el nombre de la nueva carpeta lógica creada.

Una vez cambiado el nombre de la carpeta, se da clic derecho sobre la misma y seleccionamos *New*, *Source File* Figura 5.16.

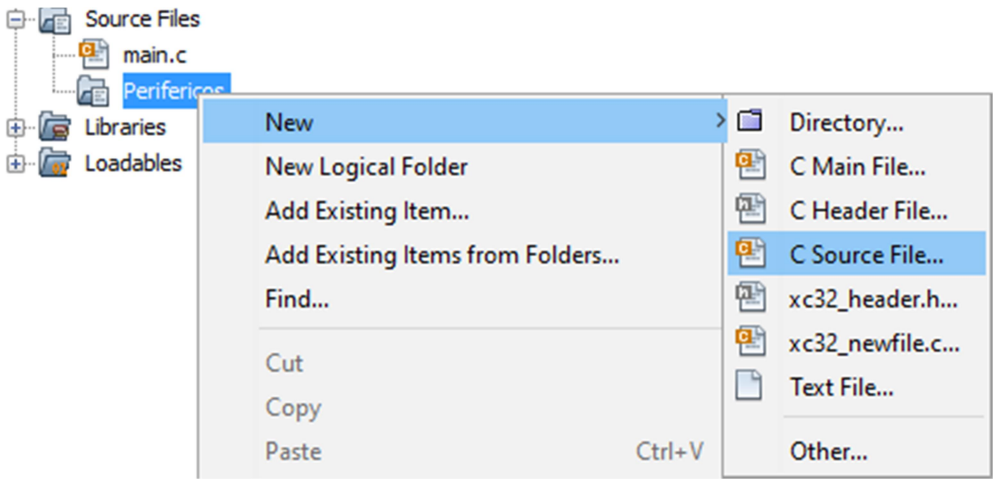


Figura 5.16. Manera de agregar un archivo a una carpeta lógica.

Dando clic sobre *C Source File* aparece una nueva ventana que nos permite ingresar el nombre del nuevo archivo .c, el cual se llamará IOs. Figura 5.17.

Damos clic en el botón *Finish* de la Figura 5.17 y el proyecto ahora tiene un nuevo archivo denominado IOs.c. Figura 5.18.

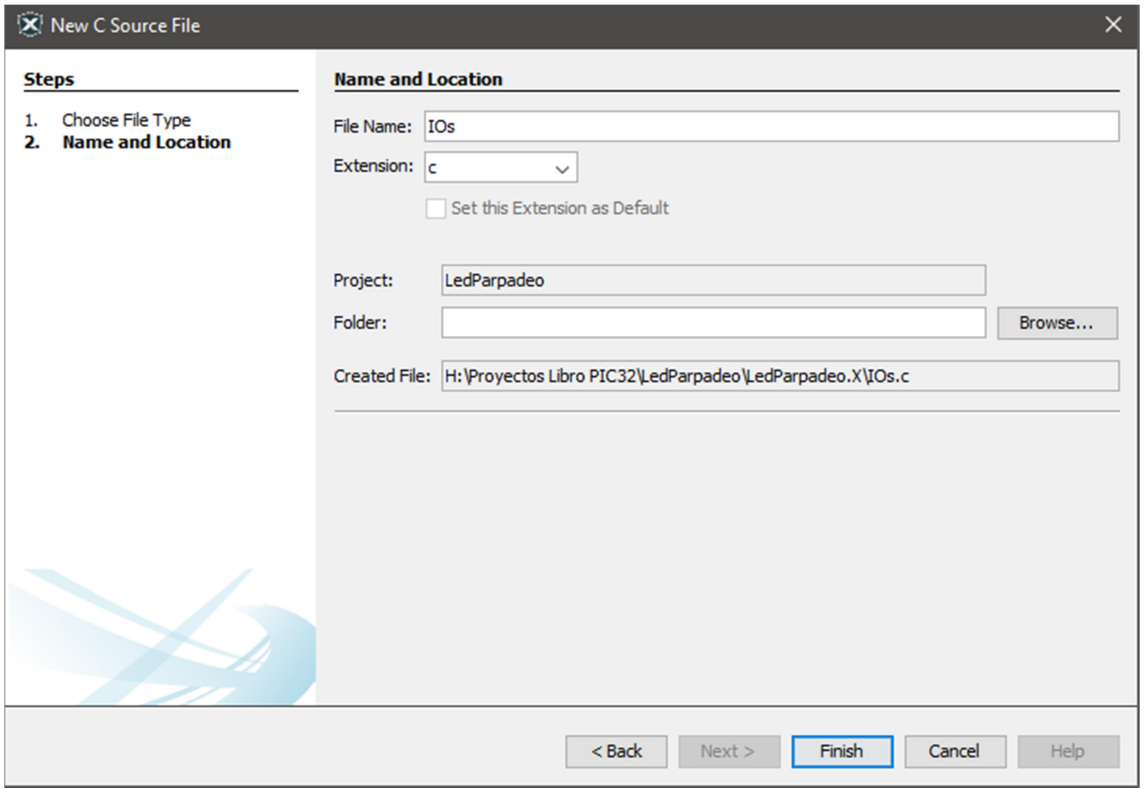


Figura 5.17. Ventana que permite crear un nuevo archivo de tipo c.

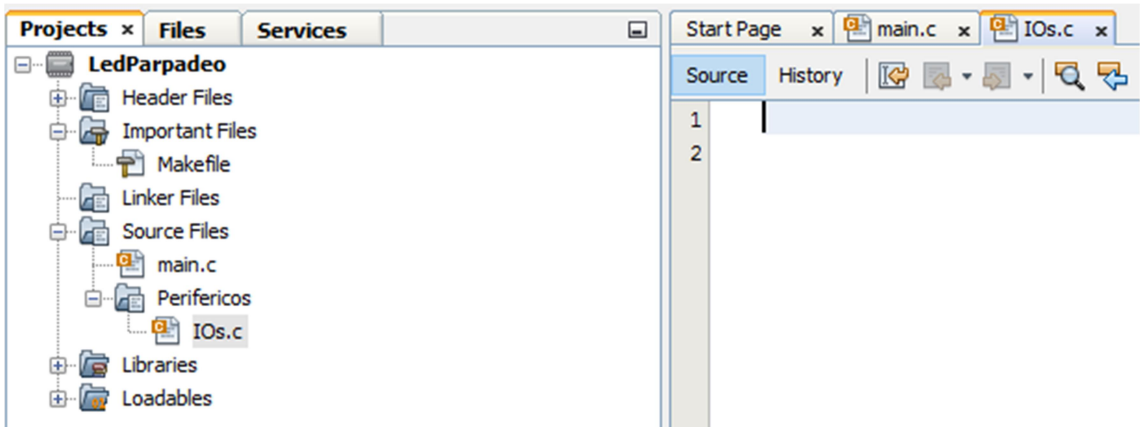


Figura 5.18. Proyecto con nuevo archivo .c denominado IOs.

En el archivo IOs.c creamos la función ConfigurarIOs como se muestra en la Figura 5.19.

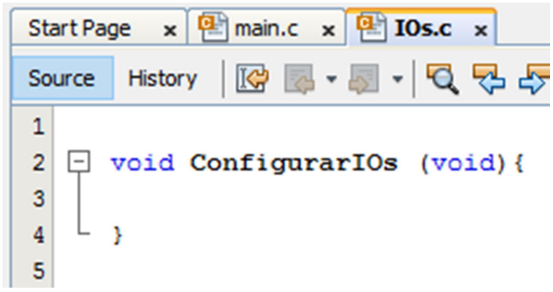


Figura 5.19. Función ConfigurarIOs dentro en el archivo IOs.

En este punto guardamos y compilamos al proyecto para conocer si todo está bien. Aquí veremos que el error que se indicaba anteriormente ha desaparecido.

Ahora dentro de la función ConfigurarIOs procedemos a escribir lo necesario para la correcta configuración de los terminales del MCU que es exactamente lo mismo del primer proyecto. Si copiamos y pegamos dichas configuraciones, veremos que MPLAB® X nos muestra varios errores. Figura 5.20.

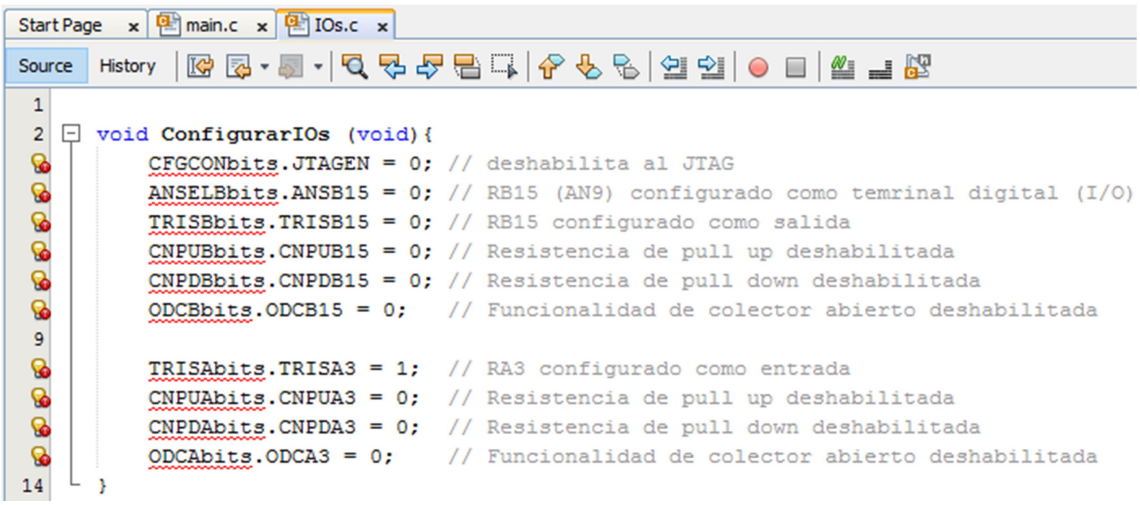


Figura 5.20. Configuraciones para los terminales del MCU dentro de la función ConfigurarIOs, se ve claramente que existe un problema.

Los errores indicados son porque para el archivo no se agregado la cabecera de los registros subrayados en rojo. La solución es incluir el archivo cabecera xc.h. Figura 5.21.

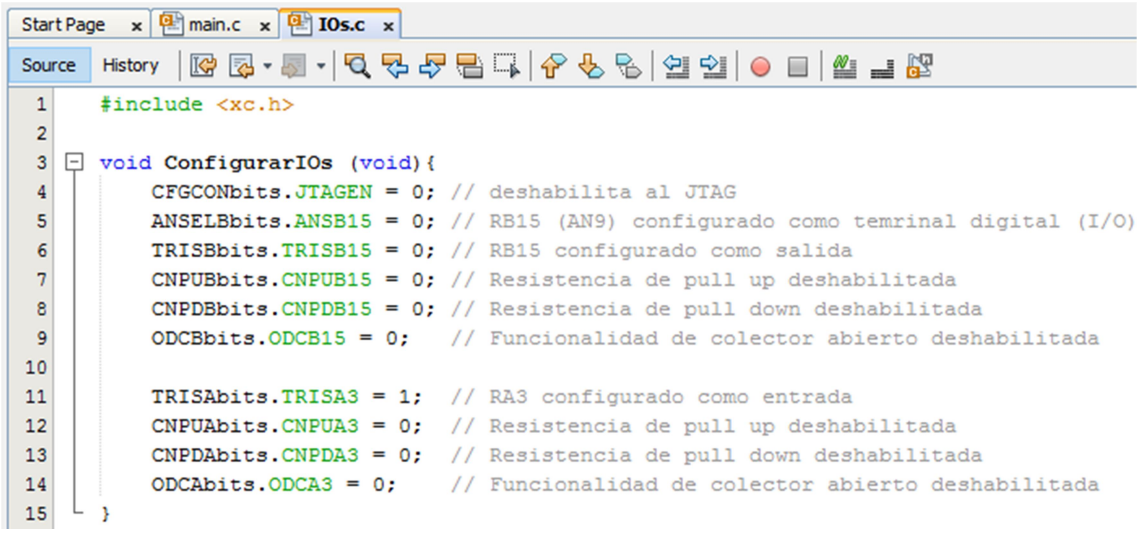


Figura 5.21. Con la inclusión del archivo xc.h los registros del MCU son reconocidos en el archivo .c.

Nuevamente, cada vez que se realice un cambio es recomendable compilar el proyecto para determinar si hay algún error.

Antes de continuar, recuerde que es recomendable declarar los prototipos de las funciones de todos los archivos .c para evitar los problemas de compilación que muchas veces el compilador desconoce a las funciones. Figura 5.22

El siguiente paso sería crear la función ConfigurarPPS (que también estará dentro de IOs.c) y *ConfigurarPerifericoN, 1, … N*, pero al momento el único periférico que se utiliza son los terminales del MCU, así que cuando sea necesario se crearán dichas funciones.

```c
#include <xc.h>
#include <stdio.h>
#include <stdlib.h>

void ConfigurarIOs(void); /* prototipo de función */
int main(void); /* prototipo de función */

int main(void) {
    ConfigurarIOs();
    return (EXIT_SUCCESS);
}
```

```c
#include <xc.h>

void ConfigurarIOs(void); /* Prototipo de función*/

void ConfigurarIOs (void){
    CFGCONbits.JTAGEN = 0; // deshabilita al JTAG
    ANSELBbits.ANSB15 = 0; // RB15 (AN9) configurado como temrinal digital (I/O)
    TRISBbits.TRISB15 = 0; // RB15 configurado como salida
    CNPUBbits.CNPUB15 = 0; // Resistencia de pull up deshabilitada
    CNPDBbits.CNPDB15 = 0; // Resistencia de pull down deshabilitada
    ODCBbits.ODCB15 = 0;   // Funcionalidad de colector abierto deshabilitada

    TRISAbits.TRISA3 = 1;  // RA3 configurado como entrada
    CNPUAbits.CNPUA3 = 0;  // Resistencia de pull up deshabilitada
    CNPDAbits.CNPDA3 = 0;  // Resistencia de pull down deshabilitada
    ODCAbits.ODCA3 = 0;    // Funcionalidad de colector abierto deshabilitada
}
```

Figura 5.22. Todos los archivos .c deben poseer los prototipos de las funciones que utilicen para que el compilador las reconozca correctamente.

Lo siguiente es inicializar la *app* denominada AppLed para lo cual creamos la carpeta lógica denominada Apps y dentro de ella creamos el archivo AppLed.c. Figura 5.23.

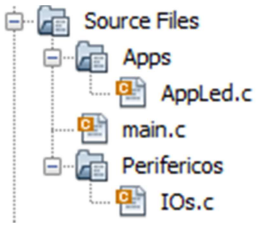


Figura 5.23. Carpeta lógica Apps conteniendo al archivo AppLed.c

Dentro del archivo AppLed creamos la función IniciarAppLed y la escribimos en la función *main*. Figura 5.24.

```
46    void ConfigurarIOs(void);
47    void InicializarAppLed(void);
48    int main (void);
49
50    int main(void) {
51        ConfigurarIOs();
52        InicializarAppLed();
53        return (EXIT_SUCCESS);
54    }
```

```
1
2     void InicializarAppLed(void);
3
4     void InicializarAppLed(void){
5
6     }
7
```

Figura 5.24. Función InicializarAppLed en archivo AppLed.c y utilizada en la función *main*.

El siguiente paso es crear los estados de la *app* como enumeradores y sus variables las cuales serán una estructura de datos. Esto se declarará en un archivo cabecera .h que corresponde con el nombre del archivo AppLed.c.

En la carpeta lógica denominada *Header Files* creamos una subcarpeta lógica denominada Apps y dando clic derecho sobre ella seleccionamos *New*, *C Header File*. Figura 5.26.

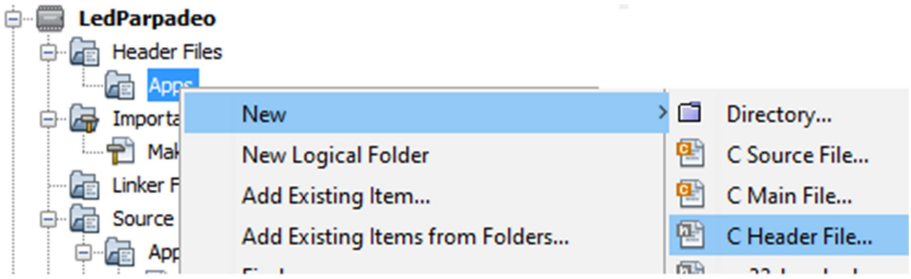


Figura 5.26. Agregando un archivo cabecera en Apps de la carpeta Header Files.

Aparece una ventana que nos permite ingresar el nombre del archivo el cual será AppLed y presionamos el botón Finish. Figura 5.27

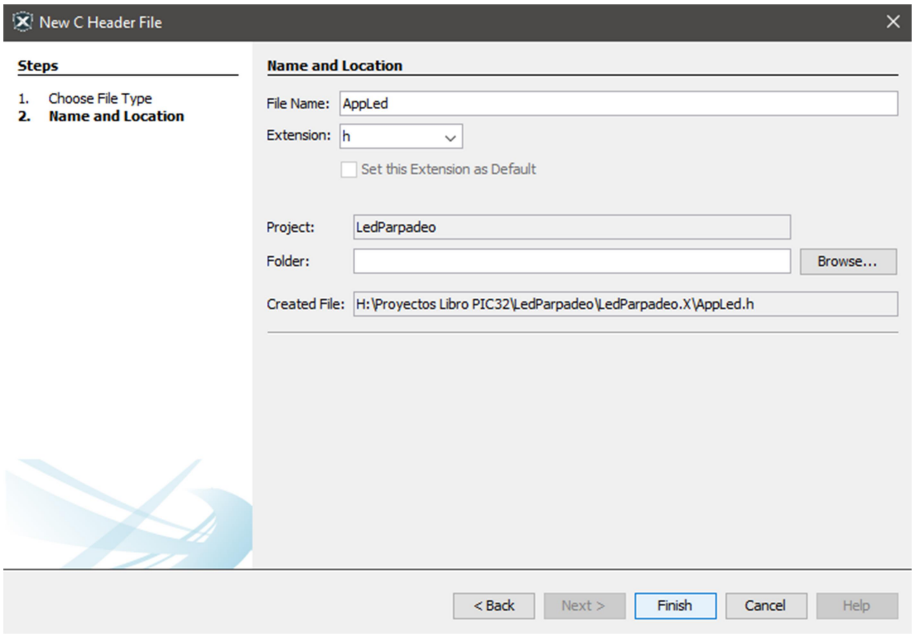


Figura 5.27. Creando un nuevo archivo cabecera.

El nuevo archivo tendrá un aspecto similar al de la Figura 5.28, las definiciones que se crean por defecto se las puede borrar, yo prefiero mantenerlas y escribir cualquier código nuevo debajo de las mismas.

El siguiente paso será escribir los estados de la app como enumeradores, para este proyecto se puede decir que el estado tendrá dos valores, un inicial y otro de conmutación y los defino como:

APP_LED_INICIO

APP_LED_CONMUTAR

En 'Inicio' se obtendrá el valor del temporizador del sistema, mientras que en el estado conmutar se analizará si se ha cumplido el tiempo establecido para conmutar el estado del led.

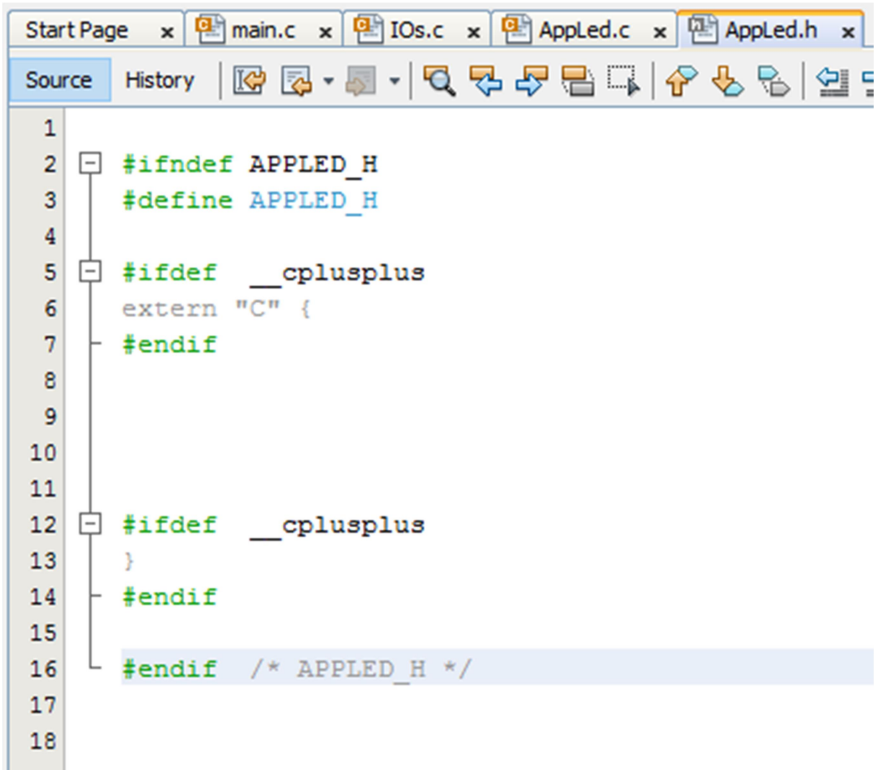


Figura 5.28 Archivo cabecera AppLed.h creado para el segundo proyecto.

Entonces en el archivo AppLed.h escribimos los dos estados de la *app* como se indica en la Figura 5.29.

```
typedef enum
{
    APP_LED_INICIO = 0x00,
    APP_LED_CONMUTAR
}APP_LED_ESTADOS;
```

Figura 5.29 Estados de la AppLed, nótese que el estado inicial es igual a cero.

A continuación creamos en este mismo archivo la variable que posee el estado de la *app* que será de tipo *char* sin signo. Figura 5.30.

```
18  typedef enum {
19      APP_LED_INICIO = 0x00,
20      APP_LED_CONMUTAR
21  }APP_LED_ESTADOS;
22
23  typedef struct {
24      unsigned char estado;
25  } APP_LED_DATOS;
```

Figura 5.30. Variable estado para AppLed.

Ahora se debe declarar una variable similar a la estructura de datos del archivo cabecera para AppLed.c, para lo cual debemos incluir el archivo cabecera como se indica en la Figura 5.31.

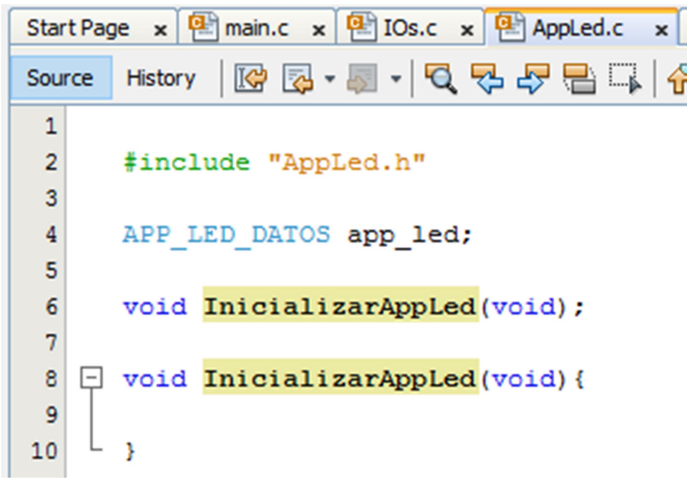


Figura 5.31. Creando una variable tipo APP_LED_DATOS para ser utilizada en el archivo AppLed.c

Finalmente dentro de la función InicializarAppLed inicializamos la variable estado de app_led. Figura 5.32.

Recuerde compilar el proyecto para determinar posibles errores. Es necesario inicializar las variables a un valor definido ya que el compilador no lo hace automáticamente y la memoria RAM del MCU para el usuario generalmente está llena con cualquier valor al energizarse el sistema.

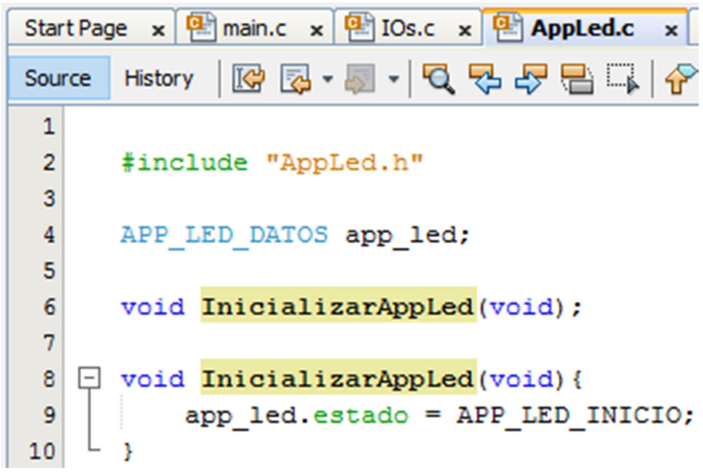


Figura 5.32. Inicializando la variable estado de app_led dentro de la función InicializarAppLed.

5.6 Creando la función AppLed

Ahora ya estamos preparados para crear la función que analiza los estados de la *app*. Esta función también estará dentro de AppLed y se llamará de igual manera. Mediante una sentencia *switch* se analizará cada estado de la tarea. Figura 5.33.

Esta función se ejecutará constantemente dentro del lazo principal en *main* como se indica en la Figura 5.34 y también luego de dicha función escribimos la sentencia para limpiar al perro guardián.

El siguiente paso sería escribir la función InicializarTemporizadorSistema que por el momento sólo se escribirá un prototipo ya que más adelante se analizará lo necesario para configurar correctamente al temporizador que necesitamos. Por el momento debo indicar que el temporizador que utilizaremos es la unión del temporizador y del temporizador 5 ya que entre los dos forman uno de 32 bits.

En la carpeta lógica Perifericos creamos un archivo de nombre Temporizador4_5.c. Figura 5.35.

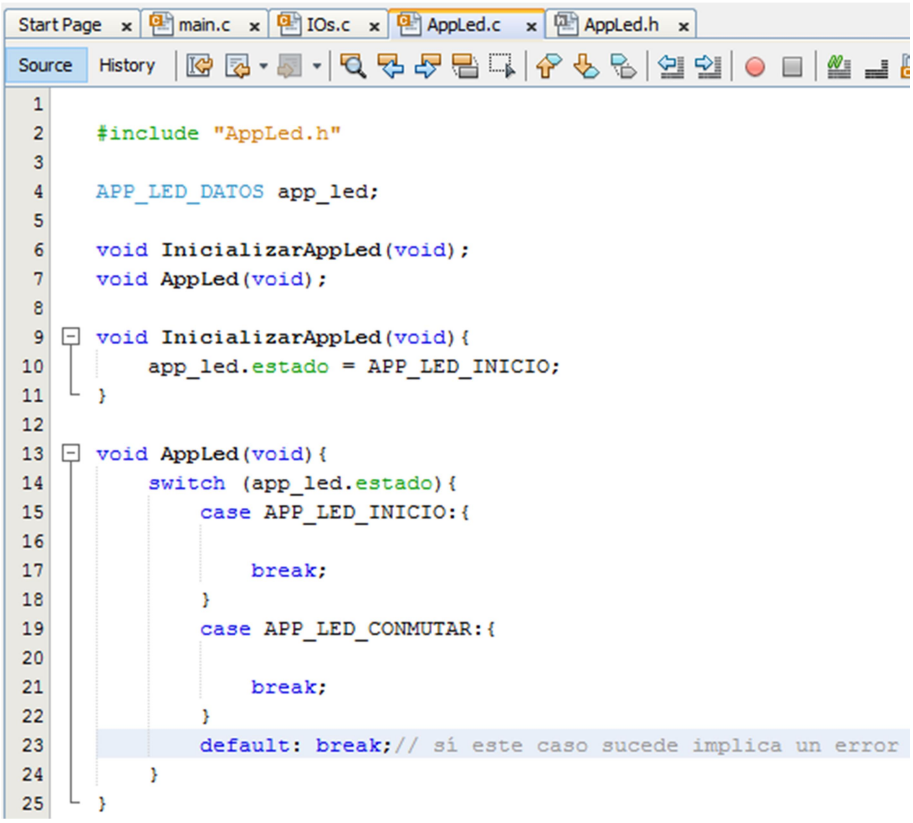


Figura 5.33.Función AppLed lista para realizar su procesamiento de datos.

```
#include <xc.h>
#include <stdio.h>
#include <stdlib.h>

void ConfigurarIOs(void);
void InicializarAppLed(void);
void AppLed(void);
int main (void);

int main(void) {
    ConfigurarIOs();
    InicializarAppLed();
    while(1){
        AppLed();
        WDTCONbits.WDTCLR = 1; // reinicia al WDT
    }
    return (EXIT_SUCCESS);
}
```

Figura 5.34.Función AppLed siendo ejecutada dentro del laso principal en la función *main*

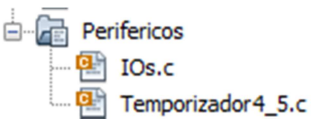


Figura 5.35. Archivo Temporizador4_5.c que contendrá las funciones que corresponden al temporizador del sistema.

En el archivo Temporizador4_5.c escribimos la función InicializarTemporizadorSistema, Figura 5.36.

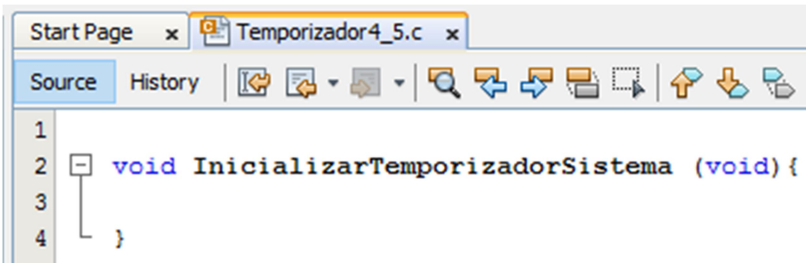


Figura 5.36. Función InicializarTemporizadorSistema que será útil para configurar al temporizador correctamente.

Esta función será invocada en la función *main* antes del lazo principal. Figura 5.36.

La siguiente función sería aquella que corresponde a la configuración de las interrupciones, pero como en este ejemplo no se las utiliza, se la escribirá y explicará en el primer ejemplo dónde se la utilice.

5.7 Configurando al Temporizador del Sistema

La velocidad del reloj para los periféricos es igual a la del CPU que es 48MHz debido a que la configuración del oscilador que es la misma del primer proyecto.

```
#include <xc.h>
#include <stdio.h>
#include <stdlib.h>

void ConfigurarIOs(void);
void InicializarAppLed(void);
void InicializarTemporizadorSistema(void);
void AppLed(void);
int main (void);

int main(void) {
    ConfigurarIOs();
    InicializarAppLed();
    InicializarTemporizadorSistema();
    while(1){
        AppLed();
        WDTCONbits.WDTCLR = 1; // reinicia al WDT
    }
    return (EXIT_SUCCESS);
}
```

Figura 5.36. Función IniciarTemporizadorSistema invocada en la función *main* antes de ejecutar el lazo principal.

Suponiendo que se utiliza un temporizador de 16 bits y que el temporizador se incrementa por cada ciclo del oscilador, es decir sin escalamientos, el valor máximo de tiempo que puede generar es:

$$Tiempo\ Máximo = \frac{65536}{48MHz} = 1.365\ ms$$

1.365 ms es un valor muy bajo ya que lo ideal sería que con algunos ciclos produzca 1 ms y con muchos más se alcance los 500 ms que es nuestro objetivo.

Se podría recurrir a variables auxiliares para llamar varias veces a dicho tiempo y alcanzar el valor deseado pero eso implicaría que nuestro código es algo engorroso y teniendo aún recursos, debemos ser más eficientes y prácticos con el sistema que se está desarrollando.

La siguiente opción es utilizar el escalamiento, por ejemplo digamos que ocupamos su máximo valor que es 256, por lo que el tiempo máximo que se generaría sería:

$$Tiempo\ Máximo = 256\frac{65536}{48MHz} = 349\ ms$$

349 ms es un valor mucho más alto, pero aun así no es la forma correcta, ya que el error para alcanzar el valor de 500ms sería muy alto, el valor más próximo sería el doble, es decir 698 ms, y lo correcto es que con pocos ciclos y no con el máximo se deben alcanzar los 500 ms.

También el tiempo mínimo que puede generarse sería:

$$Tiempo\ Mínimo = 256\frac{1}{48MHz} = 5.333\mu s$$

5.333 µs no es un valor útil para el proyecto que se está diseñando.

Ahora recurramos a un temporizador de 32 bits, de igual manera sin escalamientos el tiempo máximo sería:

$$Tiempo\ Máximo = \frac{4\ 294\ 967\ 296}{48MHz} = 89.47\ s$$

89.47 segundos es demasiado para lo que necesitamos, pero para determinar si es útil o no, hay que calcular con qué valor se puede alcanzar los 500 ms:

$$Valor = Tiempo\ Deseado\ \cdot F_{Periféricos}$$

$$Valor = 500ms\ \times 48MHz = 24000000 = 0x16E3600$$

Es decir con 24 000 000 se consigue 500 ms, lo cual está bien ya que si el valor máximo del temporizador (4 294 967 296) es el 100%, 24 000 000 es el 0.56 %.

El tiempo mínimo con el temporizador de 32 bits sería:

$$Tiempo\ Mínimo = \frac{1}{48MHz} = 20.83\ ns$$

Y para producir 1 ms sería:

$$Valor = 1\ ms\ \times 48MHz = 48000 = 0xBB80$$

En conclusión, un temporizador de 32 bits me permite alcanzar tiempos definidos más exactos sin recurrir a variables auxiliares (al menos que desee generar retardos superiores a 89.47 s)

El microcontrolador posee dos temporizadores de 32 bits, entre los temporizadores 2 y 3 conforman uno y el segundo entre los temporizadores 4 y 5.

A pesar que los registros TMRx o TMRy (donde 'x' representa a los temporizadores 2 ó 5 y donde 'y' representa a los temporizadores 3 ó 5) son de 32 bits para cada temporizador, sólo su parte baja contiene un valor útil, es decir de 16 bits y por esa razón entre dos conforman uno de 32 bits.

Para este ejemplo yo utilizaré la pareja formada por los temporizadores 4 y 5. Nótese que el registro para configurar dichos temporizadores es **T4CON,** Figura 5.37.

REGISTER 13-1: TXCON: TYPE B TIMER CONTROL REGISTER

Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
31:24	U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
	—	—	—	—	—	—	—	—
23:16	U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
	—	—	—	—	—	—	—	—
15:8	R/W-0	U-0	R/W-0	U-0	U-0	U-0	U-0	U-0
	ON(1,3)	—	SIDL(4)	—	—	—	—	—
7:0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	U-0
	TGATE(3)	TCKPS<2:0>(3)			T32(2)	—	TCS(3)	—

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 31-16 **Unimplemented:** Read as '0'

bit 15 **ON:** Timer On bit(1,3)
1 = Module is enabled
0 = Module is disabled

bit 14 **Unimplemented:** Read as '0'

bit 13 **SIDL:** Stop in Idle Mode bit(4)
1 = Discontinue module operation when the device enters Idle mode
0 = Continue module operation when the device enters Idle mode

bit 12-8 **Unimplemented:** Read as '0'

bit 7 **TGATE:** Timer Gated Time Accumulation Enable bit(3)
When TCS = 1:
This bit is ignored and is read as '0'.
When TCS = 0:
1 = Gated time accumulation is enabled
0 = Gated time accumulation is disabled

bit 6-4 **TCKPS<2:0>:** Timer Input Clock Prescale Select bits(3)
111 = 1:256 prescale value
110 = 1:64 prescale value
101 = 1:32 prescale value
100 = 1:16 prescale value
011 = 1:8 prescale value
010 = 1:4 prescale value
001 = 1:2 prescale value
000 = 1:1 prescale value

bit 3 **T32:** 32-Bit Timer Mode Select bit(2)
1 = Odd numbered and even numbered timers form a 32-bit timer
0 = Odd numbered and even numbered timers form a separate 16-bit timer

bit 2 **Unimplemented:** Read as '0'

bit 1 **TCS:** Timer Clock Source Select bit(3)
1 = External clock from TxCK pin
0 = Internal peripheral clock

bit 0 **Unimplemented:** Read as '0'

Note 1: When using 1:1 PBCLK divisor, the user's software should not read/write the peripheral SFRs in the SYSCLK cycle immediately following the instruction that clears the module's ON bit.
2: This bit is available only on even numbered timers (Timer2 and Timer4).
3: While operating in 32-bit mode, this bit has no effect for odd numbered timers (Timer3, and Timer5). All timer functions are set through the even numbered timers.
4: While operating in 32-bit mode, this bit must be cleared on odd numbered timers to enable the 32-bit timer in Idle mode.

Figura 5.37. Registro TXCON que permite configurar a los temporizadores 2,4,5 y 6

La nota 1 indicada en la Figura 5.37 advierte que no se debe escribir en un registro un valor tal que apague a un módulo cuando la frecuencia del oscilador para los periféricos tenga un valor de división igual a 1 (que es nuestro caso), pero como se va activar a dicho módulo, no existe problema.

Analizando lo indicado en la Figura 5.37, el bit15 deberá estar en 1 lógico, el bit 13 puede estar en cualquier valor ya que el proyecto no entrará a modo de bajo consumo IDLE, el bit 7 puede contener cualquier valor según la nota 3, entre los bits 6 y 4 el valor debe ser de 0, es decir sin ningún escalamiento, el bit 1 debe estar en cero ya que los pulsos que incrementarán el valor del temporizador serán los del reloj para los periféricos. Por lo tanto el valor que contendrá T4CON será:

0x8008

Entonces en la función InicializarTemporizadorSistema se procede a asignar dicho valor a T4CON. Figura 5.38.

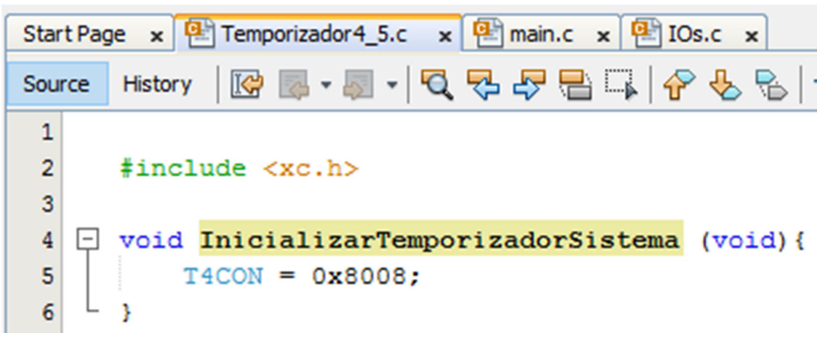


Figura 5.38. Asignado el valor correcto al registro T4CON, nótese que se necesita incluir xc.h para que el compilador reconozca a dicho registro.

5.7 Recuperando el valor instantáneo del Temporizador del Sistema

La siguiente función que se necesita implementar es aquella que obtiene el valor instantáneo del temporizador que ha sido asignado para el

sistema, en este caso el valor de TMR4 y TMR5 deben crear una variable de 32 bits y retornarlos cuando la función sea invocada.

Nótese que con las partes bajas de TMR4 y TMR5 (que son de 16 bits) se formará una variable de 32 bits. Esto puede generar una condición de carrera como se mencionó en el Capítulo 4 y la Figura 4.3 indica cómo solucionar ese posible problema sin detener al temporizador.

La función que retorna dicho valor se denominará obtenerValorTemporizador y su valor a retornar será de 32 bits sin signo, es decir *unsigned int* Figura 5.39.

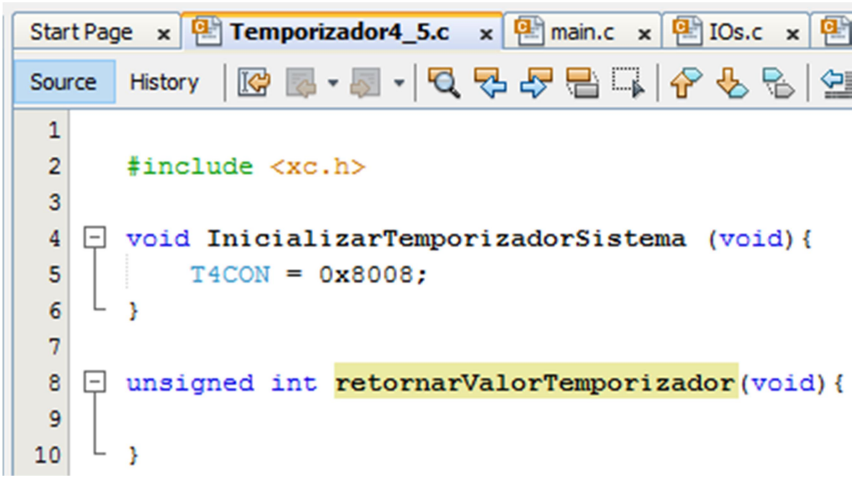


Figura 5.39. Función retornarValorTemporizador en el archivo Temporizador4_5.c

Lo primero que se hace es capturar los valores de TMR5 (parte alta) y TMR4 parte baja en variables auxiliares. Figura 5.40.

Luego, para evitar la condición de carrera, se debe verificar que la parte alta no haya cambiado, si es así, se debe capturar los valores anteriormente mencionados. Figura 5.41

```
8  unsigned int retornarValorTemporizador(void){
9      unsigned int ParteAlta;
10     unsigned int ParteBaja;
11     do
12     {
13         ParteAlta = TMR5;
14         ParteBaja = TMR4;
15     }
   }
```

Figura 5.40. Capturando el valor de TMR5 y TMR4 en variables auxiliares.

```
8  unsigned int retornarValorTemporizador(void){
9      unsigned int ParteAlta;
10     unsigned int ParteBaja;
11     do
12     {
13         ParteAlta = TMR5;
14         ParteBaja = TMR4;
15     }while (ParteAlta != TMR5);// parte alta no cambió, cotinúa
16 }
```

Figura 5.41. En la condición *while* se evita una posible condición de secuencia.

Finalmente entre los valores capturados se crea una variable de 32 bits y se retorna el valor. Figura 5.42.

```
8  unsigned int retornarValorTemporizador(void){
9      unsigned int ParteAlta;
10     unsigned int ParteBaja;
11     do
12     {
13         ParteAlta = TMR5;
14         ParteBaja = TMR4;
15     }while (ParteAlta != TMR5);
16     return ((ParteAlta << 16) | ParteBaja);
17 }
```

Figura 5.42. Finalmente se retorna el valor instantáneo del temporizador.

5.8 Encendiendo y apagando al LED cada 500 ms.

Ahora que está lista la función que permite obtener el valor instantáneo del temporizador, debemos utilizarla en la función AppLed.

Primero se debe crear una variable que contenga el valor instantáneo del temporizador, para lo cual en AppLed.h creamos la variable de 32 bits. Figura 5.43.

```
23  typedef struct {
24      unsigned char estado;
25      unsigned int valorInstantaneoTMR;
26  } APP_LED_DATOS;
```

Figura 5.43. La variable valorInstantaneoTMR capturará el valor del temporizador del sistema.

Luego, en el primer caso dela función AppLed (APP_LED_INICIO) se capturará el valor instantáneo del temporizador y el estado de la *app* se cambia a APP_LED_CONMUTAR. Figura 5.44.

En el estado APP_LED_CONMUTAR de la función AppLed se verifica que la diferencia entre el valor capturado y el valor instantáneo sea mayor que 500 ms, si es así procedemos a conmutar el estado del LED y nuevamente capturamos un nuevo valor del temporizador para la próxima vez. Figura 5.45.

Finalmente compilamos el proyecto y 'descargamos' el firmware hacia el hardware para comprobar la funcionalidad del mismo. Figura 5.46

```
#include "AppLed.h"

APP_LED_DATOS app_led;

void InicializarAppLed(void);
void AppLed(void);

void InicializarAppLed(void){
    app_led.estado = APP_LED_INICIO;
}

void AppLed(void){
    switch (app_led.estado){
        case APP_LED_INICIO:{
            app_led.valorInstantaneoTMR = retornarValorTemporizador();
            app_led.estado = APP_LED_CONMUTAR;
            break;
        }
        case APP_LED_CONMUTAR:{

            break;
        }
        default: break;// si este caso sucede implica un error
    }
}
```

Figura 5.44. En el estado APP_LED_INICIO se captura el valor del temporizador del sistema.

```
#include "AppLed.h"
#include <xc.h>

#define LED     LATBbits.LATB15
#define _500ms  0x16E3600

APP_LED_DATOS app_led;

void InicializarAppLed(void);
void AppLed(void);

void InicializarAppLed(void){
    app_led.estado = APP_LED_INICIO;
}

void AppLed(void){
    switch (app_led.estado){
        case APP_LED_INICIO:{
            app_led.valorInstantaneoTMR = retornarValorTemporizador();
            app_led.estado = APP_LED_CONMUTAR;
            break;
        }
        case APP_LED_CONMUTAR:{
            if((retornarValorTemporizador() - app_led.valorInstantaneoTMR)> _500ms){
                LED ^= 1;
                app_led.valorInstantaneoTMR = retornarValorTemporizador();
            }
            break;
        }
        default: break;// si este caso sucede implica un error
    }
}
```

Figura 5.45. En el estado APP_LED_CONMUTAR cada 500 ms aproximadamente se cambia el estado del led.

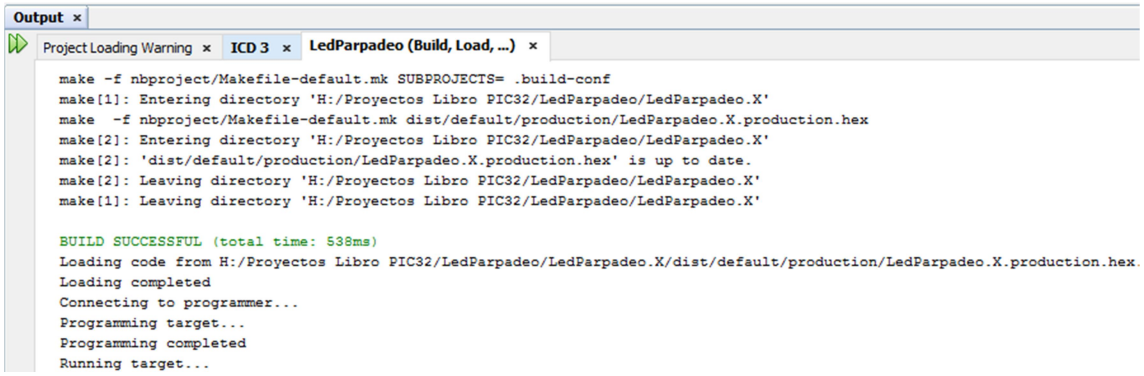


Figura 5.46. El firmware se descargó al hardware utilizado en el primer proyecto y funcionó correctamente.

Se puede comprobar que tan exacto es el retardo, para lo cual ponemos un punto de ruptura en donde el LED es conmutado y empezamos la depuración hasta cuando se lo alcanza. Figura 5.47.

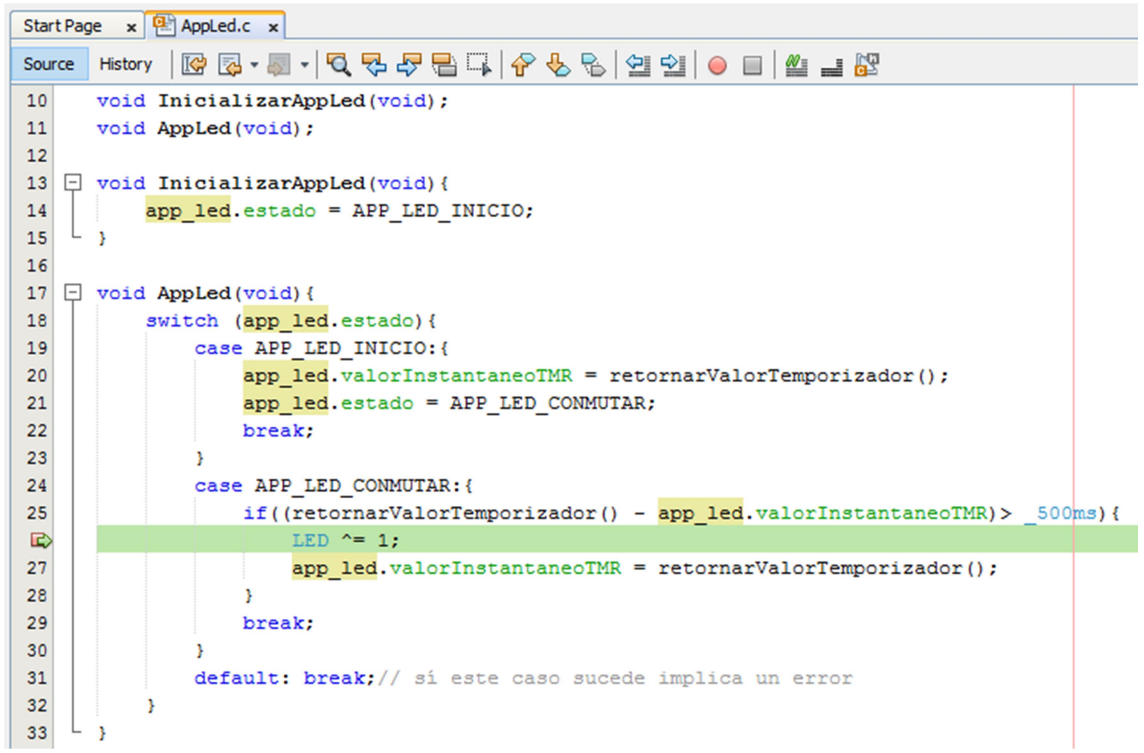


Figura 5.47. Punto de ruptura dónde se conmuta al LED.

A continuación procedemos a desensamblar el código para el archivo AppLed.c. Damos clic en *Windows* / *Debugging* / *Disassembly* Figura 5.48.

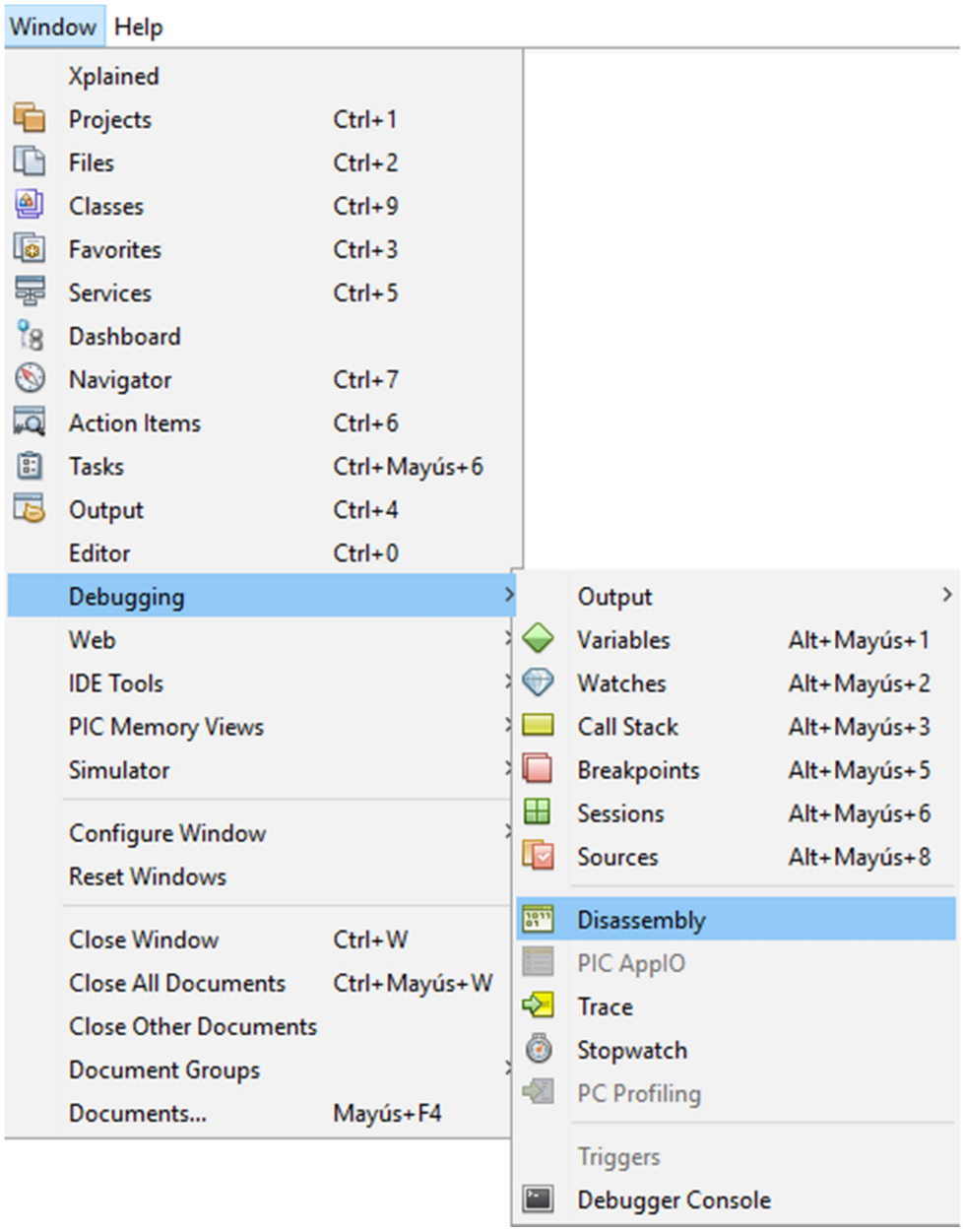


Figura 5.48. Manera de observar en lenguaje ensamblador un archivo .c

En el código en lenguaje ensamblador ponemos un punto de ruptura justo antes de realizar el proceso de conmutar al LED. Figura 5.48.

Presionamos F5 para que nuevamente se ejecute la depuración y se alcance este segundo punto de ruptura. Figura 5.49, entonces se procede a analizar el valor de TMr4, TMR5 y de valorInstantaneoTMR. Figura 5.50.

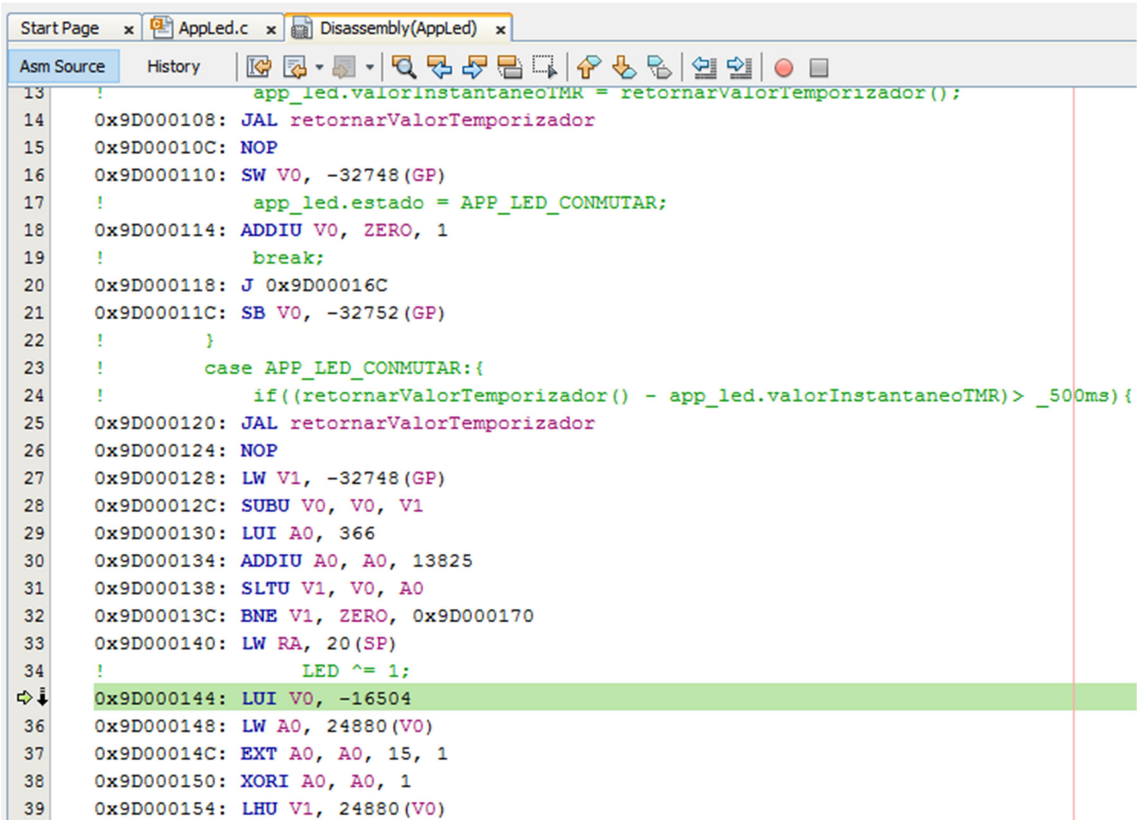


Figura 5.49. Punto de ruptura alcanzado en el código desensamblado de la función AppLed.

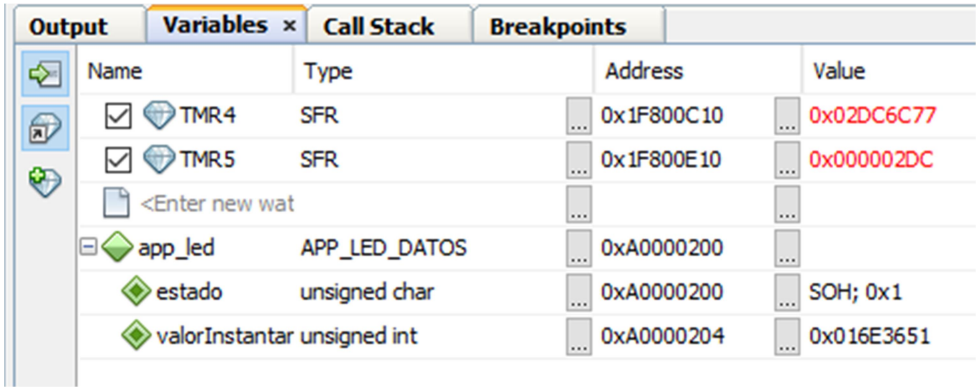


Figura 5.50. Valores de TMR4, TMR5 y valorInstantaneoTMR luego de cumplirse la condición mayor que.

Entre TMR4 y TMR5 (parte baja y alta respectivamente) forman el número:

0x02DC 6C77

A pesar que la parte alta de TMR4 tiene un valor diferente de cero, sólo es útil su parte baja de 16 bits.

La diferencia con la variable es:

0x02DC6C77 – 0x16E3651 = 0x16E 3626 = 24 000 038

Y calculando el tiempo es:

$$Tiempo\ = \frac{24000038}{48MHz} = 500.00079\ ms$$

Lo que implica que el retardo realizado es muy cercano al valor deseado.

5.9. Utilizando el Temporizador Central o *Core Timer*.

El proyecto actual funciona correctamente y para generar retardos se ha recurrido a un temporizador el cuál su única función o utilidad es de contador de ciclos de máquina para generar tiempos definidos.

Pero existe un temporizador que realiza esa misma funcionalidad sin haberlo configurado previamente y que podría afirmarse que es un desperdicio si no se lo utiliza.

La arquitectura del PIC32 incluye un temporizador central o *Core Timer* de 32 bits. Este temporizador se implementa en forma de dos registros de coprocesador: el registro de Conteo y el registro de Comparación. El registro de conteo se incrementa **cada dos ciclos de reloj del sistema** (SYSCLK).

El incremento de Conteo puede ser suspendido opcionalmente durante el modo depuración. El registro de Comparación se usa para provocar una interrupción de temporizador si se desea. Se genera una interrupción cuando ambos registros se igualan.

En el proyecto de este capítulo, la frecuencia con la que este temporizador funcionaría sería de 24 MHz, el análisis de los valores de tiempo conseguido estarían dadas por las siguientes ecuaciones:

$$Tiempo\ Máximo = \frac{4\ 294\ 967\ 296}{24MHz} = 178.95\ s$$

$$Valor = Tiempo\ Deseado\ \cdot F_{Periféricos/2}$$

$$Valor = 500ms\ \times 24MHz = 12000000 = 0xB71B00$$

El tiempo mínimo sería:

$$Tiempo\ Mínimo = \frac{1}{24MHz} = 41.67\ ns$$

El siguiente paso sería adaptar el proyecto a este temporizador para lo cual la función InicializarTemporizadorSistema no sería necesaria ya que el temporizador siempre está activado y funcional, mientras que la función retornarValorTemporizador devolvería el valor de CP0 sin realizar los procesos necesarios para evitar la condición de secuencia ya que el *Core Timer* es un verdadero temporizador de 32 bits. XC32 ofrece una función para retornar el valor de dicho registro pero añade instrucciones que harían aumentar el error del momento exacto de captura de tiempo, para lo cual es mucho mejor solo leer dicho registro. En la Figura 5.51 se puede ver los cambios que se hacen al proyecto para utilizar el temporizador central.

. Figura 5.51. Modificando el proyecto para utilizar el Temporizador Central.

Estás modificaciones se comprobaron en el hardware utilizado y el sistema funcionó como se esperaba.

El último cambio que se realiza luego de comprobar que el sistema funciona correctamente con el *Core Timer* es cambiar el nombre del archivo .c denominado Temporizador4_5 ya que no tiene nada que ver con el nuevo temporizador. Para lo cual damos clic derecho sobre el mismo y escogemos la opción *Rename…* Figura 5.52.

Cambiamos el nombre a Temporizador Central y el proyecto se verá similar al indicado en la Figura 5.53

Nuevamente, recuerde compilar y también 'descargar' el firmware al hardware para comprobar que el proyecto funciona correctamente y no se haya generado algún error en la modificación del mismo.

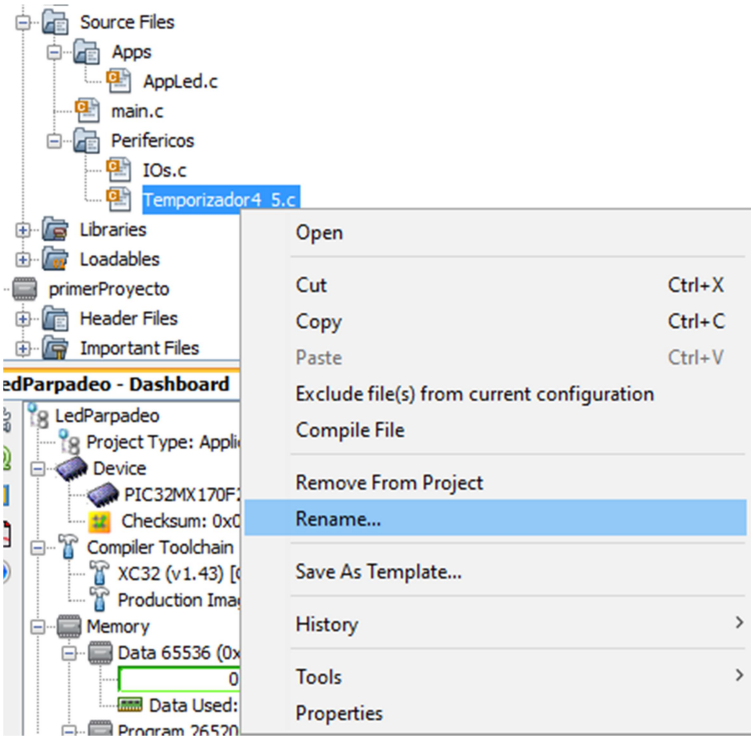


Figura 5.52. Cambiando el nombre del archivo Temporizador4_5.c

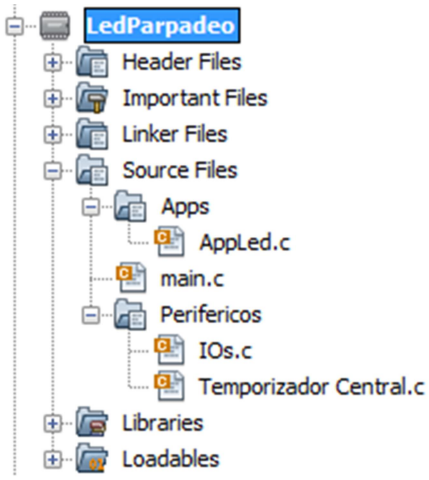


Figura 5.53. El archivo Temporizador4_5.c cambió de nombre a Temporizador Central

Aquí finaliza el segundo proyecto y se describió una manera correcta de realizar un retardo con un temporizador sin consumir todo el proceso del CPU y se aprendió a realizar una tarea por estados, lo que permite crear otras para realizar un trabajo cooperativo con un microcontrolador.