ESCUELA SUPERIOR DE INGENIERÍA Y TECNOLOGÍA

DESARROLLO DE UN JUEGO PARA PANTALLA LCD CON MICROCONTROLADOR PIC18F45K22

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Autor:
Joaquín López Montesino
Tutores:
Evelio José González González
Rafael Arnay del Arco

1. Introducción

1.1 Abstract

This project focuses on the development of a program in MikroC for PIC for the creation of a runner type game on a 2x16 LCD screen. The Easy PIC v7 board was used as the development platform, which required the installation of Windows 7 after researching and testing various options to achieve a suitable configuration. Other tasks were also carried out to ensure the correct functioning of the development environment.

The main objective was to develop an interactive and entertaining game that takes advantage of the capabilities of the LCD display, demonstrating the knowledge acquired in microcontroller programming. In summary, this project represents a combination of technical and creative skills, offering hand-on experience in embedded software development and interaction with peripheral devices.

1.2. Resumen

Este proyecto se centra en el desarrollo de un programa en MikroC para PIC para la creación de un juego tipo runner en una pantalla LCD de 2x16. Como plataforma de desarrollo se utilizó la placa Easy PIC v7, que requirió la instalación de Windows 7 tras investigar y probar diversas opciones para conseguir una configuración adecuada. También se llevaron a cabo otras tareas para asegurar el correcto funcionamiento del entorno de desarrollo.

El objetivo principal fue desarrollar un juego interactivo y entretenido que aproveche las capacidades de la pantalla LCD, demostrando los conocimientos adquiridos en programación de microcontroladores. En resumen, este proyecto representa una combinación de habilidades técnicas y creativas, ofreciendo una experiencia práctica en el desarrollo de software empotrado y la interacción con dispositivos periféricos.

1.3. Sistemas empotrados

Son sistemas construidos para llevar a cabo tareas específicas, utilizando para ello un hardware y software mínimo, cuya complejidad puede variar en función del objetivo de este sistema.El software embebido se desarrolla utilizando lenguajes de programación como C, C++ o ensamblador, y luego el microcontrolador interpreta la información que recibe de este programa, después de pasar por una serie de transformaciones, para ejecutar las acciones que el usuario necesita.

Estos programas se encargan de controlar y coordinar las funciones y operaciones del dispositivo, como el control de sensores, la gestión de la interfaz de usuario, el procesamiento de datos, la comunicación con otros dispositivos y cualquier otra tarea específica requerida por el usuario.

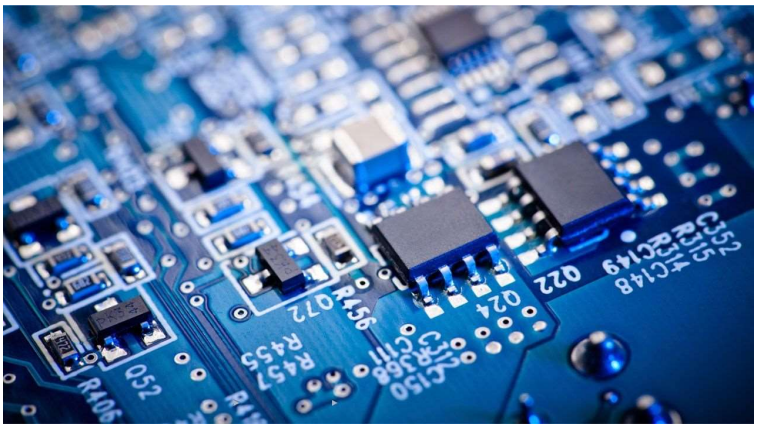


Figura 1.1. Sistema empotrado.

A diferencia de los PC convencionales, los cuales son sistemas de propósito general y se diseñan para cubrir un amplio abanico de necesidades que pueda tener el usuario, tiene como principal objetivo es reducir los costes, por lo que se intenta utilizar los materiales justos y necesarios para cubrir satisfactoriamente la necesidad.

Los sistemas empotrados pueden estar enfocados específicamente a un campo de la industria, como autómatas, cámaras de vigilancia, etc,

en nuestra propia casa como televisores, lavadoras, consolas, etc, y en muchos dispositivos que utilizamos de manera cotidiana.

Figura 1.2 - 1.4. Ejemplos de sistemas empotrados.

Este tipo de sistemas siempre llevan consigo un dispositivo el cual es el encargado de llevar a cabo las tareas deseadas por el usuario y las operaciones necesarias para ello. Este dispositivo es el microprocesador o también conocido como CPU. También contiene varías unidades de almacenamiento capaces de guardar todos los datos necesarios para realizar la tarea. Entre ellas encontramos la memoria ROM(no se borra y solo se escribe al fabricar el chip “Read Only Memory”), la memoria RAM (almacena la información necesaria para que un programa se esté ejecutando “Random Access Memory”) o la memoria EEPROM(es capaz de almacenar datos incluso cuando no hay fuente de alimentación “Electrically Erasable Programmable Read-Only Memory”).

2. Preparación de los materiales empleados

2.1. Placas de desarrollo

Figura 2.1. Placa Easy AVR v7

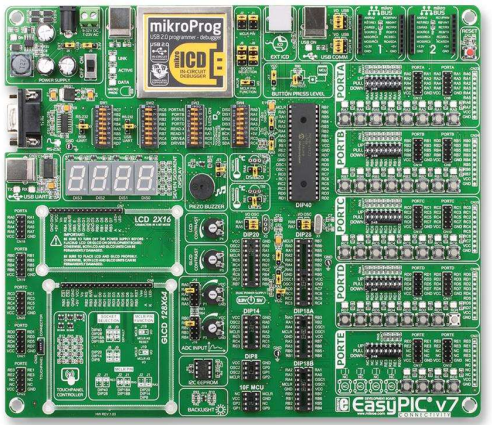


Figura 2.2. Placa Easy PIC v7

El primer día que tengo contacto con los tutores asignados para el Trabajo de Fin de Grado, obtengo dos placas de desarrollo para utilizarlas con el objetivo de hacer una aplicación, por ejemplo un juego, y se ejecute en uno de los display que traen implementados estos dispositivos.

En primer lugar se me pide que lo realice en el panel gráfico de pantalla táctil de 126x64, la cual tenía estropeado el conector habilitado para la lectura del panel táctil por lo que procedo a buscar la manera de soldar los 4 cables en una superficie muy pequeña. Al principio lo he intentado con un soldador de estaño, pero ni con ayuda adicional he conseguido alcanzar la precisión necesaria para llevar a cabo una soldadura sólida y que no se suelten con cualquier mínimo movimiento. En busca de una solución a ello, he conseguido soldar la pantalla con una resina especial que se endurece con luz ultravioleta.

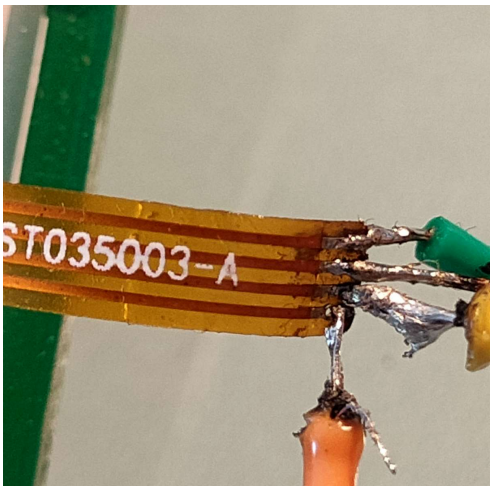


Figura 2.3. Soldadura de estaño.

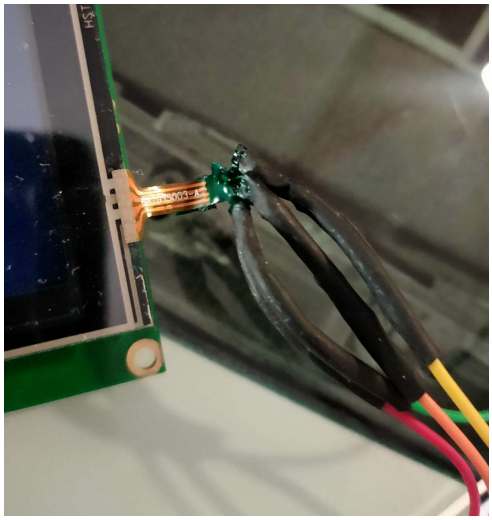


Figura 2.4. Soldadura con resina

Una vez tenemos la pantalla con posibilidad de conectarla a la placa, intentamos conseguir una correcta comunicación entre el ordenador la placa y el display. El primer problema que nos encontramos es que el ordenador no reconoce por el puerto USB a la placa debido a que esta solo trae los drivers hasta Windows 7, por lo que debemos aprender como instalar sistemas operativos en los ordenadores que tengo disponibles en casa. Luego buscamos un programa con el que se pueda programar en C la placa utilizada, en específico microcontroladores de la familia ATMEL.

Cuando llegamos a este punto, después de pasar varios programas de prueba a la placa, vemos que los resultados de ejecución en la pantalla no son los esperados, y acabamos utilizando otra distinta, la cual utiliza un micro de la familia PIC. Además, hemos tenido algún problema con la pantalla táctil por lo que finalmente, vamos a usar la placa Easy PIC v7 y la pantalla LCD 2x16.

2.2. Instalación de sistema operativo

2.2.1. Instalar Oracle Virtualbox en ordenador de sobremesa

En primer lugar, siguiendo los consejos de los tutores, he procedido a la instalación de Virtualbox para poder instalar un sistema operativo compatible con la placa sin tener que eliminar el que ya tiene instalado. Este programa fue desarrollado por la empresa Oracle y es capaz de ejecutar varios sistemas operativos simultáneamente en una sola máquina física. Este programa ofrece varías ventajas como que está disponible para instalarse en cualquier sistema operativo, ya sea Windows, Linux o macOS, entre otro, y además totalmente gratuito. El ordenador que vamos a utilizar para esta tarea tiene las siguientes especificaciones.

- Procesador: 12th Gen Intel(R) Core(TM) i7-1260P 2.10 GHz
- Ram instalada: 16,0 GB (15,8 GB usable)
- Hard Disk: 500 GB
- Tipo de sistema: Sistema operativo de 64 bits, procesador basado en x64
- Edición: Windows 11 Pro

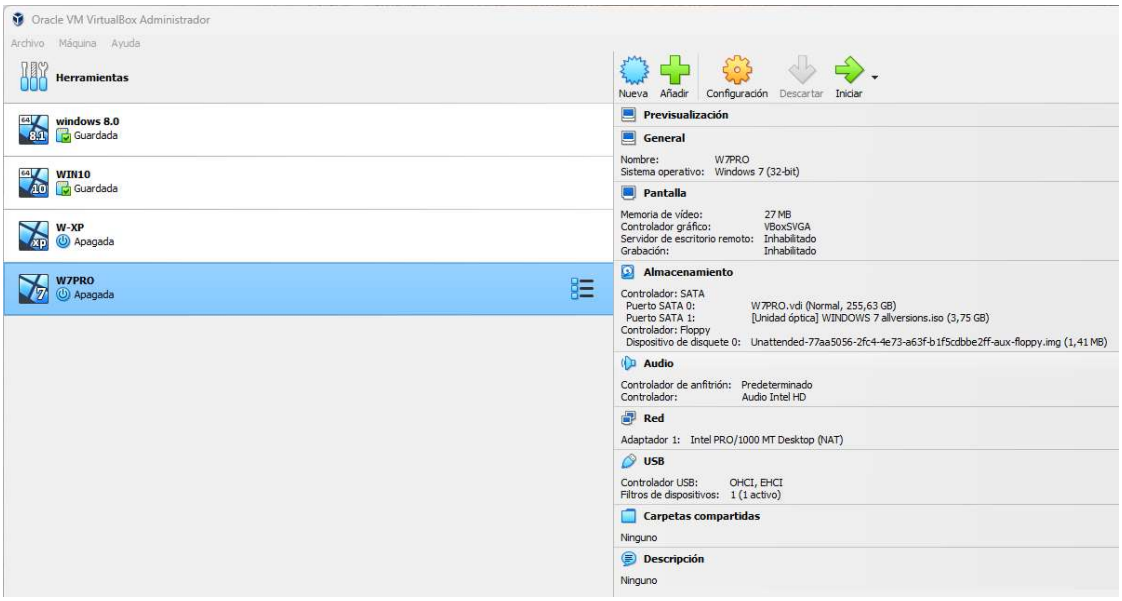


Figura 2.5. Interfaz de Oracle VM

A continuación, descargo de la página oficial de Microsoft la imagen ISO de Windows 10. Terminado el proceso de instalación procedo a conectar la tarjeta con el mismo resultado que el caso anterior, es decir, que no se instalan los controladores. Por último, descargo la ISO de Windows 7, creo nueva máquina virtual y conecto la tarjeta por puerto USB y en esta ocasión la máquina no reconoce los puertos USB´s de ordenador por lo que tampoco consigo conectar la tarjeta e instalar los drivers.

2.2.2. Instalación de W7 en portátil antiguo mediante USB booteable

Conectar la tarjeta por puerto USB a un portátil antiguo con las siguientes características:

- Procesador: Intel Core 2 Duo T7550 2.0 GHz
- Ram instalada: 4,0 GB
- Disco duro: 250 GB
- Tipo de sistema: Sistema operativo de 64 bits, procesador basado en x64
- Edición: Windows vista

Debido a que el sistema operativo Windows vista, el ordenador se bloquea constantemente y es imposible trabajar con él, procedo a instalar desde cero la versión Windows 7 profesional creando una memoria USB booteable (memoria en formato USB que sirve para llevar a cabo el proceso de 'boot' o lo que es lo mismo, arrancar el ordenador desde el USB).

Para ello en primer lugar descargamos la aplicación Rufus de la Microsoft Store que es útil para formatear y crear dispositivos flash booteables. Una vez lo tengamos, podemos descargar la imágen iso de Windows 7 Professional usando el navegador. A continuación, seleccionamos las opciones de formateo en el programa Rufus y comenzamos el proceso de montaje de la imagen ISO en el pendrive booteable.

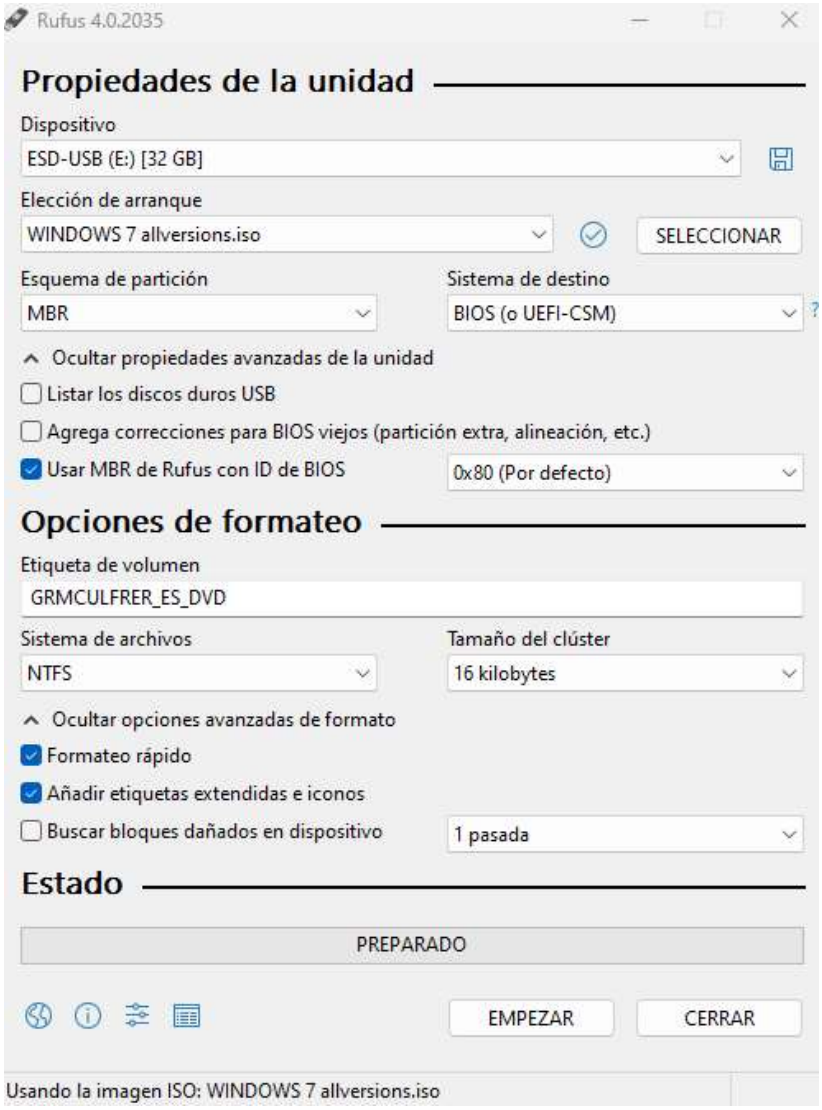


Figura 2.6. Interfaz del programa Rufus.

En la figura 2.6 podemos observar los ajustes que he utilizado "bootear" el pendrive. Una vez los hayamos elegido correctamente, seleccionamos la opción preparado, y luego empezar. Después de aproximadamente media hora, el proceso de montaje de la imagen ISO estará finalizado, y podemos retirar el pendrive para utilizarlo en otro dispositivo.

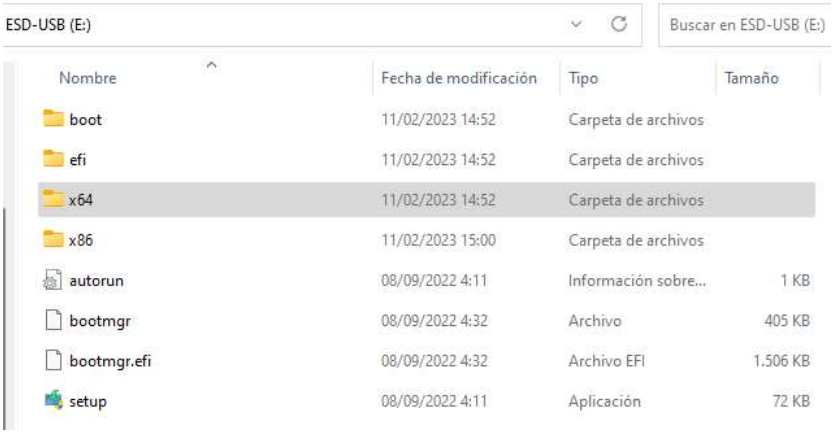


Figura 2.7. Contenido USB booteable.

Una vez tengamos el dispositivo flash preparado, lo introducimos en el ordenador donde queremos instalar el sistema operativo, encendemos el ordenador y pulsamos la tecla de función F10 para que nos muestre el boot menu con todas las posibilidades de las cuales elegimos el arranque desde el pendrive que hemos insertado anteriormente. Esta opción es mejor que la de pulsar F12 para entrar en la BIOS y elegir dentro de las opciones del BOOT que arranque primero desde USB y una vez terminado el proceso volver a la BIOS para cambiar el orden y decirle que ahora arranque primero desde el disco duro. Las teclas de función para entrar en esta funciones pueden variar según la marca del ordenador.

Si lo hemos hecho correctamente, aparecerá en la pantalla “Windows is loading files…” y luego “Starting Windows”. Luego nos saldrá un menú donde seleccionaremos las opciones deseadas para personalizar la instalación , elegimos no particionar el disco, por los que formateamos una sola partición que será la activa, es decir, desde la que arrancará el portátil. Finalmente retiramos el pendrive y reiniciamos el equipo, durante el cual se ejecutan los procesos para completar la instalación del sistema. Cuando se vuelva a iniciar aparecerá un mensaje que nos dice que el ordenador se está preparando para el primer uso, y luego se abrirá una pantalla con el usuario administrador sin contraseña, desde donde podemos manejar el sistema instalado con todos los privilegios para administrar el equipo.

2.3. Instalación del software

La idea inicial era utilizar el programa MikroC for AVR que viene diseñado para la placa que vamos a utiliza y se puede obtener con el disco que viene incluido junto con la placa y otros materiales, pero como no estaba, al descargarlo desde la web o desde el disco de la otra placa saltaba el límite de la demo, lo que me impedía hacer un programa lo suficientemente extenso como para llevar a cabo este trabajo. Entonces, he buscado otros programas con los que se pueda trabajar con el lenguaje en C para la gama de microcontroladores Atmel y he elegido el programa Atmel Studio 7.0. Además, se necesita otro programa capaz de enviarle la información del programa (mediante un archivo .hex que se genera en la carpeta del proyecto una vez esté compilado) a la placa. Para ello hemos instalado el programa AVRFLASH el cual hemos obtenido de la página web del fabricante.

Este programa tiene el inconveniente de que no tiene incluidas las librerías necesarias para trabajar con los distintos dispositivos que incluye nuestra placa. Para solucionarlo he buscado información y he encontrado algunas librerías para poder comunicarme con la pantalla gráfica LCD, específicamente he utilizado la librería u8g2 que he encontrado en la página web GitHub.

En primer lugar, vamos a crear un programa de prueba, el cual encienda o apague una serie de Leds, para ver si la comunicación entre el ordenador y la placa se produce correctamente y ejecuta el código de manera correcta. Una vez llevado a cabo, procedemos a utilizar librerías descargadas para hacer una prueba con la pantalla gráfica LCD. Para crear este programa de prueba, ha sido necesario buscar información para programar en este lenguaje específico y para utilizar las librerías descargadas correctamente.

Como vemos en las siguientes imágenes, en primer lugar incluimos las librerías que vamos a utilizar, luego definimos las conexiones entre la pantalla y el microcontrolador, después tenemos unas funciones que hemos creado que no están en el main de la foto pero fueron utilizadas anteriormente pero como no salía nada por la pantalla se intentó de varias maneras diferentes.

```
/*
 * Prueba2.c
 *
 * Created: 14/04/2023 12:30:24
 * Author : Joaquin
 */


#include <u8g2.h>
#include <avr/io.h>
#include <stdlib.h>
#include <stdio.h>
#include <util/delay.h>
#include <avr/pgmspace.h>


#define DISPLAY_CLK_DIR DDRB
#define DISPLAY_CLK_PORT PORTB
#define DISPLAY_CLK_PIN PINB5
```

Figura 2.7. Parte 1/4 programa de prueba

```
#define DISPLAY_CLK_PORT PORTB
#define DISPLAY_CLK_PIN PINB5

#define DISPLAY_DATA_DIR DDRB
#define DISPLAY_DATA_PORT PORTB
#define DISPLAY_DATA_PIN PINB3

#define DISPLAY_CS_DIR DDRB
#define DISPLAY_CS_PORT PORTB
#define DISPLAY_CS_PIN PINB2

#define DISPLAY_DC_DIR DDRB
#define DISPLAY_DC_PORT PORTB
#define DISPLAY_DC_PIN PINB1

#define DISPLAY_RESET_DIR DDRB
#define DISPLAY_RESET_PORT PORTB
#define DISPLAY_RESET_PIN PINB0
#define P_CPU_NS (1000000000UL / F_CPU)

void startTouchscreen()
{
    DDRA = 0xC;
    PORTA = 0x00;
}
```

Figura 2.8. Parte 2/4 programa de prueba

```
void drawHelloWorld()
{
    u8g2_ClearBuffer(&u8g2);

    u8g2_SetFont(&u8g2, u8g2_font_ncenB08_tr); // Selecciona la fuente
    u8g2_DrawStr(&u8g2, 10, 20, "Hola, mundo!"); // Dibuja el texto en la posición (10, 20)

    u8g2_SendBuffer(&u8g2);
}

//void dibujaColumna
//void borraColumna
//int colision


// Declaración de la pantalla GLCD ST7920
u8g2_t u8g2;

// Función de inicialización de la pantalla
void initGLCD()
{
    u8g2_Setup_st7920_s_128x64_f(&u8g2, U8G2_R0, u8x8_byte_4wire_sw_spi, u8x8_avr_gpio_and_delay);
    u8g2_InitDisplay(&u8g2);
    u8g2_SetPowerSave(&u8g2, 0);
    u8g2_ClearBuffer(&u8g2);
}
```

Figura 2.9. Parte 3/4 del programa de prueba.

```
int main()
{
    initGLCD();
    //startTouchscreen();
    //DDRD = 255;
    while (1)
    {   //PORTD=55;
        u8g2_ClearBuffer(&u8g2);
        u8g2_DrawFrame(&u8g2, 10, 10, 100, 50); // Dibujar un rectángulo en las coordenadas (10, 10) con ancho 100 y alto 50
        //u8g2_DrawPixel(&u8g2, 20, 20);
        u8g2_SendBuffer(&u8g2);

        _delay_ms(2000); // Mostrar el rectángulo durante 2 segundos
//          //PORTD=0;
//      u8g2_ClearBuffer(&u8g2);
//      u8g2_SendBuffer(&u8g2);
//
//      _delay_ms(2000); // Esperar 2 segundos antes de mostrar el rectángulo nuevamente
    }

    return 0;
}
```

Figura 2.10. Parte 4/4 del programa de prueba

2.4. Mal funcionamiento de la placa Easy AVR v7

Una vez incluidas las librerías necesarias en la carpeta del proyecto, procedemos a revisar los diferentes comandos y funciones que contiene para implementarlas en mi programa. Al llegar a esta situación, nos encontramos con que no hay ningún tipo de salida por la pantalla, aunque probemos distintas funciones como dibujar un pixel, una forma geométrica, etc. Tampoco ha funcionado al utilizar otra librería, como glcd-master del mismo repositorio online pero sin éxito. Una vez hecho esto, tampoco consigo que salga nada por esta pantalla, por lo que llego a la conclusión de que la placa tiene algún tipo de fallo.

2.5. Probamos la placa Easy PIC v7

Suponiendo que la placa Easy AVR v7 no funciona correctamente, utilizo la otra placa para controladores PIC con la misma pantalla LCD. Para ello, necesito descargar el programa MikroC for PIC a través del “Product DVD” que incluye este producto. A su vez, también se descarga el programa MikroProg Suite for Pic para trasladar la información del programa a la placa mediante conexión USB. Una vez

hecho esto, comienzo a familiarizarme con el programa y a conocer qué comandos utiliza para comunicarse con los diferentes periféricos de la placa, en particular con la pantalla LCD que voy a usar. Para comprobar si esta pantalla funciona correctamente y el fallo anterior era debido a la placa, vamos a realizar un programa de prueba sencillo.

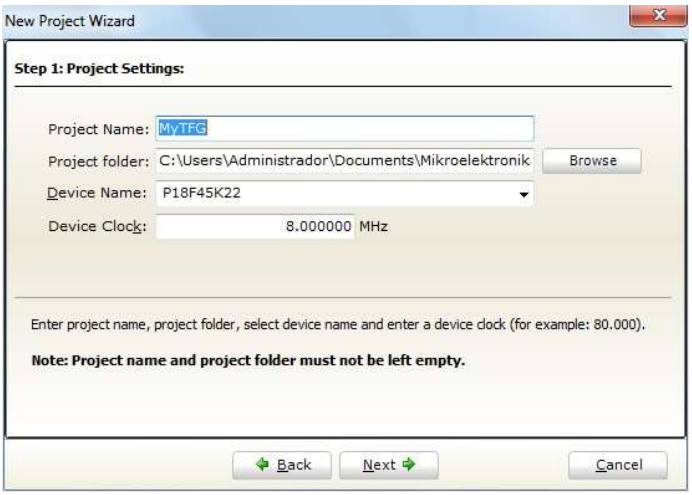


Figura 2.11. Ventana para crear nuevo proyecto en MikroC for Pic

En primer lugar, abrimos el programa MikroC for PIC y seleccionamos la opción de crear nuevo proyecto, lo cual nos creará una carpeta donde se incluyen todos los archivos relacionados con el proyecto, como el archivo main.c donde se encuentra el código principal con las instrucciones que el usuario ha querido implementar, los archivos de cabecera .h los cuales son creados por el usuario en otro fichero con el fin de utilizarlos en otro programa, los archivos .hex los cuales son los que se generan una vez se lleva a cabo la compilación de forma satisfactoria y los vamos a utilizar para pasarlos a la placa via USB para que esta lo interprete y ejecute el programa en los periféricos de la placa. Antes de crearlo, debemos asignarle un nombre al proyecto, la dirección donde se va a guardar, el microcontrolador con el que vamos a trabajar y la frecuencia de reloj.

```
sbit LCD_RS at LATB4_bit;
sbit LCD_EN at LATB5_bit;
sbit LCD_D4 at LATB0_bit;
sbit LCD_D5 at LATB1_bit;
sbit LCD_D6 at LATB2_bit;
sbit LCD_D7 at LATB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;

char txt1[] = "mikroElektronika";
char txt2[] = "EasyPIC7";
char txt3[] = "Lcd4bit";
char txt4[] = "example";

char i;

void Move_Delay() {
  Delay_ms(500);
}

void main(){
  ANSELB = 0;

  Lcd_Init();
  Lcd_Cmd(_LCD_CLEAR);
  Lcd_Cmd(_LCD_CURSOR_OFF);
  Lcd_Out(1,6,txt3);

  Lcd_Out(2,6,txt4);
  Delay_ms(2000);
  Lcd_Cmd(_LCD_CLEAR);

  Lcd_Out(1,1,txt1);
  Lcd_Out(2,5,txt2);

  Delay_ms(2000);

  // Moving text
  for(i=0; i<4; i++) {
    Lcd_Cmd(_LCD_SHIFT_RIGHT);
    Move_Delay();
  }

  while(1) {
    for(i=0; i<8; i++) {
      Lcd_Cmd(_LCD_SHIFT_LEFT);
      Move_Delay();
    }

    for(i=0; i<8; i++) {
      Lcd_Cmd(_LCD_SHIFT_RIGHT);
      Move_Delay();
    }
  }
}
```

Figuras 2.12 y 2.13. Programa prueba para pantalla LCD

Una vez tengamos creado el proyecto, procedemos a hacer el programa básico el cual nos deje claro el correcto funcionamiento de los componentes de la placa, así como la comunicación con el ordenador. Como vemos en las imágenes anteriores, la estructura del programa se basa en la declaración de las conexiones internas que va a tener la pantalla LCD con los pines del microcontrolador, luego encontramos algunas funciones y variables que vamos a utilizar posteriormente pero el programa no la ejecuta sino simplemente las lee y las guarda, y finalmente observamos el main del programa donde se encuentran todas las instrucciones necesarias para alcanzar el objetivo final del programa.

Después de conseguir esto, compilamos el programa seleccionando la opción “Build” y nos aparecerá en la parte inferior los mensajes correspondientes a la compilación. Si nos dice que la compilación se ha realizado con éxito, significa que no se han detectado errores en la lectura del programa pero ahora nos faltaría comprobar si se produce algún fallo de ejecución.

Figura 2.14. Puerto de conexión USB

Figura 2.15. Interfaz del programa mikroProg Suite

Para comprobar si la ejecución del programa se lleva a cabo correctamente, necesitaremos hacer uso del programa mikroProg Suite el cual lo hemos descargado junto con el programa utilizado para la programación, y un cable USB para conectar el ordenador y la placa. Una vez lo tengamos conectado por cable, se nos debería encender el led LINK de la placa. Si esto no ocurre puede ser debido a que el ordenador no tiene los drivers necesarios para reconocer este dispositivo externo. Es este el motivo por el que hemos tenido que utilizar un ordenador antiguo e instalar el Windows 7, ya que los drivers de esta placa van desde el Windows 98, hasta los del sistema operativo que hemos instalado.

Cuando hayamos conseguido que el led se encienda, ya podemos abrir el programa nombrado, seleccionar el microcontrolador con el que se va a comunicar, cargar el archivo .hex desde File > Load Hex y si le damos a “Write”, el programa se escribirá en la placa. Sabremos que la acción se realiza correctamente si se activa el led ACTIVE y el último led de abajo comienza a parpadear. Finalmente, vemos que todo funciona correctamente.

Figura 2.15. Resultado al ejecutar el programa de prueba.

3. Componentes de la placa EasyPic v7

3.1. Microcontrolador PIC18F45K22

Este es un microcontrolador de 8 bits perteneciente a la familia PIC18 de Microchip Technology. Este microcontrolador es ampliamente utilizado ofreciendo un alto rendimiento con bajo coste y utilizando una memoria tipo flash cubriendo variedad de necesidades, incluyendo sistemas de control industrial, automatización del hogar, electrónica de consumo y muchos otros.

Este dispositivo tiene las siguientes características principales:

- Arquitectura adaptada al compilador de C.
- EEPROM de datos a 1024 bytes.
- Operación de hasta 16 MIPS.
- Módulo convertidor de analógico a digital.
- Ancho del bus de datos: 8 bits.
- Tensiones máxima de trabajo: 5,5V.
- Memoria RAM: 1,5 kB.
- Memoria de programa: 32 kB.
- Frecuencia máxima de trabajo: 64 MHz.
- Rango de temperaturas: -40ºC - 85 ºC.

Figura 3.1. Microcontrolador PIC18F45K22.

3.2. Fuente de alimentación

Esta placa contiene una fuente de alimentación conmutada la cual es capaz de crear un voltaje estable y los valores necesarios de corriente para dar energía a cada componente. Para ello se utilizan entre otros componentes, 4 diodos colocados en la configuración de puente de wheatstone, condensadores, resistencias y reguladores de voltaje. La energía puede ser suministrada por cable USB, mediante un cable con el conector adaptado al dispositivo, o haciendo uso de una fuente de alimentación de un laboratorio.

Figura 3.2. Fuente de alimentación de la placa.

3.3. Módulo de E/S analógicas y digitales

Se utilizan fundamentalmente para digitalizar entradas analógicas procedentes de sensores, activar los diodos o conocer el estado de los pulsadores.

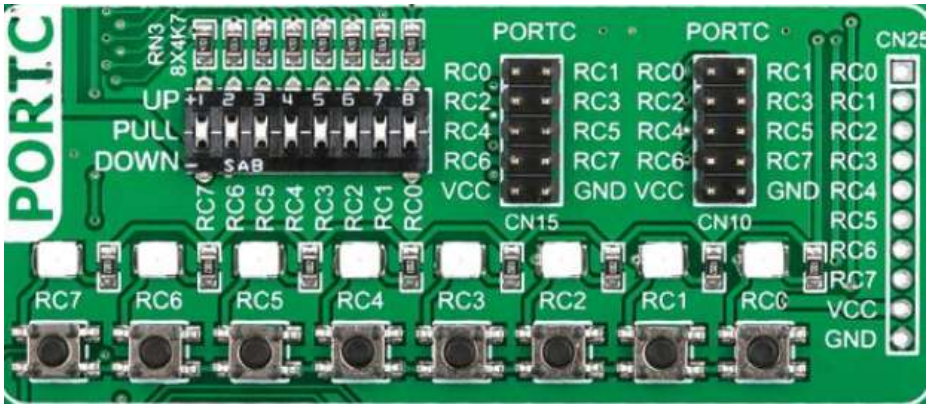


Figura 3.3. Módulo PORTC con sus leds, sus botones y pines.

3.4. Comunicaciones

Tienen gran importancia en cuanto a estos sistemas. Para conectarnos con la placa Easy PIC v7, lo podemos hacer mediante conector RS-232 o USB. En nuestro caso, utilizaremos la conexión mediante “Universal Serial Bus”.

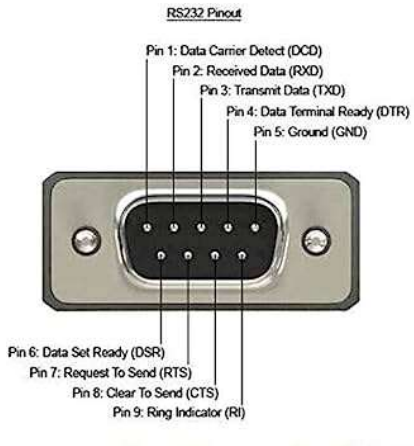


Figura 3.4. Conector RS-232

Figura 3.5. Conector USB 2.0 tipo A

3.5. Programador para USB 2.0

Esta parte de la placa incluye un depurador circuito y nos permite utilizar cualquier dispositivo del tipo PIC10, PIC12, PIC16, PIC18, es decir, soporta más de 350 microcontroladores con un excelente rendimiento y fácil.

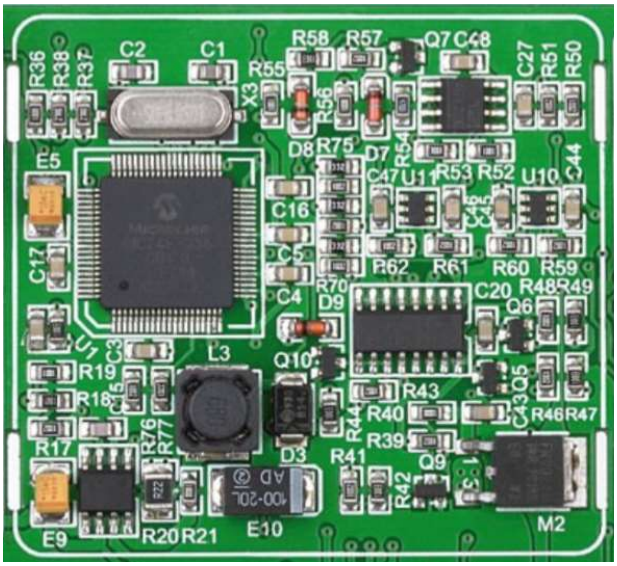


Figura 3.6. mikroProg USB 2.0 programmer - debugger.

3.6. Pantalla LCD:

La “Liquid Crystal Display” o pantalla LCD es una forma de representación de la información de una forma barata y eficaz. A través de ella se va a ejecutar el juego que hemos implementado. Esta pantalla tiene unas dimensiones de 2 filas de texto con 16 columnas de caracteres.

Esta pantalla utiliza 4 bits para transmitir la información de la imagen a la pantalla, lo cual es bueno si se busca reducir la complejidad de la placa. Sin embargo, la desventaja es que la velocidad de actualización de la pantalla puede ser más lenta que en modos de mayor número de bits, ya que se requieren múltiples ciclos de transmisión para enviar toda la información necesaria para mostrar una imagen completa.

La pantalla que vamos a utilizar contiene una serie de pines por los que se conecta con la placa y son los siguientes:

- GND and VCC: líneas de alimentación de la pantalla.
- Vo: nivel de contraste del LCD usando potenciómetro P4.
- RS: señal de selección de registro.
- R/W: indica si la pantalla está en modo lectura o escritura.
- E: línea de activación de la pantalla(“enable”).
- D0-D3: estos 4 bit de datos están conectados a tierra.
- D4-D7: 4 bits de datos por donde la pantalla recibe la información que debe sacar por pantalla.
- LED+ and LED-: conexión del ánodo y cátodo respectivamente del led de retroiluminación.

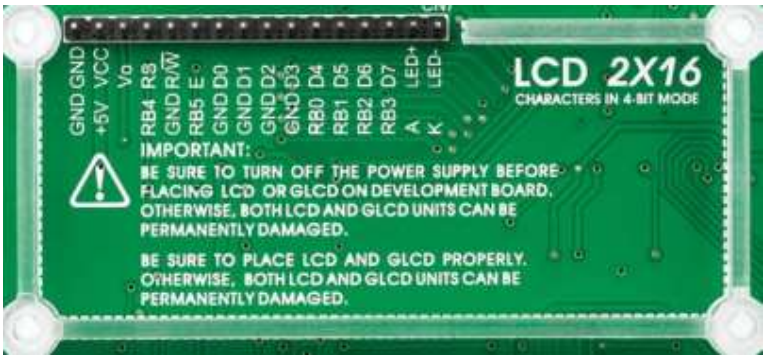


Figura 3.7. Pines de la placa para conectar la pantalla LCD.

4. Juego a implementar

El presente trabajo se centra en el desarrollo de un programa de juego utilizando MikroC for PIC, con el objetivo de implementar un juego tipo "runner" en una pantalla LCD 2x16. El programa se ha diseñado específicamente para la placa Easy PIC v7, la cual proporciona las capacidades necesarias para interactuar con la pantalla y controlar los elementos del juego.

El juego consiste en guiar a un personaje en constante movimiento a través de un entorno desafiante, evitando obstáculos y recolectando puntos para obtener la máxima puntuación posible. El personaje se desplaza horizontalmente en la pantalla, y el jugador debe utilizar los botones de la placa Easy PIC v7 para controlar los movimientos hacia arriba y hacia abajo, evitando colisionar con los obstáculos en su camino.

El resultado es un juego entretenido y desafiante que demuestra el conocimiento y las habilidades adquiridas en la programación de microcontroladores y el desarrollo de software embebido.

4.1. Librerías

Las librerías son un tipo de archivo los cuales incluimos en nuestro programa para poder usar las funciones que se definen en ella. En nuestro caso, el programa MikroC for Pic nos da un acceso a un menú donde podemos elegir las librerías necesarias para la comunicación con los distintos periféricos de la placa además de las funciones básicas para programar en C++.

En nuestro caso, hemos utilizado las librerías más comunes para programar en C, como son “stdlib”, “string”, “math”, etc, incluyendo además las necesarias para comunicarse con la pantalla LCD y para

comunicarse correctamente con los puertos de la placa y sus componentes.

4.2. Archivos de cabecera

Los archivos con extensión .h (del término en inglés “header file”), se incluyen también al principio del programa, como las librerías, y tienen una función muy similar. En nuestro caso tenemos dos archivos de cabecera: personaje1.h y obstaculo1.h.

Estos dos archivos simplemente contienen las declaraciones de las funciones y los parámetros de entrada que pueda recibir. Luego realizamos también un archivo.c con el mismo nombre que el .h, pero definiendo como va a funcionar esa función. A continuación, les muestro los que he utilizado:

En primer lugar, en el archivo obstaculo.h, tenemos simplemente la declaración de la función que vamos a implementar posteriormente en el .c. En este caso la función se va a llamar “obstaculo1” y tendrá como parámetros de entrada la fila (“pos_row”) y la columna (“pos_char”) donde se dibujara el obstáculo.

```
#ifndef OBSTACULO1_H
#define OBSTACULO1_H

void obstaculo1(char pos_row, char pos_char);

#endif
```

Figura 4.1. Obstaculo.h

Luego en el correspondiente .c encontramos la definición de la función que hemos definido anteriormente. Como observamos, en primer lugar tenemos una matriz fila de 8 números, los cuales corresponden a los 8 píxeles que forman la altura de los 32 (pantalla

LCD 2x16) rectángulos de píxeles. Esta matriz la rellenamos con el valor en decimal del número binario que quieras formar en la fila del rectángulo determinada. Muestro un ejemplo para dejarlo más claro:

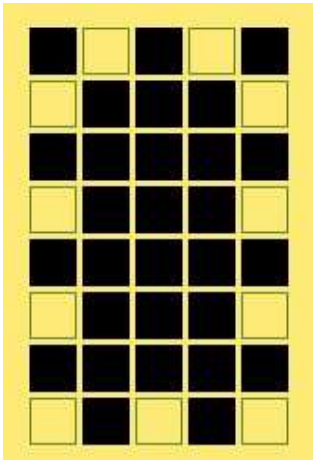


Figura 4.2. Obstáculo del juego.

Los obstáculos de nuestro juego tendrán esta forma. Por lo tanto, para explicar lo anterior, observamos que el primer valor de la matriz fila es 21, que es el resultado de pasar a decimal el número binario 10101, ya que de las primeros 5 pixeles de la fila de arriba están iluminados el primero, el tercero y el quinto. El siguiente es 14 ya que en la segunda fila el número binario correspondiente es 01110, y así sucesivamente hasta la octava fila.

Luego dentro de la función que vamos a crear, comenzamos declarando la variable “i” la cual vamos a utilizar para movernos de fila en el rectángulo donde se vaya a dibujar el obstáculo; después tenemos la sentencia “Lcd_Cmd(80)” donde le decimos la posición de memoria donde almacenar los valores de cada fila; y finalmente el bucle “for” para recorrer las filas del rectángulo con los valores de la matriz y luego pasando también en qué rectángulo se va a dibujar.

```
#include "Obstaculo1.h"

const char matObst1[] = {21,14,31,14,31,14,31,10};

void obstaculo1(char pos_row, char pos_char) {
     char i;
    Lcd_Cmd(80);
    for (i = 0; i<=7; i++) Lcd_Chr_CP(matObst1[i]);
    Lcd_Cmd(_LCD_RETURN_HOME);
    Lcd_Chr(pos_row, pos_char, 2);
}
```

Figura 4.3. Obstáculo.c

Lo mismo pasa en los archivos creados para el personaje, donde obtenemos la declaración de la función “per1Caminando” con sus respectivos parámetros de entrada para definir su posición en la pantalla. Luego en el .c Definimos la matriz de la matriz del personaje y hacemos lo mismo de antes.

```
#ifndef PERSONAJE1_H
#define PERSONAJE1_H

void per1Caminando(char pos_row, char pos_char);

#endif
```

Figura 4.4. Personaje.h

```
#include "personaje1.h"

const char matPer1[] = {14,31,21,31,31,14,10,27};

void per1Caminando(char pos_row, char pos_char) {
  char i;
    Lcd_Cmd(72);
    for (i = 0; i<=7; i++) Lcd_Chr_CP(matPer1[i]);
    Lcd_Cmd(_LCD_RETURN_HOME);
    Lcd_Chr(pos_row, pos_char, 1);
}
```

Figura 4.5. Personaje.c

4.3. Instancias de los pines de la pantalla LCD

Los pines de la pantalla LCD están explicados en el apartado 3.6 de este trabajo. Lo primero que debemos indicar en el programa que vayamos a realizar es la dirección que van a tener estos pines. En nuestro caso, los vamos a conectar a ciertos pines del microcontrolador para que este pueda controlar el periférico utilizado.

Como observamos en la figura 4.6 , en el primer grupo de declaraciones le decimos a la placa que puerto es capaz de controlar la lectura o escritura del correspondiente pin de la pantalla. Por otro lado, en el segundo párrafo de declaraciones, decidimos qué dirección van a tener esos datos que queremos leer o escribir.

```
sbit LCD_RS at LATB4_bit;
sbit LCD_EN at LATB5_bit;
sbit LCD_D4 at LATB0_bit;
sbit LCD_D5 at LATB1_bit;
sbit LCD_D6 at LATB2_bit;
sbit LCD_D7 at LATB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
```

Figura 4.6. Instancias de la pantalla en la placa.

4.4. Funciones implementadas

Para realizar un programa en, en primer lugar nos debemos preguntar, ¿cuál es el objetivo de este programa?. En nuestro caso, es el de diseñar un juego para pantalla LCD lo más entretenido posible. Una vez lo sepamos, debemos dividir el trabajo de hacer el juego, en subtareas que implementadas de una forma adecuada hacen que se ejecute el juego por la pantalla LCD.

Estas soluciones que debemos diseñar para cada subtarea la llamamos funciones y es uno de los elementos básicos de la programación en C++ y en la programación estructurada en general, junto con las estructuras de control, como bucles, condiciones, etc. En nuestro caso hemos hecho uso de las siguientes funciones:

4.4.1. Dibujar al personaje

El primero problema que se nos viene a la cabeza es el de crear el personaje, ya que es el protagonista de nuestro juego, y que salga en la pantalla en la posición deseada. Además, este personaje debe “saltar” a la fila superior de la pantalla cuando el jugador presione el pulsador RD6 de la placa de desarrollo, y volver a la fila inferior cuando se suelte dicho botón.

Lo que es el diseño del personaje ya hemos visto cómo se implementa en el apartado 4.2. de este proyecto donde se habla de los archivos de cabecera. Como esa función que hemos diseñado ya nos dibuja el personaje en pantalla en la fila y columna que le pasemos, simplemente es llamar a la función “per1Caminando” y borrar el personaje de la posición anterior.

4.4.2. Generación de obstáculos

En siguiente lugar, como todo juego tipo “runner”, debemos pensar cómo va a ser la generación de obstáculos. Este es un aspecto muy importante, ya que a parte de la complejidad del diseño, es un aspecto fundamental para que el juego pueda resultar entretenido y tenga una dificultad considerable.

En primer lugar, he intentado hacerlo de manera que se generen nuevos obstáculos cada vez que uno llegara al punto inicial, pero esto no funcionaba debido a la descoordinación y salían juntos sin ningún tipo de escapatorio. Luego me he dado cuenta que era mejor si definía previamente el número de obstáculos que iba a utilizar. Cuando mayor sea el nivel mayor será el número de obstáculos y la velocidad de estos. Estos comenzarán a aparecer en la pantalla a partir de la columna 12, ya que a partir de esta hasta la 16 la hemos reservado para mostrar la puntuación de la partida y el récord. Para producir el efecto de que se desplacen los obstáculos, le quitamos 1 al valor de la columna correspondiente y borramos la columna de la posición anterior.

También debemos aplicar un delay el cual va a ser nuestro parámetro de entrada a la función y definirá a qué velocidad se van a desplazar los elementos a esquivar.

Tras hacer varias pruebas del juego, he decidido hacer dos funciones para generar los obstáculos. Por un lado, una función donde los obstáculos saldrán cada tres casillas de la pantalla, por lo que la usaremos para el nivel inicial del juego. Luego hacemos otra para que estos salgan cada 2 casillas, por lo que dificulta el avance a través de ellas. Juntando este cambio con el aumento de la velocidad, conseguimos que el juego sea cada vez más difícil según se vaya avanzando.

4.4.3. Detectar colisión personaje - obstáculo

Está función es más sencilla de implementar. Al principio no he podido crearla debido a que los valores de las filas y columnas de los obstáculos no tenían una variable asignada, por lo que no podía igualar estos dos valores de los obstáculos con los del personaje. Una vez hayamos cambiado esto, generamos el código de la función correctamente, pasando como parámetros de entrada las filas y columnas correspondientes al personaje y a los obstáculos.

4.4.4. Mostrar puntuación actual y record

Una de las partes principales de cualquier juego, es el de la puntuación que llevas en cada momento. Para ello, hemos creado la variable “contadorObstaculos” a la cual le vamos a sumar 1 cada vez que uno de estos llegue a la posición 0 de la pantalla. Con esta función y con las 3 anteriores ya somos capaces de mostrar por pantalla la interfaz que va a haber cuando estemos en medio de una partida.

También tenemos que tener en cuenta que, como nuestras variables “contadorObstaculos” y “record” son de tipo short, al sacarla por pantalla no salen los valores numéricos que deberían, sino que en si lugar salen letras y algún símbolo. Para solucionar este problema tenemos que pasar esta variable short a un array de caracteres mediante una función de conversión que se encuentra en una de las librerías incluidas.

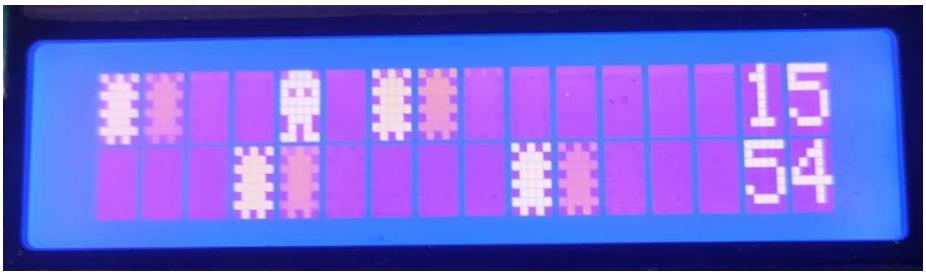


Figura 4.7. Interfaz de partida.

En la figura 4.7. podemos ver como se verá el juego cuando estemos en medio de una partida. Por un lado podemos ver al personaje que actualmente se encuentra esquivando uno de los obstáculos. También observamos las mismas barreras que se desplazan hacia la izquierda y que, en este caso el la función del nivel principal, y lo sabemos porque los obstáculos salen cada 3 casillas.Finalmente, vemos también en el lado derecho dos valores. El que está en la parte superior se encarga de informar al jugador de la puntuación que lleva. El de la parte inferior es el record de la partida, el cual se almacena en la memoria EEPROM con la función que explicamos en el apartado 4.4.6. con el fin de que se quede guardado aunque la placa se apague por completo.

4.4.5. Inicio del juego

Con las tres funciones anteriores ya podemos darnos cuenta de si el juego marcha bien o no. A continuación, vamos a crear una función la cual nos muestre un mensaje por pantalla, indicándonos el nombre del juego y que botón debemos apretar para empezar a jugar. Además he añadido un efecto de desplazamiento en el mensaje mientras aparece el personaje aparece detrás y a partir de ahí comienzan a aparecer los obstáculos.

Figura 4.8. Imagen de inicio del juego.

4.4.6. Guardar récord en memoria EEPROM

Esta función la podemos implementar utilizando la memoria de este tipo que contiene el microcontrolador utilizado. Procedente de las siglas “Electrically Erasable Programmable Read-Only Memory” es un tipo de memoria no volátil, es decir, que son capaces de almacenar datos en ella sin la necesidad de energía adicional. Este dispositivo de almacenamiento tiene un coste elevado, razón por la cual simplemente tiene implementados en el microprocesador 1024 bytes.

En primer lugar he intentado hacer uso de las funciones que incluye la librería EEPROM que contiene el programa, pero no he conseguido que funcionen de ninguna manera. Tras investigar alguna forma para utilizar esta memoria, he encontrado la forma de hacer unas funciones, una de lectura y otra de lectura, como las anteriores que hemos creado, pero con las instrucciones necesarias para que se lea o se escriba en la EEPROM. Esta función la he diseñado a partir de la tabla 3-7. de la datasheet del microcontrolador incluido en el apartado 6 de este proyecto.

TABLE 3-7: PROGRAMMING DATA MEMORY

4-bit Command	Data Payload	Core Instruction
Step 1: Direct access to data EEPROM.		
0000	9E A6	BCF EECON1, EEPGD
0000	9C A6	BCF EECON1, CFGS
Step 2: Set the data EEPROM Address Pointer.		
0000	0E <Addr>	MOVLW <Addr>
0000	6E A9	MOVWF EEADR
0000	0E <AddrH>	MOVLW <AddrH>
0000	6E AA	MOVWF EEADRH
Step 3: Load the data to be written.		
0000	0E <Data>	MOVLW <Data>
0000	6E A8	MOVWF EEDATA
Step 4: Enable memory writes.		
0000	84 A6	BSF EECON1, WREN
Step 5: Initiate write.		
0000	82 A6	BSF EECON1, WR
0000	00 00	NOP
0000	00 00	NOP write starts on 4th clock of this instruction
Step 6: Poll WR bit, repeat until the bit is clear.		
0000	50 A6	MOVF EECON1, W, 0
0000	6E F5	MOVWF TABLAT
0000	00 00	NOP
0010	<MSB><LSB>	Shift out data(1)
Step 7: Hold PGC low for time P10.		
Step 8: Disable writes.		
0000	94 A6	BCF EECON1, WREN
Repeat steps 2 through 8 to write more data.		

Note 1: See Figure 4-4 for details on shift out data timing.

Figura 4.7. Comandos para escribir en EEPROM.

4.5. Función principal o “main”

En este bloque del programa es donde vamos a combinar de manera adecuada todas las funciones creadas anteriormente para conseguir el objetivo deseado. Esta parte del programa es la encargada de llevar el control de todos los datos realizando las operaciones, asignaciones, comparaciones, etc que haya implementados en el o en alguna función a la que se llame dentro de este.

En nuestro caso, comenzamos utilizando una serie de comandos para preparar la placa, como por ejemplo establecer las salidas de los puerto del microprocesador como digitales, y para preparar también la pantalla, como la función para inicializar la pantalla, leer el récord de la EEPROM, y “limpiar” la pantalla para empezar a sacar lo que queramos mostrar. Luego asignamos valores adecuados de fila y columna a los obstáculos para que luego cuando se muevan no haya conflicto entre ellos.

Una vez tenemos todo preparado, llamamos a la función que nos muestra por pantalla la interfaz inicial, y comenzamos a abrir un bucle while donde el programa se va a quedar en la mecánica del juego hasta que tenga lugar uno de estos dos sucesos:

1. El personaje choque con una barrera, momento en el cual sacamos un mensaje por pantalla comunicando al usuario que ha perdido y cuál es el botón que hay que apretar para volver a jugar de nuevo.

2. Se alcance la puntuación de 40, momento en el cual se muestra una pantalla como que se ha superado el nivel y el pulsador correspondiente para continuar la partida.

Si el jugador pasa de nivel, deberá apretar el botón D5 para pasar al nivel definitivo. El nivel donde sabremos hasta dónde eres capaz de llegar. En el utilizaremos la segunda función de obstáculos para que

salgan mas juntos y, según se va avanzando en la partida, la velocidad irá aumentando. ¿Serás capaz de lograr la máxima puntuación?

Para ejecutar este nivel, el bucle es muy similar al del nivel principal, pero con la diferencia de que si llegas a una puntuación de 120, el se muestra que has superado el luego y te da la opción de volver a empezar. Si tiene lugar una colisión, también se le dará la opción de volver al inicio.

En el momento en el que el juego se interrumpe, se entra en un bucle donde el juego espera a que el usuario presione D5 para pasar de nivel o D7 para volver a jugar de nuevo. Para que no haya conflicto, por ejemplo cuando haya una colisión y aprietes D5 y el juego pase de nivel, hemos definido una variable llamada condición con el fin de que solo puedas apretar un botón y el otro no funcione cuando no debe.

Finalmente, para controlar que el récord cambie cuando sea superado y se escriba en la memoria EEPROM, abrimos una condición dentro del bucle donde llamamos a la función creada para escribir en este tipo de memoria de almacenamiento no volátil, y actualizamos también el valor del récord.

4.6. Programación del juego

```
#include "Obstaculo1.h"
#include "personaje1.h"

// Instancias de los pines de la pantalla
sbit LCD_RS at LATB4_bit;
sbit LCD_EN at LATB5_bit;
sbit LCD_D4 at LATB0_bit;
sbit LCD_D5 at LATB1_bit;
sbit LCD_D6 at LATB2_bit;
sbit LCD_D7 at LATB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;

//Declaración de variables del juego
unsigned char colPersonaje = 5;
unsigned char filPersonaje = 2;
unsigned char colObstaculo1;
unsigned char colObstaculo2;
unsigned char colObstaculo3;
unsigned char colObstaculo4;
unsigned char colObstaculo5;
unsigned char colObstaculo6;
unsigned char filObstaculo1;
unsigned char filObstaculo2;
unsigned char filObstaculo3;
unsigned char filObstaculo4;
```

```
unsigned char filObstaculo5;
unsigned char filObstaculo6;
unsigned short contadorObstaculos = 0;
unsigned char delayMs = 150;
unsigned short record = 0;
unsigned char variableCondicion = 0;

//Función para generar un numero aleatorio entre min y max
unsigned int random(unsigned int min, unsigned int max) {

    unsigned int result;
    unsigned int range = max - min + 1;

    result = rand() % range + min;

    return result;
}

//Función para que el personaje salte y vuelva a caer
void moverPersonaje() {

        if (Button(&PORTD, 6, 1, 1)) {

            Lcd_Chr(2, colPersonaje, ' ');
            filPersonaje = 1;
            per1Caminando(filPersonaje, colPersonaje);

        } else if (Button(&PORTD, 6, 1, 0)) {

            Lcd_Chr(1, colPersonaje, ' ');
            filPersonaje = 2;
```

```
            per1Caminando(filPersonaje, colPersonaje);


        }
}

//Función para
void moverObstaculos(unsigned char delayMs){

    if(colObstaculo1 <= 12){
    Obstaculo1(filObstaculo1, colObstaculo1);
    }
    if(colObstaculo2 <= 12){
    Obstaculo1(filObstaculo2, colObstaculo2);
    }
    if(colObstaculo3 <= 12){
    Obstaculo1(filObstaculo3, colObstaculo3);
    }
     if(colObstaculo4 <= 12){
    Obstaculo1(filObstaculo4, colObstaculo4);
    }

     Vdelay_ms(delayMs);

    Lcd_Chr(filObstaculo1, colObstaculo1, ' ');
    Lcd_Chr(filObstaculo2, colObstaculo2, ' ');
    Lcd_Chr(filObstaculo3, colObstaculo3, ' ');
    Lcd_Chr(filObstaculo4, colObstaculo4, ' ');


    colObstaculo1--;
    colObstaculo2--;
```

```
        colObstaculo3--;
        colObstaculo4--;

        if(colObstaculo1 == 0){

                colObstaculo1 = 12;
                filObstaculo1 = random(1, 2);
                //contadorObstaculos++;

        }
        if(colObstaculo2 == 0){

                colObstaculo2 = 12;
                filObstaculo2 = random(1, 2);
                contadorObstaculos++;

        }
        if(colObstaculo3 == 0){

                colObstaculo3 = 12;
                filObstaculo3 = random(1, 2);
                //contadorObstaculos++;

        }

        if(colObstaculo4 == 0){

                colObstaculo4 = 12;
                filObstaculo4 = random(1, 2);
                contadorObstaculos++;
```

```
    }

}

void moverObstaculos2(unsigned char delayMs){

    if(colObstaculo1 <= 12){
    Obstaculo1(filObstaculo1, colObstaculo1);
    }
    if(colObstaculo2 <= 12){
    Obstaculo1(filObstaculo2, colObstaculo2);
    }
    if(colObstaculo3 <= 12){
    Obstaculo1(filObstaculo3, colObstaculo3);
    }
    if(colObstaculo4 <= 12){
    Obstaculo1(filObstaculo4, colObstaculo4);
    }
    if(colObstaculo5 <= 12){
    Obstaculo1(filObstaculo5, colObstaculo5);
    }
    if(colObstaculo6 <= 12){
    Obstaculo1(filObstaculo6, colObstaculo6);
    }

    Vdelay_ms(delayMs);

    Lcd_Chr(filObstaculo1, colObstaculo1, ' ');
    Lcd_Chr(filObstaculo2, colObstaculo2, ' ');
    Lcd_Chr(filObstaculo3, colObstaculo3, ' ');
    Lcd_Chr(filObstaculo4, colObstaculo4, ' ');
```

```
    Lcd_Chr(filObstaculo5, colObstaculo5, ' ');
    Lcd_Chr(filObstaculo6, colObstaculo6, ' ');

    colObstaculo1--;
    colObstaculo2--;
    colObstaculo3--;
    colObstaculo4--;
    colObstaculo5--;
    colObstaculo6--;

    if(colObstaculo1 == 0){

            colObstaculo1 = 12;
            filObstaculo1 = random(1, 2);
            //contadorObstaculos++;

    }
    if(colObstaculo2 == 0){

            colObstaculo2 = 12;
            filObstaculo2 = random(1, 2);
            contadorObstaculos++;

    }
    if(colObstaculo3 == 0){

            colObstaculo3 = 12;
            filObstaculo3 = random(1, 2);
            //contadorObstaculos++;

    }
```

```
    if(colObstaculo4 == 0){

                colObstaculo4 = 12;
                filObstaculo4 = random(1, 2);
                contadorObstaculos++;


     }
     if(colObstaculo5 == 0){

                colObstaculo5 = 12;
                filObstaculo5 = random(1, 2);
               // contadorObstaculos++;

     }
     if(colObstaculo6 == 0){

                colObstaculo6 = 12;
                filObstaculo6 = random(1, 2);
                contadorObstaculos++;


     }

}

void mostrarPuntuacion(){

     char txt[4];
     char txt2[4];
     ShortToStr(contadorObstaculos, txt);
     Lcd_Out(1, 13, txt);
     ShortToStr(record, txt2);
```

```
    Lcd_Out(2, 13, txt2);

}
void inicioJuego(){

    char txt1[] = "LCD Epic Escape!";
    char txt2[] = "D6 para empezar";
    unsigned char i;

    int colInicial = 1;
    int colInicial2 = 1;
    Lcd_Out(1, colInicial, txt1);
    Lcd_Out(2, colInicial2, txt2);

    while(1){

        if(Button(&PORTD, 6, 1, 1)){

            for(i=0; i<11; i++) {

                Lcd_Chr(1, colInicial - 1, ' ');
                Lcd_Chr(2, colInicial2 - 1, ' ');
                Lcd_Out(1, colInicial++, txt1);
                Lcd_Out(2, colInicial2++, txt2);
                Delay_ms(300);
            }
            for(i=0; i<6; i++) {

                Lcd_Chr(2, i - 1, ' ');
                per1Caminando(2, i);
                Lcd_Chr(1, colInicial - 1, ' ');
```

```
                    Lcd_Chr(2, colInicial2 - 1, ' ');
250                 Lcd_Out(1, colInicial++, txt1);
                    Lcd_Out(2, colInicial2++, txt2);
                    Delay_ms(300);
                }
                break;
            }

        }
    }

260 unsigned char detectarColision(unsigned char filPersonaje,
                    unsigned char colPersonaje,
                    unsigned char filObstaculo, unsigned char colObstaculo) {
    if (filPersonaje == filObstaculo && colPersonaje == colObstaculo) {
        return 1;  // Colisión detectada
    } else {
        return 0;  // No hay colisión
    }
}

270 //Función para subir la velocidad de los obstaculos
void subirNivel(short puntuacion){

    if(puntuacion == 10){
    delayMs = 40;
    }
    if(puntuacion == 30){
    delayMs = 10;
    }
    if(puntuacion == 45){
```

```
280         delayMs = 10;
            }
            if(puntuacion == 50){
            delayMs = 30;
            }
            if(puntuacion == 60){
            delayMs = 10;
            }
            if(puntuacion == 70){
            delayMs = 0.1;
290         }
            if(puntuacion == 90){
            delayMs = 0.001;
            }

        }

        //Funcion para guardar record en la memoria EEPROM
        void escribirEeprom(int direccion, short dato) {

300         while (EECON1.EEPGD);//Esperar final de cualquier escritura anterior

            EEADR = direccion;
            EEDATA = dato;
            EECON1.EEPGD = 0;
            EECON1.CFGS = 0;
            EECON1.WREN = 1;

            INTCON.GIE = 0; // Deshabilitar las interrupciones globales

310         EECON2 = 0x55;
```

```
    EECON2 = 0xAA;

    EECON1.WR = 1; // Iniciar la operación de escritura

    EECON1.WREN = 0; // Deshabilitar la escritura en EEPROM

    while (EECON1.WR); // Esperar a que se complete la escritura

    INTCON.GIE = 1; // Habilitar las interrupciones globales
}

//Funcion para leer de la mamoria EEPROM
short leerEeprom(int direccion){

      EEADR = direccion;
      EECON1.EEPGD = 0;
      EECON1.RD = 1;
      return EEDATA;

}

//Función principal
void main() {

    record = leerEeprom(0x3F);  //Leer valor de record
    Lcd_Init();   // Inicialización del LCD
    Lcd_Cmd(_LCD_CLEAR);
    Lcd_Cmd(_LCD_CURSOR_OFF);
    ANSELD = 0;
    C1ON_bit = 0; // Disable comparators
    C2ON_bit = 0;
```

```
    TRISD7_bit = 1;

    Inicio:
    colObstaculo1 = 13;
    colObstaculo2 = 16;
    colObstaculo3 = 19;
    colObstaculo4 = 22;
    filObstaculo1 = 1;
    filObstaculo2 = 2;
    filObstaculo3 = 1;
    filObstaculo4 = 2;
    contadorObstaculos = 0;
    delayMs = 80;

    inicioJuego();

    while(1){

    moverObstaculos(delayMs);
    moverPersonaje();
    mostrarPuntuacion();
    subirNivel(contadorObstaculos);

if(detectarColision(filPersonaje, colPersonaje, filObstaculo1, colObstaculo1) ||
  detectarColision(filPersonaje, colPersonaje, filObstaculo2, colObstaculo2) ||
  detectarColision(filPersonaje, colPersonaje, filObstaculo3, colObstaculo3) ||
  detectarColision(filPersonaje, colPersonaje, filObstaculo4, colObstaculo4)) {
            Lcd_Cmd(_LCD_CLEAR);
            Lcd_Out(1, 2, "Has Perdido :(");
            Lcd_Out(2, 1, "D7 para empezar");
            variableCondicion = 1;
```

```
            break;
        }
        if(contadorObstaculos == 40){
            Lcd_Cmd(_LCD_CLEAR);
            Lcd_Out(1, 1, "Nivel superado:)");
            Lcd_Out(2, 3, "D5 para lvl2 ");
            break;
        }
        if(contadorObstaculos > record){
            escribirEeprom(0x3F, record);
            record = contadorObstaculos;
            }

    }

    while(1){

        if(Button(&PORTD, 5, 1, 1) && variableCondicion == 0){

            Lcd_Cmd(_LCD_CLEAR);
            break;

        }
        if(Button(&PORTD, 7, 1, 1) && variableCondicion == 1){

            Lcd_Cmd(_LCD_CLEAR);
            variableCondicion = 0;
            goto Inicio;
        }
    }
```

```
    colObstaculo1 = 14;
    colObstaculo2 = 16;
    colObstaculo3 = 18;
    colObstaculo4 = 20;
    colObstaculo5 = 22;
    colObstaculo6 = 24;
    filObstaculo1 = 1;
    filObstaculo2 = 2;
    filObstaculo3 = 1;
    filObstaculo4 = 2;
    filObstaculo5 = 1;
    filObstaculo6 = 2;
    delayMs = 70;

    while(1){

    moverObstaculos2(delayMs);
    moverPersonaje();
    mostrarPuntuacion();
    subirNivel(contadorObstaculos);

if(detectarColision(filPersonaje, colPersonaje, filObstaculo1, colObstaculo1) ||
detectarColision(filPersonaje, colPersonaje, filObstaculo2, colObstaculo2) ||
detectarColision(filPersonaje, colPersonaje, filObstaculo3, colObstaculo3) ||
detectarColision(filPersonaje, colPersonaje, filObstaculo4, colObstaculo4)) {
            Lcd_Cmd(_LCD_CLEAR);
            Lcd_Out(1, 2, "Has Perdido :(");
            Lcd_Out(2, 1, "D7 para empezar");
            break;
          }
```

```
detectarColision(filPersonaje, colPersonaje, filObstaculo2, colObstaculo2) ||
detectarColision(filPersonaje, colPersonaje, filObstaculo3, colObstaculo3) ||
detectarColision(filPersonaje, colPersonaje, filObstaculo4, colObstaculo4)) {
                Lcd_Cmd(_LCD_CLEAR);
                Lcd_Out(1, 2, "Has Perdido :(");
                Lcd_Out(2, 1, "D7 para empezar");
                break;
            }

      if(contadorObstaculos == 120){
                Lcd_Cmd(_LCD_CLEAR);
                Lcd_Out(1, 1, "Juego superado:)");
                Lcd_Out(2, 3, "D5 para volver ");
                break;

                }

       if(contadorObstaculos > record){
                escribirEeprom(0x3F, contadorObstaculos);
                record = contadorObstaculos;
                }
    }
    while(1){

        if(Button(&PORTD, 7, 1, 1)){

            Lcd_Cmd(_LCD_CLEAR);
            goto Inicio;
        }
    }
}
```

5. Conclusiones:

5.1. Posibles mejoras:

En cuanto a la programación del juego, este ha sido diseñado con una complejidad acorde con el de la pantalla LCD, aunque todavía se han podido realizar alguna mejora, la cual no ha sido posible debido a que, a la hora de la compilación, nos salta un error debido a que sobrepasamos la versión demo del programa que estoy utilizando. Algunas de estas mejoras son:

1. Efectos de sonido cuando se pasa obstáculos o al terminar un nivel: esto lo podemos hacer debido a que la placa de desarrollo contiene además un zumbador piezoeléctrico el cual, llamando a las correspondientes funciones, es capaz de inicializar el dispositivo de sonido y para emitir un cierto sonido con una frecuencia y un duración.

2. Efecto de movimiento en el personaje al saltar: esta mejora, a parte de que no la podemos implementar debido al demo limit, cuando tenía menos programa hecho intente hacerlo, pero el problema era que al hacer el efecto de agacharse y saltar, se produce un delay en los obstáculos, el cual era necesario para poder visualizar con el ojo humano el efecto del salto.

5.2. Final conclusion:

In conclusion, this Final Degree Project has been an opportunity to apply the knowledge acquired during my academic training and develop a practical project that combines C++ programming, the installation of operating systems and virtual machines, and even the soldering of some electronic components.

This work represents a personal and academic achievement that evidences my ability to tackle complex technical projects, apply acquired knowledge and demonstrate competence in the field of electronic engineering and microcontroller programming.

5.3. Conclusión final

En conclusión, este Trabajo Fin de Grado ha sido una oportunidad para aplicar los conocimientos adquiridos durante mi formación académica y desarrollar un proyecto práctico que combina la programación en C++, la instalación de sistemas operativos y máquinas virtuales, e incluso la soldadura de algunos componentes electrónicos.

Este trabajo representa un logro personal y académico que evidencia mi capacidad para abordar proyectos técnicos complejos, aplicar los conocimientos adquiridos y demostrar competencia en el campo de la ingeniería electrónica y la programación de microcontroladores.

6. Bibliografía:

https://download.mikroe.com/documents/full-featured-boards/easy/easypic-v7/easypic-v7-manual-v104c.pdf

https://github.com/olikraus/u8g2

https://www.microchip.com/en-us/tools-resources/develop/microchip-studio

https://www.alldatasheet.com/view.jsp?Searchword=Pic18f45k22%20datasheet&gclid=Cj0KCQjwnrmlBhDHARIsADJ5b_mfEoRA75zTaNRZdjQ5X_f5bOwanWaMoysuU97SbPWd0S-PXNoQK8QaAhjUEALw_wcB

7. Datasheet microcontrolador:

MICROCHIP

PIC18(L)F2XK22/4XK22

Flash Memory Programming Specification

1.0 DEVICE OVERVIEW

This document includes the programming specifications for the following devices:

- PIC18F23K22
- PIC18LF23K22
- PIC18F24K22
- PIC18LF24K22
- PIC18F25K22
- PIC18LF25K22
- PIC18F26K22
- PIC18LF26K22
- PIC18F43K22
- PIC18LF43K22
- PIC18F44K22
- PIC18LF44K22
- PIC18F45K22
- PIC18LF45K22
- PIC18F46K22
- PIC18LF46K22

2.0 PROGRAMMING OVERVIEW

The PIC18(L)F2XK22/4XK22 devices can be programmed using either the high-voltage In-Circuit Serial Programming™ (ICSP™) method or the low-voltage ICSP method. Both methods can be done with the device in the users' system. The low-voltage ICSP method is slightly different than the high-voltage method and these differences are noted where applicable. This programming specification applies to the PIC18(L)F2XK22/4XK22 devices in all package types.

2.1 Hardware Requirements

In High-Voltage ICSP mode, the PIC18(L)F2XK22/4XK22 devices require two programmable power supplies: one for VDD and one for MCLR/VPP/RE3. Both supplies should have a minimum resolution of 0.25V. Refer to **Section 6.0 "AC/DC Characteristics Timing Requirements for Program/Verify Test Mode"** for additional information.

2.1.1 LOW-VOLTAGE ICSP PROGRAMMING

In Low-Voltage ICSP mode, the PIC18(L)F2XK22/4XK22 devices can be programmed using a single VDD source in the operating range. The MCLR/VPP/RE3 does not have to be brought to a different voltage, but can instead be left at the normal operating voltage. Refer to **Section 2.6 "Entering and Exiting Low-Voltage ICSP Program/Verify Mode"** for additional hardware parameters.

Note 1: The High-Voltage ICSP mode is always available, regardless of the state of the LVP bit, by applying VIHH to the MCLR/VPP/RE3 pin.

2: While in Low-Voltage ICSP mode, MCLR is always enabled, regardless of the MCLRE bit, and the RE3 pin can no longer be used as a general purpose input.

 Advance Information

PIC18(L)F2XK22/4XK22

2.2 Pin Diagrams

The pin diagrams for the PIC18(L)F2XK22/4XK22 family are shown in Figures 2-1 through 2-5.

TABLE 2-1: PIN DESCRIPTIONS (DURING PROGRAMMING): PIC18(L)F2XK22/4XK22

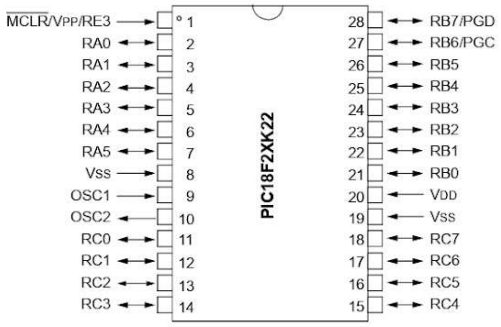
Pin Name	During Programming		
	Pin Name	Pin Type	Pin Description
$\overline{MCLR}$/VPP/RE3	VPP	P	Programming Enable
VDD(1)	VDD	P	Power Supply
VSS(1)	VSS	P	Ground
RB6	PGC	I	Serial Clock
RB7	PGD	I/O	Serial Data

Legend: I = Input, O = Output, P = Power

Note 1: All power supply (VDD) and ground (VSS) pins must be connected.

FIGURE 2-1: 28-PIN SDIP, SSOP AND SOIC PIN DIAGRAMS

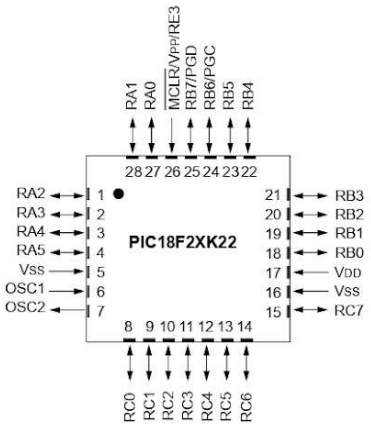
SDIP, SSOP, SOIC (300 MIL)



Note: The following devices are included in 28-pin SDIP, SSOP and SOIC parts: PIC18F23K22, PIC18LF23K22, PIC18F24K22, PIC18LF24K22, PIC18F25K22, PIC18LF25K22, PIC18F26K22, PIC18LF26K22.

FIGURE 2-2: 28-PIN QFN AND UQFN PIN DIAGRAMS

28-Pin QFN, UQFN

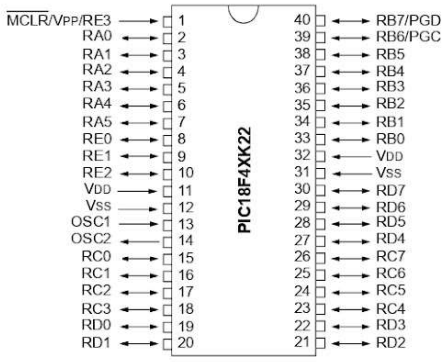


Note 1: The following devices are included in 28-pin QFN parts: PIC18F23K22, PIC18LF23K22, PIC18F24K22, PIC18LF24K22, PIC18F25K22, PIC18LF25K22, PIC18F26K22, PIC18LF26K22.

2: The following devices are included in 28-pin UQFN parts: PIC18F23K22, PIC18LF23K22, PIC18F24K22, PIC18LF24K22.

FIGURE 2-3: 40-PIN PDIP PIN DIAGRAMS

40-PIN PDIP (600 MIL)

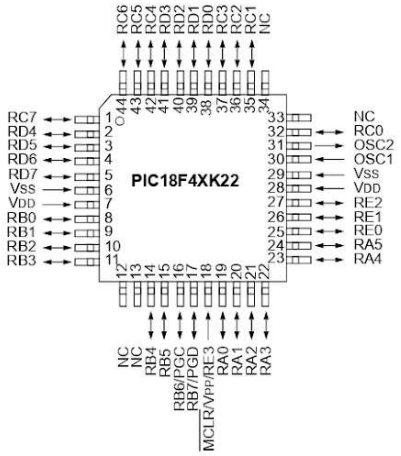


Note: The following devices are included in 40-pin PDIP parts: PIC18F43K22, PIC18LF43K22, PIC18F44K22, PIC18LF44K22, PIC18F45K22, PIC18LF45K22, PIC18F46K22, PIC18LF46K22.

PIC18(L)F2XK22/4XK22

FIGURE 2-4: 44-PIN TQFP PIN DIAGRAMS

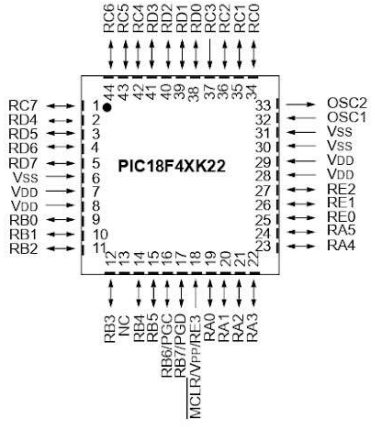
44-PIN TQFP



Note: The following devices are included in 44-pin TQFP parts: PIC18F43K22, PIC18LF43K22, PIC18F44K22, PIC18LF44K22, PIC18F45K22, PIC18LF45K22, PIC18F46K22, PIC18LF46K22.

FIGURE 2-5: 44-PIN QFN PIN DIAGRAMS

44-PIN QFN



Note: The following devices are included in 44-pin QFN parts: PIC18F43K22, PIC18LF43K22, PIC18F44K22, PIC18LF44K22, PIC18F45K22, PIC18LF45K22, PIC18F46K22, PIC18LF46K22.

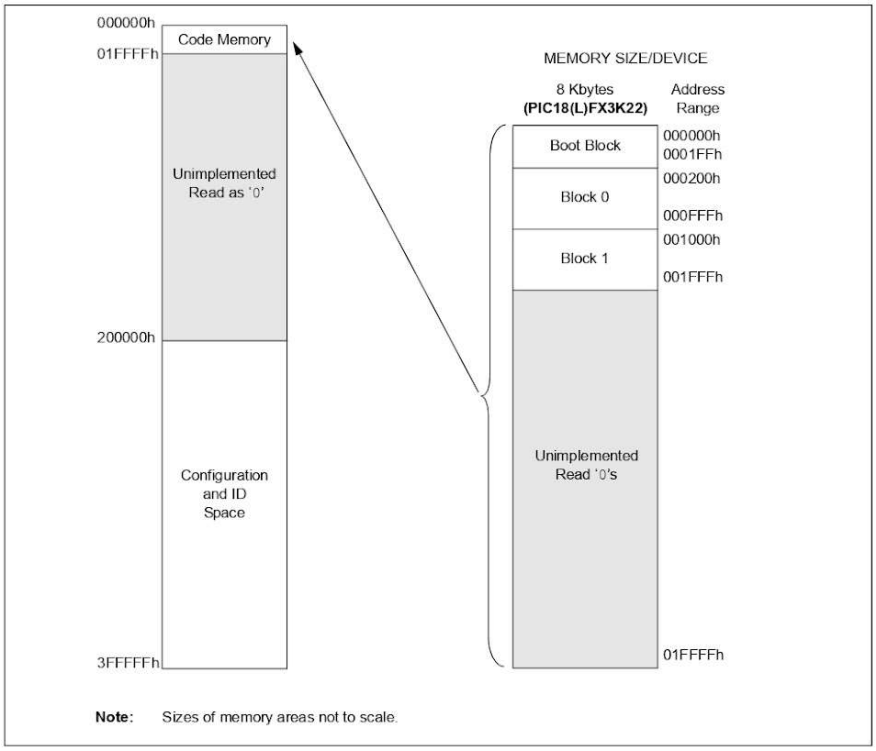
2.3 Memory Maps

For PIC18(L)FX3K22 devices, the code memory space extends from 000000h to 001FFFh (8 Kbytes) in two 4-Kbyte blocks. Addresses 000000h through 0001FFh, however, define a "Boot Block" region that is treated separately from Block 0. All of these blocks define code protection boundaries within the code memory space.

TABLE 2-2: IMPLEMENTATION OF CODE MEMORY

Device	Code Memory Size (Bytes)
PIC18F23K22	000000h-001FFFh (8K)
PIC18LF23K22	
PIC18F43K22	
PIC18LF43K22	

FIGURE 2-6: MEMORY MAP AND THE CODE MEMORY SPACE FOR PIC18(L)FX3K22 DEVICES



Note: Sizes of memory areas not to scale.

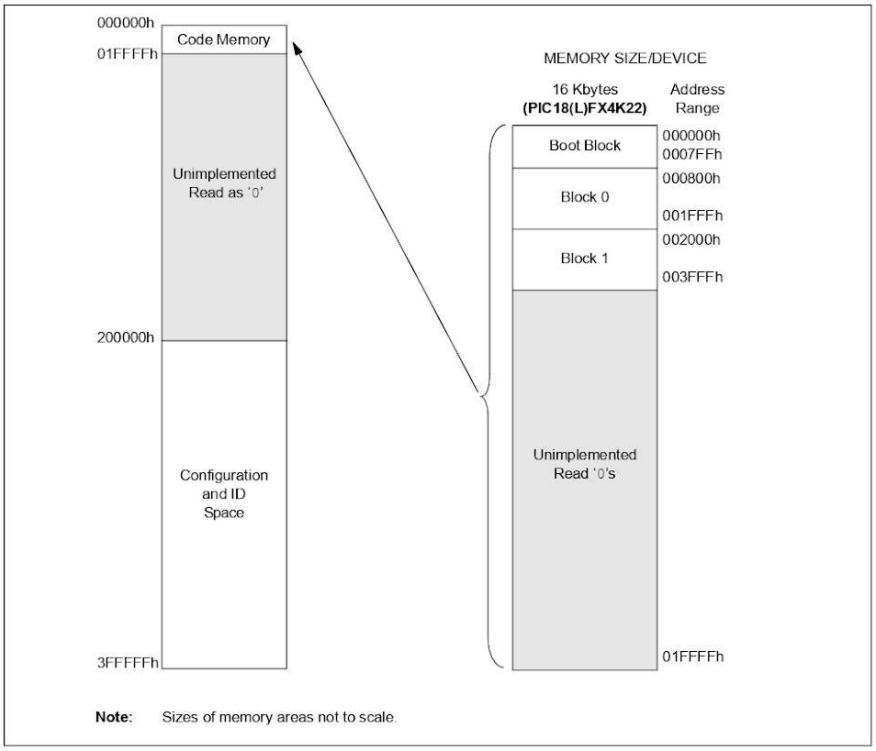
PIC18(L)F2XK22/4XK22

For PIC18(L)FX4K22 devices, the code memory space extends from 000000h to 003FFFh (16 Kbytes) in two 4-Kbyte blocks. Addresses 000000h through 0007FFh, however, define a "Boot Block" region that is treated separately from Block 0. All of these blocks define code protection boundaries within the code memory space.

TABLE 2-3: IMPLEMENTATION OF CODE MEMORY

Device	Code Memory Size (Bytes)
PIC18F24K22	000000h-003FFFh (16K)
PIC18LF24K22	
PIC18F44K22	
PIC18LF44K22	

FIGURE 2-7: MEMORY MAP AND THE CODE MEMORY SPACE FOR PIC18(L)FX4K22 DEVICES



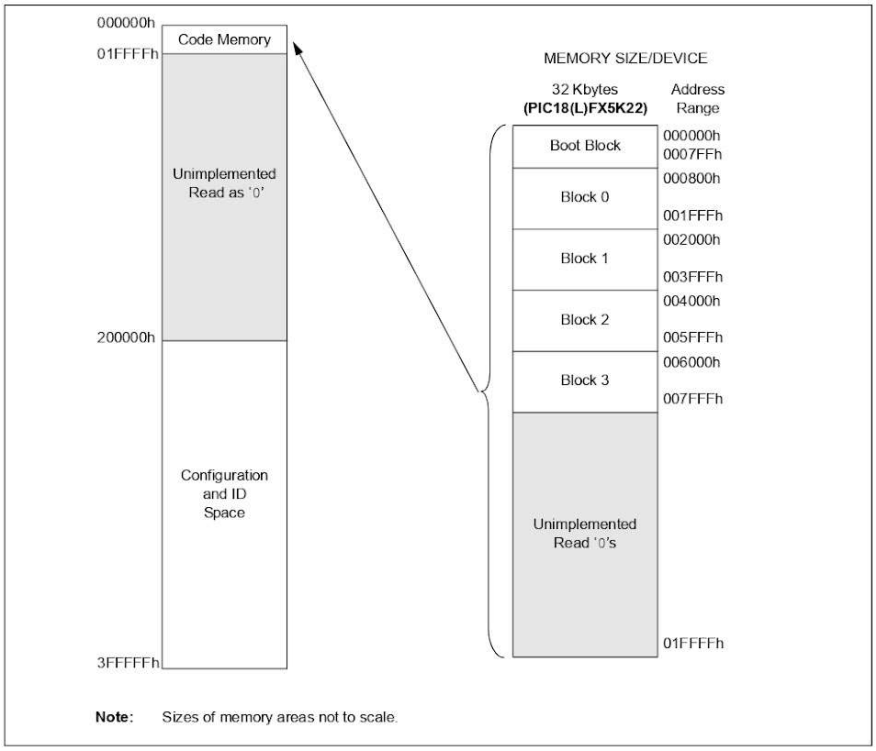
Note: Sizes of memory areas not to scale.

For PIC18(L)FX5K22 devices, the code memory space extends from 000000h to 007FFFh (32 Kbytes) in four 8-Kbyte blocks. Addresses 000000h through 0007FFh, however, define a "Boot Block" region that is treated separately from Block 0. All of these blocks define code protection boundaries within the code memory space.

TABLE 2-4: IMPLEMENTATION OF CODE MEMORY

Device	Code Memory Size (Bytes)
PIC18F25K22	000000h-007FFFh (32K)
PIC18LF25K22	
PIC18F45K22	
PIC18LF45K22	

FIGURE 2-8: MEMORY MAP AND THE CODE MEMORY SPACE FOR PIC18(L)FX5K22 DEVICES



Note: Sizes of memory areas not to scale.

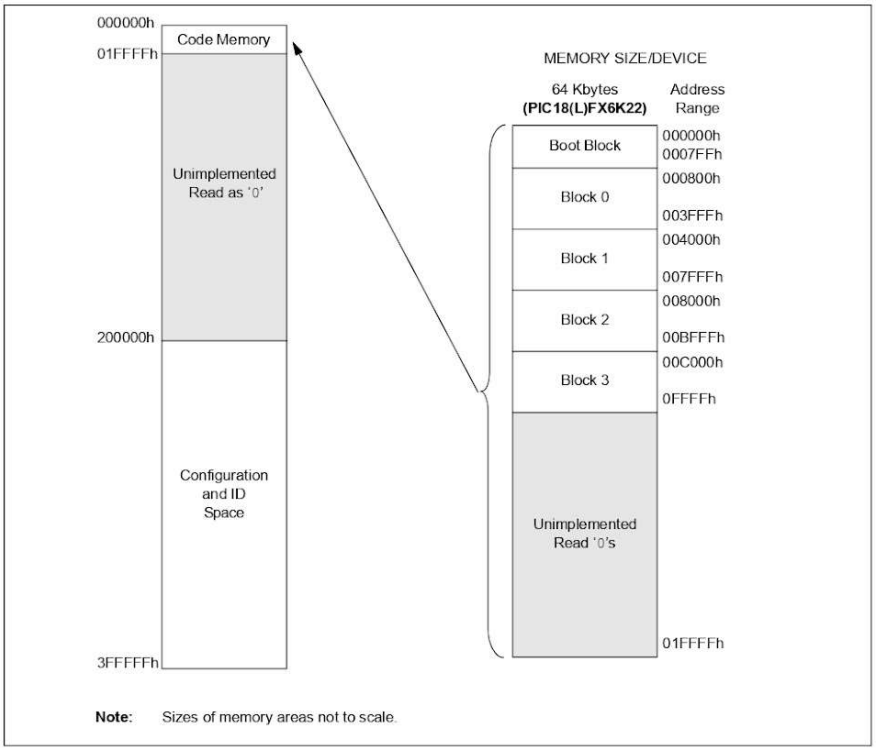
PIC18(L)F2XK22/4XK22

For PIC18(L)FX6K22 devices, the code memory space extends from 000000h to 00FFFFh (64 Kbytes) in four 16-Kbyte blocks. Addresses 000000h through 0007FFh, however, define a "Boot Block" region that is treated separately from Block 0. All of these blocks define code protection boundaries within the code memory space.

TABLE 2-5: IMPLEMENTATION OF CODE MEMORY

Device	Code Memory Size (Bytes)
PIC18F26K22	000000h-00FFFFh (64K)
PIC18LF26K22	
PIC18F46K22	
PIC18LF46K22	

FIGURE 2-9: MEMORY MAP AND THE CODE MEMORY SPACE FOR PIC18(L)FX6K22 DEVICES



Note: Sizes of memory areas not to scale.

In addition to the code memory space, there are three blocks in the configuration and ID space that are accessible to the user through table reads and table writes. Their locations in the memory map are shown in Figure 2-10.

Users may store identification information (ID) in eight ID registers. These ID registers are mapped in addresses 200000h through 200007h. The ID locations read out normally, even after code protection is applied.

Locations 300000h through 30000Dh are reserved for the Configuration bits. These bits select various device options and are described in **Section 5.0 "Configuration Word"**. These Configuration bits read out normally, even after code protection.

Locations 3FFFFEh and 3FFFFFh are reserved for the device ID bits. These bits may be used by the programmer to identify what device type is being programmed and are described in **Section 5.0 "Configuration Word"**. These device ID bits read out normally, even after code protection.

2.3.1 MEMORY ADDRESS POINTER

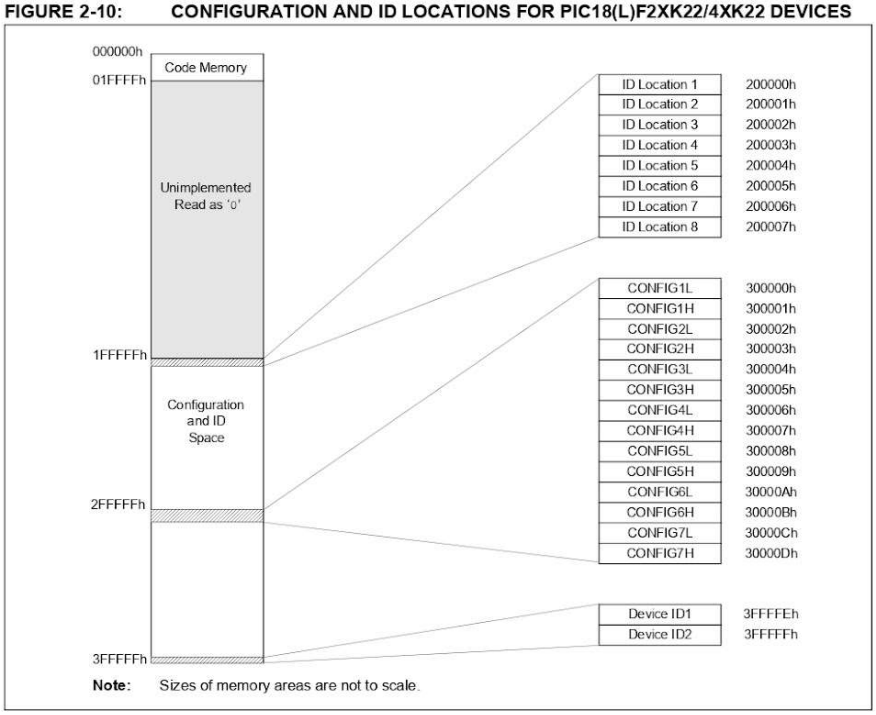
Memory in the address space, 0000000h to 3FFFFFh, is addressed via the Table Pointer register, which is comprised of three Pointer registers:

- TBLPTRU, at RAM address 0FF8h
- TBLPTRH, at RAM address 0FF7h
- TBLPTRL, at RAM address 0FF6h

TBLPTRU	TBLPTRH	TBLPTRL
Addr[21:16]	Addr[15:8]	Addr[7:0]

The 4-bit command, '0000' (core instruction), is used to load the Table Pointer prior to using any read or write operations.

FIGURE 2-10: CONFIGURATION AND ID LOCATIONS FOR PIC18(L)F2XK22/4XK22 DEVICES

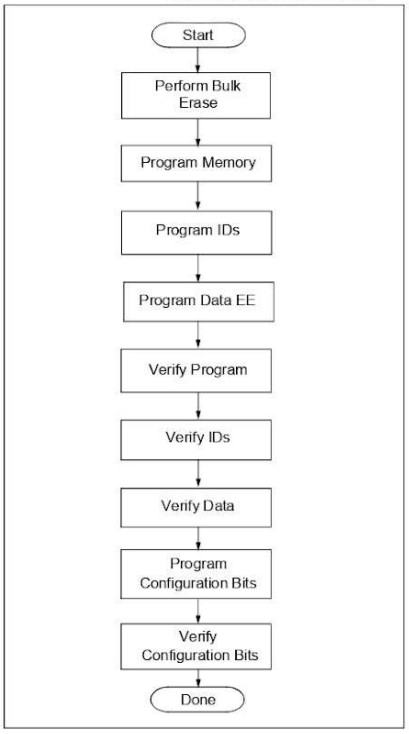


Note: Sizes of memory areas are not to scale.

2.4 High-Level Overview of the Programming Process

Figure 2-11 shows the high-level overview of the programming process. First, a Bulk Erase is performed. Next, the code memory, ID locations and data EEPROM are programmed. These memories are then verified to ensure that programming was successful. If no errors are detected, the Configuration bits are then programmed and verified.

FIGURE 2-11: HIGH-LEVEL PROGRAMMING FLOW



2.5 Entering and Exiting High-Voltage ICSP Program/Verify Mode

As shown in Figure 2-12, the High-Voltage ICSP Program/Verify mode is entered by holding PGC and PGD low and then raising $\overline{MCLR}$/VPP/RE3 to VIHH (high voltage). Once in this mode, the code memory, data EEPROM, ID locations and Configuration bits can be accessed and programmed in serial fashion. Figure 2-13 shows the exit sequence.

The sequence that enters the device into the Program/Verify mode places all unused I/Os in the high-impedance state.

FIGURE 2-12: ENTERING HIGH-VOLTAGE PROGRAM/VERIFY MODE

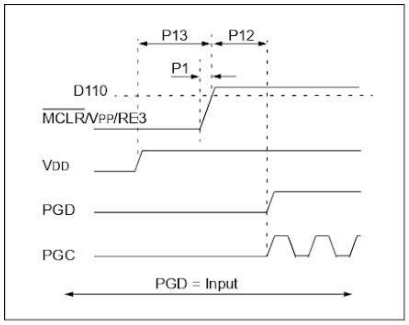
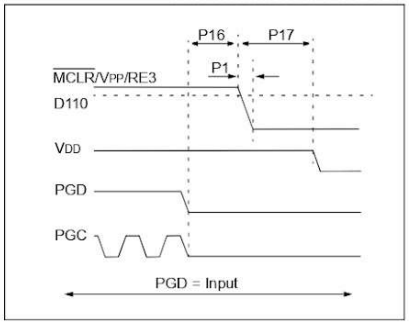


FIGURE 2-13: EXITING HIGH-VOLTAGE PROGRAM/VERIFY MODE



2.6 Entering and Exiting Low-Voltage ICSP Program/Verify Mode

Low-voltage entry into ICSP modes for PIC18(L)F2XK22/4XK22 devices is somewhat different than previous PIC18 devices. As shown in Figure 2-14, entering ICSP Program/Verify mode requires three steps:

1. Voltage is briefly applied to the $\overline{MCLR}$ pin.
2. A 32-bit key sequence is presented on PGD.
3. Voltage is reapplied to $\overline{MCLR}$.

The programming voltage applied to $\overline{MCLR}$ is VIH, or usually, VDD. There is no minimum time requirement for holding at VIH. After VIH is removed, an interval of at least P18 must elapse before presenting the key sequence on PGD.

The key sequence is a specific 32-bit pattern, '0100 1101 0100 0011 0100 1000 0101 0000' (more easily remembered as 4D434850h in hexadecimal). The device will enter Program/Verify mode only if the sequence is valid. The Most Significant bit of the most significant nibble must be shifted in first.

Once the key sequence is complete, VIH must be applied to $\overline{MCLR}$ and held at that level for as long as Program/Verify mode is to be maintained. An interval of at least time P20 and P15 must elapse before presenting data on PGD. Signals appearing on PGD before P15 has elapsed may not be interpreted as valid.

On successful entry, the program memory can be accessed and programmed in serial fashion. While in the Program/Verify mode, all unused I/Os are placed in the high-impedance state.

Exiting Program/Verify mode is done by removing VIH from $\overline{MCLR}$, as shown in Figure 2-15. The only requirement for exit is that an interval, P16, should elapse between the last clock and the program signals on PGC and PGD before removing VIH.

When VIH is reapplied to $\overline{MCLR}$, the device will enter the ordinary operational mode and begin executing the application instructions.

FIGURE 2-14: ENTERING LOW-VOLTAGE PROGRAM/VERIFY MODE

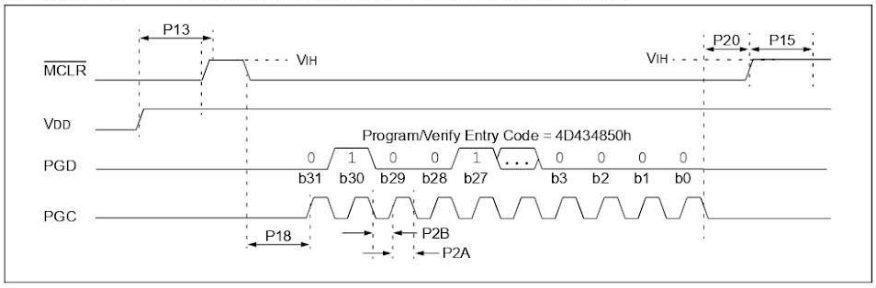
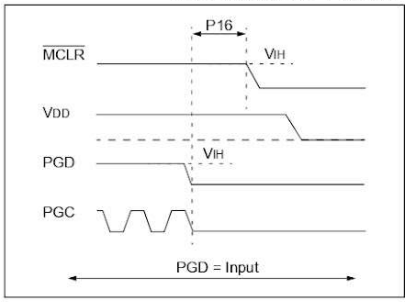


FIGURE 2-15: EXITING LOW-VOLTAGE PROGRAM/VERIFY MODE



2.7 Serial Program/Verify Operation

The PGC pin is used as a clock input pin and the PGD pin is used for entering command bits and data input/output during serial operation. Commands and data are transmitted on the rising edge of PGC, latched on the falling edge of PGC and are Least Significant bit (LSb) first.

2.7.1 4-BIT COMMANDS

All instructions are 20 bits, consisting of a leading 4-bit command followed by a 16-bit operand, which depends on the type of command being executed. To input a command, PGC is cycled four times. The commands needed for programming and verification are shown in Table 2-6.

Depending on the 4-bit command, the 16-bit operand represents 16 bits of input data or 8 bits of input data and 8 bits of output data.

Throughout this specification, commands and data are presented as illustrated in Table 2-7. The 4-bit command is shown Most Significant bit (MSb) first. The command operand, or "Data Payload", is shown <MSB><LSB>. Figure 2-16 demonstrates how to serially present a 20-bit command/operand to the device.

2.7.2 CORE INSTRUCTION

The core instruction passes a 16-bit instruction to the CPU core for execution. This is needed to set up registers as appropriate for use with other commands.

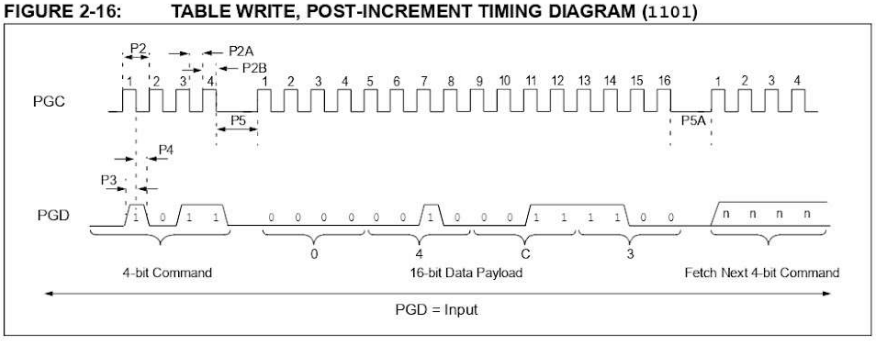
TABLE 2-6: COMMANDS FOR PROGRAMMING

Description	4-Bit Command
Core Instruction (Shift in16-bit instruction)	0000
Shift out TABLAT register	0010
Table Read	1000
Table Read, post-increment	1001
Table Read, post-decrement	1010
Table Read, pre-increment	1011
Table Write	1100
Table Write, post-increment by 2	1101
Table Write, start programming, post-increment by 2	1110
Table Write, start programming	1111

TABLE 2-7: SAMPLE COMMAND SEQUENCE

4-Bit Command	Data Payload	Core Instruction
1101	3C 40	Table Write, post-increment by 2

FIGURE 2-16: TABLE WRITE, POST-INCREMENT TIMING DIAGRAM (1101)



3.0 DEVICE PROGRAMMING

Programming includes the ability to erase or write the various memory regions within the device.

In all cases, except high-voltage ICSP Bulk Erase, the EECON1 register must be configured in order to operate on a particular memory region.

When using the EECON1 register to act on code memory, the EEPGD bit must be set (EECON1<7> = 1) and the CFGS bit must be cleared (EECON1<6> = 0). The WREN bit must be set (EECON1<2> = 1) to enable writes of any sort (e.g., erases) and this must be done prior to initiating a write sequence. The FREE bit must be set (EECON1<4> = 1) in order to erase the program space being pointed to by the Table Pointer. The erase or write sequence is initiated by setting the WR bit (EECON1<1> = 1). It is strongly recommended that the WREN bit only be set immediately prior to a program or erase.

3.1 ICSP Erase

3.1.1 HIGH-VOLTAGE ICSP BULK ERASE

Erasing code or data EEPROM is accomplished by configuring two Bulk Erase Control registers located at 3C0004h and 3C0005h. Code memory may be erased portions at a time, or the user may erase the entire device in one action. Bulk Erase operations will also clear any code-protect settings associated with the memory block erased. Erase options are detailed in Table 3-1. When any one or more blocks of code space are code protected, then all code blocks will be erased by default. If data EEPROM is code-protected (CPD = 0), the user must request an erase of data EEPROM (e.g., 0084h as shown in Table 3-1).

TABLE 3-1: BULK ERASE OPTIONS

Description	Data (3C0005h:3C0004h)
Chip Erase	0F8Fh
Erase User ID	0088h
Erase Data EEPROM	0084h
Erase Boot Block	0081h
Erase Config Bits	0082h
Erase Code EEPROM Block 0	0180h
Erase Code EEPROM Block 1	0280h
Erase Code EEPROM Block 2	0480h
Erase Code EEPROM Block 3	0880h

The actual Bulk Erase function is a self-timed operation. Once the erase has started (falling edge of the 4th PGC after the NOP command), serial execution will cease until the erase completes (parameter P11). During this time, PGC may continue to toggle but PGD must be held low.

The code sequence to erase the entire device is shown in Table 3-2 and the flowchart is shown in Figure 3-1.

Note:	A Bulk Erase is the only way to reprogram code-protect bits from an "on" state to an "off" state.

TABLE 3-2: BULK ERASE COMMAND SEQUENCE

4-Bit Command	Data Payload	Core Instruction
0000	0E 3C	MOVLW 3Ch
0000	6E F8	MOVWF TBLPTRU
0000	0E 00	MOVLW 00h
0000	6E F7	MOVWF TBLPTRH
0000	0E 05	MOVLW 05h
0000	6E F6	MOVWF TBLPTRL
1100	0F 0F	Write 0Fh to 3C0005h
0000	0E 3C	MOVLW 3Ch
0000	6E F8	MOVWF TBLPTRU
0000	0E 00	MOVLW 00h
0000	6E F7	MOVWF TBLPTRH
0000	0E 04	MOVLW 04h
0000	6E F6	MOVWF TBLPTRL
1100	8F 8F	Write 8F8Fh TO 3C0004h to erase entire device.
0000	00 00	NOP
0000	00 00	Hold PGD low until erase completes.

FIGURE 3-1: BULK ERASE FLOW

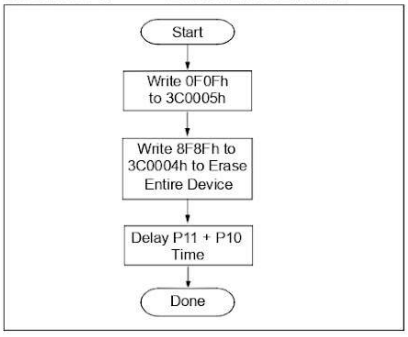
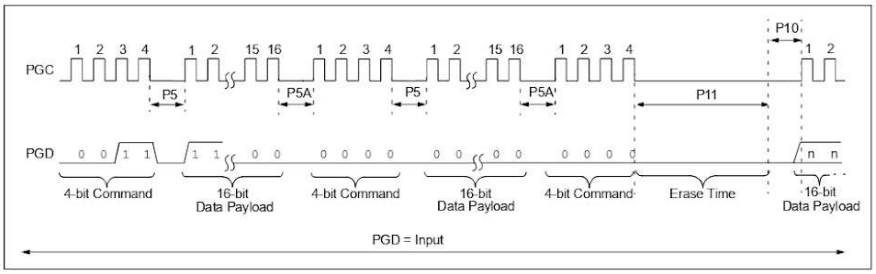


FIGURE 3-2: BULK ERASE TIMING DIAGRAM



3.1.2 LOW-VOLTAGE ICSP BULK ERASE

When using low-voltage ICSP, the part must be supplied by the voltage specified in parameter D111 if a Bulk Erase is to be executed. All other Bulk Erase details as described above apply.

If it is determined that a program memory erase must be performed at a supply voltage below the Bulk Erase limit, refer to the erase methodology described in **Section 3.1.3 "ICSP Row Erase"** and **Section 3.2.1 "Modifying Code Memory"**.

If it is determined that a data EEPROM erase must be performed at a supply voltage below the Bulk Erase limit, follow the methodology described in **Section 3.3 "Data EEPROM Programming"** and write '1's to the array.

3.1.3 ICSP ROW ERASE

Regardless of whether high or low-voltage ICSP is used, it is possible to erase one row (64 bytes of data), provided the block is not code or write-protected. Rows are located at static boundaries beginning at program memory address 000000h, extending to the internal program memory limit (see **Section 2.3 "Memory Maps"**).

The Row Erase duration is self-timed. After the WR bit in EECON1 is set, two NOPs are issued. Erase starts upon the 4th PGC of the second NOP. It ends when the WR bit is cleared by hardware.

The code sequence to Row Erase is shown in Table 3-3. The flowchart shown in Figure 3-3 depicts the logic necessary to completely erase the device. The timing diagram for Row Erase is identical to the data EEPROM write timing shown in Figure 3-7.

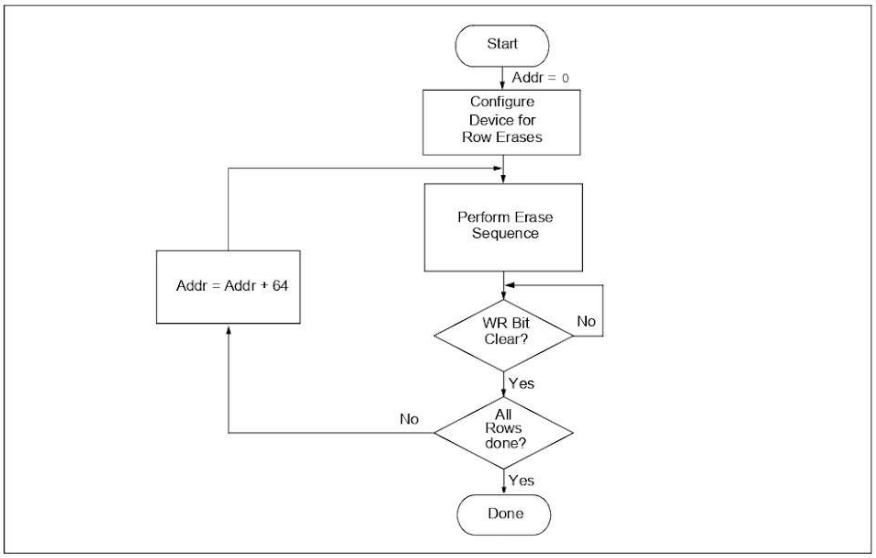
Note:	The TBLPTR register can point at any byte within the row intended for erase.

TABLE 3-3: ERASE CODE MEMORY CODE SEQUENCE

4-bit Command	Data Payload	Core Instruction
Step 1: Direct access to code memory and enable writes.		
0000	8E A6	BSF EECON1, EEPGD
0000	9C A6	BCF EECON1, CFGS
0000	84 A6	BSF EECON1, WREN
Step 2: Point to first row in code memory.		
0000	6A F8	CLRF TBLPTRU
0000	6A F7	CLRF TBLPTRH
0000	6A F6	CLRF TBLPTRL
Step 3: Enable erase and erase single row.		
0000	88 A6	BSF EECON1, FREE
0000	82 A6	BSF EECON1, WR
0000	00 00	NOP
0000	00 00	NOP Erase starts on the 4th clock of this instruction
Step 4: Poll WR bit. Repeat until bit is clear.		
0000	50 A6	MOVF EECON1, W, 0
0000	6E F5	MOVWF TABLAT
0000	00 00	NOP
0010	<MSB><LSB>	Shift out data(1)
Step 5: Hold PGC low for time P10.		
Step 6: Repeat step 3 with Address Pointer incremented by 64 until all rows are erased.		
Step 7: Disable writes.		
0000	94 A6	BCF EECON1, WREN

Note 1: See Figure 4-4 for details on shift out data timing.

FIGURE 3-3: SINGLE ROW ERASE CODE MEMORY FLOW



3.2 Code Memory Programming

Programming code memory is accomplished by first loading data into the write buffer and then initiating a programming sequence. The write and erase buffer sizes shown in Table 3-4 can be mapped to any location of the same size beginning at 000000h. The actual memory write sequence takes the contents of this buffer and programs the proper amount of code memory that contains the Table Pointer.

The programming duration is externally timed and is controlled by PGC. After a Start Programming command is issued (4-bit command, '1111'), a NOP is issued, where the 4th PGC is held high for the duration of the programming time, P9.

After PGC is brought low, the programming sequence is terminated. PGC must be held low for the time specified by parameter P10 to allow high-voltage discharge of the memory array.

The code sequence to program a device is shown in Table 3-5. The flowchart shown in Figure 3-4 depicts the logic necessary to completely write the device. The timing diagram that details the Start Programming command and parameters P9 and P10 is shown in Figure 3-5.

Note:	The TBLPTR register must point to the same region when initiating the programming sequence as it did when the write buffers were loaded.

TABLE 3-4: WRITE AND ERASE BUFFER SIZES

Devices				Write Buffer Size (bytes)	Erase Size (bytes)
PIC18F23K22	PIC18F43K22	PIC18LF23K22	PIC18LF43K22	64	64
PIC18F24K22	PIC18F44K22	PIC18LF24K22	PIC18LF44K22		
PIC18F25K22	PIC18F45K22	PIC18LF25K22	PIC18LF45K22		
PIC18F26K22	PIC18F46K22	PIC18LF26K22	PIC18LF46K22		

TABLE 3-5: WRITE CODE MEMORY CODE SEQUENCE

4-bit Command	Data Payload	Core Instruction
Step 1: Direct access to code memory.		
0000	8E A6	BSF EECON1, EEPGD
0000	9C A6	BCF EECON1, CFGS
0000	84 A6	BSF EECON1, WREN
Step 2: Point to row to write.		
0000	0E <Addr[21:16]>	MOVLW <Addr[21:16]>
0000	6E F8	MOVWF TBLPTRU
0000	0E <Addr[15:8]>	MOVLW <Addr[15:8]>
0000	6E F7	MOVWF TBLPTRH
0000	0E <Addr[7:0]>	MOVLW <Addr[7:0]>
0000	6E F6	MOVWF TBLPTRL
Step 3: Load write buffer. Repeat for all but the last two bytes.		
1101	<MSB><LSB>	Write 2 bytes and post-increment address by 2.
Step 4: Load write buffer for last two bytes and start programming.		
1111	<MSB><LSB>	Write 2 bytes and start programming.
0000	00 00	NOP - hold PGC high for time P9 and low for time P10.
To continue writing data, repeat steps 2 through 4, where the Address Pointer is incremented by 2 at each iteration of the loop.		

FIGURE 3-4: PROGRAM CODE MEMORY FLOW

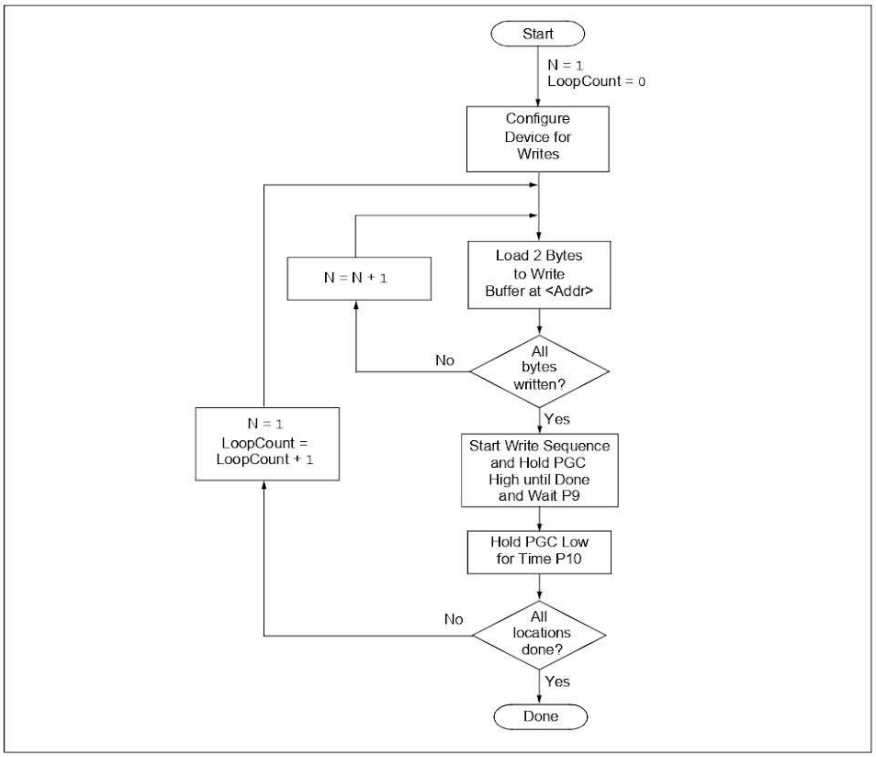
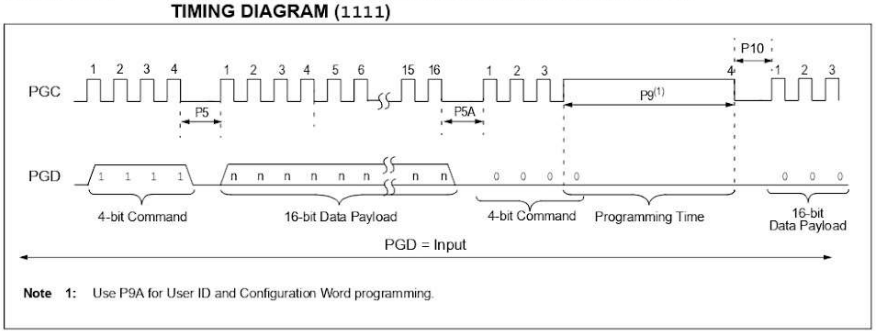


FIGURE 3-5: TABLE WRITE AND START PROGRAMMING INSTRUCTION TIMING DIAGRAM (1111)



Note 1: Use P9A for User ID and Configuration Word programming.

3.3 Data EEPROM Programming

Data EEPROM is accessed one byte at a time via an Address Pointer (register pair EEADRH:EEADR) and a data latch (EEDATA). Data EEPROM is written by loading EEADRH:EEADR with the desired memory location, EEDATA with the data to be written and initiating a memory write by appropriately configuring the EECON1 register. A byte write automatically erases the location and writes the new data (erase-before-write).

When using the EECON1 register to perform a data EEPROM write, both the EEPGD and CFGS bits must be cleared (EECON1<7:6> = 00). The WREN bit must be set (EECON1<2> = 1) to enable writes of any sort and this must be done prior to initiating a write sequence. The write sequence is initiated by setting the WR bit (EECON1<1> = 1).

The write begins on the falling edge of the 24th PGC after the WR bit is set. It ends when the WR bit is cleared by hardware.

After the programming sequence terminates, PGC must be held low for the time specified by parameter P10 to allow high-voltage discharge of the memory array.

FIGURE 3-6: PROGRAM DATA FLOW

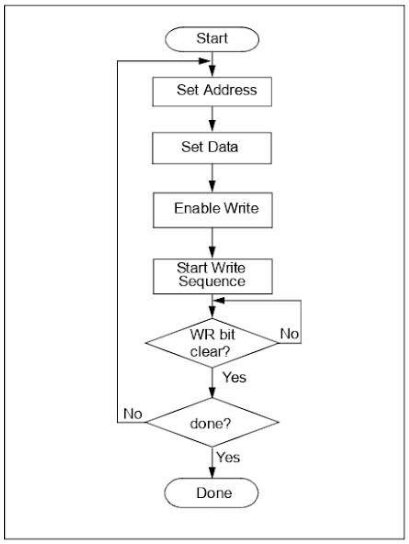


FIGURE 3-7: DATA EEPROM WRITE TIMING DIAGRAM

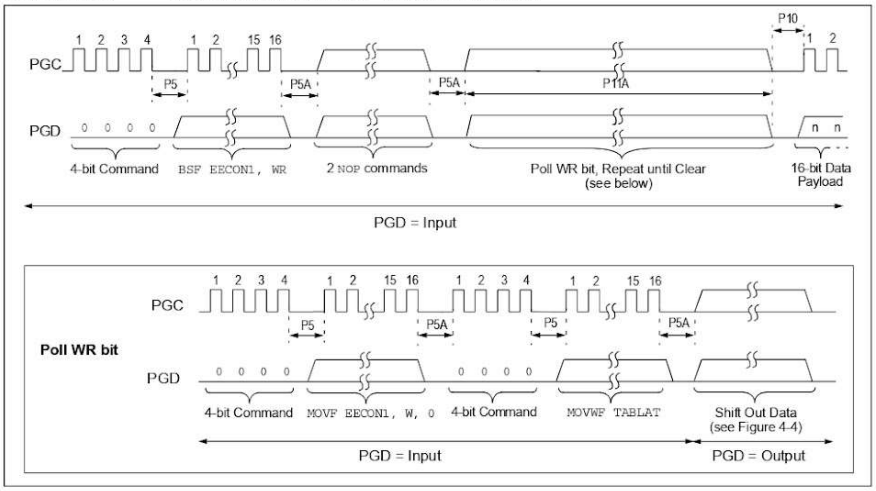


TABLE 3-7: PROGRAMMING DATA MEMORY

4-bit Command	Data Payload	Core Instruction
Step 1: Direct access to data EEPROM.		
0000	9E A6	BCF EECON1, EEPGD
0000	9C A6	BCF EECON1, CFGS
Step 2: Set the data EEPROM Address Pointer.		
0000	0E <Addr>	MOVLW <Addr>
0000	6E A9	MOVWF EEADR
0000	OE <AddrH>	MOVLW <AddrH>
0000	6E AA	MOVWF EEADRH
Step 3: Load the data to be written.		
0000	0E <Data>	MOVLW <Data>
0000	6E A8	MOVWF EEDATA
Step 4: Enable memory writes.		
0000	84 A6	BSF EECON1, WREN
Step 5: Initiate write.		
0000	82 A6	BSF EECON1, WR
0000	00 00	NOP
0000	00 00	NOP ;write starts on 4th clock of this instruction
Step 6: Poll WR bit, repeat until the bit is clear.		
0000	50 A6	MOVF EECON1, W, 0
0000	6E F5	MOVWF TABLAT
0000	00 00	NOP
0010	<MSB><LSB>	Shift out data(1)
Step 7: Hold PGC low for time P10.		
Step 8: Disable writes.		
0000	94 A6	BCF EECON1, WREN
Repeat steps 2 through 8 to write more data.		

Note 1: See Figure 4-4 for details on shift out data timing.

3.4 ID Location Programming

The ID locations are programmed much like the code memory. The ID registers are mapped in addresses 200000h through 200007h. These locations read out normally even after code protection.

Note:	The user only needs to fill the first 8 bytes of the write buffer in order to write the ID locations.

Table 3-8 demonstrates the code sequence required to write the ID locations.

In order to modify the ID locations, refer to the methodology described in **Section 3.2.1 "Modifying Code Memory"**. As with code memory, the ID locations must be erased before being modified.

When V_{DD} is below the minimum for Bulk Erase operation, ID locations can be cleared with the Row Erase method described in **Section 3.1.3 "ICSP Row Erase"**.

TABLE 3-8: WRITE ID SEQUENCE

4-bit Command	Data Payload	Core Instruction
Step 1: Direct access to code memory.		
0000	8E A6	BSF EECON1, EEPGD
0000	9C A6	BCF EECON1, CFGS
0000	84 A6	BSF EECON1, WREN
Step 2: Set Table Pointer to ID. Load write buffer with 8 bytes and write.		
0000	0E 20	MOVLW 20h
0000	6E F8	MOVWF TBLPTRU
0000	0E 00	MOVLW 00h
0000	6E F7	MOVWF TBLPTRH
0000	0E 00	MOVLW 00h
0000	6E F6	MOVWF TBLPTRL
1101	<MSB><LSB>	Write 2 bytes and post-increment address by 2.
1101	<MSB><LSB>	Write 2 bytes and post-increment address by 2.
1101	<MSB><LSB>	Write 2 bytes and post-increment address by 2.
1111	<MSB><LSB>	Write 2 bytes and start programming.
0000	00 00	NOP - hold PGC high for time P9 and low for time P10.

3.5 Boot Block Programming

The code sequence detailed in Table 3-5 should be used, except that the address used in "Step 2" will be in the range of 000000h to 0007FFh.

3.6 Configuration Bits Programming

Unlike code memory, the Configuration bits are programmed a byte at a time. The Table Write, Begin Programming 4-bit command ('1111') is used, but only 8 bits of the following 16-bit payload will be written. The LSB of the payload will be written to even addresses and the MSB will be written to odd addresses. The code sequence to program two consecutive configuration locations is shown in Table 3-9. See Figure 3-5 for the timing diagram.

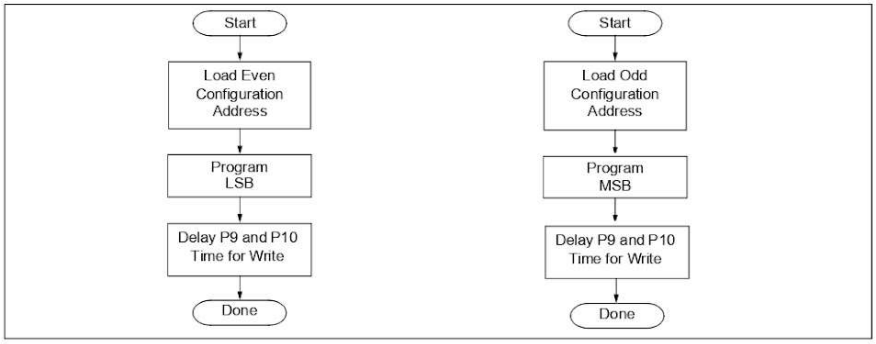
Note:	The address must be explicitly written for each byte programmed. The addresses can not be incremented in this mode.

TABLE 3-9: SET ADDRESS POINTER TO CONFIGURATION LOCATION

4-bit Command	Data Payload	Core Instruction
Step 1: Direct access to config memory.		
0000	8E A6	BSF EECON1, EEPGD
0000	8C A6	BSF EECON1, CFGS
0000	84 A6	BSF EECON1, WREN
Step 2(1): Set Table Pointer for config byte to be written. Write even/odd addresses.		
0000	0E 30	MOVLW 30h
0000	6E F8	MOVWF TBLPTRU
0000	0E 00	MOVLW 00h
0000	6E F7	MOVWF TBLPRTH
0000	0E 00	MOVLW 00h
0000	6E F6	MOVWF TBLPTRL
1111	<MSB ignored><LSB>	Load 2 bytes and start programming.
0000	00 00	NOP - hold PGC high for time P9 and low for time P10.
0000	0E 01	MOVLW 01h
0000	6E F6	MOVWF TBLPTRL
1111	<MSB><LSB ignored>	Load 2 bytes and start programming.
0000	00 00	NOP - hold PGC high for time P9A and low for time P10.

Note 1: Enabling the write protection of Configuration bits (WRTC = 0 in CONFIG6H) will prevent further writing of Configuration bits. Always write all the Configuration bits before enabling the write protection for Configuration bits.

FIGURE 3-8: CONFIGURATION PROGRAMMING FLOW



4.0 READING THE DEVICE

4.1 Read Code Memory, ID Locations and Configuration Bits

Code memory is accessed one byte at a time via the 4-bit command, '1001' (table read, post-increment). The contents of memory pointed to by the Table Pointer (TBLPTRU:TBLPTRH:TBLPTRL) are serially output on PGD.

The 4-bit command is shifted in LSb first. The read is executed during the next 8 clocks, then shifted out on PGD during the last 8 clocks, LSb to MSb. A delay of P6 must be introduced after the falling edge of the 8th PGC of the operand to allow PGD to transition from an input to an output. During this time, PGC must be held low (see Figure 4-1). This operation also increments the Table Pointer by one, pointing to the next byte in code memory for the next read.

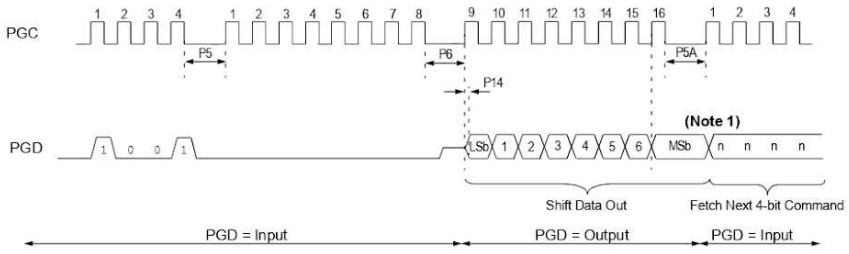
This technique will work to read any memory in the 000000h to 3FFFFFh address space, so it also applies to the reading of the ID and Configuration registers.

Note:	When table read protection is enabled, the first read access to a protected block should be discarded and the read repeated to retrieve valid data. Subsequent reads of the same block can be performed normally.

TABLE 4-1: READ CODE MEMORY SEQUENCE

4-bit Command	Data Payload	Core Instruction
Step 1: Set Table Pointer		
0000	0E <Addr[21:16]>	MOVLW Addr[21:16]
0000	6E F8	MOVWF TBLPTRU
0000	0E <Addr[15:8]>	MOVLW <Addr[15:8]>
0000	6E F7	MOVWF TBLPTRH
0000	0E <Addr[7:0]>	MOVLW <Addr[7:0]>
0000	6E F6	MOVWF TBLPTRL
Step 2: Read memory and then shift out on PGD, LSb to MSb		
1001	00 00	TBLRD *+

FIGURE 4-1: TABLE READ POST-INCREMENT INSTRUCTION TIMING DIAGRAM (1001)



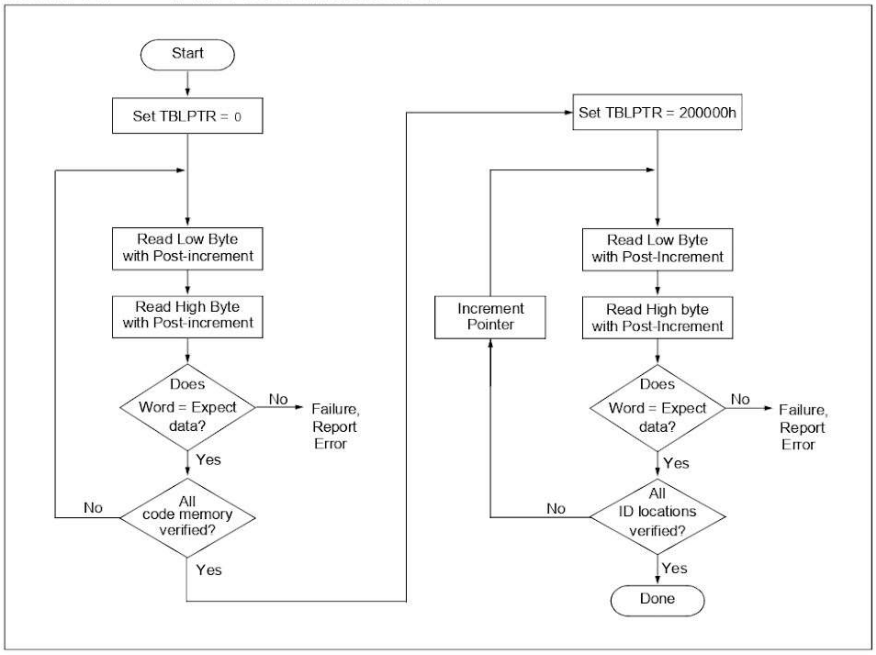
Note 1: Magnification of the high-impedance delay between PGC and PGD is shown in Figure 4-6.

4.2 Verify Code Memory and ID Locations

The verify step involves reading back the code memory space and comparing it against the copy held in the programmer's buffer. Memory reads occur a single byte at a time, so two bytes must be read to compare against the word in the programmer's buffer. Refer to **Section 4.1 "Read Code Memory, ID Locations and Configuration Bits"** for implementation details of reading code memory.

The Table Pointer must be manually set to 200000h (base address of the ID locations) once the code memory has been verified. The post-increment feature of the table read 4-bit command can not be used to increment the Table Pointer beyond the code memory space. In a 64-Kbyte device, for example, a post-increment read of address FFFFh will wrap the Table Pointer back to 000000h, rather than point to unimplemented address 010000h.

FIGURE 4-2: VERIFY CODE MEMORY FLOW



4.3 Verify Configuration Bits

A configuration address may be read and output on PGD via the 4-bit command, '1001'. Configuration data is read and written in a byte-wise fashion, so it is not necessary to merge two bytes into a word prior to a compare. The result may then be immediately compared to the appropriate configuration data in the programmer's memory for verification. Refer to **Section 4.1 "Read Code Memory, ID Locations and Configuration Bits"** for implementation details of reading configuration data.

4.4 Read Data EEPROM Memory

Data EEPROM is accessed one byte at a time via an Address Pointer (register pair EEADRH:EEADR) and a data latch (EEDATA). Data EEPROM is read by loading EEADRH:EEADR with the desired memory location and initiating a memory read by appropriately configuring the EECON1 register. The data will be loaded into EEDATA, where it may be serially output on PGD via the 4-bit command, '0010' (Shift Out Data Holding register). A delay of P6 must be introduced after the falling edge of the 8th PGC of the operand to allow PGD to transition from an input to an output. During this time, PGC must be held low (see Figure 4-4).

The command sequence to read a single byte of data is shown in Table 4-2.

FIGURE 4-3: READ DATA EEPROM FLOW

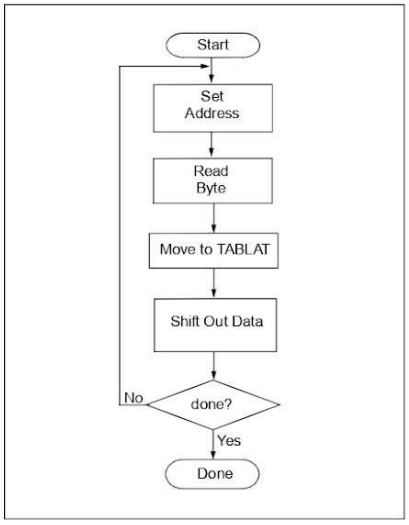
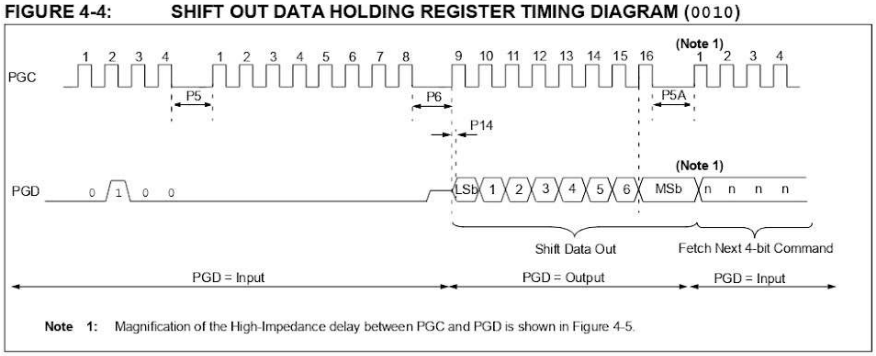


TABLE 4-2: READ DATA EEPROM MEMORY

4-bit Command	Data Payload	Core Instruction
Step 1: Direct access to data EEPROM.		
0000	9E A6	BCF EECON1, EEPGD
0000	9C A6	BCF EECON1, CFGS
Step 2: Set the data EEPROM Address Pointer.		
0000	0E <Addr>	MOVLW <Addr>
0000	6E A9	MOVWF EEADR
0000	OE <AddrH>	MOVLW <AddrH>
0000	6E AA	MOVWF EEADRH
Step 3: Initiate a memory read.		
0000	80 A6	BSF EECON1, RD
Step 4: Load data into the Serial Data Holding register.		
0000	50 A8	MOVF EEDATA, W, 0
0000	6E F5	MOVWF TABLAT
0000	00 00	NOP
0010	<MSB><LSB>	Shift Out Data(1)

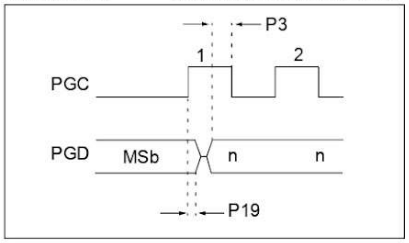
Note 1: The <LSB> is undefined. The <MSB> is the data.

FIGURE 4-4: SHIFT OUT DATA HOLDING REGISTER TIMING DIAGRAM (0010)



Note 1: Magnification of the High-Impedance delay between PGC and PGD is shown in Figure 4-5.

FIGURE 4-5: HIGH-IMPEDANCE DELAY



4.5 Verify Data EEPROM

A data EEPROM address may be read via a sequence of core instructions (4-bit command, '0000') and then output on PGD via the 4-bit command, '0010' (TABLAT register). The result may then be immediately compared to the appropriate data in the programmer's memory for verification. Refer to **Section 4.4 "Read Data EEPROM Memory"** for implementation details of reading data EEPROM.

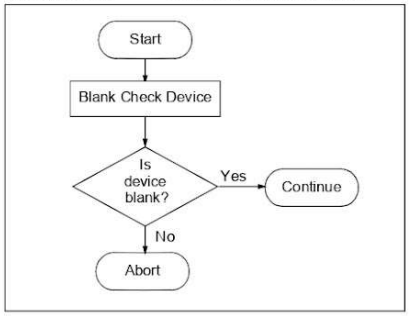
4.6 Blank Check

The term "Blank Check" means to verify that the device has no programmed memory cells. All memories must be verified: code memory, data EEPROM, ID locations and Configuration bits. The device ID registers (3FFFFEh:3FFFFFh) should be ignored.

A "blank" or "erased" memory cell will read as a '1'. Therefore, Blank Checking a device merely means to verify that all bytes read as FFh except the Configuration bits. Unused (reserved) Configuration bits will read '0' (programmed). Refer to Table 5-1 for blank configuration expect data for the various PIC18(L)F2XK22/4XK22 devices.

Given that Blank Checking is merely code and data EEPROM verification with FFh expect data, refer to **Section 4.4 "Read Data EEPROM Memory"** and **Section 4.2 "Verify Code Memory and ID Locations"** for implementation details.

FIGURE 4-6: BLANK CHECK FLOW



5.0 CONFIGURATION WORD

The PIC18(L)F2XK22/4XK22 devices have several Configuration Words. These bits can be set or cleared to select various device configurations. All other memory areas should be programmed and verified prior to setting Configuration Words. These bits may be read out normally, even after read or code protection. See Table 5-1 for a list of Configuration bits and device IDs and Table 5-3 for the Configuration bit descriptions.

5.1 User ID Locations

A user may store identification information (ID) in eight ID locations mapped in 200000h:200007h. It is recommended that the Most Significant nibble of each ID be Fh. In doing so, if the user code inadvertently tries to execute from the ID space, the ID data will execute as a NOP.

5.2 Device ID Word

The device ID word for the PIC18(L)F2XK22/4XK22 devices is located at 3FFFFEh:3FFFFFh. These bits may be used by the programmer to identify what device type is being programmed and read out normally, even after code or read protection. See Table 5-2 for a complete list of device ID values.

FIGURE 5-1: READ DEVICE ID WORD FLOW

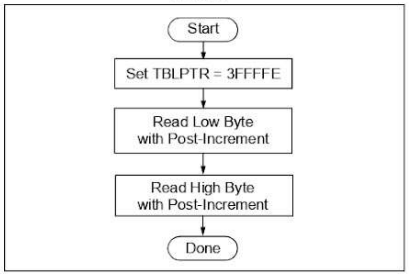


TABLE 5-1: CONFIGURATION BITS AND DEVICE IDs

File Name		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Default/ Unprogrammed Value
300001h	CONFIG1H	IESO	FCMEN	PRI_CLK_EN	PLLEN	FOSC3	FOSC2	FOSC1	FOSC0	0010 0101
300002h	CONFIG2L	—	—	—	BORV1	BORV0	BOREN1	BOREN0	PWRTEN	---1 1111
300003h	CONFIG2H	—	—	WDTPS3	WDTPS2	WDTPS1	WDTPS0	WDTEN1	WDTEN0	--11 1111
300005h	CONFIG3H	MCLRE	—	P2BMX	T3CMX	HFOFST	CCP3MX	PBADEN	CCP2MX	1-11 1111
300006h	CONFIG4L	DEBUG	XINST	—	—	—	LVP	—	STVREN	10-- -1-1
300008h	CONFIG5L	—	—	—	—	CP3(1)	CP2(1)	CP1	CP0	---- 1111
300009h	CONFIG5H	CPD	CPB	—	—	—	—	—	—	11-- ----
30000Ah	CONFIG6L	—	—	—	—	WRT3(1)	WRT2(1)	WRT1	WRT0	---- 1111
30000Bh	CONFIG6H	WRTD	WRTB	WRTC	—	—	—	—	—	111- ----
30000Ch	CONFIG7L	—	—	—	—	EBTR3(1)	EBTR2(1)	EBTR1	EBTR0	---- 1111
30000Dh	CONFIG7H	—	EBTRB	—	—	—	—	—	—	-1-- ----
3FFFFEh	DEVID1(2)	DEV2	DEV1	DEV0	REV4	REV3	REV2	REV1	REV0	See Table 5-2
3FFFFFh	DEVID2(2)	DEV10	DEV9	DEV8	DEV7	DEV6	DEV5	DEV4	DEV3	See Table 5-2

Legend: x = unknown, u = unchanged, – = unimplemented. Shaded cells are unimplemented, read as '0'.

Note 1: These bits are only implemented on specific devices. Refer to **Section 2.3 "Memory Maps"** to determine which bits apply based on available memory.

2: DEVID registers are read-only and cannot be programmed by the user.

TABLE 5-2: DEVICE ID VALUE

Device	Device ID Value	
	DEVID2	DEVID1
PIC18F45K22	55h	000x xxxx
PIC18LF45K22	55h	001x xxxx
PIC18F25K22	55h	010x xxxx
PIC18LF25K22	55h	011x xxxx
PIC18F23K22	57h	010x xxxx
PIC18LF23K22	57h	011x xxxx
PIC18F24K22	56h	010x xxxx
PIC18LF24K22	56h	011x xxxx
PIC18F26K22	54h	010x xxxx
PIC18LF26K22	54h	011x xxxx
PIC18F43K22	57h	000x xxxx
PIC18LF43K22	57h	001x xxxx
PIC18F44K22	56h	000x xxxx
PIC18LF44K22	56h	001x xxxx
PIC18F46K22	54h	000x xxxx
PIC18LF46K22	54h	001x xxxx

Note: The 'x's in DEVID1 contain the device revision code.

TABLE 5-3: PIC18(L)F2XK22/4XK22 BIT DESCRIPTIONS

Bit Name	Configuration Words	Description
IESO	CONFIG1H	Internal External Switchover bit 1 = Internal External Switchover mode enabled 0 = Internal External Switchover mode disabled
FCMEN	CONFIG1H	Fail-Safe Clock Monitor Enable bit 1 = Fail-Safe Clock Monitor enabled 0 = Fail-Safe Clock Monitor disabled
PRICLKEN	CONFIG1H	1 = Primary clock enabled 0 = Primary clock disabled
FOSC<3:0>	CONFIG1H	Oscillator Selection bits 1111 = External RC oscillator, CLKOUT function on OSC2 1110 = External RC oscillator, CLKOUT function on OSC2 1101 = EC oscillator (low power) 1100 = EC oscillator, CLKOUT function on OSC2 (low power) 1011 = EC oscillator (medium power, 4 MHz-16 MHz) 1010 = EC oscillator, CLKOUT function on OSC2 (medium power, 4 MHz-16 MHz) 1001 = Internal RC oscillator, CLKOUT function on OSC2 1000 = Internal RC oscillator 0111 = External RC oscillator 0110 = External RC oscillator, CLKOUT function on OSC2 0101 = EC oscillator (high power, >16 MHz) 0100 = EC oscillator, CLKOUT function on OSC2 (high power, >16 MHz) 0011 = HS oscillator (medium power, 4 MHz-16 MHz) 0010 = HS oscillator (high power, >16 MHz) 0001 = XT oscillator 0000 = LP oscillator
BORV<1:0>	CONFIG2L	Brown-out Reset Voltage bits 11 = V_{BOR} set to 1.9V 10 = V_{BOR} set to 2.2V 01 = V_{BOR} set to 2.5V 00 = V_{BOR} set to 2.85V
BOREN<1:0>	CONFIG2L	Brown-out Reset Enable bits 11 = Brown-out Reset enabled in hardware only (SBOREN is disabled) 10 = Brown-out Reset enabled in hardware only and disabled in Sleep mode (SBOREN is disabled) 01 = Brown-out Reset enabled and controlled by software (SBOREN is enabled) 00 = Brown-out Reset disabled in hardware and software
$\overline{\text{PWRTEN}}$	CONFIG2L	Power-up Timer Enable bit 1 = PWRT disabled 0 = PWRT enabled

TABLE 5-3: PIC18(L)F2XK22/4XK22 BIT DESCRIPTIONS (CONTINUED)

Bit Name	Configuration Words	Description
WDTPS<3:0>	CONFIG2H	Watchdog Timer Postscaler Select bits 1111 = 1:32,768 1110 = 1:16,384 1101 = 1:8,192 1100 = 1:4,096 1011 = 1:2,048 1010 = 1:1,024 1001 = 1:512 1000 = 1:256 0111 = 1:128 0110 = 1:64 0101 = 1:32 0100 = 1:16 0011 = 1:8 0010 = 1:4 0001 = 1:2 0000 = 1:1
WDTEN<1:0>	CONFIG2H	Watchdog Timer Enable bits 11 = WDT enabled in hardware; SWDTEN bit is disabled 10 = WDT controlled by the SWDTEN bit 01 = WDT enabled when device is active, disabled when device is in Sleep; SWDTEN bit is disabled 00 = WDT disabled in hardware; SWDTEN bit is disabled
MCLRE	CONFIG3H	$\overline{\text{MCLR}}$ Pin Enable bit 1 = $\overline{\text{MCLR}}$ pin enabled, RE3 input pin disabled 0 = RE3 input pin enabled, $\overline{\text{MCLR}}$ pin disabled
P2BMX	CONFIG3H	CCP2 B Output MUX bit On 28-pin devices: 1 = P2B is on RB5 0 = P2B is on RC0 On 40-pin devices: 1 = P2B is on RD2 0 = P2B is on RC0
T3CMX	CONFIG3H	1 = T3CKI is on RC0 0 = T3CKI is on RB5
HFOFST	CONFIG3H	HFINTOSC Fast Start bit 1 = HFINTOSC output is not delayed 0 = HFINTOSC output is delayed until oscillator is stable (IOFS = 1)
CCP3MX	CONFIG3H	CCP3 MUX bit On 28-pin devices: 1 = CCP3 input/output is multiplexed with RB5 0 = CCP3 input/output is multiplexed with RC6 On 40-pin devices: 1 = CCP3 input/output is multiplexed with RB5 0 = CCP3 input/output is multiplexed with RE0
PBADEN	CONFIG3H	PORTB A/D Enable bit 1 = PORTB A/D<5:0> pins are configured as analog input channels on Reset 0 = PORTB A/D<5:0> pins are configured as digital I/O on Reset
CCP2MX	CONFIG3H	CCP2 MUX bit 1 = CCP2 input/output is multiplexed with RC1 0 = CCP2 input/output is multiplexed with RB3

TABLE 5-3: PIC18(L)F2XK22/4XK22 BIT DESCRIPTIONS (CONTINUED)

Bit Name	Configuration Words	Description
DEBUG	CONFIG4L	Background Debugger Enable bit 1 = Background debugger disabled, RB6 and RB7 configured as general purpose I/O pins 0 = Background debugger enabled, RB6 and RB7 are dedicated to In-Circuit Debug
XINST	CONFIG4L	Extended Instruction Set Enable bit 1 = Instruction set extension and Indexed Addressing mode enabled 0 = Instruction set extension and Indexed Addressing mode disabled (Legacy mode)
LVP	CONFIG4L	Low-Voltage Programming Enable bit If MCLRE = 1, then: 1 = Low-Voltage Programming enabled 0 = Low-Voltage Programming disabled If MCLRE = 0, then: LVP is disabled
STVREN	CONFIG4L	Stack Overflow/Underflow Reset Enable bit 1 = Reset on stack overflow/underflow enabled 0 = Reset on stack overflow/underflow disabled

TABLE 5-3: PIC18(L)F2XK22/4XK22 BIT DESCRIPTIONS (CONTINUED)

Bit Name	Configuration Words	Description
CP3	CONFIG5L	Code Protection bits (Block 3 code memory area) 1 = Block 3 is not code-protected 0 = Block 3 is code-protected
CP2	CONFIG5L	Code Protection bits (Block 2 code memory area) 1 = Block 2 is not code-protected 0 = Block 2 is code-protected
CP1	CONFIG5L	Code Protection bits (Block 1 code memory area) 1 = Block 1 is not code-protected 0 = Block 1 is code-protected
CP0	CONFIG5L	Code Protection bits (Block 0 code memory area) 1 = Block 0 is not code-protected 0 = Block 0 is code-protected
CPD	CONFIG5H	Code Protection bits (Data EEPROM) 1 = Data EEPROM is not code-protected 0 = Data EEPROM is code-protected
CPB	CONFIG5H	Code Protection bits (Boot Block memory area) 1 = Boot Block is not code-protected 0 = Boot Block is code-protected
WRT3	CONFIG6L	Write Protection bits (Block 3 code memory area) 1 = Block 3 is not write-protected 0 = Block 3 is write-protected
WRT2	CONFIG6L	Write Protection bits (Block 2 code memory area) 1 = Block 2 is not write-protected 0 = Block 2 is write-protected
WRT1	CONFIG6L	Write Protection bits (Block 1 code memory area) 1 = Block 1 is not write-protected 0 = Block 1 is write-protected
WRT0	CONFIG6L	Write Protection bits (Block 0 code memory area) 1 = Block 0 is not write-protected 0 = Block 0 is write-protected
WRTD	CONFIG6H	Write Protection bit (Data EEPROM) 1 = Data EEPROM is not write-protected 0 = Data EEPROM is write-protected
WRTB	CONFIG6H	Write Protection bit (Boot Block memory area) 1 = Boot Block is not write-protected 0 = Boot Block is write-protected
WRTC	CONFIG6H	Write Protection bit (Configuration registers) 1 = Configuration registers are not write-protected 0 = Configuration registers are write-protected

TABLE 5-3: PIC18(L)F2XK22/4XK22 BIT DESCRIPTIONS (CONTINUED)

Bit Name	Configuration Words	Description
EBTR3	CONFIG7L	Table Read Protection bit (Block 3 code memory area) 1 = Block 3 is not protected from table reads executed in other blocks 0 = Block 3 is protected from table reads executed in other blocks
EBTR2	CONFIG7L	Table Read Protection bit (Block 2 code memory area) 1 = Block 2 is not protected from table reads executed in other blocks 0 = Block 2 is protected from table reads executed in other blocks
EBTR1	CONFIG7L	Table Read Protection bit (Block 1 code memory area) 1 = Block 1 is not protected from table reads executed in other blocks 0 = Block 1 is protected from table reads executed in other blocks
EBTR0	CONFIG7L	Table Read Protection bit (Block 0 code memory area) 1 = Block 0 is not protected from table reads executed in other blocks 0 = Block 0 is protected from table reads executed in other blocks
EBTRB	CONFIG7H	Table Read Protection bit (Boot Block memory area) 1 = Boot Block is not protected from table reads executed in other blocks 0 = Boot Block is protected from table reads executed in other blocks
DEV<10:3>	DEVID2	Device ID bits These bits are used with the DEV<2:0> bits in the DEVID1 register to identify part number.
DEV<2:0>	DEVID1	Device ID bits These bits are used with the DEV<10:3> bits in the DEVID2 register to identify part number.
REV<4:0>	DEVID1	Revision ID bits These bits are used to indicate the revision of the device.

.

5.3 Single-Supply ICSP Programming

The LVP bit in Configuration register, CONFIG4L, enables Single-Supply (Low-Voltage) ICSP Programming. The LVP bit defaults to a '1' (enabled) from the factory.

If Single-Supply Programming mode is not used, the LVP bit can be programmed to a '0'. However, the LVP bit may only be programmed by entering the High-Voltage ICSP mode, where $\overline{MCLR}$/VPP/RE3 is raised to VIHH. Once the LVP bit is programmed to a '0', only the High-Voltage ICSP mode is available and only the High-Voltage ICSP mode can be used to program the device.

Note 1: The High-Voltage ICSP mode is always available, regardless of the state of the LVP bit, by applying VIHH to the $\overline{MCLR}$/VPP/RE3 pin.

5.4 Embedding Configuration Word Information in the HEX File

To allow portability of code, a PIC18(L)F2XK22/4XK22 programmer is required to read the Configuration Word locations from the hex file. If Configuration Word information is not present in the hex file, then a simple warning message should be issued. Similarly, while saving a hex file, all Configuration Word information must be included. An option to not include the Configuration Word information may be provided. When embedding Configuration Word information in the hex file, it should start at address 300000h.

Microchip Technology Inc. feels strongly that this feature is important for the benefit of the end customer.

5.5 Embedding Data EEPROM Information In the HEX File

To allow portability of code, a PIC18(L)F2XK22/4XK22 programmer is required to read the data EEPROM information from the hex file. If data EEPROM information is not present, a simple warning message should be issued. Similarly, when saving a hex file, all data EEPROM information must be included. An option to not include the data EEPROM information may be provided. When embedding data EEPROM information in the hex file, it should start at address F00000h.

Microchip Technology Inc. believes that this feature is important for the benefit of the end customer.

5.6 Checksum Computation

The checksum is calculated by summing the following:

- The contents of all code memory locations
- The Configuration Word, appropriately masked
- ID locations (Only if any portion of program memory is code-protected)

The Least Significant 16 bits of this sum are the checksum.

Code protection limits access to program memory by both external programmer (code-protect) and code execution (table read protect). The ID locations, when included in a code protected checksum, contain the checksum of an unprotected part. The unprotected checksum is distributed: one nibble per ID location. Each nibble is right justified.

Table 5-4 describes how to calculate the checksum for each device.

Note: The checksum calculation differs depending on the code-protect setting. Since the code memory locations read out differently depending on the code-protect setting, the table describes how to manipulate the actual code memory values to simulate the values that would be read from a protected device. When calculating a checksum by reading a device, the entire code memory can simply be read and summed. The Configuration Word and ID locations can always be read.

TABLE 5-4: CHECKSUM COMPUTATION

Device	Code-Protect	Checksum	Blank Value	0xAA at 0 and Max Address
PIC18FX3K22 PIC18LFX3K22	None	SUM[0000:01FF]+SUM[0200:0FFF]+SUM[1000:1FFF]+ (CONFIG1L & 00h)+ (CONFIG1H & FFh)+(CONFIG2L & 1Fh)+(CONFIG2H & 3Fh)+ (CONFIG3L & 00h)+(CONFIG3H & BFh)+(CONFIG4L & C5h)+ (CONFIG4H & 00h)+(CONFIG5L & 03h)+(CONFIG5H & C0h)+ (CONFIG6L & 03h)+(CONFIG6H & E0h)+(CONFIG7L & 03h)+ (CONFIG7H & 40h)	E3B0	E306
	Boot Block	SUM[0200:0FFF]+SUM[1000:1FFF]+ (CONFIG1L & 00h)+(CONFIG1H & FFh)+(CONFIG2L & 1Fh)+ (CONFIG2H & 3Fh)+(CONFIG3L & 00h)+(CONFIG3H & BFh)+ (CONFIG4L & C5h)+(CONFIG4H & 00h)+(CONFIG5L & 03h)+ (CONFIG5H & C0h)+(CONFIG6L & 03h)+(CONFIG6H & E0h)+ (CONFIG7L & 03h)+(CONFIG7H & 40h)+SUM_ID	E58C	E532
	Boot/ Block 0	SUM[1000:1FFF]+(CONFIG1L & 00h)+ (CONFIG1H & FFh)+(CONFIG2L & 1Fh)+(CONFIG2H & 3Fh)+ (CONFIG3L & 00h)+(CONFIG3H & BFh)+(CONFIG4L & C5h)+ (CONFIG4H & 00h)+(CONFIG5L & 03h)+(CONFIG5H & C0h)+ (CONFIG6L & 03h)+(CONFIG6H & E0h)+(CONFIG7L & 03h)+ (CONFIG7H & 40h)+SUM_ID	F38B	F331
	All	(CONFIG1L & 00h)+(CONFIG1H & FFh)+(CONFIG2L & 1Fh)+ (CONFIG2H & 3Fh)+(CONFIG3L & 00h)+(CONFIG3H & BFh)+ (CONFIG4L & C5h)+(CONFIG4H & 00h)+(CONFIG5L & 03h)+ (CONFIG5H & C0h)+(CONFIG6L & 03h)+(CONFIG6H & E0h)+ (CONFIG7L & 03h)+(CONFIG7H & 40h)+SUM_ID	0389	0384
PIC18FX4K22 PIC18LFX4K22	None	SUM[0000:07FF]+SUM[0800:1FFF]+SUM[2000:3FFF]+ (CONFIG1L & 00h)+ (CONFIG1H & FFh)+(CONFIG2L & 1Fh)+(CONFIG2H & 3Fh)+ (CONFIG3L & 00h)+(CONFIG3H & BFh)+(CONFIG4L & C5h)+ (CONFIG4H & 00h)+(CONFIG5L & 03h)+(CONFIG5H & C0h)+ (CONFIG6L & 03h)+(CONFIG6H & E0h)+(CONFIG7L & 03h)+ (CONFIG7H & 40h)	C3B0	C306
	Boot Block	SUM[0800:1FFF]+SUM[2000:3FFF]+ (CONFIG1L & 00h)+(CONFIG1H & FFh)+(CONFIG2L & 1Fh)+ (CONFIG2H & 3Fh)+(CONFIG3L & 00h)+(CONFIG3H & BFh)+ (CONFIG4L & C5h)+(CONFIG4H & 00h)+(CONFIG5L & 03h)+ (CONFIG5H & C0h)+(CONFIG6L & 03h)+(CONFIG6H & E0h)+ (CONFIG7L & 03h)+(CONFIG7H & 40h)+SUM_ID	CB8A	CB30
	Boot/ Block 0	SUM[2000:3FFF]+(CONFIG1L & 00h)+ (CONFIG1H & FFh)+(CONFIG2L & 1Fh)+(CONFIG2H & 3Fh)+ (CONFIG3L & 00h)+(CONFIG3H & BFh)+(CONFIG4L & C5h)+ (CONFIG4H & 00h)+(CONFIG5L & 03h)+(CONFIG5H & C0h)+ (CONFIG6L & 03h)+(CONFIG6H & E0h)+(CONFIG7L & 03h)+ (CONFIG7H & 40h)+SUM_ID	D389	D32F
	All	(CONFIG1L & 00h)+(CONFIG1H & FFh)+(CONFIG2L & 1Fh)+ (CONFIG2H & 3Fh)+(CONFIG3L & 00h)+(CONFIG3H & BFh)+ (CONFIG4L & C5h)+(CONFIG4H & 00h)+(CONFIG5L & 03h)+ (CONFIG5H & C0h)+(CONFIG6L & 03h)+(CONFIG6H & E0h)+ (CONFIG7L & 03h)+(CONFIG7H & 40h)+SUM_ID	0387	0382

Legend:

Item		Description
CONFIGx	=	Configuration Word
SUM[a:b]	=	Sum of locations, a to b inclusive
SUM_ID	=	Byte-wise sum of lower four bits of all customer ID locations
+	=	Addition
&	=	Bit-wise AND

TABLE 5-4: CHECKSUM COMPUTATION (CONTINUED)

Device	Code-Protect	Checksum	Blank Value	0xAA at 0 and Max Address
PIC18FX5K22 PIC18LFX5K22	None	SUM[0000:07FF]+SUM[0800:1FFF]+SUM[2000:3FFF]+ SUM[4000:5FFF]+SUM[6000:7FFF]+(CONFIG1L & 00h)+ (CONFIG1H & FFh)+(CONFIG2L & 1Fh)+(CONFIG2H & 3Fh)+ (CONFIG3L & 00h)+(CONFIG3H & BFh)+(CONFIG4L & C5h)+ (CONFIG4H & 00h)+(CONFIG5L & 0Fh)+(CONFIG5H & C0h)+ (CONFIG6L & 0Fh)+(CONFIG6H & E0h)+(CONFIG7L & 0Fh)+ (CONFIG7H & 40h)	83D4	832A
	Boot Block	SUM[0800:1FFF]+SUM[2000:3FFF]+SUM[4000:5FFF]+SUM[6000:7FFF]+ (CONFIG1L & 00h)+(CONFIG1H & FFh)+(CONFIG2L & 1Fh)+ (CONFIG2H & 3Fh)+(CONFIG3L & 00h)+(CONFIG3H & BFh)+ (CONFIG4L & C5h)+(CONFIG4H & 00h)+(CONFIG5L & 0Fh)+ (CONFIG5H & C0h)+(CONFIG6L & 0Fh)+(CONFIG6H & E0h)+ (CONFIG7L & 0Fh)+(CONFIG7H & 40h)+SUM_ID	8BB0	8B56
	Boot/ Block 0/ Block 1	SUM[4000:5FFF]+SUM[6000:7FFF]+(CONFIG1L & 00h)+ (CONFIG1H & FFh)+(CONFIG2L & 1Fh)+(CONFIG2H & 3Fh)+ (CONFIG3L & 00h)+(CONFIG3H & BFh)+(CONFIG4L & C5h)+ (CONFIG4H & 00h)+(CONFIG5L & 0Fh)+(CONFIG5H & C0h)+ (CONFIG6L & 0Fh)+(CONFIG6H & E0h)+(CONFIG7L & 0Fh)+ (CONFIG7H & 40h)+SUM_ID	C3AD	C353
	All	(CONFIG1L & 00h)+(CONFIG1H & FFh)+(CONFIG2L & 1Fh)+ (CONFIG2H & 3Fh)+(CONFIG3L & 00h)+(CONFIG3H & BFh)+ (CONFIG4L & C5h)+(CONFIG4H & 00h)+(CONFIG5L & 0Fh)+ (CONFIG5H & C0h)+(CONFIG6L & 0Fh)+(CONFIG6H & E0h)+ (CONFIG7L & 0Fh)+(CONFIG7H & 40h)+SUM_ID	03A1	039C
PIC18FX6K22 PIC18LFX6K22	None	SUM[0000:07FF]+SUM[0800:3FFF]+SUM[4000:7FFF]+ SUM[8000:BFFF]+SUM[C000:FFFF]+(CONFIG1L & 00h)+ (CONFIG1H & FFh)+(CONFIG2L & 1Fh)+(CONFIG2H & 3Fh)+ (CONFIG3L & 00h)+(CONFIG3H & BFh)+(CONFIG4L & C5h)+ (CONFIG4H & 00h)+(CONFIG5L & 0Fh)+(CONFIG5H & C0h)+ (CONFIG6L & 0Fh)+(CONFIG6H & E0h)+(CONFIG7L & 0Fh)+ (CONFIG7H & 40h)	03D4	032A
	Boot Block	SUM[0800:3FFF]+SUM[4000:7FFF]+SUM[8000:BFFF]+SUM[C000:FFFF] + (CONFIG1L & 00h)+(CONFIG1H & FFh)+(CONFIG2L & 1Fh)+ (CONFIG2H & 3Fh)+(CONFIG3L & 00h)+(CONFIG3H & BFh)+ (CONFIG4L & C5h)+(CONFIG4H & 00h)+(CONFIG5L & 0Fh)+ (CONFIG5H & C0h)+(CONFIG6L & 0Fh)+(CONFIG6H & E0h)+ (CONFIG7L & 0Fh)+(CONFIG7H & 40h)+SUM_ID	0BA8	0B4E
	Boot/ Block 0/ Block 1	SUM[8000:BFFF]+SUM[C000:FFFF]+(CONFIG1L & 00h)+ (CONFIG1H & FFh)+(CONFIG2L & 1Fh)+(CONFIG2H & 3Fh)+ (CONFIG3L & 00h)+(CONFIG3H & BFh)+(CONFIG4L & C5h)+ (CONFIG4H & 00h)+(CONFIG5L & 0Fh)+(CONFIG5H & C0h)+ (CONFIG6L & 0Fh)+(CONFIG6H & E0h)+(CONFIG7L & 0Fh)+ (CONFIG7H & 40h)+SUM_ID	43A5	434B
	All	(CONFIG1L & 00h)+(CONFIG1H & FFh)+(CONFIG2L & 1Fh)+ (CONFIG2H & 3Fh)+(CONFIG3L & 00h)+(CONFIG3H & BFh)+ (CONFIG4L & C5h)+(CONFIG4H & 00h)+(CONFIG5L & 0Fh)+ (CONFIG5H & C0h)+(CONFIG6L & 0Fh)+(CONFIG6H & E0h)+ (CONFIG7L & 0Fh)+(CONFIG7H & 40h)+SUM_ID	0399	0394

Legend:

Item		Description
CONFIGx	=	Configuration Word
SUM[a:b]	=	Sum of locations, a to b inclusive
SUM_ID	=	Byte-wise sum of lower four bits of all customer ID locations
+	=	Addition
&	=	Bit-wise AND

6.0 AC/DC CHARACTERISTICS TIMING REQUIREMENTS FOR PROGRAM/ VERIFY TEST MODE

Standard Operating Conditions
Operating Temperature: 25°C is recommended

Param No.	Sym.	Characteristic		Min.	Max.	Units	Conditions
D110	VIHH	High-Voltage Programming Voltage on MCLR/VPP/RE3		VDD + 4.5	9	V	
D111	VDD	Supply Voltage During Programming	PIC18LF	1.80	3.60	V	Row Erase/Write
				2.7	3.60	V	Bulk Erase operations
			PIC18F	1.8	5.5	V	Row Erase/Write
				2.7	5.5	V	Bulk Erase operations
D112	IPP	Programming Current on MCLR/VPP/RE3		—	300	µA	
D113	IDDP	Supply Current During Programming		—	10	mA	
D031	VIL	Input Low Voltage		VSS	0.2 VDD	V	
D041	VIH	Input High Voltage		0.8 VDD	VDD	V	
D080	VOL	Output Low Voltage		—	0.6	V	IOL = 8.5 mA @ 3.0V
D090	VOH	Output High Voltage		VDD – 0.7	—	V	IOH = 3.0 mA @ 3.0V
D012	CIO	Capacitive Loading on I/O pin (PGD)		—	50	pF	To meet AC specifications
P1	TR	MCLR/VPP/RE3 Rise Time to enter Program/Verify mode		—	1.0	µs	**(Note 1)**
P2	TPGC	Serial Clock (PGC) Period		100	—	ns	VDD = 3.6V
				1	—	µs	VDD = 1.8V
P2A	TPGCL	Serial Clock (PGC) Low Time		40	—	ns	VDD = 3.6V
				400	—	ns	VDD = 1.8V
P2B	TPGCH	Serial Clock (PGC) High Time		40	—	ns	VDD = 3.6V
				400	—	ns	VDD = 1.8V
P3	TSET1	Input Data Setup Time to Serial Clock ↓		15	—	ns	
P4	THLD1	Input Data Hold Time from PGC ↓		15	—	ns	
P5	TDLY1	Delay between 4-bit Command and Command Operand		40	—	ns	
P5A	TDLY1A	Delay between 4-bit Command Operand and next 4-bit Command		40	—	ns	
P6	TDLY2	Delay between Last PGC ↓ of Command Byte to First PGC ↑ of Read of Data Word		20	—	ns	
P9	TDLY5	PGC High Time (minimum programming time)		1	—	ms	Externally Timed
P9A	TDLY5A	PGC High Time		5		ms	Configuration Word programming time
P10	TDLY6	PGC Low Time after Programming (high-voltage discharge time)		200	—	µs	
P11	TDLY7	Delay to allow Self-Timed Bulk Erase to occur	PIC18(L)F X5/X6	15	—	ms	
			PIC18(L)F X3/X4	12	—	ms	
P11A	TDRWT	Data Write Polling Time		4	—	ms	
P11B	TDLY7B	Delay for Self-Timed Memory Write		2	—	ms	
P12	THLD2	Input Data Hold Time from MCLR/VPP/RE3 ↑		2	—	µs	
P13	TSET2	VDD ↑ Setup Time to MCLR/VPP/RE3 ↑		100	—	ns	

Note 1: Do not allow excess time when transitioning MCLR between VIL and VIHH; this can cause spurious program executions to occur. The maximum transition time is:
1 TCY + TPWRT (if enabled) + 1024 TOSC (for LP, HS, HS/PLL and XT modes only) + 2 ms (for HS/PLL mode only) + 1.5 µs (for EC mode only) where TCY is the instruction cycle time, TPWRT is the Power-up Timer period and TOSC is the oscillator period. For specific values, refer to the Electrical Characteristics section of the device data sheet for the particular device.

Standard Operating Conditions Operating Temperature: 25°C is recommended						
Param No.	**Sym.**	**Characteristic**	**Min.**	**Max.**	**Units**	**Conditions**
P14	TVALID	Data Out Valid from PGC ↑	10	—	ns	
P15	THLD4	Input data hold time from $\overline{MCLR}$ ↑	400	—	µs	
P16	TDLY8	Delay between Last PGC ↓ and $\overline{MCLR}$/VPP/RE3 ↓	0	—	s	
P17	THLD3	$\overline{MCLR}$/VPP/RE3 ↓ to VDD ↓	—	100	ns	
P18	TKEY1	Delay from First $\overline{MCLR}$ ↓ to first PGC ↑ for Key Sequence on PGD	1	—	ms	
P19	THIZ	Delay from PGC ↑ to PGD High-Z	3	10	ns	
P20	TKEY2	Delay from Last PGC ↓ for Key Sequence on PGD to Second $\overline{MCLR}$ ↑	40	—	ns	

Note 1: Do not allow excess time when transitioning $\overline{MCLR}$ between VIL and VIHH; this can cause spurious program executions to occur. The maximum transition time is:
1 TCY + TPWRT (if enabled) + 1024 TOSC (for LP, HS, HS/PLL and XT modes only) + 2 ms (for HS/PLL mode only) + 1.5 µs (for EC mode only) where TCY is the instruction cycle time, TPWRT is the Power-up Timer period and TOSC is the oscillator period. For specific values, refer to the Electrical Characteristics section of the device data sheet for the particular device.

NOTES:

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Trademarks

ISBN: 978-1-60932-156-7

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

MICROCHIP

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Cleveland
Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Chongqing
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-6578-300
Fax: 886-3-6578-370

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

01/05/10

8. Esquema placa de desarrollo: