UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

INGENIERÍA INDUSTRIAL

"MANEJO DE UNA PANTALLA TÁCTIL CON EL PIC32 PARA EL CONTROL DE DISPOSITIVOS EXTERNOS"

PROYECTO FIN DE CARRERA

Enero – 2010

AUTOR: Jesús Fernández González

DIRECTORA: María Asunción Vicente Ripoll

VISTO BUENO Y CALIFICACIÓN DEL PROYECTO

Título proyecto:

MANEJO DE UNA PANTALLA TÁCTIL CON EL PIC32 PARA EL CONTROL DE DISPOSITIVOS EXTERNOS.

Proyectante: Jesús Fernánez González

Directora: María Asunción Vicente Ripoll

VºBº directora del proyecto:

Fdo.: María Asunción Vicente Ripoll

Lugar y fecha: _____

CALIFICACIÓN NUMÉRICA ☐ MATRÍCULA DE HONOR ☐

Méritos justificativos en el caso de conceder Matrícula de Honor:

_____

_____

_____

_____

Conforme presidente:	Conforme secretario:	Conforme vocal:
Fdo.:	Fdo.:	Fdo.:

Lugar y fecha: _____

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE FIGURAS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN Y OBJETIVOS

CAPÍTULO 1. INTRODUCCIÓN Y OBJETIVOS

1.1. OBJETIVOS

El presente Proyecto Fin de Carrera se centra en el desarrollo de aplicaciones prácticas para la programación de microcontroladores. En concreto, el trabajo ha sido realizado con microcontroladores PIC de gama alta (familia de microcontroladores de 32bits), en particular con los modelos PIC32MX360F512L y el PIC32MX460F512L.

En los últimos años han tenido un gran auge los microcontroladores PIC fabricados por Microchip Technology Inc. Los PIC (Peripheral Interface Controller) son una familia de microcontroladores que ha tenido gran aceptación y desarrollo gracias a que sus buenas características, bajo precio, reducido consumo, pequeño tamaño, gran calidad, fiabilidad y abundancia de información, los convierten en muy fáciles, cómodos y rápidos de utilizar.

El trabajo realizado en este proyecto se puede dividir en 4 tareas básicas:

- Estudio del hardware y software (juego de instrucciones) del PIC32 dada su reciente puesta en el mercado.
- Análisis de programas ejemplo para la placa PIC32 Starter Kit y desarrollo de los primeros programas ejemplo para el PIC32.
- Estudio del sistema de desarrollo Explorer16 y desarrollo de código fuente para este sistema.
- Desarrollo de aplicaciones de control mediante el uso de la pantalla táctil incorporando posteriormente un acelerómetro.

1.2. ESTRUCTURA

El proyecto fin de carrera *Manejo de una pantalla táctil con el PIC32 para el control de dispositivos externos* se divide en ocho capítulos y tres anexos. En este primer capítulo se realiza una breve introducción detallando los objetivos del proyecto y describiendo su estructura.

En el segundo capítulo se realiza una introducción a la arquitectura de los microcontroladores en general, y se describe con detalle la familia de los microcontroladores PIC de 32 bits.

En el tercer capítulo se presentan los distintos programas que se han utilizado a lo largo del proyecto, tanto para la grabación de los programas en los microcontroladores, como para la depuración y simulación del código fuente.

Seguidamente, en el cuarto capítulo se describe la metodología realizada durante la evaluación del PIC32 Starter Kit, desarrollando los primeros programas fuente para el PIC32.

Posteriormente, en el quinto capítulo se analiza el sistema de desarrollo Explorer16, se presenta su arquitectura física y el software necesario para hacer uso de ella. Además se presentan una serie de programas ejemplo desarrollados para esta tarjeta.

En el capítulo sexto se estudia el hardware de la pantalla táctil presente en la placa de expansión “Graphics PICtail Plus Daughter Board (versión 2)” y se analiza el funcionamiento de la librería “Microchip Graphic Library versión 1.6.”.

A continuación, en el capítulo séptimo se detallan las aplicaciones desarrolladas para la tarjeta gráfica “Graphics PICtail Plus Daughter Board (versión 2)” usando tanto el sistema de desarrollo Explorer16 como la tarjeta “I/O Expansion Board” a la cual se ha incorporado el acelerómetro ADXL330.

Finalmente, en el capítulo octavo se hace una recopilación de conclusiones, así como de los posibles trabajos futuros.

Por último, al final del documento se encuentran los Anexos A, B y C. En el anexo A se describe de qué manera se puede configurar la memoria del PIC32. En el anexo B se describen distintos aspectos a tener en cuenta a la hora de programar sobre el PIC32 que no se comentan en profundidad en ninguno de los capítulos del presente proyecto. Mientras que el anexo C se realiza un breve estudio del hardware del acelerómetro ADLX330 para la detección de movimiento en el espacio.

2. EL MICROCONTROLADOR PIC32: HARDWARE Y SOFTWARE

CAPÍTULO 2. EL MICROCONTROLADOR PIC32: HARDWARE Y SOFTWARE

2.1. INTRODUCCIÓN

Hace unos años, los sistemas de control se implementaban usando exclusivamente la lógica de componentes, lo que hacía que fuesen dispositivos de gran tamaño y muy pesados. Para facilitar una velocidad más alta y mejorar la eficiencia de estos dispositivos de control, se trató de reducir su tamaño apareciendo así los microprocesadores. Siguiendo con el proceso de miniaturización, el siguiente paso consistió en la fabricación de un controlador que integrase todos sus componentes en un solo chip. A esto se le conoce con el nombre de microcontrolador.

2.1.1. DIFERENCIAS ENTRE UN MICROCONTROLADOR Y UN MICROPROCESADOR

Un microprocesador es un circuito integrado que contiene la Unidad Central de Proceso (CPU) de un computador, que consta de la Unidad de Control y de los buses necesarios para comunicarse con los distintos módulos. Mientras que un microcontrolador, es un circuito integrado programable que contiene todos los bloques necesarios para controlar el funcionamiento de una tarea determinada:

- Procesador o CPU: que interpreta las instrucciones de programa.
- Memoria RAM para almacenar las variables necesarias.
- Memoria EPROM/PROM/ROM para el programa tipo.
- Puertos E/S para comunicarse con el exterior.
- Diversos módulos para el control de periféricos.
- Generador de impulsos de reloj que sincroniza el funcionamiento de todo el sistema.

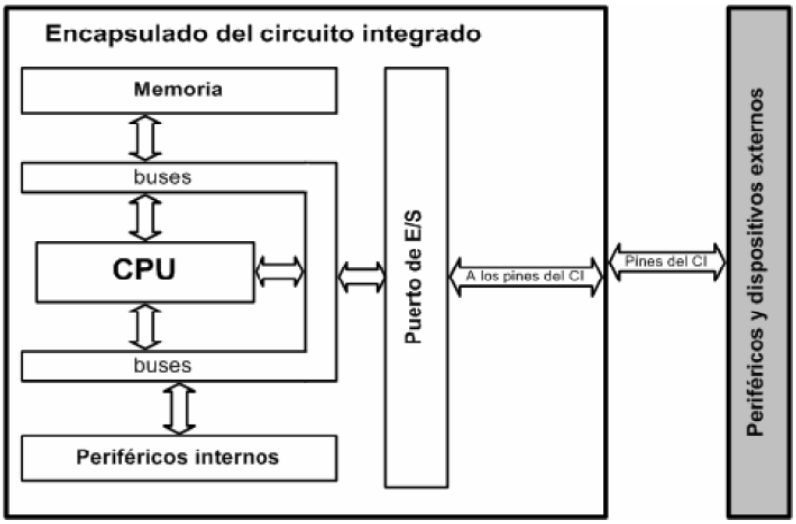


Figura 2.1: Estructura básica de un microcontrolador.

En la Figura 2.1, se puede ver al microcontrolador embebido dentro de un encapsulado de circuito integrado, con su procesador, buses, memoria, periféricos y puertos de entrada/salida. Fuera del encapsulado se ubican otros circuitos para completar periféricos internos y dispositivos que pueden conectarse a los pines de entrada/salida.

Un microcontrolador es, por tanto, un circuito o chip que incluye en su interior las tres unidades funcionales de un ordenador: CPU, Memoria y Unidades de E/S, es decir, se trata de un computador completo en un solo circuito integrado.

2.1.2. APLICACIONES DE LOS MICROCONTROLADORES

Cada vez existen más productos que incorporan un microcontrolador con el fin de aumentar sustancialmente sus prestaciones, reducir su tamaño y coste, mejorar su fiabilidad y disminuir el consumo.

Los microcontroladores a menudo se encuentran en aplicaciones domésticas y en equipos informáticos tales como: microondas, refrigeradores, lavadoras, televisión, equipos de música, ordenadores, impresoras, módems, lectores de discos, etc.

Figura 2.2: Ejemplos de aplicaciones de microcontroladores.

Los microcontroladores también son muy utilizados en robótica, donde la comunicación entre controladores es muy importante. Esto hace posible muchas tareas específicas al distribuir un gran número de microcontroladores por todo el sistema. La comunicación entre cada microcontrolador y uno central, permite procesar la información por un ordenador central, o transmitirlo a otros microcontroladores del sistema.

Otro ejemplo de aplicación de los microcontroladores, es la de la utilización para monitorizar y gravar parámetros medioambientales (temperatura, humedad, precipitaciones, etc.). El pequeño tamaño, el bajo consumo de potencia y su flexibilidad hacen de este dispositivo, una herramienta adecuada para este tipo de aplicaciones.

2.1.3. MODELOS DE MICROCONTROLADORES DE LA MARCA MICROCHIP

Diversos fabricantes ofrecen amplias gamas de microcontroladores para todas las necesidades. Nosotros vamos a utilizar microcontroladores de la marca microchip, al ser la marca con un mayor número de modelos de estos y por su mayor utilización tanto profesionalmente como por aficionados.

Dentro de la marca de microchip, nos encontramos con varios modelos de PIC. Estos se clasifican de acuerdo a la longitud de sus instrucciones dando lugar a 3 grandes tipos de PIC.

Clasificación	Longitud de instrucciones	Modelos de PIC
Gama Baja	8 bits	Pic10, Pic12, Pic16, Pic18
Gama Media	16 bits	Pic24F, Pic24H, dsPIC30, dsPIC33
Gama Alta	32 bits	Pic32

Tabla 2.1: Modelos de microcontroladores de la marca Microchip.

Además, conforme aumentamos la longitud de las instrucciones va a aumentar la funcionalidad, las prestaciones ofrecidas pero también la complejidad de las instrucciones y de su uso.

Figura 2.3: PIC16F84A, modelo de microcontrolador muy popular en la docencia de PICs.

2.2. MICROCONTORLADOR PIC32

2.2.1. ELECCIÓN DEL MICROCONTROLADOR

Existe una gran diversidad de microcontroladores, como se ha podido comprobar con anterioridad. Dependiendo de la potencia y características que se necesiten, se pueden elegir microcontroladores de 8, 16 ó 32 bits. Aunque las prestaciones de los microcontroladores de 16 y 32 bits son superiores a los de 8 bits, la realidad es que los microcontroladores de 8 bits dominan el mercado. Esto es debido a que los microcontroladores de 8 bits son apropiados para la gran mayoría de las aplicaciones, por lo que no es necesario emplear microcontroladores más potentes y en consecuencia más caros.

Sin embargo, los modernos microcontroladores de 32 bits (puestos en el mercado a finales del año 2007) se van afianzando en el mercado, siendo las áreas de más interés el procesamiento de imágenes, las comunicaciones, las aplicaciones militares, los procesos industriales y el control de los dispositivos de almacenamiento masivo de datos.

A la hora de seleccionar el microcontrolador a utilizar en un diseño concreto se ha de tener en cuenta multitud de factores como la documentación, herramientas de desarrollo disponibles y su precio, la cantidad de fabricantes que lo producen y por supuesto las características del microcontrolador (tipo de memoria de programa, número de temporizadores, interrupciones, etc.).

En el caso particular de este proyecto, la elección del microcontrolador vino influenciada por su reciente puesta en el mercado del mismo, con motivo de analizar y estudiar las nuevas posibilidades que nos ofrecía el PIC32 respecto a sus anteriores. Dentro de los microcontroladores de gama alta vamos a utilizar dos modelos de 32 bits, el PIC32MX360F512L y el PIC32MX460F512L.

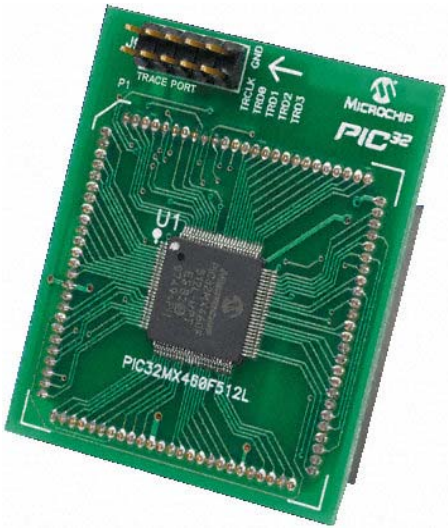


Figura 2.4: PIC32MX460512L.

2.2.2. CARACTERÍSTICAS DEL MICROCONTROLADOR PIC32MXXX

El PIC32X360F512L, introduce una nueva línea de dispositivos de microchip correspondientes a la familia de microcontroladores de 32 bit RISC (Reduced Instruction Set Computer). Además esta familia ofrece la opción de una nueva migración para estas aplicaciones de altas prestaciones las cuales pueden quedarse pequeñas para las plataformas de 16-bit.

A continuación se muestran las características generales del PIC32:

- Frecuencia de operación: Dc-80MHZ.
- Memoria de programa: 512Kbytes
- Memoria de Datos: 32Kbytes
- Recursos de interrupción/vectores: 95/63
- Puertos de E/S: Puertos A, B, C, D, F, G
    - Total pins E/S: 85
- Canales DMA(Direct memory Access): 4
- Timers:
    - Número total (16bit): 5
    - 32-bit (pareja de 16-bit): 2
    - Timer de núcleo de 32 bit: 1

- Canales de captura de entrada: 5
- Canales de salida Comparadores/PWN
- Notificación de cambio de entradas por interrupción: 22
- Comunicaciones en serie:
  - UART: 2
  - SPI(3cables/4cables): 2
  - I^2C: 2
- Comunicaciones paralelas(PMP/PSP): 8bit/16bit
- JTAG boundary SCAN
- JTAG debug and program
- ICSP 2-wire debug and program:
- Instrucción trace
- Hardware break points: 6 instructions, 2 Data
- Modulo de 10-bit analógico-digital (Canales de entrada): 16
- Comparadores analógicos: 2
- Interno LDO
- Resets (y retrasos): POR, BOR, MCLR, WDT, SWT(software reset), CM(configuration Bit Mismatch)
- 100 pin TQFP

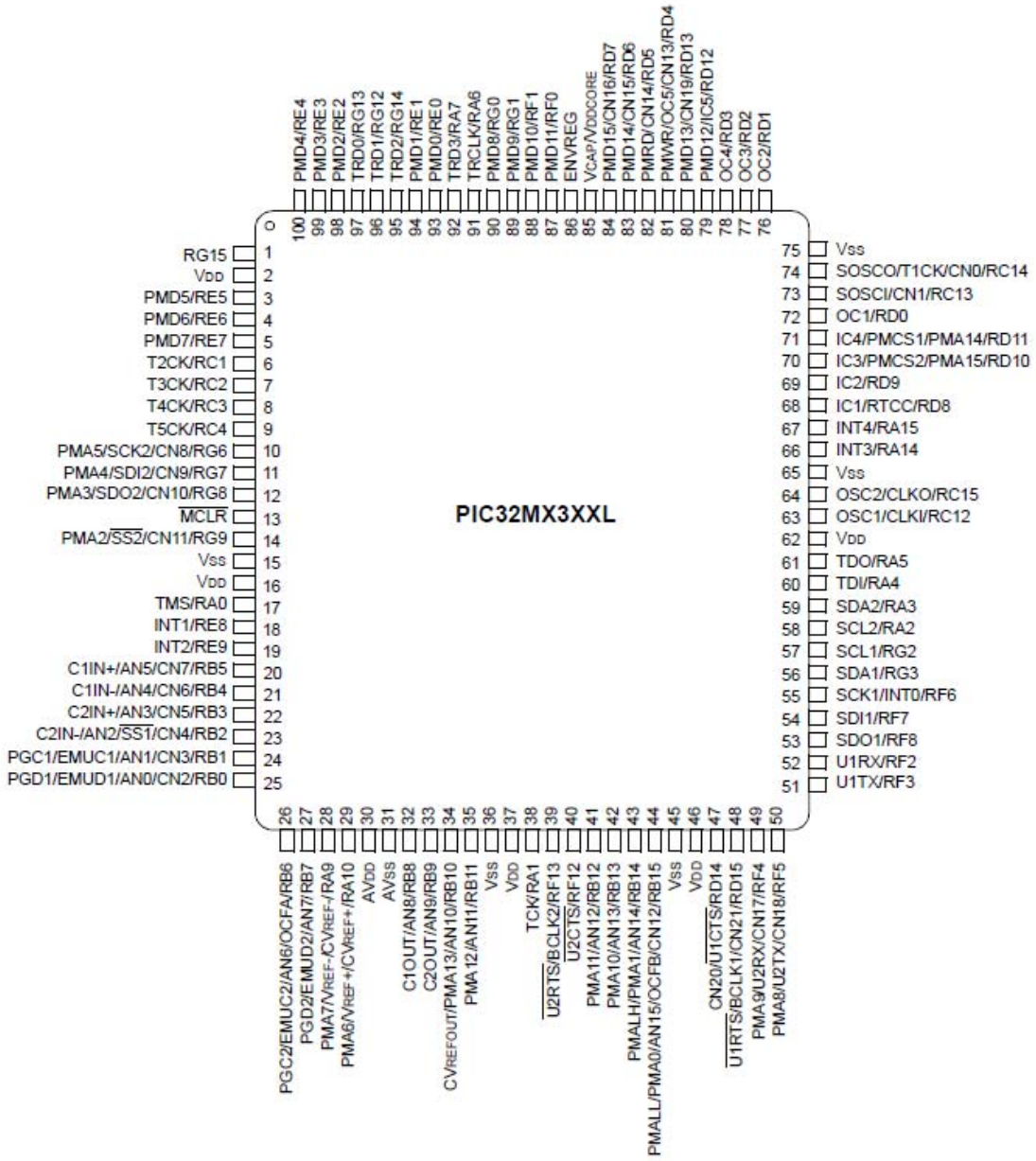


Figura 2.5: 100 pines del PIC32MX3XXL.

Por otra parte, la arquitectura del PIC32 ha sido descompuesta en los siguientes bloques funcionales:

- Núcleo MCU
- Memoria del sistema
- Periféricos
- Integración del sistema

2.2.3. NÚCLEO MCU

El corazón del PIC32 es el núcleo M4K CPU basado en la tecnología MIPS32 [1]. La CPU realiza las operaciones bajo el programa de control. Las instrucciones son leídas y cargadas por la CPU, decodificadas y ejecutadas síncronamente. Estas instrucciones se pueden almacenar en la memoria flash de programa o bien en la memoria de datos RAM.

Esta CPU del PIC32 se basa, por tanto, en una arquitectura de carga-almacenamiento la cual realiza la mayoría de operaciones en base a una serie de registros internos (contiene 32 registros de propósito general de 32 bits). Estas instrucciones específicas de carga y almacenamiento se utilizan para mover datos entre estos registros internos así como fuera del PIC.

Las características principales del núcleo MCU son las siguientes:

- Núcleo RISC MIPS32 M4K de 32 bits.
- ALU de un solo ciclo.
- Unidad de ejecución de carga-almacenamiento.
- 5 estados pipeline.
- Buses de direcciones y datos de 32 bits.
- Archivos de registro de propósito general de 32 bits.
- FMT-Fixed Mapping Translation.
- FMDU-Fast-Multiply-Divide Unit.
- MIPS32 Compatible instruction set.
- MIPS16e Code Compression Instruction Set Architecture Support.
- Instrucciones de 16 y 32 bits, optimizadas para lenguajes de alto nivel como C.
- Puerto debug EJTAG
- Rendimiento hasta 1.5DMIPS/MHz
- Protección del código interno para ayudar a proteger la propiedad intelectual.

A continuación vamos a ir describiendo algunas de estas características así como los conceptos más importantes del Núcleo MCU.

2.2.3.1 Estados pipeline

Los estados del pipeline son 5:

- Estado de instrucción (I)
- Estado de ejecución (E)
- Estado de memoria (M)
- Estado de alinear (A)
- Estado de reescribir (W)

Estado I: Instrucción Fetch

Durante el estado I una instrucción es cargada y leída desde SRAM y las instrucciones MIPS16e son convertidas a instrucciones en MIPS32.

Estado E: Ejecución

Durante el estado E:

- Los operandos son leídos y cargados desde el archivo de registro.
- Operandos de los estados M y A son desviados a este estado.
- La unidad aritmético lógica (ALU) empieza las operaciones aritméticas o lógicas para las instrucciones registro a registro.
- La ALU calcula la dirección virtual de los datos para las instrucciones de cargar y guardar y el MMU (Memory Management Unit) realiza el traslado de la dirección virtual a la física.
- La ALU determina si para una condición de bifurcación esta es verdadera y calcula la dirección objetivo de la ramificación.
- Las instrucciones lógicas seleccionan una dirección de instrucción y el MMU realiza la traslación de la dirección virtual a la física.
- Todas las operaciones de división y multiplicación empiezan en este estado.

Estado M: Memory Fetch

Durante este estado:

- Terminan las operaciones aritméticas o lógicas de la ALU.
- Se realiza el acceso a los datos SRAM para las instrucciones de carga y almacenamiento.
- Los cálculos de multiplicaciones y divisiones continúan en la MDU (Multiply/Divide Unit). Si el cálculo termina antes que el IU (Integer Unit)

mueva la instrucción pasada al estado M, entonces la MDU mantiene el resultado en un registro temporal hasta que el IU mueve la instrucción al Estado A (y consecuentemente sabrá que no será eliminado)

Estado A: Alinear

Durante el estado A:

- Una operación MUL hace que el resultado esté disponible para la reescritura. El registro actual reescrito es ejecutado en el estado W.
- Desde este estado, la carga de datos o el resultado de la MDU están disponibles para su bypassing (comentado a continuación) en el estado E.

Estado W: Reescribir

Durante el registro W: Para registros a registros o cargar instrucciones, el resultado se reescribe en el archivo de registro.

El núcleo M4k implementa un mecanismo de desviación (bypassing) que permite que el resultado de una operación sea enviado directamente a la instrucción que lo necesita sin tener que escribir el resultado en el registro y entonces volver a leerlo.

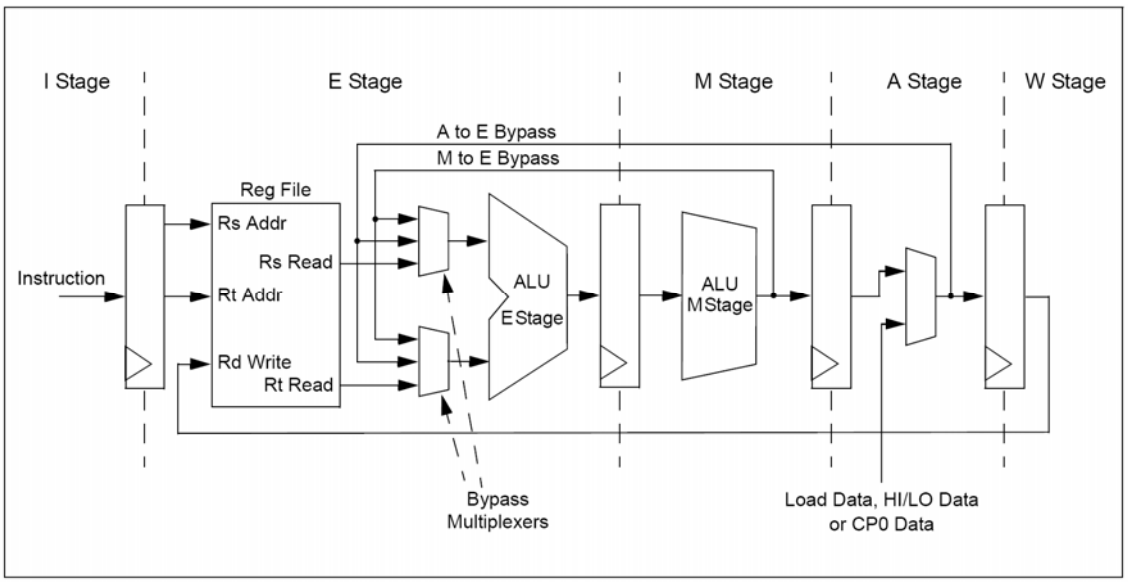


Figura 2.6: Esquema de los estados pipeline de la CPU del PIC32MX.

2.2.3.2 Unidad de Ejecución

La unidad de ejecución del PIC32 es la responsable de llevar a cabo el procesamiento de la mayoría de instrucciones del MIPS. Esta proporciona una ejecución para la mayoría de instrucciones en un solo ciclo, mediante la ejecución en pipeline (pipelining), donde las complejas operaciones son divididas en pequeñas partes llamados estados (explicados anteriormente).

2.2.3.3. MDU: Unidad de Multiplicación y División

La unidad de multiplicación y división realiza como su nombre indica, las operaciones de división y multiplicación. La MDU consiste en un multiplicador de 32x16, registros de resultado de acumulación (HI-LO) y todos los multiplexores y la lógica de control requerida para realizar estas funciones. Esta unidad tiene unas altas prestaciones ya que soporta la ejecución de una operación de multiplicación de 16x16 o 32x16 cada ciclo de reloj, y una operación de multiplicación de 32x32 se realiza cada dos ciclos de reloj. Por otra parte, las operaciones de división se implementan con un solo bit cada ciclo de reloj mediante un algoritmo iterativo y requiere 35 ciclos de reloj en el peor de los casos para que se complete la operación. Tan pronto como el algoritmo detecta el signo del dividendo, si el tamaño actual es 24, 16 o 8 bits, el divisor salta 7, 15 o 23 de las 32 iteraciones.

Además, el M4K implementa una instrucción adicional de multiplicación, MUL, la cual consiste en que los 32 bits más bajos del resultado de la multiplicación se guardan en el archivo de registro en lugar de en el par de registros HI/LO. Para ello, se han diseñado dos instrucciones adicionales que se usan para ejecutar las operaciones de multiplicación y suma así como multiplicación y resta, esta son: multiplicación-suma (MADD/MADDU) y multiplicación-resta (MSUB/MSUBU). La instrucción MADD multiplica dos números y entonces suma el producto al contenido actual de los registros HI y LO. Similarmente, la instrucción MSUB multiplica los dos operandos y entonces resta el producto de los registros HI y LO. Estas operaciones se suelen usar en algoritmos de Digital Signal Processor (DSP).

2.2.3.4. Shadow Register Sets

El procesador del PIC32 implementa una copia de los registros de propósito general para usarlos como interrupciones de alta prioridad. Este banco extra de registro es conocido como shadow register set (controlados por los registros ubicados en la CP0) [2]. Cuando ocurre una interrupción de alta prioridad, el procesador automáticamente conmuta a los shadow register set sin la necesidad de una intervención del software.

Este banco de registros especiales permite reducir eficazmente, tanto la sobrecarga en el manejo de interrupciones como el tiempo de latencia.

2.2.3.5. Register Bypassing

Un Interlock o bloqueo, ocurre cuando una instrucción en el estado pipeline no puede avanzar debido a una dependencia de datos o una condición externa similar, por ejemplo, cuando una instrucción depende del resultado de una instrucción previa. El tipo de bloqueo hardware en el procesador MIPs es Slips. Estos permiten que una parte de la pipeline avance mientras otra parte de la pipeline permanece estática, se muestra un ejemplo en la siguiente figura:

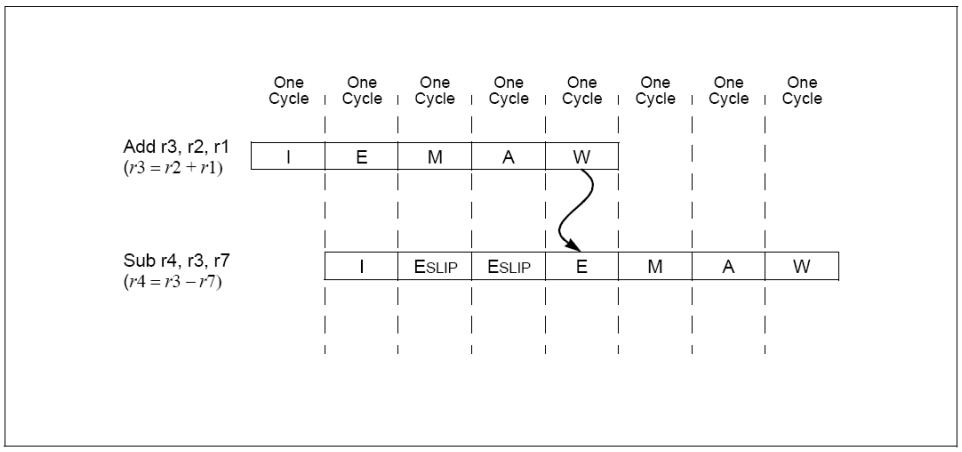


Figura 2.7: Esquema de los estados pipeline de la CPU del PIC32MX ante bloqueo hardware Slip.

Sin embargo, el procesador del PIC32 implementa un mecanismo llamado register bypassing que ayuda a reducir estos slips en la pipeline durante la ejecución. Cuando una instrucción esta en el estado E de la pipeline, el operando debe estar disponible para la siguiente instrucción.

El registro bypassing permite un atajo para conseguir directamente el operando desde la pipeline. De tal forma que una instrucción en el estado E puede recuperar un operando de otra instrucción que se está ejecutando o en el estado M o en el estado A de la pipeline. Mediante la figura que se muestra a continuación podemos ver las interdependencias descritas:

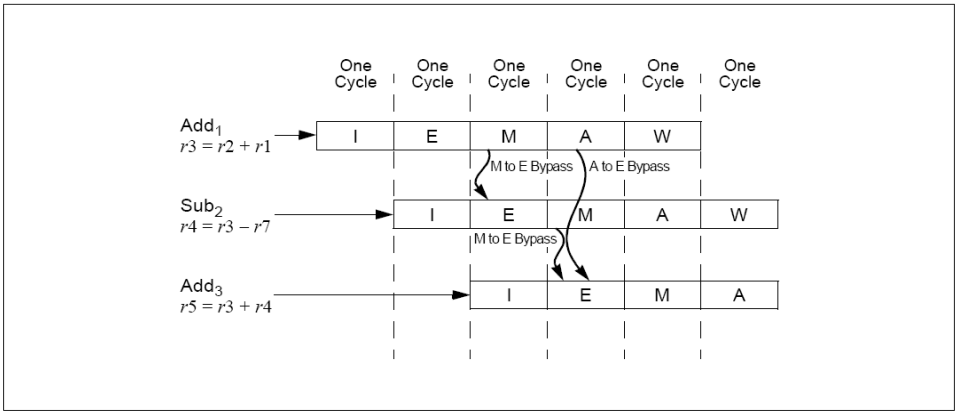


Figura 2.8: Esquema de los estados pipeline de la CPU del PIC32MX usando el mecanismo bypassing.

Evidentemente el uso del bypassing aumenta el rendimiento en el rango de una instrucción por ciclo reloj para las operaciones de la ALU.

2.2.3.6. BITS de estado de la ALU

A diferencia de la mayoría de los otros microcontroladores PIC, el PIC32 no usa un registro de Status con flags. Las banderas o flags se usan en la mayoría de los procesadores a ayudar a tomar una decisión para realizar operaciones durante la ejecución de un programa. Estas banderas se basan en, resultados de comparación de operaciones o bien en operaciones aritméticas. De tal forma que el programa ejecuta instrucciones basadas en estos valores de las banderas.

Sin embargo el PIC32 usa instrucciones que realizan una comparación y almacenan una bandera o valor en un registro de propósito general, ejecutándose entonces una rama condicionada, usando este registro de propósito general que actúa como un operando.

2.2.3.7. Mecanismo de interrupciones y excepciones

La familia de procesadores del PIC32 implementa un mecanismo eficiente y flexible para el manejo de las interrupciones y excepciones. Ambos se comportan de manera similar ya que la instrucción actual cambia temporalmente para ejecutarse un procedimiento especial. La diferencia entre las dos es que las interrupciones son usualmente un comportamiento normal y las excepciones son el resultado de un error en las condiciones, como por ejemplo un error en el bus al enviar o recibir datos.

Cuando ocurre una interrupción o una excepción, el procesador realiza los siguientes pasos:

- El PC (contador de programa) de la siguiente instrucción a ejecutar se guarda en el registro del coprocesador.
- La causa del registro es actualizada para conocer la razón de la excepción o de la interrupción.
- Status EXL o ERL es puesto a 1 como causa del modo de ejecución Kernel.
- Desde los valores de EBASE y SPACING se calcula el PC.
- El procesador comienza la ejecución del programa desde un nuevo PC.

2.2.4. JUEGO DE INSTRUCCIONES

La familia de los microprocesadores PIC32 están diseñados para usarse con un leguaje de programación de alto nivel como es el leguaje de programación C. El PIC32 soporta varios tipos de datos y usa un modo de direccionamiento simple pero necesario para el lenguaje de alto nivel. De manera que dispone de 32 registros de propósito general y 2 registros especiales para realizar las operaciones de multiplicación y división (Hi-Lo).

Existen tres tipos diferentes de formatos para las instrucciones en lenguaje máquina presentes en el procesador.

- Instrucciones Inmediatas o tipo I.
- Instrucciones de Salto o tipo J.
- Instrucciones de Registros o tipo R.

La mayoría de estas instrucciones son ejecutadas en registros. Las instrucciones de Registros tienen tres operandos: 2 fuentes y un destino. Las instrucciones inmediatas tienen una fuente y un destino, mientras que las instrucciones de salto tienen una instrucción relativa de 26 bit, que se usa para calcular el destino del salto.

Field	Description
opcode	6-bit primary operation code
rd	5-bit specifier for the destination register
rs	5-bit specifier for the source register
rt	5-bit specifier for the target (source/destination) register or used to specify functions within the primary opcode REGIMM
immediate	16-bit signed immediate used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacement
instr_index	26-bit index shifted left two bits to supply the low-order 28 bits of the jump target address
sa	5-bit shift amount
function	6-bit function field used to specify functions within the primary opcode SPECIAL

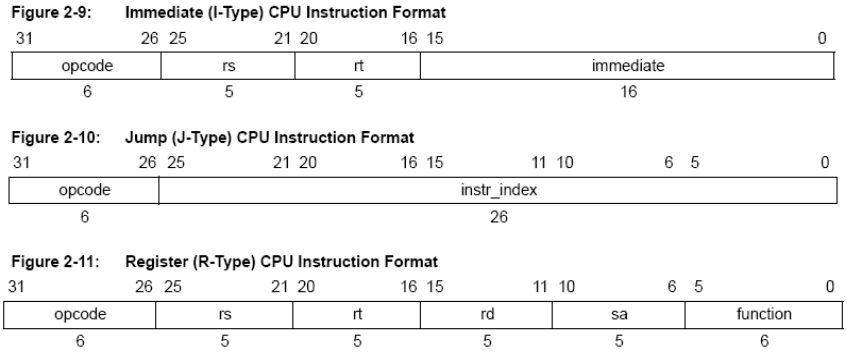


Figura 2.9: Formato de los 3 tipos de instrucciones.

A continuación mostramos una tabla resumen con las instrucciones que están implementadas en los núcleos de las siguientes familias de microcontroladores de 32 bits, PIC32MX3XX/4XX:

Instruction	Description	Function
ADD	Integer Add	Rd = Rs + Rt
ADDI	Integer Add Immediate	Rt = Rs + Immed
ADDIU	Unsigned Integer Add Immediate	Rt = Rs $+_U$ Immed
ADDIUPC	Unsigned Integer Add Immediate to PC (MIPS16e™ only)	Rt = PC $+_u$ Immed
ADDU	Unsigned Integer Add	Rd = Rs $+_U$ Rt
AND	Logical AND	Rd = Rs & Rt
ANDI	Logical AND Immediate	Rt = Rs & (0_{16} || Immed)
B	Unconditional Branch (Assembler idiom for: BEQ r0, r0, offset)	PC += (int)offset
BAL	Branch and Link (Assembler idiom for: BGEZAL r0, offset)	GPR[31> = PC + 8 PC += (int)offset
BEQ	Branch On Equal	if Rs == Rt PC += (int)offset

BEQL	Branch On Equal Likely	if Rs == Rt
		  PC += (int)offset
		else
		  Ignore Next Instruction

BGEZ	Branch on Greater Than or Equal To Zero	if !Rs[31> PC += (int)offset
BGEZAL	Branch on Greater Than or Equal To Zero And Link	GPR[31> = PC + 8 if !Rs[31> PC += (int)offset

BGEZALL	Branch on Greater Than or Equal To Zero And Link Likely	GPR[31> = PC + 8
		if !Rs[31>
		  PC += (int)offset
		else
		  Ignore Next Instruction

BGEZL	Branch on Greater Than or Equal To Zero Likely	if !Rs[31>
		  PC += (int)offset
		else
		  Ignore Next Instruction

BGTZ	Branch on Greater Than Zero	if !Rs[31> && Rs != 0 PC += (int)offset

BGTZL	Branch on Greater Than Zero Likely	if !Rs[31> && Rs != 0
		  PC += (int)offset
		else
		  Ignore Next Instruction

Instruction	Description	Function
BLEZ	Branch on Less Than or Equal to Zero	if Rs[31> || Rs == 0 PC += (int)offset

BLEZL	Branch on Less Than or Equal to Zero Likely	if Rs[31> || Rs == 0
		  PC += (int)offset
		else
		  Ignore Next Instruction

BLTZ	Branch on Less Than Zero	if Rs[31> PC += (int)offset
BLTZAL	Branch on Less Than Zero And Link	GPR[31> = PC + 8 if Rs[31> PC += (int)offset

BLTZALL	Branch on Less Than Zero And Link Likely	GPR[31> = PC + 8
		if Rs[31>
		  PC += (int)offset
		else
		  Ignore Next Instruction

BLTZL	Branch on Less Than Zero Likely	if Rs[31>
		  PC += (int)offset
		else
		  Ignore Next Instruction

BNE	Branch on Not Equal	if Rs != Rt PC += (int)offset

BNEL	Branch on Not Equal Likely	if Rs != Rt
		  PC += (int)offset
		else
		  Ignore Next Instruction

BREAK	Breakpoint	Break Exception
CLO	Count Leading Ones	Rd = NumLeadingOnes(Rs)
CLZ	Count Leading Zeroes	Rd = NumLeadingZeroes(Rs)
COP0	Coprocessor 0 Operation	See Software User's Manual
DERET	Return from Debug Exception	PC = DEPC Exit Debug Mode
DI	Atomically Disable Interrupts	Rt = Status; $Status_{IE}$ = 0
DIV	Divide	LO = (int)Rs / (int)Rt HI = (int)Rs % (int)Rt
DIVU	Unsigned Divide	LO = (uns)Rs / (uns)Rt HI = (uns)Rs % (uns)Rt
EHB	Execution Hazard Barrier	Stop instruction execution until execution hazards are cleared
EI	Atomically Enable Interrupts	Rt = Status; $Status_{IE}$ = 1

ERET	Return from Exception	if SR[2>
		  PC = ErrorEPC
		else
		  PC = EPC
		  SR[1> = 0
		SR[2> = 0
		LL = 0

EXT	Extract Bit Field	Rt = ExtractField(Rs, pos, size)

Instruction	Description	Function
INS	Insert Bit Field	Rt = InsertField(Rs, Rt, pos, size)
J	Unconditional Jump	PC = PC[31:28> || offset<<2
JAL	Jump and Link	GPR[31> = PC + 8 PC = PC[31:28> || offset<<2
JALR	Jump and Link Register	Rd = PC + 8 PC = Rs
JALR.HB	Jump and Link Register with Hazard Barrier	Like JALR, but also clears execution and instruction hazards
JALRC	Jump and Link Register Compact – do not execute instruction in jump delay slot (MIPS16e™ only)	Rd = PC + 2 PC = Rs
JR JR.HB	Jump Register Jump Register with Hazard Barrier	PC = Rs Like JR, but also clears execution and instruction hazards
JRC	Jump Register Compact – do not execute instruction in jump delay slot (MIPS16e only)	PC = Rs
LB	Load Byte	Rt = (byte)Mem[Rs+offset>
LBU	Unsigned Load Byte	Rt = (ubyte))Mem[Rs+offset>
LH	Load Halfword	Rt = (half)Mem[Rs+offset>
LHU	Unsigned Load Halfword	Rt = (uhalf)Mem[Rs+offset>
LL	Load Linked Word	Rt = Mem[Rs+offset> LL = 1 LLAdr = Rs + offset
LUI	Load Upper Immediate	Rt = immediate << 16
LW	Load Word	Rt = Mem[Rs+offset>
LWPC	Load Word, PC relative	Rt = Mem[PC+offset>
LWL	Load Word Left	See Architecture Reference Manual
LWR	Load Word Right	See Architecture Reference Manual
MADD	Multiply-Add	HI | LO += (int)Rs * (int)Rt
MADDU	Multiply-Add Unsigned	HI | LO += (uns)Rs * (uns)Rt
MFC0	Move From Coprocessor 0	Rt = CPR[0, Rd, sel>
MFHI	Move From HI	Rd = HI
MFLO	Move From LO	Rd = LO
MOVN	Move Conditional on Not Zero	if Rt ¼ 0 then Rd = Rs
MOVZ	Move Conditional on Zero	if Rt = 0 then Rd = Rs
MSUB	Multiply-Subtract	HI | LO -= (int)Rs * (int)Rt
MSUBU	Multiply-Subtract Unsigned	HI | LO -= (uns)Rs * (uns)Rt
MTC0	Move To Coprocessor 0	CPR[0, n, Sel> = Rt
MTHI	Move To HI	HI = Rs
MTLO	Move To LO	LO = Rs
MUL	Multiply with register write	HI | LO =Unpredictable Rd = ((int)Rs * (int)Rt)$_{31..0}$
MULT	Integer Multiply	HI | LO = (int)Rs * (int)Rd
MULTU	Unsigned Multiply	HI | LO = (uns)Rs * (uns)Rd

Instruction	Description	Function
NOP	No Operation (Assembler idiom for: SLL r0, r0, r0)	
NOR	Logical NOR	Rd = ~(Rs | Rt)
OR	Logical OR	Rd = Rs | Rt
ORI	Logical OR Immediate	Rt = Rs | Immed
RDHWR	Read Hardware Register	Allows unprivileged access to registers enabled by HWREna register
RDPGPR	Read GPR from Previous Shadow Set	Rt = SGPR[SRSCtl$_{PSS}$, Rd>
RESTORE	Restore registers and deallocate stack frame (MIPS16e™ only)	See Architecture Reference Manual
ROTR	Rotate Word Right	Rd = Rt$_{sa-1..0}$ || Rt$_{31..sa}$
ROTRV	Rotate Word Right Variable	Rd = Rt$_{Rs-1..0}$ || Rt$_{31..Rs}$
SAVE	Save registers and allocate stack frame (MIPS16e only)	See Architecture Reference Manual
SB	Store Byte	(byte)Mem[Rs+offset> = Rt
SC	Store Conditional Word	if LL = 1 mem[Rs+offset> = Rt Rt = LL
SDBBP	Software Debug Break Point	Trap to SW Debug Handler
SEB	Sign-Extend Byte	Rd = (byte)Rs
SEH	Sign-Extend Half	Rd = (half)Rs
SH	Store Half	(half)Mem[Rs+offset> - Rt
SLL	Shift Left Logical	Rd - Rt << sa
SLLV	Shift Left Logical Variable	Rd = Rt << Rs[4:0>

SLT	Set on Less Than	if (int)Rs < (int)Rt
		  Rd = 1
		else
		  Rd = 0

SLTI	Set on Less Than Immediate	if (int)Rs < (int)Immed
		  Rt = 1
		else
		  Rt = 0

SLTIU	Set on Less Than Immediate Unsigned	if (uns)Rs < (uns)Immed
		  Rt = 1
		else
		  Rt = 0

SLTU	Set on Less Than Unsigned	if (uns)Rs < (uns)Immed
		  Rd = 1
		else
		  Rd = 0

SRA	Shift Right Arithmetic	Rd = (int)Rt >> sa
SRAV	Shift Right Arithmetic Variable	Rd = (int)Rt >> Rs[4:0>
SRL	Shift Right Logical	Rd - (uns)Rt >> sa
SRLV	Shift Right Logical Variable	Rd - (uns)Rt >> Rs[4:0>
SSNOP	Superscalar Inhibit No Operation	NOP
SUB	Integer Subtract	Rt = (int)Rs - (int)Rd
SUBU	Unsigned Subtract	Rt = (uns)Rs - (uns)Rd
SW	Store Word	Mem[Rs+offset> = Rt
SWL	Store Word Left	See Architecture Reference Manual

Instruction	Description	Function
SWR	Store Word Right	See Architecture Reference Manual
SYNC	Synchronize	See Software User's Manual
SYSCALL	System Call	SystemCallException
TEQ	Trap if Equal	if Rs == Rt TrapException
TEQI	Trap if Equal Immediate	if Rs == (int)Immed TrapException
TGE	Trap if Greater Than or Equal	if (int)Rs >= (int)Rt TrapException
TGEI	Trap if Greater Than or Equal Immediate	if (int)Rs >= (int)Immed TrapException
TGEIU	Trap if Greater Than or Equal Immediate Unsigned	if (uns)Rs >= (uns)Immed TrapException
TGEU	Trap if Greater Than or Equal Unsigned	if (uns)Rs >= (uns)Rt TrapException
TLT	Trap if Less Than	if (int)Rs < (int)Rt TrapException
TLTI	Trap if Less Than Immediate	if (int)Rs < (int)Immed TrapException
TLTIU	Trap if Less Than Immediate Unsigned	if (uns)Rs < (uns)Immed TrapException
TLTU	Trap if Less Than Unsigned	if (uns)Rs < (uns)Rt TrapException
TNE	Trap if Not Equal	if Rs != Rt TrapException
TNEI	Trap if Not Equal Immediate	if Rs != (int)Immed TrapException
WAIT	Wait for Interrupts	Stall until interrupt occurs
WRPGPR	Write to GPR in Previous Shadow Set	SGPR[SRSCtl$_{PSS}$, Rd> = Rt
WSBH	Word Swap Bytes Within Halfwords	Rd = Rt$_{23..16}$ || Rt$_{31..24}$ || Rt$_{7..0}$ || Rt$_{15..8}$
XOR	Exclusive OR	Rd = Rs ^ Rt
XORI	Exclusive OR Immediate	Rt = Rs ^ (uns)Immed
ZEB	Zero-extend byte (MIPS16e™ only)	Rt = (ubyte) Rs
ZEH	Zero-extend half (MIPS16e only)	Rt = (uhalf) Rs

Tabla 2.2: Juego de instrucciones completo presente en la familia PIC32MX3XX/4XX.

Como podemos observar existen 124 instrucciones diferentes, por lo que dada su extensión y su dificultad, la familia de los microprocesadores PIC32 suelen programarse en lenguaje de programación de alto nivel, accediendo a los registros específicos usando las funciones proporcionadas por el compilador o asignado directamente los unos y los ceros en las posiciones deseadas.

2.2.4.1. Registros de la CPU

Tal y como hemos comentando antes los registros de la CPU son los siguientes:

- 32 registros de propósito general de 32 bits.
- 2 registros de propósito general para almacenar el resultado de las multiplicaciones, divisiones y operaciones de multiplicación acumulativa (HI y LO).
- Y un registro de propósito especial, contador de programa (PC), al cual solo le afectan indirectamente ciertas instrucciones, tal y como hemos visto anteriormente, las interrupciones y excepciones. Este registro no es un registro visible en la arquitectura.

En la siguiente figura se muestra la distribución de los registros de la CPU:

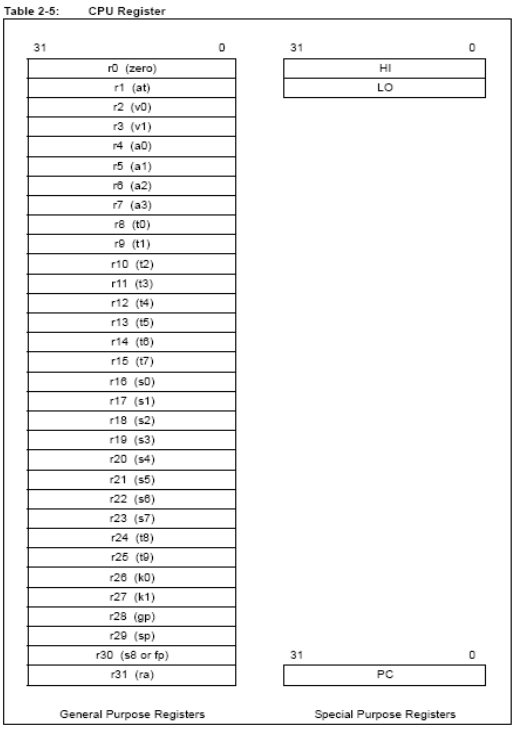


Figura 2.10: Registros de la CPU.

2.2.4.2. Modos del procesador:

Hay tres maneras distintas de ejecución de la CPU del PIC32: dos modos de operación y un modo especial: modo User, modo Kernel y modo Debug. El procesador comienza la ejecución en el modo Kernel y si se quiere se puede permanecer en este modo durante la operación normal. El modo usuario es un modo opcional que permite

al diseñador partir el código entre software privilegiado y no privilegiado. Por otra parte, el modo DEBUG solo se usa solo mediante un depurador.

Una de las principales diferencias entre los distintos modos de operación es el direccionamiento de la memora, de tal forma que el software solo permite el acceso a determinadas regiones de esta. Por ejemplo los periféricos no son accesibles en el modo usuario. La siguiente figura muestra la organización de la memoria en cada modo:

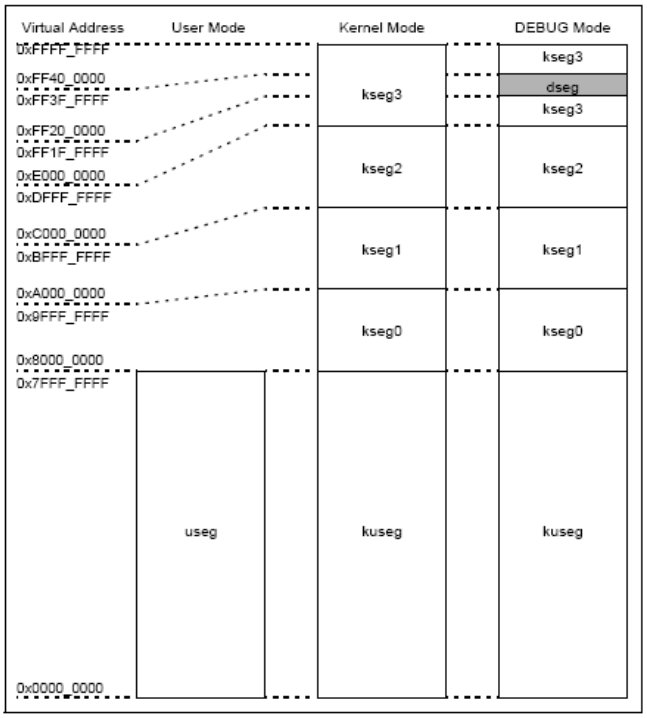


Figura 2.11: Modos de operación de la CPU.

Modo Kernel:

Para poder acceder a todos los recursos hardware, el procesador debe estar en este estado. Tal y como podemos ver en la figura anterior, este modo permite el acceso software a todo el espacio de direcciones del procesador así como también a las instrucciones privilegiadas.

De tal forma que el procesador operará en este modo cuando el bit DM del registro DEBUG sea “0” y el registro STATUS contenga alguno de los siguientes valores: UM=0; ERL=1; EXL=1.

Modo User:

Para poder operar en este modo, el registro STATUS debe contener los siguientes valores: UM=1; EXL=0; ERL=0.

Mientras se está ejecutando el procesador en este modo, el software se restringe a una serie de recursos del procesador. De tal forma, que en este modo se tiene únicamente acceso al área de memoria USEG.

Modo Debug:

Este modo especial del procesador, requiere que para su funcionamiento se ejecute una excepción debug. De tal forma que se accederá a todos los recursos como en el modo kernel así como a los recursos hardwares especiales usados en las aplicaciones de debug. Cuando se entra en este modo el bit DM del registro DEBUG es 1. Para salir de este modo basta con ejecutar la instrucción DERET.

2.2.4.3. Registros CP0:

El PIC32 usa un registro especial como interface para comunicar el estatus y la información de control entre el software del sistema y la CPU. Esta interfaz se llama coprocesador 0. El software del sistema accede a los registros del CP0 usando las instrucciones del coprocesador como MFC0 y MTC0. Los registros CP0 en el MCU del PIC32 son los siguientes:

Table 2-8: CP0 Registers

Register Number	Register Name	Function
0-6	Reserved	Reserved in the PIC32MX core
7	HWREna	Enables access via the RDHWR instruction to selected hardware registers in Non-privileged mode
8	BadVAddr	Reports the address for the most recent address-related exception
9	Count	Processor cycle count
10	Reserved	Reserved in the PIC32MX core
11	Compare	Timer interrupt control
12	Status/ IntCtl/ SRSCtl/ SRSMap	Processor status and control; interrupt control; and shadow set control
13	Cause	Cause of last exception
14	EPC	Program counter at last exception
15	PRId/ EBASE/	Processor identification and revision; exception base address
16	Config/ Config1/ Config2/ Config3	Configuration registers
17-22	Reserved	Reserved in the PIC32MX core
23	Debug/ Debug2/	Debug control/exception status and EJTAG trace control
24	DEPC	Program counter at last debug exception
25-29	Reserved	Reserved in the PIC32MX core
30	ErrorEPC	Program counter at last error
31	DeSAVE	Debug handler scratchpad register

Figura 2.12: Registros CP0.

Entre los registros más importantes del coprocesador destaca el registro STATUS (registro 12 del CP0, selección 0). Este registro STATUS es de lectura y escritura y contiene el modo de operaciones, la habilitación de interrupciones y el estado del diagnostico del procesador.

- Interrupciones habilitadas cuando:
  - IE=1 EXL=0 ERL=0 DM=0
- Modos de operación:
  - DEBUG cuando el bit DM es "1" en cualquier otro caso Kernel o user
  - Modo user cuando UM=1 EXL=0 ERL=0
  - Modo kernel cuando UM=0 EXL=1 ERL=1

La posición que ocupan estos bits son:

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	—	—	UM	—	ERL	EXL	IE
bit 7							bit 0

Figura 2.13: Primeros 8 bits del registro STATUS.

2.2.5. MEMORIA DEL SISTEMA

El PIC32 proporciona 4GB (2^{32}) de espacio de direcciones de memoria virtual unificada. Todas las regiones de memoria, incluidas la memoria de programa, memoria de datos, SFRs y registros de configuración residen en este espacio de direcciones con sus respectivas direcciones únicas. Opcionalmente, la memoria de datos y de programa se pueden partir en memoria de usuario y kernel. Además, la memoria de datos puede ponerse como ejecutable, permitiendo al PIC32 ejecutarse desde esta.

Características principales de la organización de la memoria:

- Ancho de datos de 32 bits.
- Separación del espacio de direcciones del modo User y kernel
- Flexibilidad para poder realizar una partición de la memoria flash de programa.
- Flexibilidad para poder realizar una partición de la memoria RAM para datos y otro espacio para programa.
- Separación de la memoria Flash boot para proteger el código.
- Manejo robusto de las excepciones del bus para interceptar el código fuera de control.

- Mapeo de la memoria mediante la unidad FMT(Fixed Mapping Translation)
- Regiones de direcciones cacheables y no cacheables.

En el Anexo A del presente proyecto, se detalla con más precisión la forma en la que se separan los espacios de memoria en modo user y kernel, los registros usados para configurar la memoria del microcontrolador, así como la manera de realizar una partición tanto de la memoria flash de programa como de la memoria RAM y las consideraciones que hay que tener en cuenta a la hora de llevarlo a cabo.

2.2.6. RECURSOS ESPECIALES

En este apartado se detallan los recursos especiales más comunes que pueden poseer los microcontroladores:

2.2.6.1. Perro guardián o Watchdog:

El perro guardián está diseñado para inicializar automáticamente el microcontrolador en el caso de que exista un mal funcionamiento del sistema.

Consiste en un temporizador cuyo objetivo es generar automáticamente un reset cuando se desborda y pasa por 0, a menos que se inhabilite en la palabra de configuración. Cuando el perro guardián está habilitado, se ha de diseñar el programa de manera que refresque o inicialice al perro guardián antes de que se provoque dicho reset. Si el programa falla o se bloquea, no se refrescará al perro guardián y cuando complete su temporización provocará un reset al sistema.

2.2.6.2. Tecnología de ahorro energético:

Todos los dispositivos de la familia del PIC32 incorporan un rango de características que pueden significativamente reducir el consumo eléctrico durante su funcionamiento. Se debe básicamente a dos características:

- Cambio de reloj sobre la marcha: El reloj del dispositivo puede ser cambiado bajo un software que controle alguno de los 4 relojes durante la operación.
- Instrucciones basadas en modos de ahorro de energía. El microcontrolador puede suspender todas las operaciones, o selectivamente apagar el núcleo mientras deja activo los periféricos, utilizando una instrucción del software.

2.2.6.3. Osciladores:

Toda la familia del PIC32 nos ofrece 4 osciladores diferentes con sus correspondientes opciones, lo que le permite al usuario un gran rango de posibles elecciones en el desarrollo de las aplicaciones hardware. Estas incluyen:

- **FRC:** Internal oscillator, para velocidades altas de operación con un consumo bajo, salida nominal de 8MHz.

- **LPRC:** Internal low-frequency and low-power oscillator. Designado para velocidades de operación pequeñas con bajo consumo, baja precisión, 32KHZ.

- **POSC:** External primary oscillator. Hasta 20MHZ (XT-hasta 10MHz; HS para más de 10MHz), dependiendo si usamos cristal de cuarzo o un resonador cerámico.

- **SOSC:** External low-frequency and low-power, designado para bajas velocidades con un cristal externo de 32,768Khz. Utilizado para temporizar tiempos exactos. Además es necesario si se quiere usar el módulo RTCC (Real-Time Clock and Calendar).

- **EC:** External clock, permite conectar un circuito externo y reemplazarlo por el oscilador, proporcionando al microcontrolador una onda de la frecuencia que queramos.

La señal producida por cada reloj puede ser multiplicada o dividida para ofrecer un ancho rango de frecuencias a través del circuito PLL. Este circuito proporciona un rango de frecuencias desde 32KHZ hasta la frecuencia máxima especificada por el PIC32 80MHZ, comentado en mayor profundidad en el Anexo B del presente proyecto.

2.2.6.4. Puertos de comunicación:

Los puertos de comunicación son herramientas que dotan al microcontrolador de la posibilidad de comunicarse con otros dispositivos externos, otros buses de microprocesadores, buses de sistemas, buses de redes y poder adaptarlos con otros elementos bajo otras normas y protocolos. En las aplicaciones de los PIC, estas se reducen a entender los protocolos. En general en el PIC32 existen disponibles dos tipos, la interfaz de comunicación serie asíncrona (UART) y la síncrona (SPI y I2C).

El PIC32 proporciona 7 formas de comunicación periférica de las cuales 6 son en serie. Estas últimas son:

- 2 Universal Asynchronous Receiver and Transmitters (UARTs).
- 2 SPI y 2 I2C (síncrono).

La diferencia principal entre una comunicación síncrona y otra asíncrona es la necesidad de pasar información temporal desde el que transmite al receptor. Es decir, la comunicación síncrona necesita de una línea dedicada para una señal de reloj que proporcione una sincronización entre ambos dispositivos. Así, el que origina la señal de reloj se le denomina maestro y al otro esclavo.

Interfaz I2C:

Utiliza dos cables, dos pines del microcontrolador: 1 para el reloj (SCL) y otro para trasmitir los datos de forma bidireccional (SDA).

Requiere 10-bits para configurar el dispositivo antes de que se envíe cualquier dato. Esto permite que con el mismo cable se pueda usar hasta 1000 dispositivos.

Interfaz SPI:

Esta interfaz separa la transmisión de datos en dos líneas: una para los datos de entrada (SDI) y otra para los de salida (SDO), por tanto requiere otro cable pero permite transferir datos simultáneamente en ambas direcciones.

Sin embargo, esta interfaz requiere una línea física adicional (selección del esclavo, SS) para conectar cada dispositivo. La principal ventaja de la interfaz SPI es que es muy simple y la velocidad puede ser mucho más alta que con la mayor velocidad del bus I2C.

Interfaz UART:

No requiere una línea de reloj. Existen 2 líneas de datos, TX y RX, las cuales se usan para la entrada y salida de datos, y se pueden colocar adicionalmente dos líneas que pueden ser usadas para proporcionar un hardware handshake (líneas CTS y RTS). La sincronización se obtiene añadiendo un bit de start y otro de stop a los datos.

Generalmente se usa esta interfaz cuando la distancia física es grande.

Interfaz pararela:

El Parallel Master Port (PMP) se ha añadido a la arquitectura del PIC32 para dotar de un bus de E/S flexible para realizar tareas cotidianas del control de periféricos externos. El PMP proporciona la habilidad de transferir datos de 8 o 16 bits de una manera bidireccional y hasta 64K de espacio de direcciones. Además se puede ajustar el tiempo para adecuar el PMP a la velocidad de los periféricos con los que queremos interactuar.

2.2.6.5. Conversor A/D:

Como es muy frecuente el trabajo con señales analógicas, éstas deben ser convertidas a digital y por ello muchos microcontroladores incorporan un conversor A/D, el cual se utiliza para tomar datos de varias entradas diferentes que se seleccionan mediante un multiplexor.

El PIC32 ofrece la posibilidad de convertir la información analógica a digital concretamente a través de un modulo ADC de 10 bits.

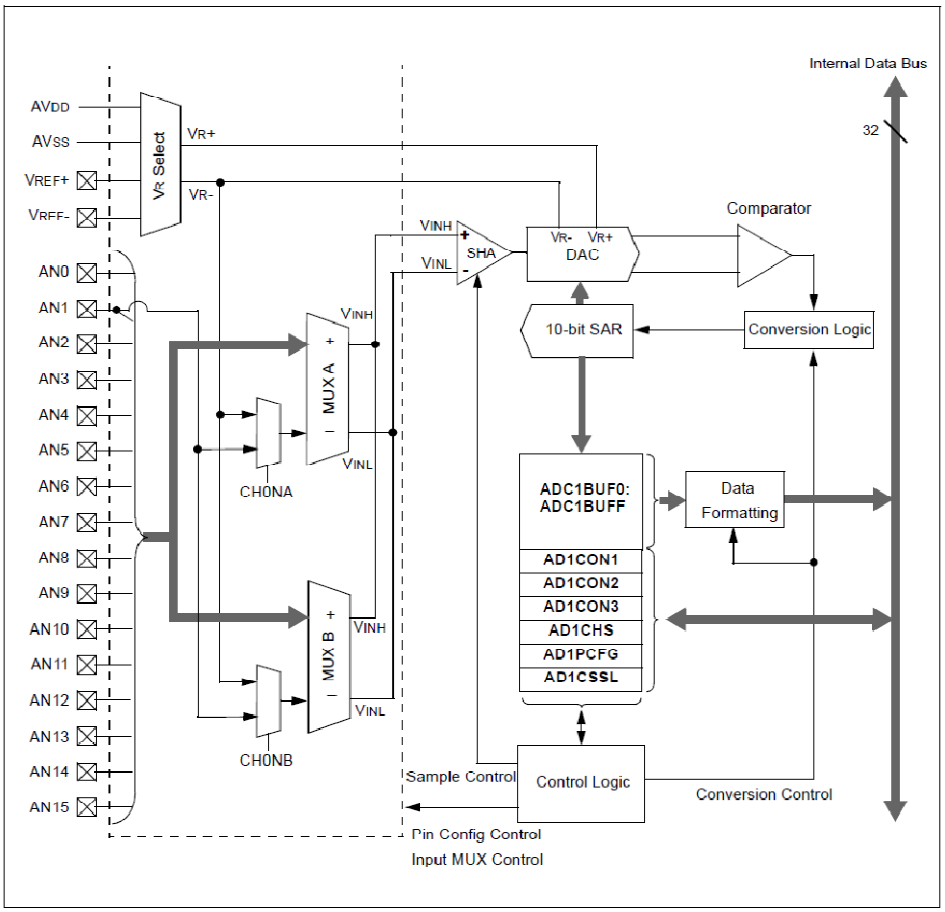


Figura 2.14: Diagrama de bloques del módulo ADC de 10 bits.

Como se puede ver en el esquema de bloques anterior, el módulo A/D pose hasta 16 pines de entradas que se pueden usar para recibir entradas analógicas. Estos pines están conectados a 2 multiplexores para seleccionar los distintos canales y las diferentes referencias para cada uno. La salida del convertidor de 10bits se puede pasar a enteros con signo o sin signo de 16 o 32 bits.

Para más información sobre el microcontrolador se recomienda ver la hoja de características del componente disponible en el CD-ROM adjunto a la memoria del presente Proyecto Fin de Carrera [3], el documento "*PIC32MX Family Reference Manual*" [4] o bien la dirección web del fabricante Microchip [5].

3. HERRAMIENTAS SOFTWARE DE DESARROLLO

CAPÍTULO 3. HERRAMIENTAS SOFTWARE DE DESARROLLO

3.1. MPLAB IDE

3.1.1. INTRODUCCIÓN

El proceso de escritura de una aplicación se describe a menudo como un ciclo de desarrollo, ya que es muy difícil que todos los pasos efectuados desde el diseño hasta la implementación se realicen correctamente a la primera. La mayoría de las veces se escribe el código, se testea y luego se modifica para crear una aplicación que funcione correctamente (ver figura 3.1).

MPLAB IDE integra todas estas funciones con el fin de no tener que utilizar distintas herramientas y diferentes modos de operación.

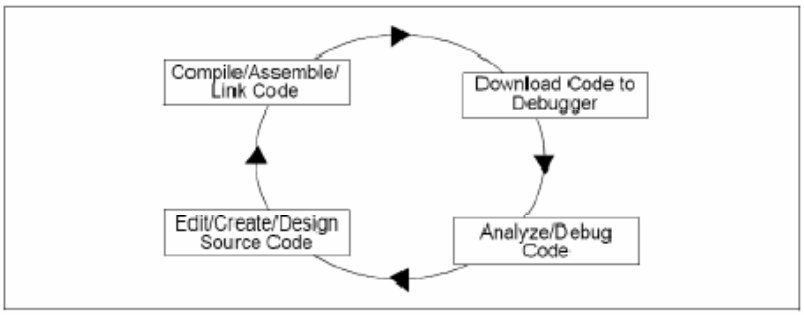


Figura 3.1: Proceso de escritura de una aplicación.

El software MPLAB IDE de Microchip [6], es un entorno de desarrollo integrado bajo Windows, que permite editar, ensamblar, linkar, depurar y simular proyectos para los distintos dispositivos PIC de Microchip.

Dicho entorno incorpora todas las herramientas necesarias para la realización de cualquier proyecto:

- Editor de texto.
- Ensamblador.
- Linkador.

- Simulador.
- Menús de ayuda.

Además de las herramientas incorporadas, se pueden añadir otras como por ejemplo:

- Compiladores de C.
- Emuladores.
- Programadores.
- Depuradores.

Por tanto, el entorno MPLAB es un software que junto con un emulador y un programador, forman un conjunto de herramientas de desarrollo.

3.1.2. CARACTERÍSTICAS GENERALES

Este software esta en inglés y está diseñado para trabajar bajo el sistema operativo de Windows en un entorno gráfico con ventanas y botones. A lo largo del presente trabajo vamos a trabajar con la versión del MPLAB IDE v8.33.

Si observamos la pantalla inicial del programa podemos observar tres áreas principales, la ventana de edición MPLAB IDE (color azul), ventana de salida (color naranja) y el panel de navegación a la izquierda de la misma (color verde). La ventana de edición es el editor de código, donde vamos a escribir nuestro programa, mientras que en la ventana de salida se mostrará la información y el estado de la última acción realizada.

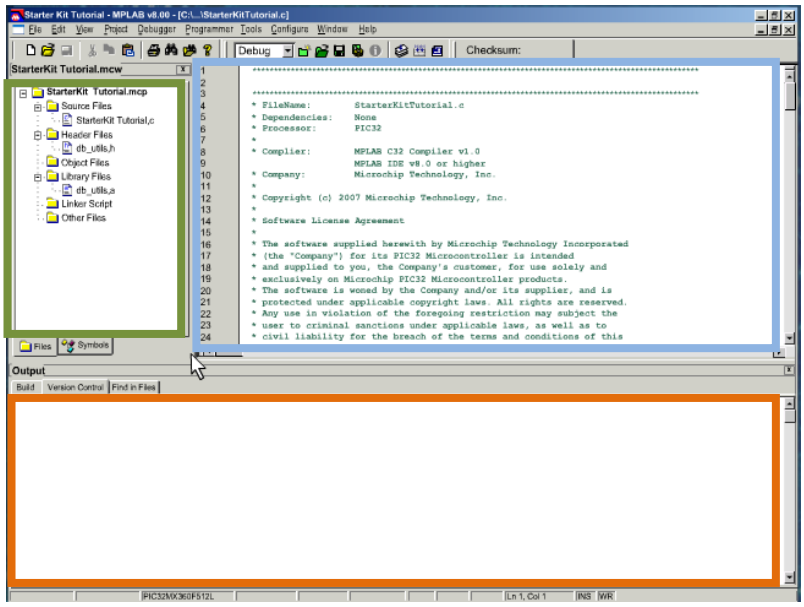


Figura 3.2: Pantalla inicial del MPLAB IDE.

Por otra parte, en el panel de navegación se pueden ver dos tabulaciones etiquetadas como Archivos y Símbolos. La etiqueta de archivos nos da acceso al organizador "Project manager", el cual nos permite organizar nuestros archivos para un proyecto. Es de destacar que no es necesario incluir el archivo linker en la ventana de nuestro proyecto. Con el PIC32, se usará automáticamente para el enlazado un linker por defecto, sin embargo, en el caso de que se considere oportuno se podrá incluir uno en concreto.

Además, una de las características del depurador para el PIC32 que nos ofrece el MPLAB IDE, es que posee un puerto de datos para depurar. Esto proporciona al depurador la capacidad de enviar y recibir mensajes a través de una consola, la cual la vamos a utilizar en algunas de las aplicaciones desarrolladas (se muestra un ejemplo de esta consola en el apartado 2 del capítulo 4), mediante el uso de la librería "*db_utils.h*".

3.1.3. ESTRUCTURA DE UN PROYECTO

Un proyecto es un conjunto de programas que se integran en un único módulo con el fin de generar un código ejecutable para un determinado microcontrolador. Dichos programas pueden estar escritos en diferentes lenguajes de programación, como por ejemplo ensamblador, C, Basic, Pascal.

A continuación se muestra la estructura de un proyecto creado con MPLAB, así como sus ficheros de entrada y salida:

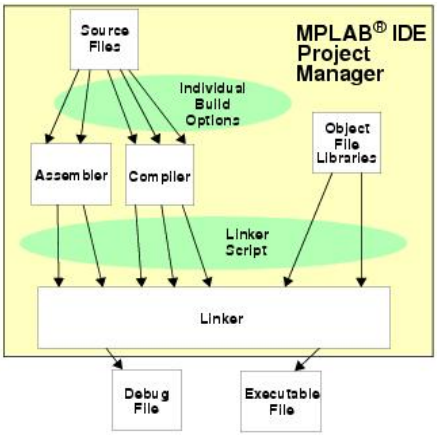


Figura 3.3: Estructura de un proyecto con el MPLAB IDE.

Los ficheros fuente (pueden ser varios), pueden estar escritos en ensamblador o en C, con extensiones .asm y .c respectivamente, y mediante los programas ensamblador y compilador, se obtienen los ficheros objeto con extensión .o.

Todos los ficheros objeto, junto a otros ficheros procedentes de librerías, son linkados, generando una serie de ficheros de salida de los cuales el más importante es el ejecutable .HEX que será el que se grabará en el dispositivo.

3.1.4. CREACIÓN DE UN PROYECTO

Un proyecto contiene todos los archivos necesarios para construir una aplicación (código fuente, ficheros de cabecera, librerías, etc), así como sus correspondientes opciones de construcción. Generalmente habrá un solo proyecto por workspace.

Por otra parte, un worskspace contiene: uno o más proyectos, información del dispositivo seleccionado, la herramienta de programación y depuración y las opciones de configuración del MPLAB IDE.

Además, el MPLAB IDE contiene un asistente para ayudar a crear nuevos proyectos. De tal forma que el proceso de creación de un proyecto se puede dividir en 8 tareas las cuales vamos a ir describiendo paso a paso con tal de crear un proyecto para ejecutarlo en el Starter Kit (Capitulo 4), primera tarjeta evaluada:

1.- Selección del dispositivo:

Abrimos el MPLAB IDE y lo primero que tenemos que realizar es cerrar cualquier worksapce que estuviera abierto, para ello realizamos clic en file>Close. Posteriormente hacemos clic en Project>Project Wizard, para abrir el asistente de creación de un proyecto nuevo, de tal forma que se nos abrirá una ventana como la que se muestra a continuación:

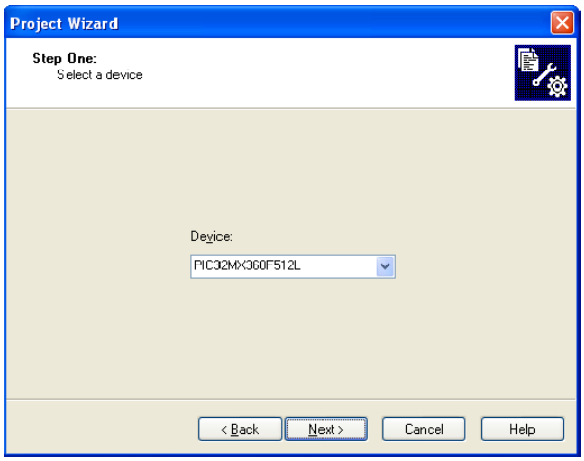


Figura 3.4: Creación de un proyecto, selección del dispositivo.

En el menú desplegable seleccionamos el dispositivo, en nuestro caso "PIC32MX360F512L" y le damos a NEXT. Entonces se nos abrirá otra ventana, como la que mostramos en el siguiente punto.

2.- Selección del Lenguaje de trabajo:

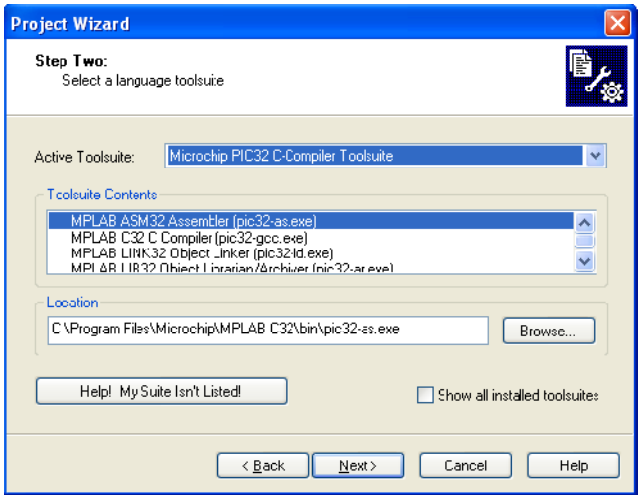


Figura 3.5: Creación de un proyecto, selección del compilador y el linkado.

En esta ventana tenemos que seleccionar el compilador que vamos a usar en el programa así como el linker, ambos se han descargado de la página de microchip [7].

En "Active Toolsuite" seleccionamos de la lista desplegable "Microchip PIC32 C Compiler ToolsSuite". La herramienta de trabajo incluye el ensamblador y el linkado que usaremos. Si la opción del compilador no está disponible, hay que activar la casilla "show all installed toolsuites".

En el desplegable de contenidos de las herramientas, seleccionamos "MPLAB C32 C Compiler (pic32-gcc.exe)" y abajo, en "Location", hacemos clic en "Browse" y seleccionamos la siguiente ruta que contiene el archivo seleccionado antes: "C:\Program Files\Microchip\MPLAB C32\bin\pic32-gcc.exe".

Por último seleccionamos en el desplegable anterior "MPLAB 32 LINK Object linker (pic20-ld.exe)" y comprobamos que la ruta es la siguiente"c:\Program Files\Microchip\MPLAB C32\bin\pic32-ld.exe". Hacemos clic en NEXT para continuar.

3.- Nombrar el proyecto

Se nos abrirá una pantalla como la que se muestra a continuación:

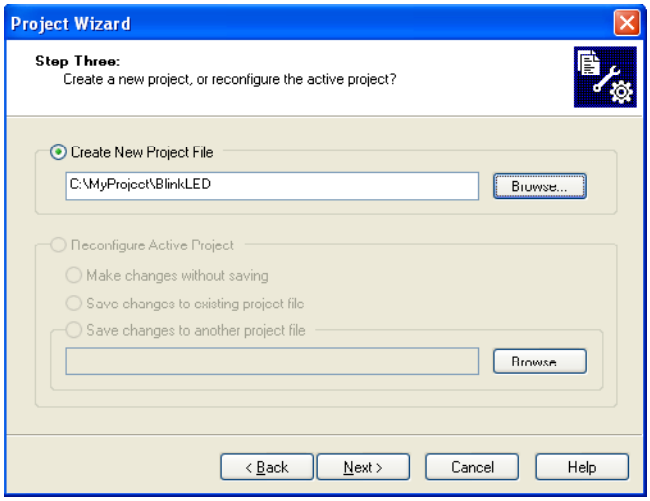


Figura 3.6: Creación de un proyecto, nombrar el proyecto.

En esta una nueva ventana seleccionamos la carpeta en la que queremos guardar nuestro proyecto y le damos un nombre al mismo, por ejemplo "C:\MiProyecto\Luces". Hacemos clic en NEXT para continuar.

4.-Añadir los archivos a tu proyecto:

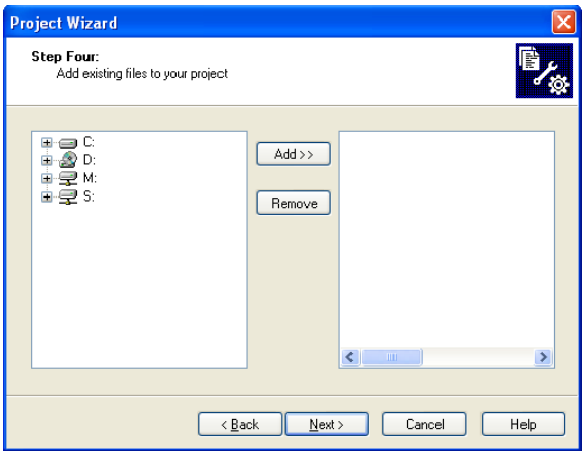


Figura 3.7: Creación de un proyecto, añadir archivos al proyecto.

Si los archivos en ".c" aun no los hemos creado podremos saltar esta pantalla y ya los añadiremos posteriormente. Hacemos clic en Finish para terminar y cerramos la ventana. Al cerrar la ventana, se ha creado en el MPLAB IDE el proyecto y el workspace, con los siguientes nombres: el workspace Luces.mcw y el archivo del proyecto Luces.mcp.

Hacemos clic en Files>New en la barra del menú para crear un archivo nuevo en el que comenzar a escribir nuestro programa. Antes que nada, lo guardamos en File>Save As como "*Luces.c*" en la misma carpeta en la que hemos creado el proyecto. Es importante escribir la extensión ".c" para que el MPLAB lo reconozca como un fichero fuente. Ahora escribimos el código en c para ejecutar el ejemplo. Una vez escrito el código añadimos el archivo "*Luces.c*" al directorio de fuentes tal y como se puede apreciar en la siguiente figura:

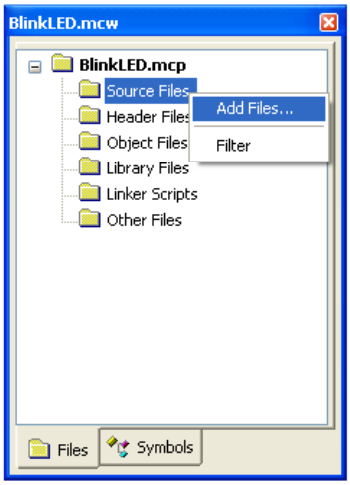


Figura 3.8: Añadir archivos al proyecto desde el menú principal.

Posteriormente, seleccionamos nuestro depurador, clic en Debugger>Select Tool>PIC32MX Starter Kit de la barra de menú. Antes de seleccionarlo, hay que asegurarse que nuestro Starter Kit esté conectado a nuestro PC a través del cable USB. Una vez lo hayamos conectado, en la ventana de salida del MPLAB, se mostrará un mensaje en el que nos indicará "Starter kit Found", a la vez que el LED naranja de depuración de la tarjeta se encenderá.

5.- Confirmar las opciones de configuración.

Hacemos clic en Configure>Configuration Bits para confirmar que las opciones de configuración de nuestro microcontrolador son las que queramos. A continuación se muestra una configuración típica para el Starter Kit:

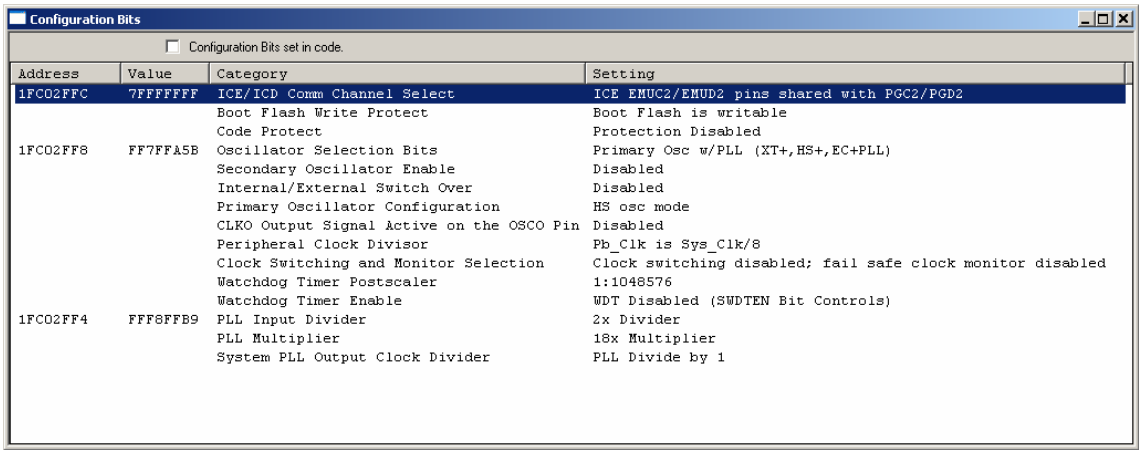


Figura 3.9: Resumen de las opciones de configuración, configuration bits.

6.- Construir el proyecto:

Hacemos clic en Project>Build all en la barra del menú y observamos (en la ventana de salida) que el proceso transcurra sin errores hasta que aparezca el mensaje "BUILD SUCCEEDED", momento en el que el programa estará preparado para que lo probemos en el Starter Kit.

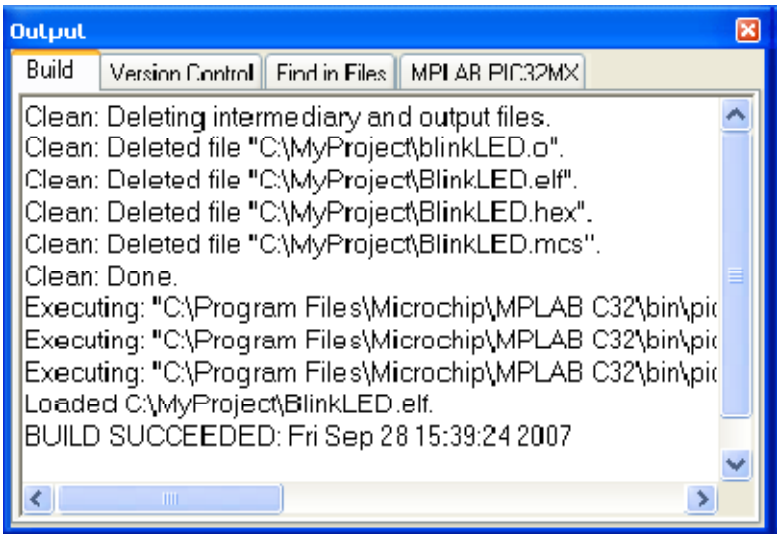


Figura 3.10: Construcción del programa, build.

7.- Programar el dispositivo

Hacemos clic en el icono de Program All Memories en la barra Herramientas del dispositivo, tal y como muestra la figura, con tal de programar el mismo:

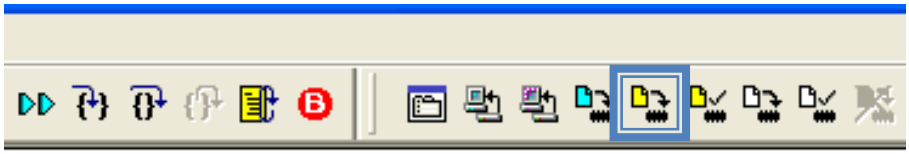


Figura 3.11: Programar el microcontrolador.

Una vez pulsado, nos aparecerá un mensaje advirtiéndonos acerca de que vamos a sobrescribir la memoria, le damos a Yes para continuar. En la ventana de salida nos fijamos en el progreso y cuando aparezca “DONE” significará que la programación del dispositivo ha sido completada. Esto verifica que la imagen ha sido cargada en la memoria Flash y está preparada para su ejecución.

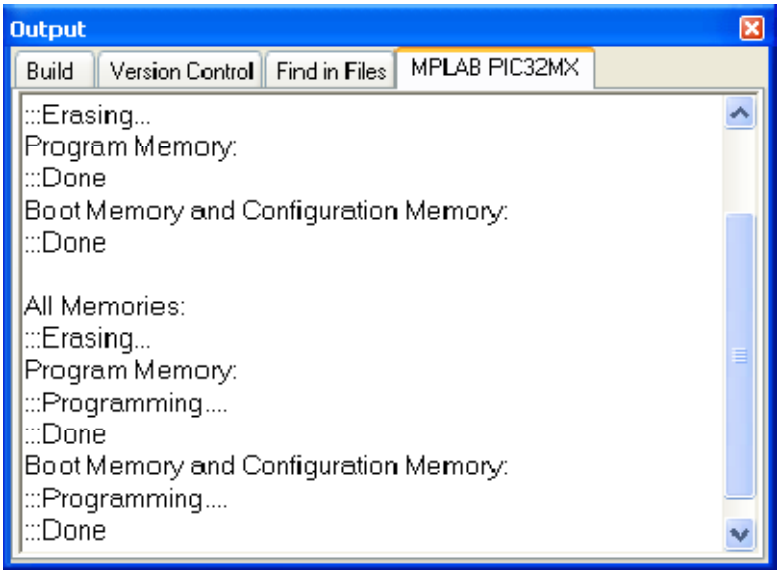


Figura 3.12: Verificación de la programación en el dispositivo.

8.- Probar el programa:

Una vez programada la memoria del dispositivo hacemos clic en Debugger>Run en la barra del menú o en el icono de Run (color azul) en el menú del depurador tal y como indica la siguiente figura, comenzando el funcionamiento del programa en nuestro Starter Kit.

Figura 3.13: Ejecutar el programa en modo debugger.

Para parar el programa primero tenemos que hacer clic en “Halt” (pausa) y posteriormente en “Reset” (color purpura).

3.2. SOFTWARE DE GRABACIÓN

3.2.1. INTRODUCCIÓN

Los microcontroladores en general, y los de la empresa Microchip en particular, necesitan de un circuito electrónico que permita transferirles el programa realizado desde el ordenador. Existen muchas maneras de afrontar este problema, y en general se suele utilizar alguno de los puertos disponibles en cualquier ordenador para este fin. Por ello, en el mercado es posible conseguir programadores de PICs con conexión para puerto USB, paralelo o serie (RS-232).

3.2.2. SOFTWARE DE GRABACIÓN

Para grabar los programas en el PIC32, existen múltiples opciones así como distintos software. En el presente trabajo se ha utilizado el MPLAB ICD3 In-Circuit Debugger para la grabación y depuración del código, el cual se conecta directamente a la Explorer16 (Capítulo 5) a través del conector reservado para ello. De tal forma que este grabador se conecta a la Explorer16 mediante un cable modular y al ordenador mediante un cable USB. Esta versión del programador ICD3 es hasta 15 veces más rápida que su versión anterior [8]. Además, no necesita de una fuente de alimentación externa, ya que esta la recibe desde el ordenador a través del cable USB.

Por otra parte, para que el ordenador pueda realizar las diferentes opciones de grabación, se necesita tener instalado el programa MPLAB IDE en su versión 8.15a o superior. En nuestro caso vamos a trabajar con la versión MPLAB IDE v8.33, tal y como ya hemos comentado anteriormente.

Figura 3.14: MPLAB ICD3 In-Circuit Debugger.

La pastilla de grabación dispone también de 3 luces indicadoras las cuales poseen los siguientes significados. El Led de “Power” se pondrá de color verde cuando se conecte al ordenador, indicando que ya está conectado a corriente. El LED “active” se pondrá de color azul cuando se le suministre energía o cuando se conecte a la tarjeta. Por otra parte el LED “Status” puede tener 3 configuraciones distintas: si esta de color verde, significa que el depurador está operando correctamente o bien que este se encuentra en standby; si es de color rojo es que alguna operación ha fallado y por último si es de color naranja significa que el depurador está ocupado.

Además dispone de una tarjeta de Test, la cual se usa para verificar que el MPLAB ICD3 funciona correctamente. Esta tarjeta la podemos ver en la figura 3.13, en el extremo al que se encuentra conectado el cable modular.

3.2.3. GRABAR EN LA EXPLORER16 MEDIANTE ICD3

En este apartado vamos a describir los pasos que hay que realizar a la hora de depurar o grabar un programa mediante el uso del MPLAB ICD3 bajo el entorno MPLAB IDE v8.33, además se indican todos los aspectos que hay que tener en cuenta para llevarlo a cabo.

Los pasos necesarios para grabar un programa y verificar su funcionamiento mediante el uso de MPLAB ICD 3 In-circuit Debugger son los siguientes:

- Primero tenemos que asegurarnos que al abrir el programa tengamos seleccionado nuestro dispositivo en Configure->Select Device.
- Nos aseguramos que los bits de configuración son los correctos, Configure->Configuration Bits y posteriormente seleccionamos Debugger->Select Tool->ICD 3, MPLAB ICD3 in-circuit debugger, para depurar el código.
- Construimos el programa para crear el archivo de grabación con la extensión .HEX, mediante la opción Build All. Nos tenemos que asegurar que el interruptor S2 está en PIM, para que funcione correctamente nuestro microcontrolador, en caso contrario dará un error.
- Cargamos el programa en la Explorer16 para su depuración. Para ello debugger->Program. Una vez realizado ya podemos comprobar el funcionamiento del programa dándole a run. Hasta este punto es muy similar respecto a la forma de depurar en el Starter Kit.

- También podemos usar los breakpoints para realizar paradas del programa en una línea específica de código. Estos breakpoints pueden ser de dos tipos, hardware and software.

- Así mismo, podemos usar la herramienta stopwatch (se ejecuta desde la ventana de dialogo de los breakpoints) para determinar el tiempo entre distintos breakpoints, el tiempo que se tarda en ejecutar determinadas instrucciones. El stopwatch nos devuelve en la ventana de dialogo de salida del ICD3 el número de ciclos entre las dos instrucciones seleccionadas. Por lo tanto, para saber el tiempo que tardan en ejecutarse las instrucciones seleccionadas entre los 2 breakpoints tendremos que saber a qué frecuencia está trabajando nuestro microcontrolador.

- Una vez depurado el programa, vamos a programarlo en el microcontrolador. Para ello primero deseleccionamos nuestro depurador y seleccionamos el programador ICD3 (Si el PIC lo requiere tendremos que actualizar la ID del dispositivo en configure->ID Memory).

- Por finalizar, seleccionamos las opciones de grabación en programmer->Settings y construimos el programa, y para grabarlo finalmente le damos a programmer->program y ya tendremos grabado el código en nuestro microcontrolador.

4. TARJETA DE EVALUACIÓN PARA EL PIC32: STARTER KIT

CAPÍTULO 4. TARJETA DE EVALUACIÓN PARA EL PIC32: STARTER KIT

4.1. INTRODUCCIÓN

Como ya hemos comentado en el capítulo anterior, la idea de realizar este proyecto surgió tras el lanzamiento al mercado del potente microcontrolador PIC32 junto con una serie de herramientas de desarrollo que facilitaban su uso. En concreto, el equipo de evaluación PIC32 Starter kit, el cual facilita mucho la interacción con este novedoso microcontrolador. Este kit inicial para el PIC32 representa un método simple y de bajo coste para llevar a cabo un primer contacto con los microcontroladores de 32 bits. Además el kit incluye todo lo necesario para escribir un programa, efectuar el depurado y ejecutar el código en un microcontrolador PIC32.

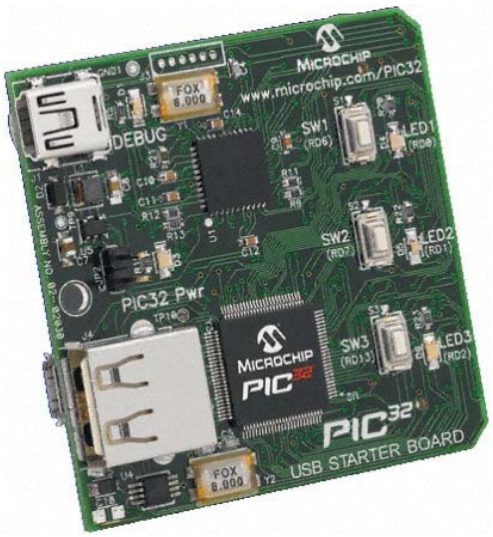


Figura 4.1: Sistema de evaluación PIC32 Starter Kit.

Además, junto con el kit, se incluye un Cd que contiene el software necesario para la grabación y depuración vía USB de los programas ejemplos incluidos en el mismo y que comentaremos en el apartado 4.2. Por lo que estos programas pueden servir como base a la hora de elaborar los primeros programas con la familia de microcontroladores PIC32.

4.1.1. CARACTERÍSTICAS GENERALES

A continuación se enumeran las principales características del sistema de evaluación PIC32 Starter Kit:

1. Microcontrolador de 32 bit PIC32MX360F512L con 512KB de memoria Flash y 32KB de RAM .
2. En la tarjeta, reloj de 8 MHz.
3. Microcontrolador PIC18LF4550 para depurar y programar.
4. Conectividad mediante USB para las comunicaciones de depuración.
5. Indicador Led naranja de depuración.
6. Tres pulsadores.
7. Tres Leds de distintos colores, definidos por el usuario.

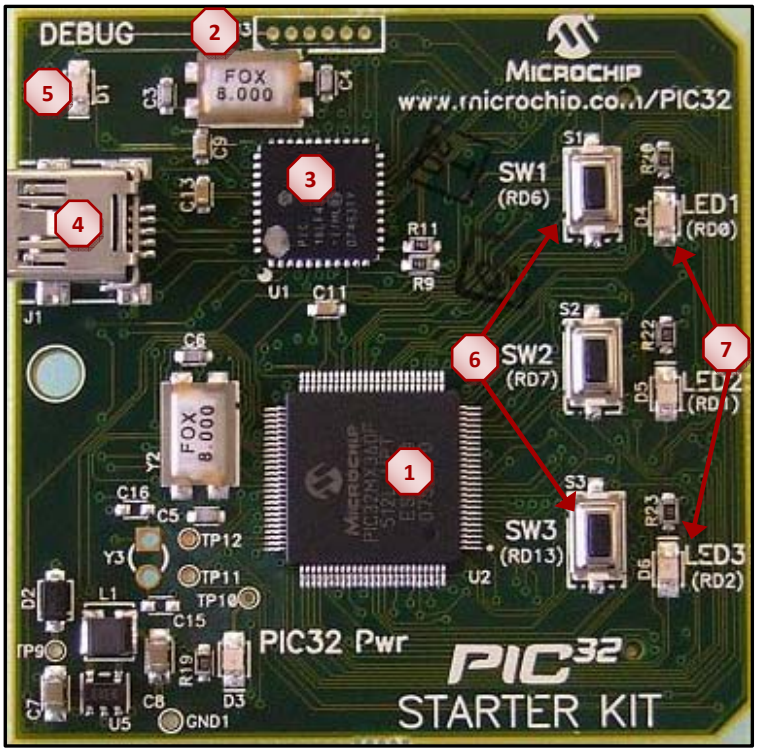


Figura 4.2: Componentes del sistema de evaluación PIC32 Starter Kit.

4.1.2. ARQUITECTURA DE LA TARJETA PIC32 STARTER KIT

En este apartado se realiza una explicación exhaustiva del sistema de evaluación PIC32 Starter Kit.

Suministro eléctrico:

Hay dos formas de suministrar la energía al PIC32 del Starter Kit:

- Cable USB conectado a J1.
- Una tarjeta externa con un suministro eléctrico regulado para corriente continua que proporcione +5V y que se conecte a J2 del Starter Kit, situado en la parte posterior de la tarjeta de evaluación.

Conectividad USB:

El PIC32MX Starter Kit incluye un microcontrolador PIC18LF4550 USB, el cual proporciona la conectividad y el soporte del protocolo USB. El cableado desde PIC18LF4550 al PIC32MX nos proporciona dos tipos de conectividad:

-Pins de E/S del PIC18LF4550 a ICSP pins del PIC32MX

-Pins de E/S del PIC18LF4550 a JTAG pins del PIC32MX

El PIC32MX Starter Kit usa comúnmente los pines del JTAG del PIC32MX para programar y depurar. La primera vez que se enchufa, el PIC18LF4550 se carga con el firmware USB. Además, mediante la conexión USB se puede actualizar de una forma muy sencilla el firmware del dispositivo.

Interruptores:

La tarjeta de evaluación dispone de los siguientes pulsadores:

-SW1: Interruptor activo bajo conectado a RD6

-SW2: Interruptor activo bajo conectado a RD7

-SW3: Interruptor activo bajo conectado a RD13

Estos interruptores no tienen ningún circuito anti-rebote y requieren el uso interno de resistencias conectadas a +5V. Por tanto, los interruptores estarán a nivel alto (3,3V) cuando no se pulsan y cuando los presionamos se encontrarán a nivel bajo.

LEDS:

Los 3 leds disponibles se encuentran conectados a las líneas 0, 1 y 2 del puerto D. Para que se enciendan estos leds, los pines del PORTD correspondientes se tendrán que poner a nivel alto para que se enciendan. Además, cada uno de estos leds tienen un color distinto, rojo, naranja y verde tal y como podemos ver en la Figura 4.3.

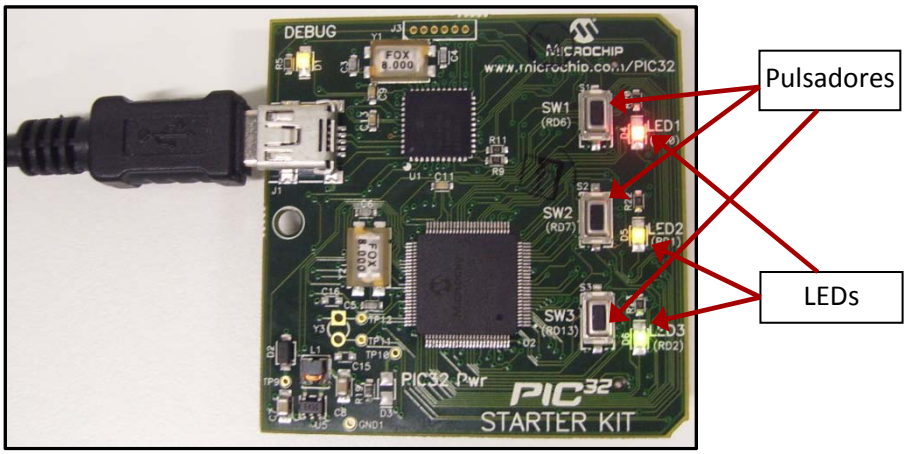


Figura 4.3: PIC32 Starter Kit, Leds encendidos.

Opciones del Oscilador:

El microcontrolador de la tarjeta de evaluación tiene un circuito oscilador el cual se encuentra conectado a este. El oscilador principal (Y2) usa un reloj de 8MHz y actúa como el oscilador primario del controlador. Por tanto, no es necesario el uso de un cristal externo para usar el PIC32. En caso de que queramos se podrá usar el oscilador interno del microcontrolador.

Por otra parte, el reloj del PIC18LF4550 es independiente del PIC32MX y tiene su propio cristal de 8MHz (Y1).

Conector para expansión modular de 120-pin (J2):

La placa del PIC32MX Starter Kit ha sido diseñada con una interfaz de expansión modular de 120 pins (situada en la parte trasera de la tarjeta, ver figura 4.4). Esta interfaz proporciona a la placa un elemento modular para posibles ampliaciones de nuevas tecnologías que puedan aparecer en el futuro, haciendo de la tarjeta un elemento flexible. Así mismo este conector nos va a servir más adelante para poder conectar esta tarjeta a la Explorer16 así como a la I/O Expansión Board que veremos en los capítulos 5 y 7 respectivamente.

Figura 4.4: Conector para expansión modular de 120 pins y/o alimentación de la tarjeta.

A continuación mostramos el diagrama de bloques de alto nivel de la tarjeta de evaluación PIC32 Starter Kit:

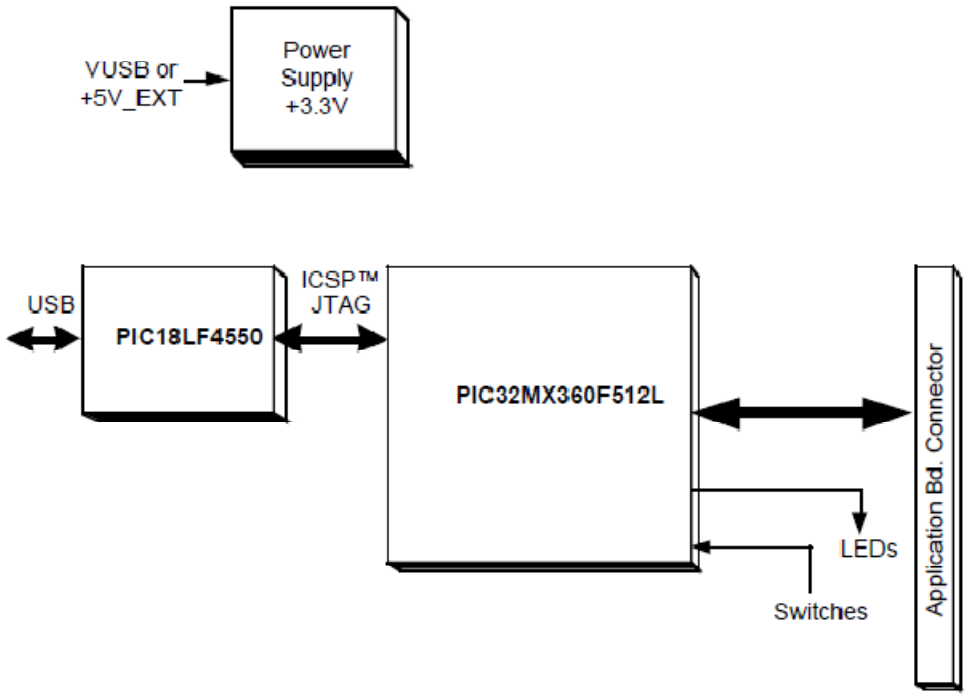
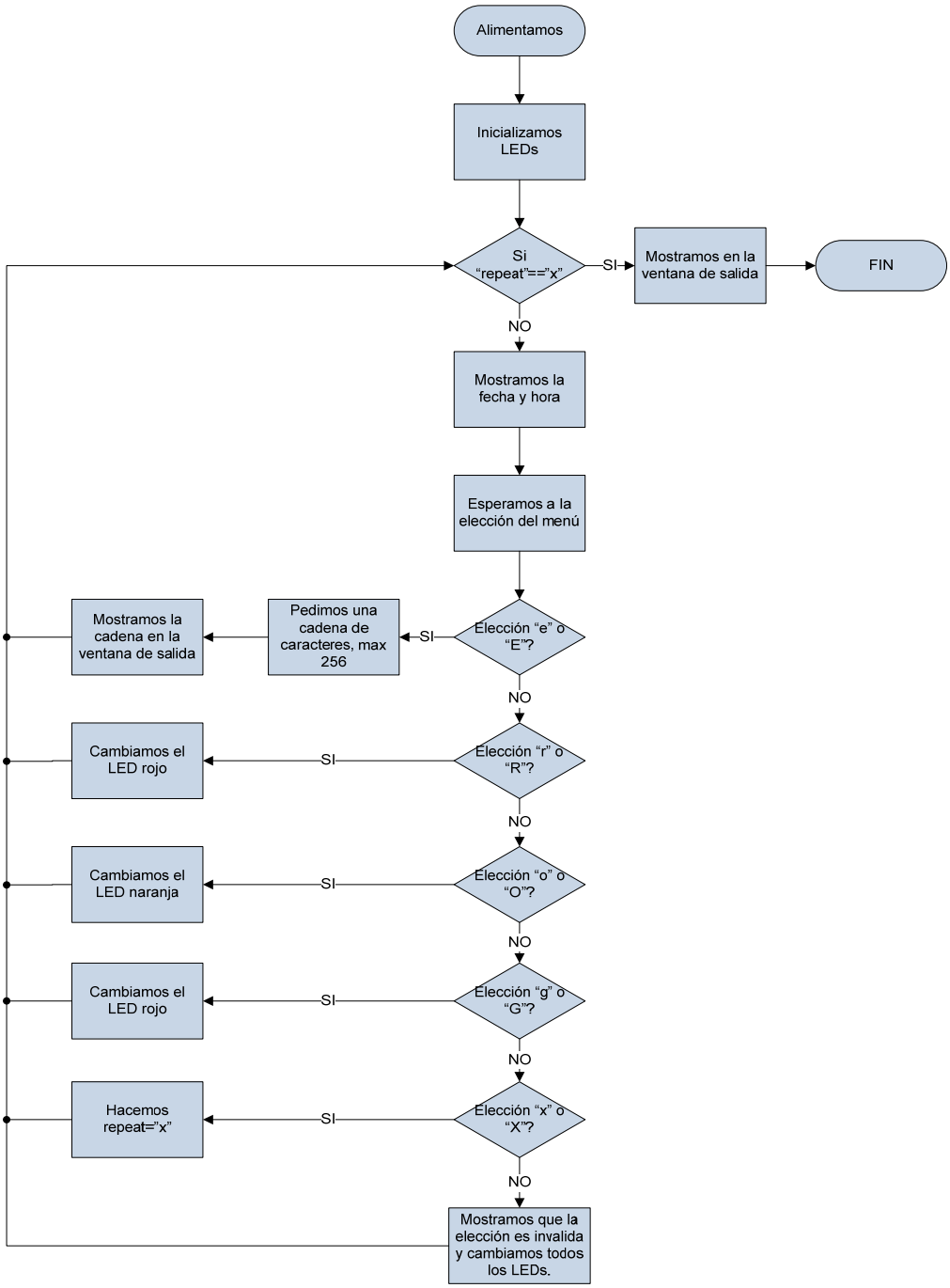


Figura 4.5: Diagrama de Bloques de la tarjeta de evaluación PIC32 Starter Kit.

4.2. PROGRAMAS UTILIZADOS PARA VERIFICAR STARTER KIT

A continuación se detallan algunos de los programas utilizados para comprobar el funcionamiento de la tarjeta de evaluación. Hay que tener en cuenta, que al ser una tarjeta de evaluación sin muchos módulos con lo único que se puede operar es con los pulsadores y con los leds de distintos colores.

El primer programa evaluado (*StarterKitTutorial.c*) sigue el siguiente diagrama de flujo:



Este primer programa incluye la librería "Debug print" (*db_utils.h*) con la que vamos a ser capaces de mostrar mensajes por pantalla a la espera de una respuesta por parte del usuario. En este cuadro se nos va a indicar que debemos de introducir una de las letras que se nos indican en la ventana de salida, estas son: E, R, O, G, X, si la entrada no se corresponde con ninguna de estas letras se mostrará un mensaje de error y se intercambiará el estado de cada LED (ver diagrama de flujo anterior). Por tanto, el programa responderá con una acción u otra dependiendo de la letra introducida. A continuación mostramos el cuadro de diálogo (figura 4.6) en el que introducimos nuestra elección así como los mensajes que se muestran en la ventana de salida (Figura 4.7) del MPLAB IDE:

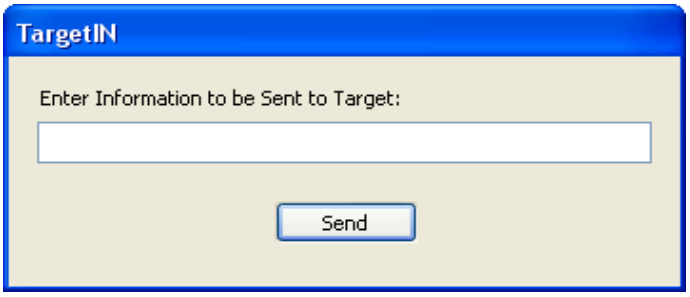


Figura 4.6: Cuadro de diálogo para la introducción de la acción a realizar, programa "*Starterkitturorial.c*".

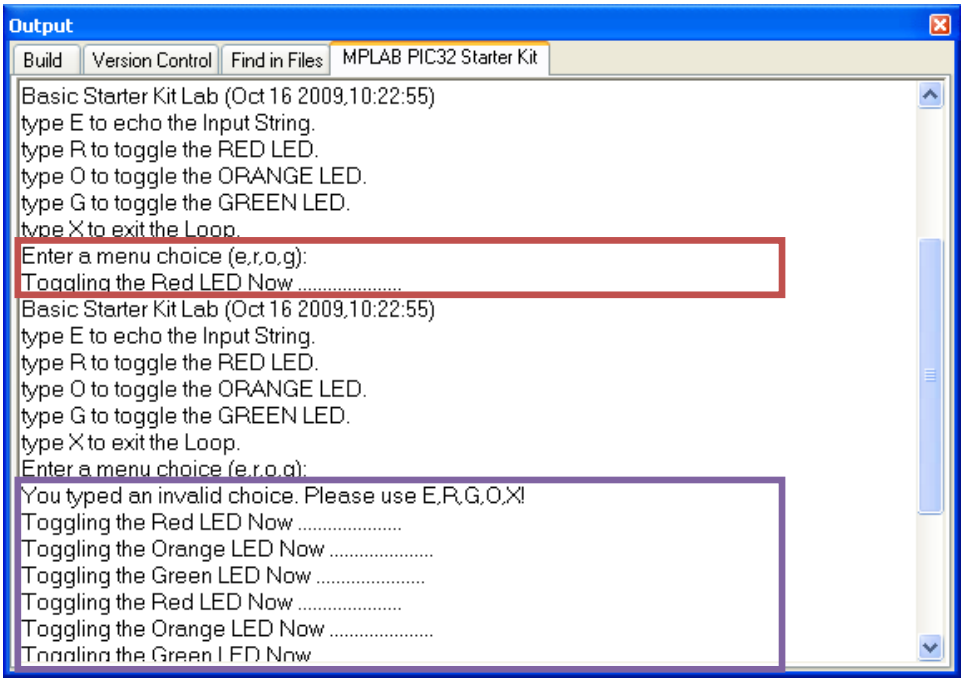


Figura 4.7: Ventana de Salida del programa "*StaterKitTutorial.c*".

En la Figura 4.7, podemos ver el resultado de la ventana de salida tras haber conectado la tarjeta mediante el cable USB al ordenador e introducir “R” en el cuadro de diálogo (color rojo). En el siguiente recuadro podemos ver el mensaje de error tras la introducción de un carácter erróneo, lo que lleva a intercambiar el estado de los Leds.

El segundo programa (*“Simon_says.c”)* evaluado, consiste en imitar al juego clásico “Simon dice”. Primeramente, cuando programamos el Starter kit con el juego y conectamos el cable USB, los 3 Leds comenzarán a iluminarse de forma intermitente indicando el comienzo del juego. El juego comenzará presionando uno de los 3 pulsadores, con tal de indicar la dificultad del juego. De manera que pulsaremos el interruptor SW3 si queremos el nivel más fácil, SW2 para un nivel intermedio y SW1 para el más difícil. El objetivo del juego es imitar las luces que se van encendiendo de forma aleatoria tanto tiempo como podamos presionando el pulsador correspondiente del led que se haya encendido. El Juego termina cuando cometamos un error al presionar el interruptor equivocado, momento en el que todos los LEDS se encenderán indicando que el juego ha terminado. Después de una pequeña pausa, podremos seleccionar un nuevo nivel de juego y comenzar otra vez.

4.3. PROGRAMAS REALIZADOS PARA EL PIC32 STARTER KIT

Como ya hemos dicho, el modelo exacto de 32 bits montado sobre nuestra tarjeta Starter Kit es el PIC32MX360F512L. Por tanto sobre este vamos a grabar los programas realizados en este apartado, sin embargo, también se podrían grabar en el otro microcontrolador que disponemos, el PIC32MX460MX512L.

Las opciones de grabación para todos los programas desarrollados en este apartado se muestran en la siguiente figura:

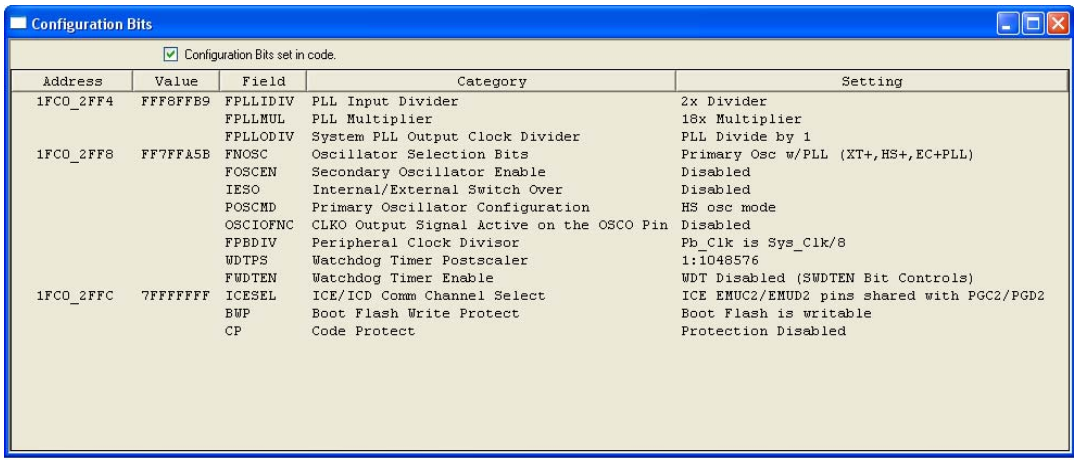


Figura 4.8: Configurations bits para el PIC32 Starter Kit.

Programa 1: "Semaforosv1.0.c"

El programa simula el comportamiento de un semáforo. Primeramente se supone que el semáforo se encuentra en rojo. De tal forma que estará en este estado durante 15 segundos, momento en el que se encenderá el led verde durante 25 segundos y posteriormente el led naranja durante 5seg, para volver a encenderse el led rojo. Para contabilizar el tiempo se ha utilizado el timer1 mediante el cual se ha generado una función que cuenta 0.1 segundos.

Para controlar la mayoría de las funciones del Timer1 vamos a tener que acceder a 3 registros de propósito general, el TMR1, el cual contiene el valor del contador de 16 bits, T1CON, el cual controla la activación y el modo de operación del Timer y el registro PR1, el cual se usa para producir un reset periódico del Timer (no lo vamos a necesitar en nuestro caso). Para que el Timer1 comience a contar desde cero simplemente tendremos que poner el registro TMR1 a 0. Para configurar el registro T1CON consultamos el manual:

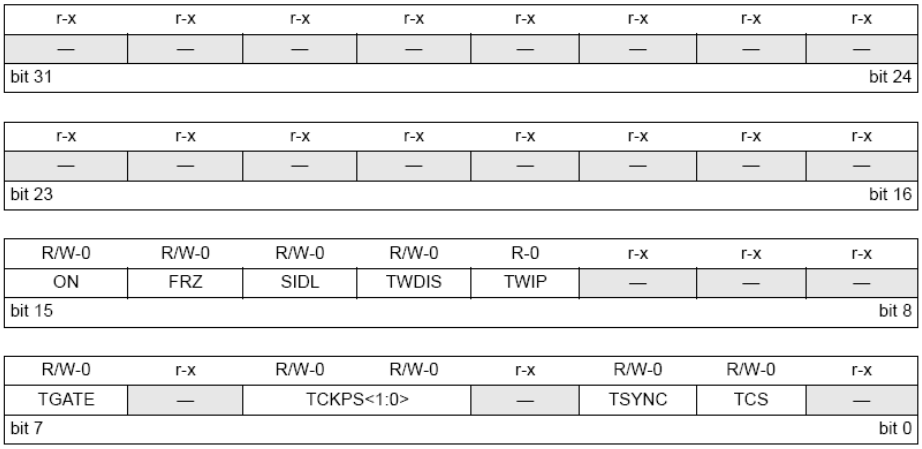


Figura 4.9: Registro T1CON, asociado al Timer1.

Tendremos que operar sobre los siguientes bits de este registro:

- TON, lo pondremos a 1 para activar el Timer1
- TCS, como fuente usamos el reloj principal del MCU, por tanto lo ponemos a 0.
- TCKPS, seleccionamos como preesacalado el valor mínimo (1:1), luego TCKPS=00.
- TGATE y TSYNC, como usamos directamente para que cuente el Timer el reloj interno del MCU, ambos bits los pondremos a 0.

Por tanto, el valor que tendremos que asignar al registro es:

T1CON= 1000 0000 0011 0000 (binario) equivalente a T1CON=0x8000 (hexadecimal)

Una vez asignadas las opciones de operación al Timer1 habrá que calcular el número total de impulsos a contar por el mismo antes de que desborde. Para ello usamos la siguiente fórmula:

$$Tdelay = (Fpb) * Prescaler * DELAY$$

Donde, Tdelay es el tiempo deseado (nuestro caso 1 ms), Fpb es el periodo de un ciclo máquina (hay que tener en cuenta que estamos operando a una frecuencia del bus periférico de 36MHZ), Prescaler es el preescalado seleccionado (1 en nuestro caso) y Delay los impulsos a contar por el Timer1.

Por tanto operando para nuestro caso sale un valor Delay igual a 36000.

El programa se muestra a continuación:

```
/*********************************************************************************************
                                    Semáforo v1.0 README
 *********************************************************************************************
 * Objetivo: Simula el funcionamiento de un semáforo. En rojo durante 15 seg,
 * en verde durante 25 seg y en ámbar durante 5 seg.
 *
 * Tools:
 *                          1. MPLAB with PIC32MX support
 *                          2. C32 Compiler
 *                          3. Starter Kit.
 *
 *********************************************************************************************/
#include <p32xxxx.h>            // Processor defs
#include <plib.h>               // Peripheral Library's

////////////////////////////////////////////////////////////////////////////
//                      PROTOTYPES
////////////////////////////////////////////////////////////////////////////
void Initialize(void);
void SemaforoTask(void);
void Delayms( unsigned t);

// Configuration Bit settings
//  System Clock = 72 MHz,  Peripherial Bus = 36 MHz
//  Primary Osc w/PLL (XT+,HS+,EC+PLL)
//  Input Divider    2x Divider
//  Multiplier      18x Multiplier
//  WDT disabled
//Other options are don't care
//
//#pragma config PLLMUL = MUL_18, PLLIDIV = DIV_2, FWDTEN = OFF
//#pragma config POSCMOD = HS

// LED Macros
#define mSetAllLedsOFF()        (mSetRedLedOFF(), mSetYellowLedOFF(), mSetGreenLedOFF())

#define mSetRedLedON()          mPORTDSetBits(BIT_0)
#define mSetRedLedOFF()         mPORTDClearBits(BIT_0)

#define mSetYellowLedON()       mPORTDSetBits(BIT_1)
#define mSetYellowLedOFF()      mPORTDClearBits(BIT_1)

#define mSetGreenLedON()        mPORTDSetBits(BIT_2)
#define mSetGreenLedOFF()       mPORTDClearBits(BIT_2)

//Delay
#define FPB 36000000L

int main(void)
{
   // Set Periph Bus Divider 72MHz / 2 = 36MHz Fpb
   SYSTEMConfigWaitStatesAndPB(72000000);

  // Initiallize board
  Initialize();
          while (1)
          {
                  SemaforoTask();
```

```
        }
        return 0;
}//main

// Initialize routine
void Initialize(void)
{
        // Turn All Led's Off
        mSetAllLedsOFF();

        // Set LED Pins to Outputs
        PORTSetPinsDigitalOut(IOPORT_D, BIT_0 | BIT_1 | BIT_2);
}//Initialize

void SemaforoTask(void)
{
        mSetRedLedON();
        Delayms(15000); //wait 15 sec.
        mSetRedLedOFF();
        mSetGreenLedON();
        Delayms(25000); //wait 25 sec.
        mSetGreenLedOFF();
        mSetYellowLedON();
        Delayms(5000); //wait 5 sec.
        mSetYellowLedOFF();
}//SemaforoTask

/*********************************************************************
** Delay
**
** uso del:    Timer1: 0.1 segundos.
 *********************************************************************/

void Delayms( unsigned t)
{
   T1CON = 0x8000;     // Habilitacion del TMR1, Tpb, 1:1
   while (t--)
   {  // t x 1ms bucle
      TMR1 = 0;
      while (TMR1 < FPB/1000);
   }
} // Delayms
```

Programa 2: "Semaforosv1.2.c"

En esta siguiente versión del programa anterior, vamos a añadir la posibilidad de indicar el tiempo que queremos que este cada Led encendido.

Para ello vamos a utilizar los interruptores presentes en la tarjeta, funcionando del siguiente modo: El led que tengamos que configurar se encenderá previamente durante 5 segundos y para asignar el tiempo pulsaremos SW1 tantas veces como segundos queramos, teniendo en cuenta que después de cada pulsación se encenderá el Led Rojo durante un segundo indicando que se ha contabilizado tal segundo. Una

vez apagado podremos volver a pulsar de nuevo. Cuando ya hayamos pulsado todas las veces que queramos el interruptor, pulsaremos SW3, tras lo cual se encenderá durante 5 seg el siguiente LED a configurar. Una vez realizada esta operación con los 3 LEDs, se mostrarán de forma intermitentemente para señalar que ya hemos finalizado con la configuración de los mismos. Terminado el paso anterior se mostrará el semáforo actuando como en el programa anterior (*semaforov1.0.c)*, pero en el que cada Led se encenderá el periodo de tiempo indicado. A continuación mostramos las modificaciones realizadas respecto al programa anterior:

```
********************************************************************************
                                Semaforo v1.2 README
 ********************************************************************************
 * Objetivo: Simula el funcionamiento de un semáforo. En el que introducimos mediante
 * los pulsadores de la tarjeta los segundos que queremos que se encienda cada uno.
 *
 * Tools:
 *                          1. MPLAB with PIC32MX support
 *                          2. C32 Compiler
 *                          3. Starter Kit.
********************************************************************************/
#include <p32xxxx.h>            // Processor defs
#include <plib.h>               // Peripheral Library's

////////////////////////////////////////////////////////////////////////////
//                      PROTOTYPES
////////////////////////////////////////////////////////////////////////////
void Initialize(void);
int SelectTime(void);
void ConfigureTime(void);
void ToogleLeds(void);
void SemaforoTask(void);
void Delayms( unsigned t);

// Configuration Bit settings
//  System Clock = 72 MHz,  Peripherial Bus = 36 MHz
//  Primary Osc w/PLL (XT+,HS+,EC+PLL)
//  Input Divider    2x Divider
//  Multiplier      18x Multiplier
//  WDT disabled
//Other options are don't care
//
//#pragma config PLLMUL = MUL_18, PLLIDIV = DIV_2, FWDTEN = OFF
//#pragma config POSCMOD = HS

// LED Macros
#define mSetAllLedsOFF()        (mSetRedLedOFF(), mSetYellowLedOFF(), SetGreenLedOFF())
#define mSetAllLedsON()         (mSetRedLedON(), mSetYellowLedON(), mSetGreenLedON())

#define mSetRedLedON()                  mPORTDSetBits(BIT_0)
#define mSetRedLedOFF()                 mPORTDClearBits(BIT_0)

#define mSetYellowLedON()               mPORTDSetBits(BIT_1)
#define mSetYellowLedOFF()              mPORTDClearBits(BIT_1)
```

```
#define mSetGreenLedON()                mPORTDSetBits(BIT_2)
#define mSetGreenLedOFF()               mPORTDClearBits(BIT_2)

// Switch Macros
#define mGetRawRedSwitch()              (!mPORTDReadBits(BIT_6))  //SW1
#define mGetRawYellowSwitch()           (!mPORTDReadBits(BIT_7))  //SW2
#define mGetRawGreenSwitch()            (!mPORTDReadBits(BIT_13)) //SW3

//Delay
#define FPB 36000000L

//Globar variables
int TimeR,TimeY,TimeG;

int main(void)
{
   // Set Periph Bus Divider 72MHz / 2 = 36MHz Fpb
   SYSTEMConfigWaitStatesAndPB(72000000);

        // Initiallize board
        Initialize();

        //Configure time
        ConfigureTime();

        while (1)
        {
                SemaforoTask();
        }
        return 0;
}//main

// Initialize routine
void Initialize(void)
{
        // Turn All Led's Off
        mSetAllLedsOFF();

        // Set LED Pins to Outputs
        PORTSetPinsDigitalOut(IOPORT_D, BIT_0 | BIT_1 | BIT_2);

        // Set Switch Pins to Inputs
        PORTSetPinsDigitalIn(IOPORT_D, BIT_6 | BIT_7 | BIT_13);
}//Initialize

int SelectTime(void)
{
int Time,c=0;
while(mGetRawGreenSwitch()==0)
{
        if (mGetRawRedSwitch()==1)
        {
        mSetRedLedON();
        Delayms(1000);
        mSetRedLedOFF();
        c++;
        }
}
Time=c;
return Time;
```

```
}//SelecTime

void ToogleLeds(void)
{
        mSetAllLedsON();
        Delayms(1000);
        mSetAllLedsOFF();
        Delayms(1000);
}//ToogleLeds

void ConfigureTime(void)
{
        mSetRedLedON();
        Delayms(5000);
        mSetRedLedOFF();
        TimeR=SelectTime();
        mSetGreenLedON();
        Delayms(5000);
        mSetGreenLedOFF();
        TimeG=SelectTime();
        mSetYellowLedON();
        Delayms(5000);
        mSetYellowLedOFF();
        TimeY=SelectTime();
        //Finalizado
        ToogleLeds();
        ToogleLeds();
        ToogleLeds();
        ToogleLeds();
}//ConfigureTime

void SemaforoTask(void)
{
        mSetRedLedON();
        Delayms(TimeR*1000); //wait TimeR sec.
        mSetRedLedOFF();
        mSetGreenLedON();
        Delayms(TimeG*1000); //wait TimeG sec.
        mSetGreenLedOFF();
        mSetYellowLedON();
        Delayms(TimeY*1000); //wait TimeY sec.
        mSetYellowLedOFF();
}//SemaforoTask

/*********************************************************************
** Delay
**
** uso del:    Timer1: 0.1 segundos.
 *********************************************************************/

void Delayms( unsigned t)
{
   T1CON = 0x8000;     // Habilitacion del TMR1, Tpb, 1:1
   while (t--)
   {  // t x 1ms bucle
      TMR1 = 0;
      while (TMR1 < FPB/1000);
   }
} // Delayms
```

Programa 3: "Semaforosv2.0.c"

Por último hemos realizado una última mejora al programa. Seguramente en el programa anterior, podría ocurrir que no supiéramos muy bien cómo funciona realmente el programa ni cuántas veces hemos pulsado el interruptor asignando de esta manera los segundos a cada led. Este problema lo vamos a resolver usando la librería *db_utils.h,* la cual como ya hemos comentando se utiliza para mostrar mensajes en la ventana de salida mientras se encuentra el depurador funcionando.

Este programa ya más completo que los anteriores no lo vamos a mostrar debido a su extensión, sin embargo, el código del mismo se puede consultar en el Cd adjunto al presente proyecto.

A continuación mostramos la ventana de salida del MPLAB IDE para una configuración de 20 segundos en rojo, 30 en verde y 5 en ámbar.

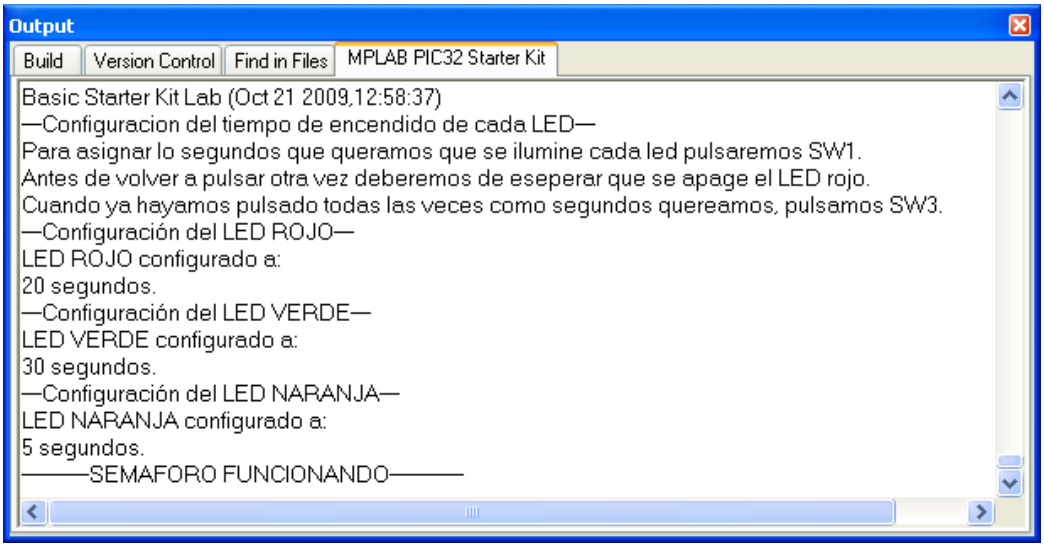


Figura 4.10: Ventana de Salida del programa "*semáforos v2.0.c*".

Podríamos seguir mejorando este programa, añadiéndole nuevas funcionalidades, pero no merece la pena realizar programas más complejos con el PIC32 STARTER KIT, ya que no dispone de más módulos que podamos probar y ver de una forma sencilla, pues esta tarjeta ha sido diseñada para realizar la primera toma de contacto con la familia de microcontroladores de microchip de 32 bits y no para realizar programas complejos.

5. SISTEMA DE DESARROLLO EXPLORER16

CAPÍTULO 5. SISTEMA DE DESARROLLO EXPLORER16

5.1. INTRODUCCIÓN

El sistema de desarrollo Explorer16, consiste en un entrenador o placa de evaluación para aplicaciones basadas en los microcontroladores PIC de Microchip. Se ha diseñado para permitir a estudiantes e ingenieros explorar y trabajar con las capacidades de los microcontroladores PIC. Permite además, concentrarse principalmente en el desarrollo del software puesto que las conexiones entre microcontroladores PIC y circuitos externos son muy sencillas de realizar.

Dispone de una serie de periféricos básicos de E/S con los que se puede verificar el funcionamiento de una aplicación, así como los circuitos necesarios para la grabación de diversos modelos de microcontroladores PIC24F/24H/dsPIC33F/PIC32MX de 100 pines.

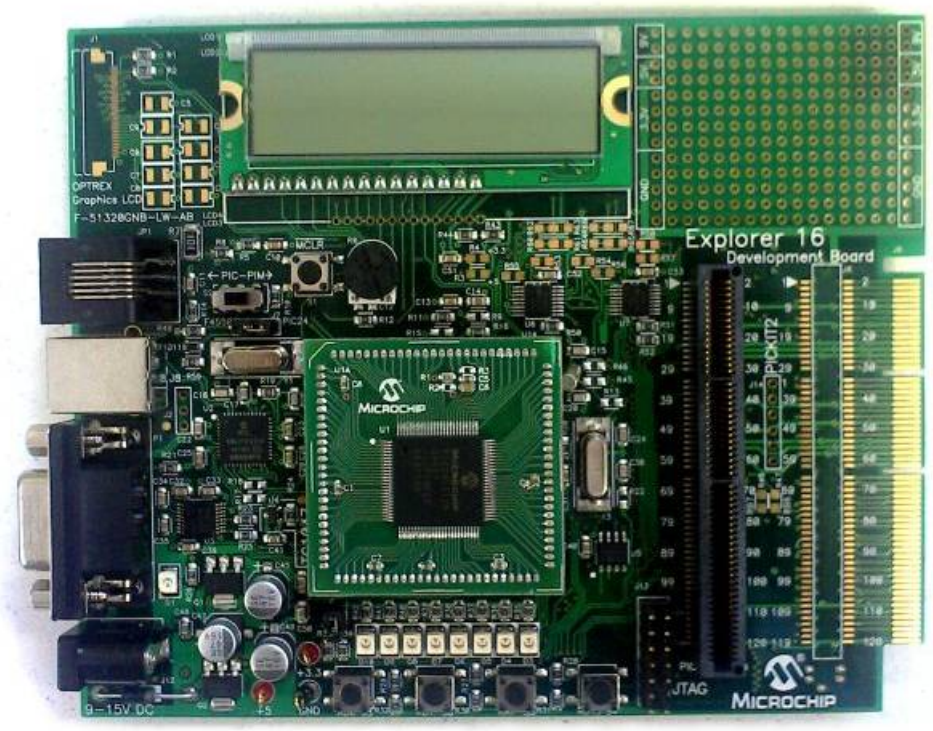


Figura 5.1: Sistema de desarrollo Explorer16.

5.1.1. CARACTERÍSTICAS GENERALES

A continuación se enumeran las principales características del sistema Explorer16:

1.-Zocalo de 100 pines, compatible con las versiones de todos los dispositivos de Microchip siguientes: PIC24F/24H/dsPIC33F/PIC32MX.

2.- Alimentación mediante fuente de alimentación externa AC/DC de 9 a 15V proporcionando +3.3V y +5V en toda la placa.

3.- Led indicador de Alimentación.

4.- Puerto serie RS-232 y hardware asociado.

5.- Sensor térmico analógico TC1047A.

6.- Conector USB para comunicaciones y programación/depuración del dispositivo.

7.- Conector para la familia de programadores/depuradores ICD.

8.- Interruptor de selección del procesador.

9.- LCD de 16 caracteres por 2 líneas.

10.- PCB para añadir una LCD gráfica.

11.- Pulsadores, reset y entradas definidas por el usuario.

12.- Potenciómetro de entrada analógica.

13.- 8 Leds.

14.- Multiplexores relacionados con la comunicación serie.

15.- EEPROM serie.

16.- Relojes de precisión.

17.- Área para el desarrollo de aplicaciones propias.

18.- Conector para las tarjetas PICtail Plus.

19.- Interfaz de 6 pins para el programador PICkit2.

20.- Conector JTAG.

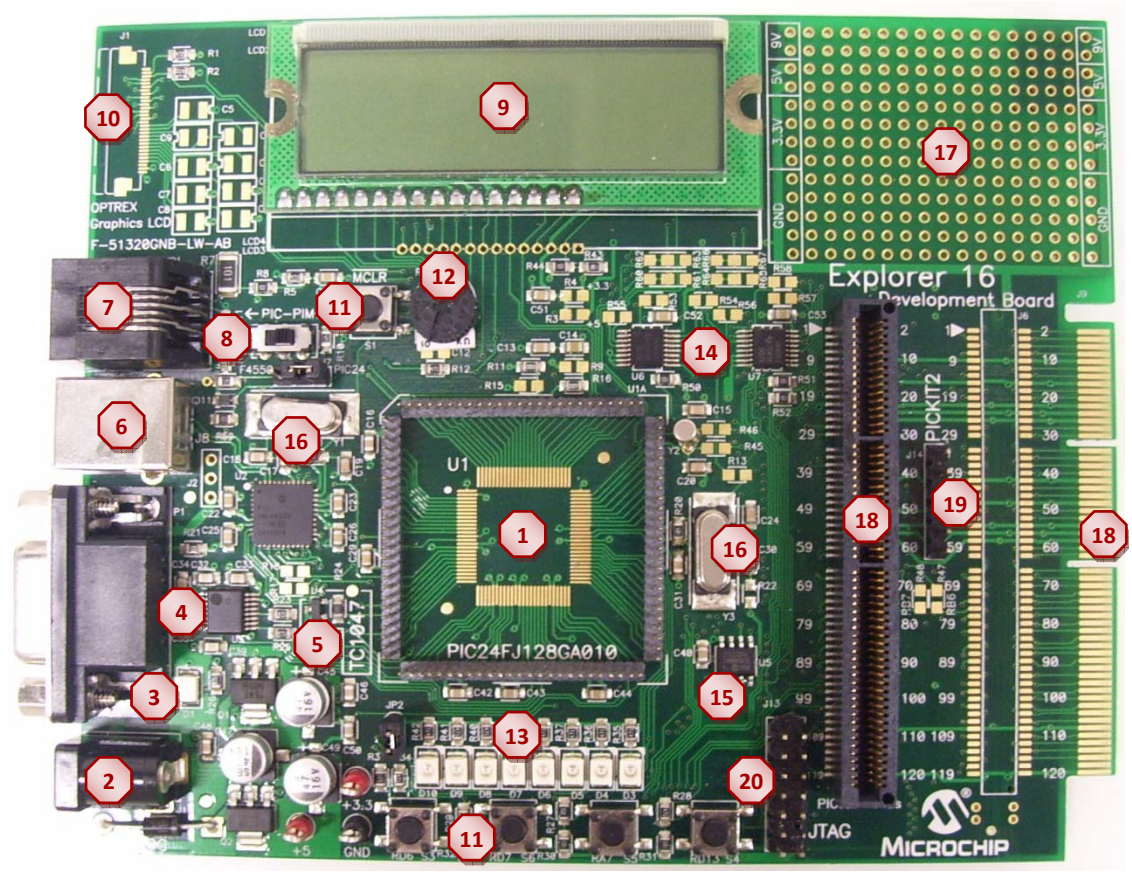


Figura 5.2: Componentes del sistema de desarrollo Explorer16.

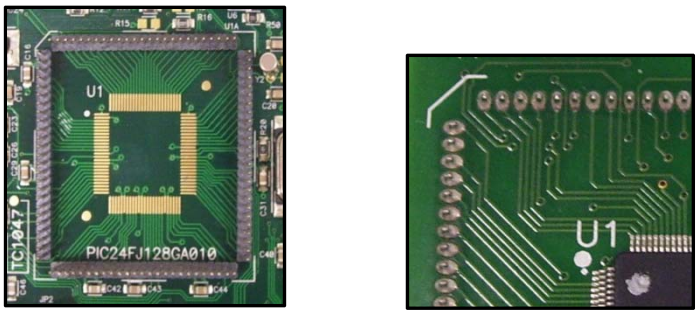
5.1.2. ARQUITECTURA DE LA TARJETA EXPLORER16

En este apartado se realiza una explicación exhaustiva del sistema de desarrollo Explorer16.

Soporte del microcontrolador:

La tarjeta de desarrollo Explorer16 ha sido diseñada para que los microcontroladores estén montados permanentemente (U1) o bien que estos se puedan desacoplar siempre que queramos a través del zócalo de 100 pins. El interruptor S2, nos indica de qué forma vamos a utilizar nuestro microcontrolador, el cual se explica más adelante en el apartado “Interruptor de selección de zócalo”.

Por otra parte, siempre que coloquemos cualquier microcontrolador en la placa nos tenemos que fijar que la marca con la esquina cortada quede orientada hacia la esquina superior izquierda. De lo contrario estaríamos colocando mal el dispositivo y no funcionaría correctamente.



a) b)

Figura 5.3: a) Zócalo de 100 pins. b) Detalle de la esquina para la colocación del PIC.

Interruptor de selección de zócalo:

Este interruptor tiene dos posiciones “PIM” y “PIC”. Cuando en la Explorer 16 no tengamos montado permanentemente un microcontrolador tipo PIC24FJ, tendremos que asegurarnos que el interruptor se encuentra en la posición “PIM”, por el contrario, si lo tenemos de forma permanente tendrá que estar en la posición “PIC”.

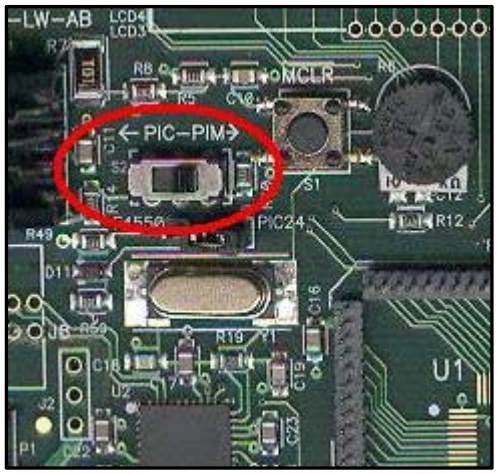


Figura 5.4: Selector del procesador, PIC-PIM.

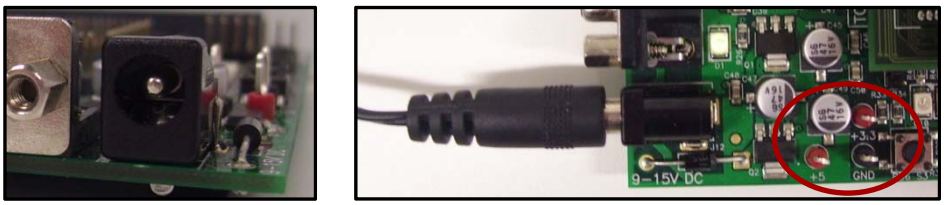
Fuente de alimentación:

Existen 2 formas de suministrar corriente a tarjeta Explorer16:

1.- Mediante fuente de alimentación externa AC/DC de 9 a 15V (preferiblemente DC 9V) conectada a J12, proporcionando +3.3V y +5V en toda la placa.

2.- Mediante una fuente de alimentación externa DC que proporcione +5V y +3.3V y que se conecte a los terminales mostrados en la figura 5.5.b.

Una vez que se suministre corriente a la tarjeta mediante cualquiera de esos dos métodos, se encenderá el LED D1.



a) b)

Figura 5.5: Suministro de energía eléctrica mediante fuente de alimentación externa conectada a: a) Conector J12. b) Patillaje situado en la parte inferior izquierda de la placa.

LEDs:

Los diodos emisores de luz (LED) son los componentes más utilizados, principalmente para mostrar el nivel lógico de los pines a los que están conectados. El sistema Explorer16 tiene 8 LEDs que están conectados al puerto A del microcontrolador. De tal forma que estos LEDs se iluminaran cuando los pines correspondientes del puerto A tengan un valor alto. Este grupo de ocho LEDs puede ser habilitado o deshabilitado mediante el interruptor JP2.

Además la tarjeta Explorer16 dispone de un LED verde (D1) que se enciende cuando se le suministra a la tarjeta una fuente de alimentación, indicando la presencia de +3.3V.

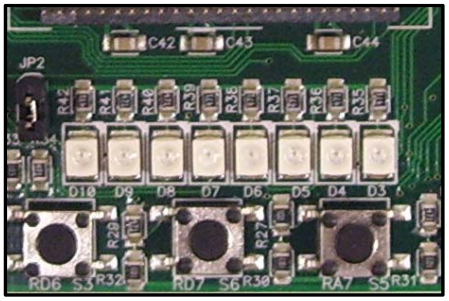


Figura 5.6: LEDs presentes en la tarjeta Explorer16.

Pulsadores:

La placa Explorer16 tiene 4 pulsadores, con los que se puede interactuar a través de diferentes programas. Esos pulsadores se denominan S3, S6, S4 y S5 los cuales están conectados a las líneas 6, 7 y 13 del puerto D y a la línea 7 del puerto A respectivamente. También la placa dispone de un pulsador RESET, que cuando se pulsa el programa del microcontrolador empieza a ejecutarse desde el principio, es decir, resetea el procesador. Todos los pulsadores son activos bajos, es decir, cuando se presionan pasan a tierra, 0V.

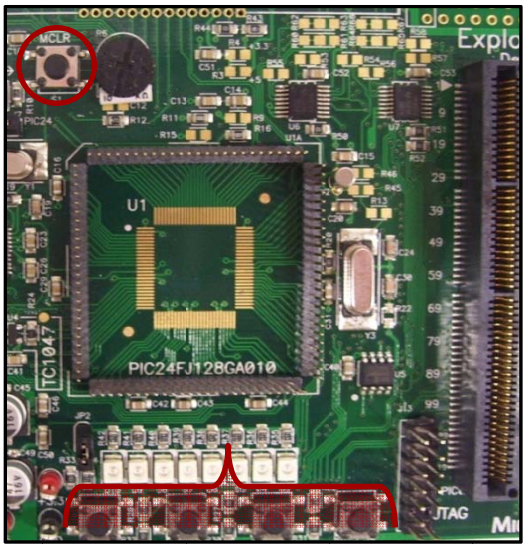
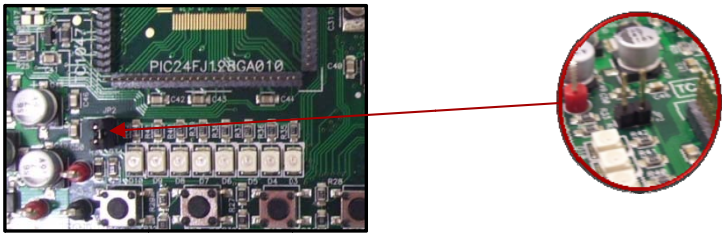


Figura 5.7: Pulsadores presentes el sistema de desarrollo Explorer16.

Jumpers:

Los jumpers, al igual que los interruptores, realizan la conexión entre dos puntos. Debajo de la cubierta de plástico del jumper existe un contacto metálico que establece la conexión al situar el jumper entre dos pines desconectados.

Este es el caso del jumper utilizado para habilitar el grupo de ocho LEDs verdes disponibles en la Explorer16, de tal forma que la conexión se realiza cuando el jumper (JP2) se sitúa entre ambos contactos.

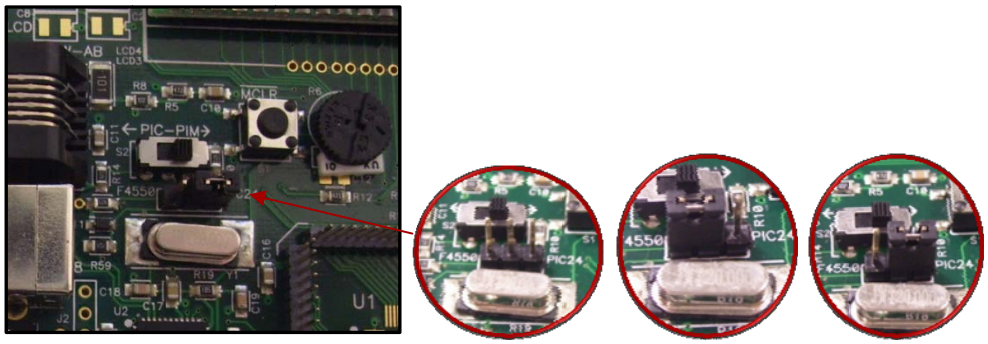


a) b)

Figura 5.8: Jumper en modo Interruptor: a) Jumper en modo ON. b) Jumper en modo OFF.

A menudo, los jumpers se utilizan como selectores entre dos posibles conexiones utilizando conectores de tres pines. Como se muestra en la Figura 5.9, el contacto del centro puede conectarse al pin de la derecha o de la izquierda dependiendo de la posición del jumper.

En la Explorer16 el jumper J7 decide el camino del conector ICD (JP1). Si el jumper esta en lado "PIC24", JP1 se comunica directamente con el dispositivo conectado en el zócalo de 100 pins. Si el jumper esta en lado "F4450", el conector JP1 se comunica primeramente con el microcontrolador PIC18LF4550 de la placa.



a) b) c) d)

Figura 5.9: Jumper J7 en modo multiplexor: a) Ubicación del jumper en la placa Explorer16 b) Detalle de los 3 pines. c) Jumper en el lado F4450. d) Jumper en el lado "PIC24".

Módulo LCD:

En la explorer16 tenemos un display con dos filas por 16 caracteres cada una y un modulo LCD alfanumérico de 3V compatible con los controladores HD44780.

En los módulos alfanumérico podemos colocar directamente un carácter en ASCII en el buffer de la RAM del controlador del modulo LCD (Data RAM Buffer, DDRAM). La imagen de salida se produce usando una tabla 5x7 en la que se representa cada carácter. Además se dispone de la posibilidad de crear o modificar nuevos caracteres accediendo a un segundo buffer interno (Character Generator RAM buffer, CGRAM), ofreciendo la posibilidad de crear 2 nuevos caracteres.

Este módulo LCD se comunica con el PIC a través del puerto paralelo (PMP), por lo que para comunicarnos con este módulo vamos a tener que utilizar el bus paralelo de 8 bits a través del puerto D y E (concretamente las líneas del puerto D, RD4, RD5 y RD15; y del puerto E, RE7:RE0).

A continuación mostramos el juego de caracteres disponibles en el módulo LCD, así como la visualización del LCD mientras se está ejecutando un programa:

Char.code	0 0 0 0 0 0 0 0	0 0 0 1 1 1 1 0	0 1 1 0 0 1 1 1	0 0 1 0 1 0 1 0	1 1 1 1 1 1 1 1	0 0 0 1 1 1 1 0	1 0 0 1 1 0 1 1	1 1 0 1 0 1 0 1	1 1 1 0 0 1 0 1	1 1 1 1 1 0 1 1
××××0000			0	@	P	`	p		ー	タ	ミ	α	p
××××0001		!	1	A	Q	a	q	。	ア	チ	ム	ä	q
××××0010		"	2	B	R	b	r	「	イ	ツ	メ	ß	θ
××××0011		#	3	C	S	c	s	」	ウ	テ	モ	ε	∞
××××0100		$	4	D	T	d	t	、	エ	ト	ヤ	μ	Ω
××××0101		%	5	E	U	e	u	・	オ	ナ	ユ	σ	ü
××××0110		&	6	F	V	f	v	ヲ	カ	ニ	ヨ	ρ	Σ
××××0111		'	7	G	W	g	w	ァ	キ	ヌ	ラ	q	π
××××1000		(	8	H	X	h	x	ィ	ク	ネ	リ	ɾ	x̄
××××1001		)	9	I	Y	i	y	ゥ	ケ	ノ	ル	⁻¹	4
××××1010		*	:	J	Z	j	z	ェ	コ	ハ	レ	i	千
××××1011		+	;	K	[	k	{	ォ	サ	ヒ	ロ	×	万
××××1100		,	<	L	¥	l	|	ャ	シ	フ	ワ	¢	円
××××1101		−	=	M	]	m	}	ュ	ス	ヘ	ン	ŧ	÷
××××1110		.	>	N	^	n	→	ョ	セ	ホ	゛	ñ	
××××1111		/	?	O	_	o	←	ッ	ソ	マ	°	ö	■

Tabla 5.1: Juego de caracteres del módulo LCD

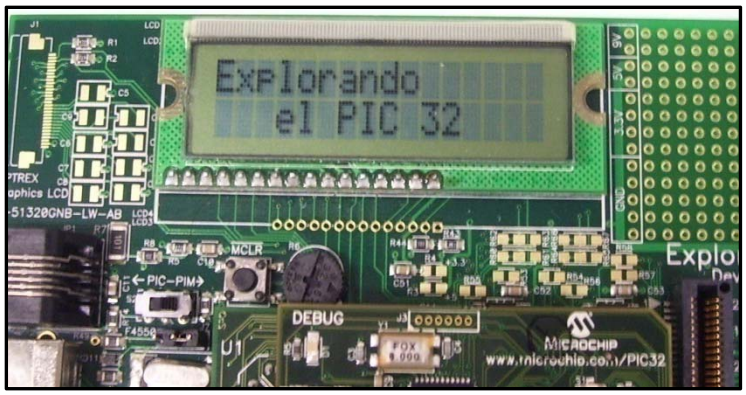


Figura 5.10: Módulo LCD de la placa Explorer16.

Sensor térmico analógico TC1047A:

La placa dispone de un sensor de temperatura con salida analógica (TC1074A [9], U4) conectado a uno de los canales del controlador analógico/digital, concretamente al canal AN4. Este sensor es muy práctico para medir la temperatura ambiente soportando un rango de temperaturas entre -40°C y 125°C. Dentro de este

rango su variación respecto la tensión es lineal de acuerdo a 10mV/°C y con una precisión de +/-0.5 °C.

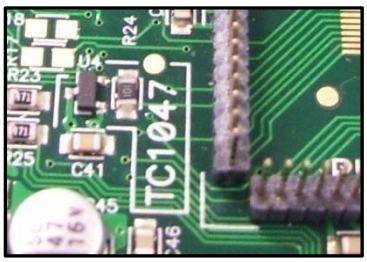


Figura 5.11: Sensor de Temperatura TC1047A.

Potenciómetro:

La tarjeta también dispone de un potenciómetro (R6) el cual está conectado a través de una resistencia (R12) al canal analógico AN5. Este potenciómetro se puede ajustar desde la tensión de alimentación (3.3V) hasta tierra (0V), por lo que su salida estará comprendida entre esos valores.

Figura 5.12: Potenciómetro R6 de 10KΩ.

Conector ICD:

El modulo MPLAB ICD en cualquiera de sus versiones, se puede conectar a la tarjeta Explorer 16 a través del conector modular JP1 tanto para grabar los programas en el microcontrolador como para depurar los mismos. El conector ICD utiliza los pines del puerto B seis y siete, permitiendo el uso del depurador en línea.

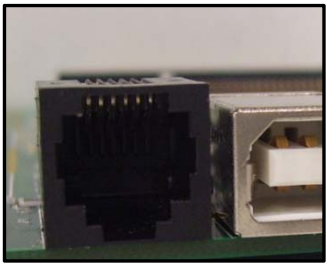


Figura 5.13: Conector para la familia de programadores/depuradores ICD.

Comunicación RS-232:

La comunicación RS-232 permite transferir datos punto a punto. Se suele utilizar en aplicaciones de adquisición de datos para la transferencia de datos entre el microcontrolador y el ordenador. Puesto que los niveles de tensión de un microcontrolador y un ordenador no son compatibles con los del RS-232, se utiliza el adaptador de niveles MAX32320.

Esta interfaz permite realizar todo tipo de comunicaciones serie entre la Explorer16 y cualquier otro equipo mediante el protocolo estándar RS-232. La velocidad de transferencia irá en función del tipo de microcontrolador empleado y su velocidad de trabajo.

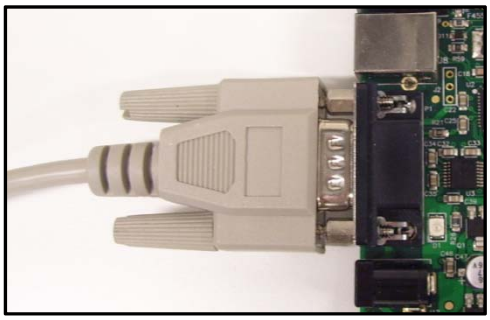


Figura 5.14: Puerto serie RS-232.

Comunicación USB:

El conector de comunicación USB se encuentra en el lado izquierdo de la placa Explorer16. Además, incluye un PUC18LF4550 el cual proporciona el soporte para la transmisión de datos, el controlador USB. Para que se pueda usar la comunicación USB hay que utilizar microcontroladores PIC específicos que soporten la interfaz USB tales como la gama de PIC32MX460F.

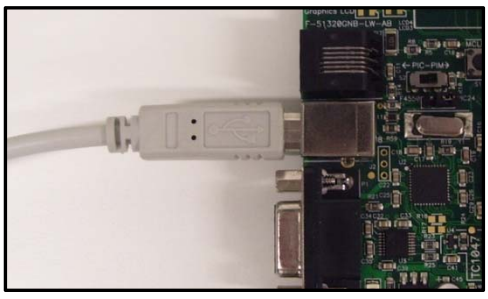


Figura 5.15: Conector USB.

EEPROM serie:

En la placa Explorer16 se ha incluido una memoria EEPROM en serie, 25LC256 (U5), la cual posee 32Kbytes de memoria no volátil. Esta se encuentra conectada al módulo SPI2 con el objetivo de mostrar cómo se utiliza este tipo de comunicación síncrona. Además hay que tener en cuenta que el número de veces que se puede grabar y borrar una memoria EEPROM es finito, por lo que no es recomendable una reprogramación continua de esta.

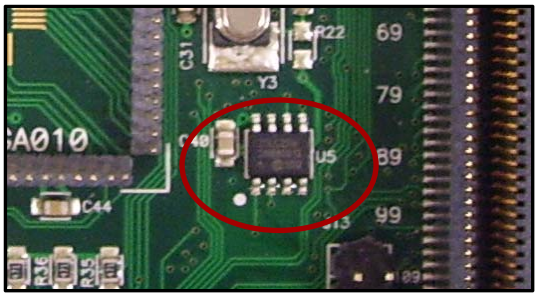
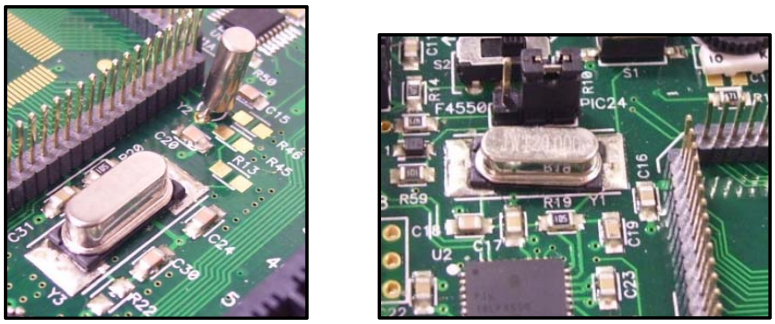


Figura 5.16: Memoria EEPROM (25LC256) presente en la Explorer16.

Relojes:

El oscilador o reloj se encarga de generar la frecuencia principal de trabajo del microcontrolador. En la placa nos podemos encontrar 2 osciladores diferentes. El oscilador principal usa una frecuencia de 8MHz (Y3) y actúa como el oscilador principal. Además existe otro con una frecuencia de 32.768kHz (Y2) el cual actúa como oscilador del Timer1 y sirve como recurso para el contador de tiempos y para el oscilador secundario.

Por otra parte, el PIC18LF4550 presente en la placa, tiene su propio reloj con una frecuencia de 20MHz (Y1).



a) b)

Figura 5.17: Relojes: a) Oscilador primario 8MHz y secundario 32.768kHz (cilíndrico). b) Oscilador para el PIC18LF4550 20MHz.

LCD gráfica:

La explorer16 tiene también un espacio reservado (situado en la esquina superior izquierda) así como el diagrama necesario y el circuito asociado, para colocar una LCD gráfica Optrex con una matriz de puntos de 128x64 (F-51320GNB-LW-AB).

Figura 5.18: Explorer16, PCB para añadir una LCD gráfica.

Conector para las tarjetas de expansión PICtail Plus:

La tarjeta Explorer16 ha sido diseñada con una interfaz para colocar módulos de expansión a través de los conectores PICtail plus. Estos conectores proporcionan una gran flexibilidad para nuevas tecnologías que puedan estar disponibles en el futuro.

Los conectores PICtail Plus se basan en una conexión de 120 pins divididos en 3 secciones de 30 pins, 30 pins y 56 pins. Cada sección de 30 pins proporciona conexiones para todas las comunicaciones con los periféricos así como puertos de E/S, interrupciones externas y canales de A/D. Todo esto proporciona suficientes señales para desarrollar diferentes interfaces de expansión como la tarjeta Ethernet. Por tanto, las tarjetas de expansión PICtail plus de 30 pins se podrán usar tanto en los 30 pins superiores como en los del medio.

En la placa nos encontramos con tres posibles sitios en los que vamos a ser capaces de colocar las tarjetas de expansión, dos conectores hembras J5 (el más popular dado su fácil uso) y J6, así como un conector macho en el lado derecho (J9).

Además la interfaz PICtail Plus nos va a permitir conectar dos tarjetas Explorer16 sin la necesidad de usar un conector adicional, permitiendo el intercambio de datos entre los dos microcontroladores mediante el uso de la interfaz SPI o UART y teniendo en cuenta que solo uno de ellos tendrá que estar conectado a alimentación.

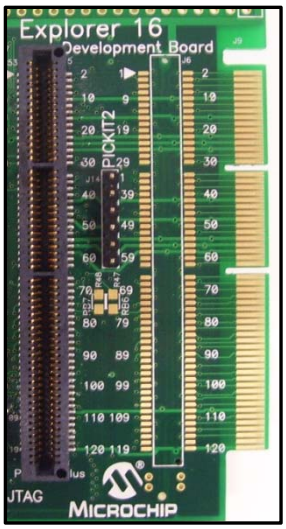


Figura 5.19: Explorer16, conector para tarjetas de expansión PICtaill Plus.

Conector PICkit 2:

El conector J14 proporciona el espacio necesario para colocar la interfaz del programador de 6 pins PICkit 2. Este proporciona la tercera opción para programar a bajo coste el microcontrolador, en lugar de usar la familia MPLAB ICD o bien la interfaz JTAG.

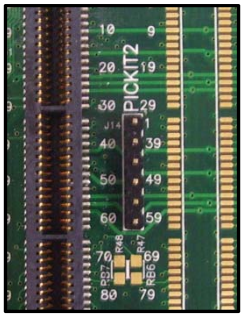


Figura 5.20: Explorer16, conector PICkit 2.

Conector JTAG:

El conector J13 nos va a proporcionar la interfaz estándar JTAG permitiéndonos conectar y programar los dispositivos a través de esta.

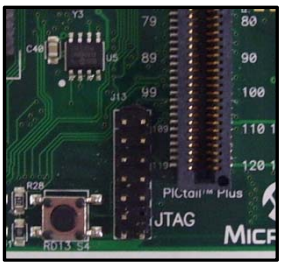


Figura 5.21: Explorer16, conector JTAG.

A continuación mostramos el diagrama de bloques de alto nivel de la tarjeta Explorer16:

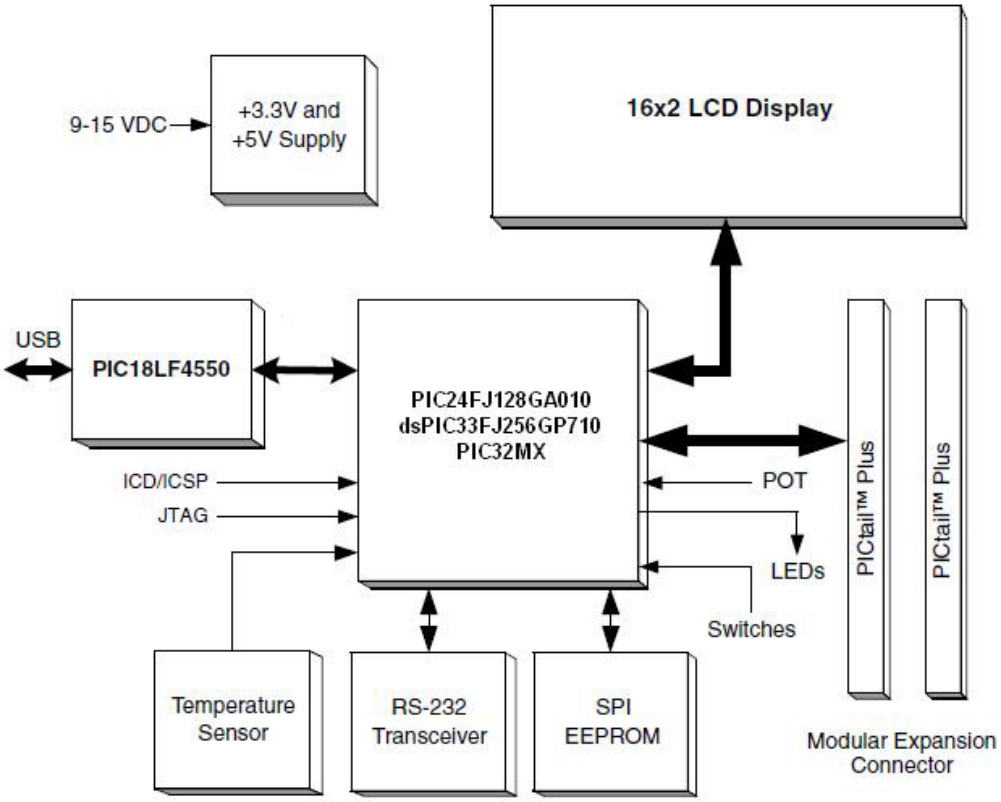


Figura 5.22: Diagrama de Bloques de la tarjeta Explorer16.

5.2. SOFTWARE DE GRABACIÓN

El sistema de desarrollo Explorer16 no dispone de un programador integrado en la placa, como pudiera ser el ejemplo del sistema de desarrollo EasyPIC, por lo que va a ser necesario el uso de un equipo externo durante el proceso de programación.

Como hemos comentado en el Capítulo 5.1.1 existen diferentes formas de grabar en el microcontrolador los programas realizados en el computador. En el presente trabajo se ha utilizado el MPLAB ICD3 In-Circuit Debugger para la grabación y depuración del código, tal y como hemos descrito en el capítulo 3.

Además hay que tener en cuenta que si queremos usar la placa PIC32 Starter Kit vamos a tener que usar la tarjeta 100L PIM Adaptor, para poder conectar la anterior al sistema de desarrollo Explorer16.

Figura 5.23: Adaptador “PIC32 Starter Kit 100L Pim Adaptor”.

A continuación mostramos una figura en la que se está usando el MPLAB ICD3 para grabar en la Explorer16 utilizando la placa PIC32 Starter Kit:

Figura 5.24: MPLAB ICD3 In-Circuit Debugger en funcionamiento en la Explorer16.

5.3. EJEMPLOS DE PROGRAMAS PARA LA EXPLORER16

Tras describir en el Apartado anterior los componentes y características del sistema de desarrollo Explorer16, se realiza una serie de pruebas con el fin de verificar que todos los componentes que integran la placa se encuentran en perfecto estado y no presentan ningún comportamiento anómalo.

Estas pruebas consisten en cargar una serie de programas en el microcontrolador, y comprobar que el comportamiento de la placa se corresponde con el código grabado. Para realizar dichas pruebas se han utilizado algunos de los programas que se encuentran disponibles en el libro "Programming 32-bit Microcontrollers in C: Exploring The PIC32" [10] así como la página web de Microchip [11] en la cual aparecen también muchos programas ejemplos. Sin embargo, en este apartado únicamente vamos a ver algunos de los ejemplos del libro citado anteriormente así como los programas realizados basándonos en ellos.

A continuación mostramos una tabla resumen con los programas que vamos a detallar en este capítulo así como los que únicamente hemos probado de alguna de las dos referencias citadas en el párrafo anterior. Todos estos programas se encuentran en la carpeta "PROGRAMAS_EXPLORER16" del CD adjunto al presente proyecto:

Programa	Carpeta en el CD adjunto al presente proyecto
LED.c	PROGRAMAS_EXPLORER16/LED
LED_TIMER.c	…/LED_TIMER
SPI2.c	…/SPI2
SEEtest.c	…/SPI2
Serial.c	…/UART
U2test.c	…/UART
LCDtest.c	…/LCD_PMP
ProgressBar.c	…/LCD_PMP
Bounce.c	…/PULSADORES
Buttons.c	…/PULSADORES
Pot.c	…/POTENCIOMETRO
Pot-man.c	…/POTENCIOMETRO
Temperatura.c	…/TEMPERATURA
Message.c	…/OTROS_PROGRAMAS/MESSAGE
String.c	…/OTROS_PROGRAMAS/STRING
Punteros.c	…/OTROS_PROGRAMAS/STRING
Liquid.c	…/OTROS_PROGRAMAS/LCD_MANUAL

Counter.c	…/OTROS_PROGRAMAS/ICD_3
Timer.c	…/OTROS_PROGRAMAS/ICD_3
Todos (para el PIC24)	…/OTROS_PROGRAMAS/TUTORIAL_EXPLORER16
Programas Web Microchip	…/OTROS_PROGRAMAS/All_PIC32_v1-00-04.zip

Tabla 5.2: Lista de todos los programas evaluados sobre el sistema Explorer16.

Vistos todos los programas presentes en el CD adjunto, mostramos una tabla resumen con todos los programas que vamos a detallar con mayor profundidad en los siguientes apartados, indicando el nombre del programa, sus elementos implicados y una breve descripción del mismo:

Programa	Elementos Implicados	Funcionamiento
LED.c	LEDS	Encendido de los LEDs conectados al puerto A
LED_TIMER.c	LEDS y Timer1	Apagado y encendido de leds cada 256ms
SPI2.c	Interfaz SPI y EEPROM 25LC256	Escritura en la EEPROM a través de la interfaz síncrona SPI
SEEtest.c	Interfaz SPI y EEPROM 25LC256	Verificación de la funcionalidad de las librerías SEE.c y SEE.h, incrementando una variable de la EEPROM.
Serial.c	Interfaz UART e Hyperterminal	Comunicación entre el PIC32 y el ordenador mediante el RS-232, escribimos datos en el teclado y se muestran en la consola del Hyperterminal
U2test.c	Interfaz UART e Hyperterminal	Verificación de la funcionalidad de las librerías conU2.h y conU2.c, lo que escribimos en la consola del programa se duplica hasta 128 caracteres.
LCDtest.c	Interfaz PMP y modulo LCD	Verificación de la funcionalidad de las librerías LCD.c y LCD.h mostrando en el modulo LCD un mensaje.
ProgressBar.c	Interfaz PMP y modulo LCD	Simulación de una barra que se va cargando mediante la creación de un carácter nuevo cada 100ms.
Bounce.c	Pulsadores	Ejemplo que muestra el efecto de rebote en los pulsadores
Buttons.c	Pulsadores, interfaz PMP y modulo LCD	Se muestra en el modulo LCD que botón se ha presionado, codificado en hexadecimal.
Pot.c	Potenciómetro, modulo	Nos muestra en el display una barra

	ADC, Interfaz PMP y modulo LCD	asociada al potenciómetro cuya carga depende del valor de este, indicando así mismo el valor del potenciómetro.
Pot-man.c	Potenciómetro, modulo ADC, Interfaz PMP y modulo LCD	Simulación del juego Pac-Man en el LCD, moviendo este último girando el potenciómetro. Añadido el número de veces que "comemos" y posibilidad de reinicio del juego.
Temperatura.c	Sensor de temperatura, modulo ADC, Interfaz PMP y modulo LCD	Programa Pot-man.c aplicado al sensor de temperatura, en el que además se muestra la temperatura en la segunda línea del display.

Tabla 5.3: Programas detallados ejecutados en el sistema de desarrollo Explorer16.

Vamos a detallar algunos de los códigos utilizados (Tabla 5.3), así como una breve descripción del comportamiento del sistema de desarrollo Explorer16. Para todos estos programas se ha utilizado el esquema de la figura 5.24, usamos el PIC32 Starter Kit, por tanto los programas son ejecutados para el modelo PIC32MX360F512L.

Las siguientes aplicaciones se han clasificado en función del dispositivo implicado en la ejecución del programa.

5.3.1. EJEMPLO 1: LEDS

Como hemos comentado anteriormente, el grupo de 8 leds presentes en nuestra placa están conectados al Puerto A. Lo primero que tendremos que realizar si queremos que se iluminen los Leds es configurar este puerto como entrada o salida, para lo cual tendremos que hacer uso del registro TRISA. Si queremos configurar el puerto como salida, lo pondremos a 0 y si lo queremos configurar como entrada a 1. En nuestro caso lo que queremos realizar es que se iluminen todos los leds del Puerto A, luego por tanto lo tendremos que configurar como salida.

Sin embargo, hay que tener en cuenta que el puerto JTAG (recordemos que la interfaz JTAG es otra de las opciones disponibles para grabar o depurar en línea a parte de de la interfaz ICD) se encuentra conectado a los pines del puerto A 0, 1, 4, 5, y que tiene prioridad sobre otras acciones. Por tanto, como nosotros usamos el MPLAB ICD3 in-circuit debuggers, tendremos que deshabilitar el puerto JTAG para tener acceso a todos los pines del puerto A.

A continuación mostramos el código del programa y una imagen en la que se pueden ver los leds encendidos:

```
#include <p32xxxx.h>

main()
{
DDPCONbits.JTAGEN=0;  // Disable the JTAG port
TRISA = 0;                      // Configure all POTA as output
PORTA =0xff;
}//main
```

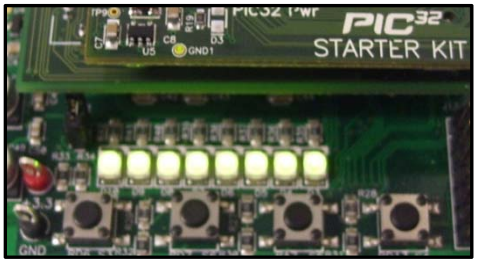


Figura 5.25: LEDS encendidos, programa "*LED.c".*

Como segundo programa, modificamos el anterior, introduciendo el Timer1 (visto en el Capítulo 4 para la realización del programa "*Semaforosv1.0.c"),* el cual lo utilizamos para apagar y encender los Leds cada 256ms. A continuación mostramos el código del programa:

```
/*
** LED_TIMER.c
** Calcule Tdelay-> Tdelay=Fpb*256*DELAY, Tdelay=256ms y FPB->36 MHz
*/

#include <p32xxxx.h>
#define DELAY   36000        // 256ms delay

main()
{
   // Initialization
   DDPCONbits.JTAGEN = 0;  // disable JTAGport, free up PORTA
   TRISA = 0xff00;         // all PORTA as output
   T1CON = 0x8030;      // TMR1 on, prescale 1:256 PB=36MHz
   PR1 = 0xFFFF;           // set period register to max

   //Main loop
   while( 1)
   {
      //Turn all LED ON
      PORTA = 0xff;
      TMR1 = 0;
      while ( TMR1 < DELAY){
         //wait
      }
```

```
        // Turn all LED OFF
        PORTA = 0;
        TMR1 = 0;
        while ( TMR1 < DELAY){
           //wait
        }
    } // main loop
} // main
```

5.3.2. EJEMPLO 2: INTERFAZ SPI

En el apartado 2.2.6.4 del Capítulo 2 comentamos los puertos de comunicación disponibles para nuestro PIC32. Vamos ahora a ver un ejemplo usando la interfaz SPI (comunicación síncrona) para acceder a la memoria EEPROM serie (25LC256, también llamada SEE) presente en la tarjeta de desarrollo Explorer16.

Como podemos ver en la siguiente figura necesitamos 3 líneas, dos para la transmisión de datos, una para los datos de entrada (SDI) y otra para los de salida (SDO) y otra para la línea del reloj (SCK). Sin embargo, se necesita otra línea adicional para conectar cada dispositivo (SS).

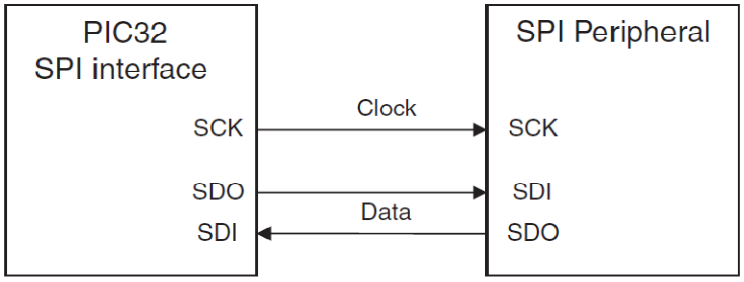


Figura 5.26: Diagrama de bloques de la interfaz SPI.

Esta interfaz se compone básicamente de un registro shift, circuito de alta velocidad que permite desplazar un conjunto de bits a derecha o izquierda. El tamaño de este registro shift varía entre 8, 16 y 32. Estos bits son movidos rápidamente desde el más significante, desde la línea SDI, y hacia afuera, por la línea SDO, síncronamente con el reloj del pin SCK.

El dispositivo se puede configurar de dos formas, dependiendo de la configuración, la señal de reloj provendrá de una u otra parte. Si el dispositivo se configura como bus master, el reloj se generará internamente derivado del reloj periférico (Fpb), generando la onda por el pin SCK. Mientras que si el dispositivo actúa como esclavo recibirá el reloj por el pin SCK.

Las opciones de configuración de esta interfaz se controlan mediante el registro SPIxCON mientras que la señal se genera con el registro SPIxBRG (disponemos de 9 bits para formar la onda). La forma más fácil y rápida de configurar la interfaz es asignando el valor correcto a cada bit del registro SPI2CON. Para ello hay que tener en cuenta las características de la menoría EEPROM a la cual queremos acceder: deshabilitar el modo de 16 bits y el de 32 (luego actuar en modo 8bits, MODE16=0, MODE32=0), reloj activo alto (CKP=0), y los cambios en la salida se producen de activo a bajo (CKE=1). Además hay que configurar el PIC32 como bus master (MSTEN=1), luego:

```
#define SPI_CONF 0x8120 // SPI on, 8 bit master, CKE=1, CKP=0
```

Por otra parte, para determinar la frecuencia de reloj utilizamos la siguiente fórmula:

$$F_{SCK} = \frac{F_{PB}}{2 * (SPIxBRG + 1)}$$

El programa selecciona un valor de SPI2BRG=15, para que a una frecuencia de 9MHZ del bus periférico tengamos una frecuencia F_{SCK} de 280kHZ, con tal de que la comunicación se produzca de forma adecuada y reducir el consumo de la EEPROM.

Lo único que nos falta, es saber cómo está conectada la interfaz SPI con la EEPROM. Consultando el conexionado [12], observamos que el pin12 del puerto D está conectado al pin de selección de memoria CS y que este es activo bajo, nuestro SS.

Una vez sabemos cómo se configura la interfaz vamos a ver de qué manera vamos a poder enviar y recibir datos a través de esta interfaz. Para enviar datos vamos a utilizar el registro SPIxSTAT y el SPIxBUF (para más detalles de estos registros consultar las paginas 388-390 de la referencia [3]). De entre los bits del registro SPIxBUF cabe destacar el bit SPIRBF, con el que vamos a esperar hasta que una transferencia termine (1 = Receive buffer, SPIxRXB is full, 0 = Receive buffer, SPIXRXB is not full).

Ejemplo de cómo enviar un dato:

```
int writeSPI2(int i){
        SPI2BUF=i;
         while (!SPI2STATbits.SPIRBF)
         return SPI2BUF;
}//writeSPI2
```

Una vez disponemos de esta función hay que saber que, tras enviar el comando apropiado con este función, necesitamos enviar un segundo byte (falso, dummy) para

capturar la respuesta desde el dispositivo de memoria. De tal forma que para enviar un dato a la EEPROM se requieren 4 pasos:

- Activar la memoria, CS pin bajo.
- Shift out los comandos de 8 bits
- Enviar o recibir datos
- Desactivar la memoria para completar el comando. Después de esto el dispositivo volverá al modo de bajo consumo

Es decir, si queremos leer el registro Status de la EEPROM el código en c será:

```
CSEE=0;                  //seleccionamos el EEPROM
writeSPI2( SEE_STAT);    //enviamos el comando leer status
i=writeSPI2( 0);         //enviamos y recibimos
CSEE=1;                  //deshabilitamos terminado el comando
```

Si nos fijamos en este trozo de programa estamos escribiendo en la EEPROM SEE_STAT que es uno de los 6 posibles comandos para leer y escribir datos a o desde el dispositivo de memoria, estos comandos se muestran en el código del programa "*SPI2.c*".

Sin embargo, tenemos que tener en cuenta que antes de poder escribir y modificar el contenido de la EEPROM tenemos que poner a 1 el bit WEL del registro STATUS (figura 5.27), bit de habilitación de escritura, ya que por defecto el valor del registro status es 0x00. Por tanto, por defecto no están activados los bits de protección BP1 y BP0, ni el bit WEL de habilitación de escritura ni tampoco el bit WIP de escritura en progreso.

7	6	5	4	3	2	1	0
W/R	–	–	–	W/R	W/R	R	R
WPEN	x	x	x	BP1	BP0	WEL	WIP
W/R = writable/readable; R = read-only.							

Figura 5.27: Registro STATUS de la EEPROM serie, 25LC256.

Una vez conocido como funciona y que registros tenemos que utilizar para configurar la interfaz SPI así como la memoría EEPROM ya podemos probar el siguiente programa, "*SPI2.c*":

```
/*
** SPI2.c
*/
#include <p32xxxx.h>
//Configuración de las opciones de bit, reloj, Fcy=72Mhz, FPb=9Mhz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
```

```
#pragma config FPBDIV=DIV_8, FWDTEN=OFF, CP=OFF, BWP= OFF
// Fcy=8(/2)(*18)(/1) =72MHz   Fpb=72/8= 9Mhz

//Configuración del periferico
#define SPI_CONF 0x8120 // SPI on, 8 bit master, CKE=1, CKP=0
#define SPI_BAUD 15 // clock divider frecuencia SPI=FPB/2*(15+1)=281,25KHz

//I/O Definiciones
#define CSEE  _RD12                 //seleccionamos la linea EEPROM
#define TCSEE _TRISD12   // control para el pin CSEE
//Comandos para el 25LC256
#define SEE_WRSR        1          //escribe en el registro status
#define SEE_WRITE       2          //escribe un comando
#define SEE_READ        3          //lee un comando
#define SEE_WDI         4          //escribe deshablitiar
#define SEE_STAT        5          //lee el registro status
#define SEE_WEN         6          //escribe habilitar

//Función para transferir los datos y devolverlos al dispositivo serie EEPROM

//Envia un byte de datos y lo devuelve al mismo tiempo
/* Inmediantamente escribe un caracter para transmitirlo al buffer, y
** entonces entra en un lazo y espera pa recibir el flag que se haya
** puesto a 1 indicando que la transmissión se ha completado.  El dato
** se devuelve como retorno de la funcion. */
int writeSPI2(int i)
{
          SPI2BUF=i; //escribimos en el buffer
          while (!SPI2STATbits.SPIRBF); //esperamos a que se complete
          return SPI2BUF; //nos devuelve el valor recibido en el dispositivo

}//write SPI2

main()
{
int i;
//1.- Inicializamos el SPI periférico
TCSEE=0;
CSEE=1;
SPI2CON=SPI_CONF;
SPI2BRG=SPI_BAUD;

//BUCLE PRINCIPAL
while (1)
{
// 2.1.-Enviamos el comando de habilitar escritura
CSEE=0;           //habilitamos el EEPROM
writeSPI2( SEE_WEN); //enviamos el comando, ignoramos dato
CSEE=1;
//Habilitado el bit WEL del registro Status de la SEE podemos
//empezar a escribir comandos y modificar el contenido del
// dispositivo.

// 2.2.- Comprobamos el status del EEPROM
          CSEE=0;                   //seleccionamos el EEPROM
          writeSPI2( SEE_STAT);  //enviamos el comando leer status
          i=writeSPI2( 0);          //enviamos y recibimos
          CSEE=1;                   //deshabilitamos terminado el comando
}//bucle principal
}//main
```

En este programa lo que vemos es el valor leído en el apartado 2.2 del mismo a través de la variable "i". Este valor es 2, correspondiéndose a lo esperado tras habilitar el bit WEL del registro STATUS.

Una vez probado el programa anterior veamos la forma general de escribir y leer más de un dato en la EEPROM.

```
/*ESCRIBIR UN DATO EN LA EEPROM
//escribir un dato en la EEPROM
CSEE=0;
writeSPI2(SEE_WRITE); //envia el comando
writeSPI2(ADDR_MSB); // envia msb de la dirección de memoria
writeSPI2(ADDR_LSB); //          envia lsb de la dirección de memoria
writeSPI2(data);  // envia el dato actual
//enviar mas datos aqui para realizar una escritura de pagina
CSEE=1; //Comienza el actual ciclo de escritura en la EEPROM
*/

/*LEER CONTENIDO DE LA MEMORIA
CSEE=0;
writeSPI2(SEE_READ); //envia el comando
writeSPI2(ADDR_MSB); // envia MSB de la dirección de memoria
writeSPI2(ADDR_LSB); //          envia LSB de la dirección de memoria
data=writeSPI2(0);          // envia un dato falso y lee los datos
//leer más datos aqui incrementando la dirección
CSEE=1; //Termina la secuencia de lectura y regresa a bajo consumo
*/
```

Como podemos ver en el código anterior, la EEPROM comienza el ciclo de escritura una vez que la línea CS regresa al valor alto, CSEE=1, por lo que hay que dejar un tiempo (consultar el data sheet de la EEPROM [13]) antes de volver a enviar cualquier otro comando. Hay dos formas de estar seguros de que la memoria ha dejado el tiempo suficiente para completar la escritura. El más simple consiste en insertar un retraso fijo después de la escritura de comando, para el caso más desfavorable sería de 5ms (máximo según [13]). O bien, chequear el registro STATUS antes de leer o escribir cualquier comando, utilizando el bit WIP (esperaremos hasta que se ponga a 0, momento en el que habrá terminado, lo cual coincide con que el bit WEN se pone a 0, reset). Con este último esperaremos únicamente el mínimo tiempo requerido.

Con todo lo desarrollado hasta aquí se crea una pequeña librería para acceder a la memoria EEPROM presente en la Explorer16, "*see.c*" y "*see.h*", las cuales se encuentran en el Cd adjunto a este proyecto.

Para verificar la funcionalidad de las librerías se crea el siguiente programa:

```
/*
** SEE Access library
** Programa para probar la funcionalidad de la libreria SEE
```

```
** Programa que lee repetidamente el contenido de memoria de 16
** e incrementa su valor y lo vuelve a escribir en la memoria.
*/

#include <p32xxxx.h>
#include "see.h"

//Configuracion de las opciones de bit, reloj, Fcy=72Mhz, FPb=9Mhz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_8, FWDTEN=OFF, CP=OFF, BWP= OFF
// Fcy=8(/2)(*18)(/1) =72MHz   Fpb=72/8= 9Mhz

main()
{
int data;
// Inicializacion del puerto SPI2 y CS y acceso al 25LC256
intSEE();
//bucle principal
while(1){
        //lectura del contenido actual de la memoria
        data=readSEE (16);
        //incrementamos el valor actual
        data++; //Breakpoint

        //reescribimos el nuevo valor
        writeSEE( 16,data);
        //address++;
}//bucle principal
}//main
```

Una vez que compilamos el programa, usamos el depurador para probarlo, colocando un breakpoint en el sitio indicado, pues de lo contrario podríamos sobrepasar el límite de escritura de la EEPROM, establecido en un millón de ciclos [13]. Utilizando la herramienta Watch Windows observamos como el contenido de 16 se va incrementando sucesivamente. Además si desconectamos la fuente de alimentación, el contenido de la EEPROM no variará.

5.3.3. EJEMPLO 3: INTERFAZ UART

Esta interfaz también la comentamos en el apartado 2.2.6.4 del Capítulo 2. Vamos ahora a ver un ejemplo usando esta interfaz (comunicación asíncrona) para comunicarnos con un ordenador mediante el puerto serie RS232. Además, tendremos que hacer uso de un programa de emulación para comprobar su funcionamiento (Hyperterminal).

El módulo UART del PIC32 soporta las 4 principales funciones de una aplicación serie asíncrona: RS232 point-to-point connection (puerto serie), RS485 multi-point serial connection, LIN bus e infrared wireless communication. Como hemos dicho nosotros vamos a usar el módulo UART2 conectado a un cable RS232 hembra.

Los parámetros necesarios para configurar el modulo UART son: Ancho de baudio, número de bits de datos, paridad (si es necesaria), número de bits de stop y el protocolo Handshake. Para el programa vamos a usar una configuración rápida y práctica de la interfaz, 115.200 baudio, 8 bits de datos, No paridad, 1 bit de parada y Hardware handshake usando las líneas CTS y RTS.

El Hardware Handshake es necesario para la comunicación con un ordenador Windows, ya que Windows es un sistema de operación multitarea y en estas aplicaciones pueden ocurrir largos retrasos que pueden significar una pérdida de datos. Para ello, usamos el pin RF12 (CTS) de la Explorer16 como entrada para determinar cuando el ordenador está preparado para recibir un nuevo carácter (Clear to send) y un pin como salida (RF13, RTS), para avisar cuando nuestra aplicación está preparada para recibir un nuevo carácter (Request to send).

Por otra parte, para seleccionar el ancho de baudio utilizamos el registro U2BREG, un contador de 16 bits que alimenta al reloj del bus periférico. En modo alta velocidad (BREGH=1) opera con un divisor 1:4, y la fórmula para calcular la configuración ideal de este es:

$$\mathrm{UxBRG} = \frac{\mathrm{F_{PB}}}{4 * (\mathrm{Baude\ Rate})} - 1$$

En nuestro caso Fpb 36Mhz, luego U2BRG=77,125 aproximadamente 77.

Los otros dos registros que nos permiten configurar el módulo UART2 son los registros llamados U2MODE y U2STA (para más detalles de estos registros consultar las paginas 428-433 de la referencia [3]). Del registro U2MODE configuraremos el modo de alta velocidad con el bit BREGH (como hemos visto en el párrafo anterior), el bit de parada (bit STSEL) y el bit de paridad (PDSEL). Mediante el registro U2STA habilitaremos el transmisor TX (bit UTXEN) y deshabilitaremos los flags de errores (bit PERR y FERR).

Por tanto de acuerdo a las características vistas anteriormente tendremos que inicializar ambos registros a:

```
//Inicializacion para el U2MODE
#define U_ENABLE 0x8008 //enable, BREGH=1, 1 stop, no paridad

//Inicializacion para el U2STA
#define U_TX 0x1400 //enable tx & rx, clear all flags
```

Una vez configurada la interfaz UART procedemos igual que con la interfaz SPI, que pasos hay que realizar para enviar un dato al puerto serie:

- El ordenador tiene que estar encendido y con la consola del hyperterminal preparada, chequeando la línea CTS (Clear to send).
- Hay que estar seguro que la interfaz no está ocupada enviando datos. El modulo UART del PIC32 tiene 4 niveles en el buffer de tipo FIFO (First in first out), por tanto tenemos que esperar hasta que al menos el nivel alto este libre. Chequeamos que el flag de buffer lleno UTXBF, este a 0.
- Finalmente transferimos el nuevo carácter al buffer.

```
//Funcion para realizar los 3 pasos de enviar un caracter al puerto serie
int putU2(int c)
{
while (CTS); //esperamos a CTS
while (U2STAbits.UTXBF);  //esperamos mientras buffer lleno
U2TXREG=c; //enviamos el caracter c
return c;
}//putU2
```

Para recibir un dato por el puerto serie, hay que realizar una secuencia parecida a la anterior:

- Avisar al ordenador que estamos preparados para recibir un dato, RTS activo bajo.
- Esperar a la llegada de algún dato en el buffer, chequeando el flag URXDA dentro del registro status del UART2, U2STA.
- Leer y cargar el carácter desde el buffer (FIFO).

```
//Funcion que recibe un caracter del puerto serie
char getU2(void)
{
RTS=0;
while(!U2STAbits.URXDA); //esperamos a la llegada de un nuevo caracter
RTS=1;
return U2RXREG; //lee el caracter recibido en el buffer
}//getU2
```

El programa a probar ("*serial.c*") utiliza las dos funciones anteriores además de la inicialización comentada anteriormente, a continuación mostramos únicamente la función main del programa:

```
main()
{
char c;
//1.-Inicializamos el puerto UART2
initU2();
//2.-enviamos un guia, prompt
putU2('>');
//3.- bucle principal
while(1){
```

```
            //3.1 esperamos a un caracter
            c=getU2();

            //3.2.- visualizamos el caracter por pantalla
            putU2(c);
}//bulce principal
}//main
```

Como hemos dicho antes, para ver cómo funciona la interfaz UART, vamos a utilizar el programa hyperterminal, mediante el cual dispondremos de la consola necesaria para visualizar los datos enviados y recibidos a través del PIC32. Para ello, tenemos que configurar la comunicación con los mismos parámetros que hemos puesto en el programa, es decir:

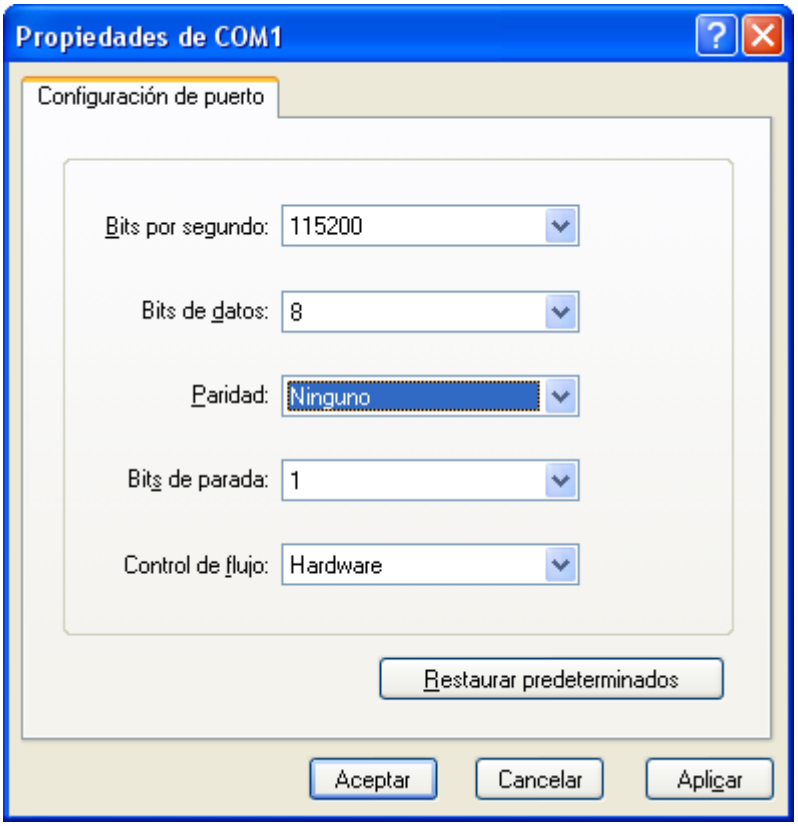


Figura 5.28: Configuración del Programa Hyperterminal.

A continuación mostramos la salida del programa anterior, que como podemos ver en la Figura 5.29, primero recibimos el carácter de guía, para ver que todo funciona correctamente y después, lo que escribimos en el teclado se muestra en la consola.

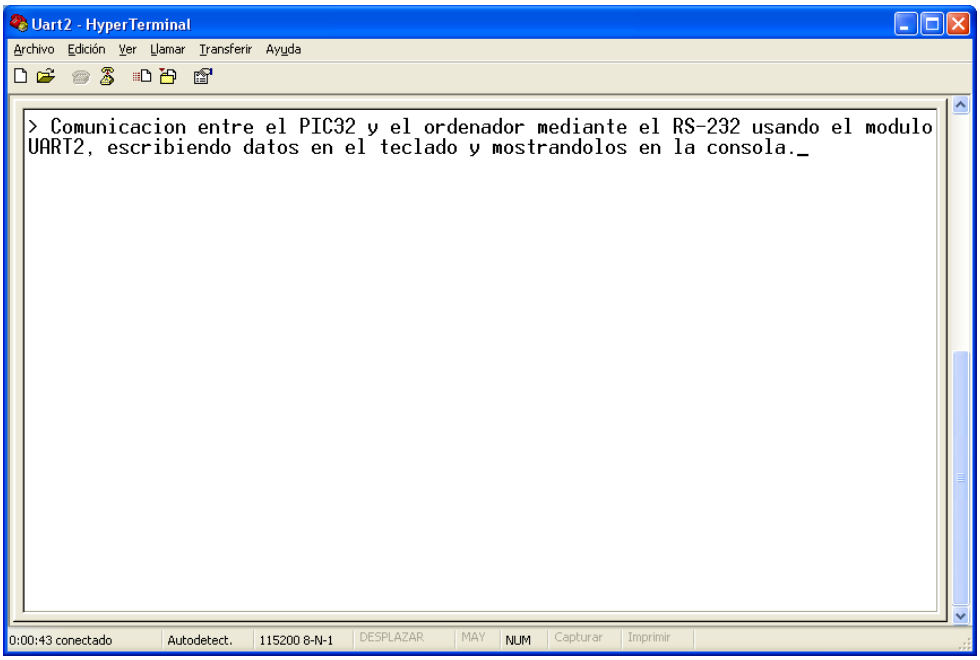


Figura 5.29: Consola del Programa Hyperterminal, ejecución programa "*serial.c*".

Al igual que lo realizado con la interfaz SPI, sería útil tener una librería con las funciones realizadas anteriormente así como la creación de una función que muestre una cadena de caracteres de una vez y no solo un carácter, y de la misma manera una función que lea una cadena de caracteres completa siempre que no sobrepase el buffer, limitado a 128. Por tanto generamos una librería para la comunicación asíncrona añadiendo estas dos funciones.

- Puts(char *s): función que muestra una cadena entera hasta que se escribe un 0.
- Char *getsn(char *s, int len): Lee una cadena desde la consola y la almacena en el buffer (teniendo cuidado de no sobrepasar su límite).
- Creamos una librería para el manejo del modulo UART2, *conU2.h* y *conU2.c.*

Para verificar la funcionalidad de la librería anterior se crea un programa que realiza las siguientes tareas:

- Inicializa el puerto serie y limpia la ventana del hyperterminal.
- Envía un mensaje de Bienvenida a la consola y posteriormente un prompt.
- Se queda esperando a leer una línea de texto y lo vuelve a escribir en una nueva línea.

A continuación mostramos el código de programa de la función main (*U2test.c*):

```
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
```

```
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF
#include <p32xxxx.h>
#include <stdlib.h>
#include "CONU2.h"

#define BUF_SIZE 128

main(){
   //int i = RAND_MAX;
   char s[BUF_SIZE];
   // 1. init the console serial port
    initU2();

   // 2. text prompt
   clrscr();
   home();
   puts( "Exploring the PIC32!");

   // 3. main loop
   while ( 1){
      // 3.1 read a full line of text
      getsn( s, sizeof(s));
      // 3.2 send a string to the serial port
      puts( s);
   } // main loop
}// main
```

Como podemos observar, este programa nos va a permitir usar el puerto serie como una herramienta de depuración. Por tanto, utilizando esta librería vamos a ser capaces de mostrar variables críticas de funciones por pantalla así como otra información adicional que queramos. En la siguiente figura se muestra la consola de salida de este programa.

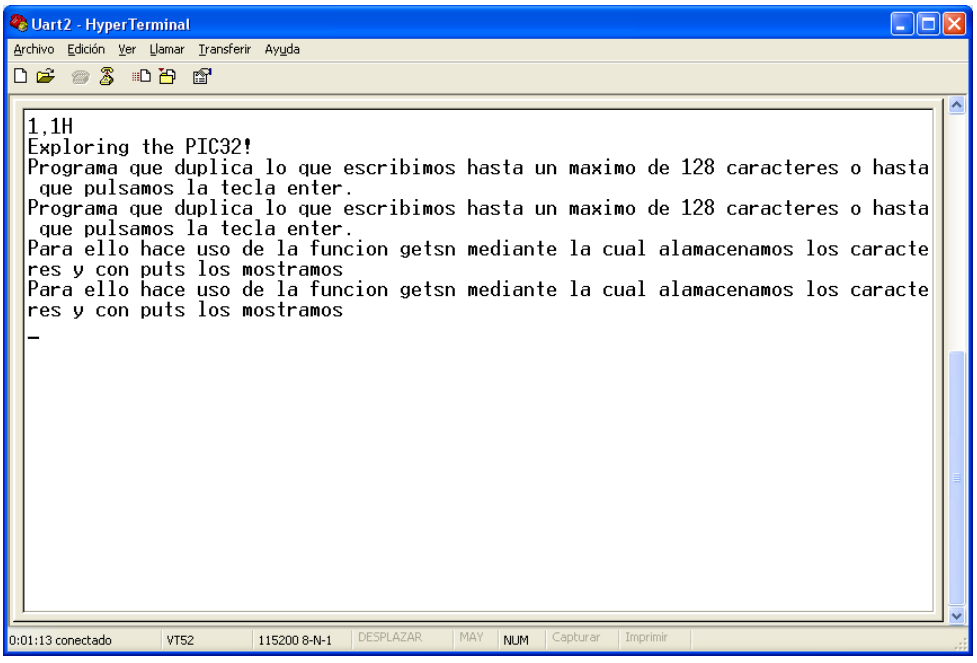


Figura 5.30: Consola del Programa Hyperterminal, ejecución programa "*U2Test.c*".

5.3.4. EJEMPLO 4: MODULO LCD Y PUERTO PARALELO (PMP)

En el apartado 5.1.2 de este Capítulo hemos visto que el sistema de desarrollo Explorer16 dispone de un módulo LCD, el cual se comunica con el PIC32 a través del puerto paralelo (PMP). Veamos ahora un ejemplo de cómo configurar el puerto paralelo para acceder al LCD y mostrar caracteres.

El modulo LCD dispone de un controlador HD44780 el cual contiene 2 registros de 8 bits, uno para datos (DR) y otro para las instrucciones (IR). Para controlar el LCD existen 10 instrucciones predefinas en el HD44780 cuya codificación se puede consultar en la página 24 de la siguiente referencia [14]. Por tanto para operar con el LCD tendremos que enviar estas instrucciones usando el puerto paralelo.

Lo primero que tenemos que realizar, de la misma manera que con las interfaces de comunicación anteriores, es configurar el puerto paralelo para controlar el modulo LCD. Para ello, esta vez existe una gran cantidad de registros destinados a configurar el PMP, de entre los cuales el más importante es el PMCON. Sin embargo, también habrá que inicializar los siguientes registros, PMMODE, PMADDR, PMSTAT y PMAEN. De tal forma que para manejar el LCD tendremos que configurar el PMP de la siguiente manera:

- Habilitar el PMP.
- Separación entre líneas de datos y direcciones.
- Habilitar la señal strobe, en pin RD4.
- Señal de lectura, en pin RD5.
- Habilitar el strobe, activo alto.
- Lectura activo alto, y escritura activo bajo.
- Modo master con señales de lectura y escritura en el mismo pin, RD5.
- 8-bit bus (usando el PORTE).
- Solo se necesita un bit de dirección, por lo que usaremos la mínima configuración para el PMP, usando PMA0 (RB15) y PMA1 sin usar.

No obstante, para configurar el PMP se puede usar una librería específica, *pmp.h*. En particular 4 funciones con todas las herramientas que necesitamos para controlar el PMP así como el dialogo con el display LCD:

- mPMPOpen(), nos ayuda a configurar el PMP.
- PMPSetAddress(), nos permite acceder al registro de direcciones.
- PMPMasterWrite(), inicia la secuencia de escritura.
- mPMPMasterReadByte(), inicia la secuencia de lectura y devuelve el valor.

A la hora de inicializar el puerto paralelo hay que tener en cuenta que el modulo LCD es un dispositivo extremadamente lento, y que el controlador HD44780 necesita de un cierto tiempo para llevar a cabo la secuencia de Inicialización. Luego tendremos que añadir una espera, que en nuestro caso tendrá que ser de al menos 30ms según la referencia [14]. Además, para realizar la secuencia completa de escritura en el bus paralelo hay que seguir una secuencia de pasos específica, que en total se tarda en ejecutar al menos otros 40us. Por lo que a la hora de llevar a cabo la inicialización habrá que tener en cuenta que hay introducir este retraso. El código en c necesario para inicializar el puerto paralelo se muestra a continuación:

```
void initLCD( void){
   //Inicializacion  del PMP
   mPMPOpen( PMP_ON | PMP_READ_WRITE_EN | 3,
            PMP_DATA_BUS_8 | PMP_MODE_MASTER1 |
            PMP_WAIT_BEG_4 | PMP_WAIT_MID_15 |
            PMP_WAIT_END_4,
            0x0001,          // solo PMA0 habilitado
            PMP_INT_OFF);    // No usamos interrupciones

            //espera más de 30ms
   Delayms( 30);

            //Inicialización del HD4470, secuencia
   PMPSetAddress( LCDCMD);    // Seleccionamos el registro de comando(ADDR=0)
   PMPMasterWrite( 0x38);     // 8-bit int, 2 lineas, 5x7
   Delayms( 1);               // > 48 us

   PMPMasterWrite( 0x0c);     // ON, no cursor, no blink
   Delayms( 1);               // > 48 us

   PMPMasterWrite( 0x01);     // clear display
   Delayms( 2);               // > 1.6ms

   PMPMasterWrite( 0x06);     //incrementamos cursor, no shift
   Delayms( 2);               // > 1.6ms
} // initLCD
```

Además como es el caso de las otras interfaces vistas en los apartados anteriores se crea una librería para utilizar de forma más sencilla el LCD. Entre estas funciones destacan las 3 siguientes:

- readLCD(int addr), lee desde el registro de datos o comandos una posición concreta (addr).
- writeLCD(int addr, char c), escribe en una posición concreta (addr) en uno de los registros, el valor c.
- putsLCD(char *s), función que usa la anterior pero que tiene la capacidad de interpretar caracteres especiales como final de línea, tabulación o nueva línea.

Por tanto con las funciones anteriores y con una serie de macros útiles como posicionar el cursor en la primera posición o segunda línea, se crea la librería *LDC.c* y *LCD.h.* A continuación mostramos un pequeño ejemplo (*LCDtest.c*) utilizando la librería anterior.

```
main(){
initEX16();
initLCD();
clrLCD();
putsLCD( "Explorando \n el PIC32");
while(1);
}//main
```

Como podemos observar también se ha incluido la librería *Explore.c* la cual contiene la inicialización del PIC32 así como la función de retraso (*Delayms()*) usada por la librería del módulo LCD .

Figura 5.31: Programa "*LCDtest.c"* ejecutado en la Explorer16.

Por otra parte, como ya hemos anteriormente, mediante el modulo LCD se pueden crear 2 caracteres propios accediendo a un segundo buffer interno (CGRAM). Vamos a ver de qué forma podemos llevarlo a cabo. Primero vamos a necesitar que el puntero del buffer de la RAM del LCD apunte al principio del área CGRAM. Usando las funciones descritas en la librería *LCD.c*, podremos usar la siguiente macro la cual emplea la función *writeLCD()*:

```
#define setLCDG( a) writeLCD(LCDCMD, (a & 0x3F)|0x40)
```

Una vez apunta al principio del buffer (*setLDG(0)*), podemos usar la función *putLCD()* para colocar 8 bytes de datos en el buffer. Cada byte de datos contendrá 5 bits (Lsb), es decir, formaremos el nuevo carácter de 5x8. El nuevo carácter lo podemos definir con el código 0x00.

Vamos ahora a crear un nuevo carácter para simular el comportamiento de una barra de progreso. Para ello se utiliza el último carácter definido en la tabla (0xff) y además se crea un bloque para dar más precisión al programa. Este bloque estará formado por Wx8, donde W será el ancho del carácter (código 0x00), de acuerdo a la siguiente figura:

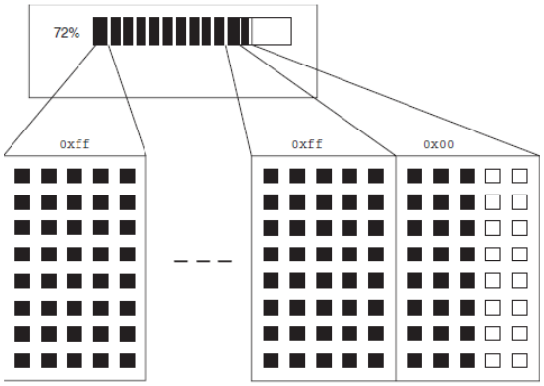


Figura 5.32: Esquema del nuevo carácter a crear en el modulo LCD (0x00).

El programa en c de este ejemplo ("*ProgressBar.c"*) se puede consultar en el CD adjunto al presente proyecto en la carpeta Explorer16/LCD_PMP. Básicamente consiste en simular una barra la cual se va cargando de 0 a 100. Para lo cual se va creando el carácter nuevo que sea requerido cada 100ms. A continuación mostramos una imagen del programa siendo ejecutado en la placa.

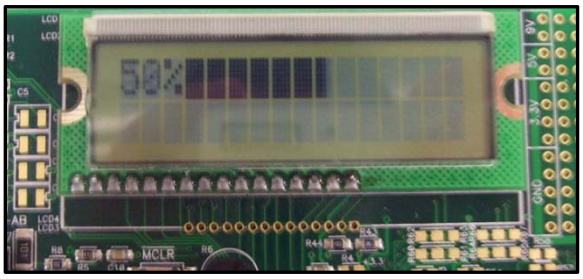


Figura 5.33: Programa "*ProgressBar.c"* ejecutado en la Explorer16.

5.3.5. EJEMPLO 5: PULSADORES

El sistema de desarrollo Explorer16, como hemos comentado en el apartado de arquitectura de la Explorer16, dispone de 4 pulsadores, por tanto tiene 4 entradas digitales (1-0). Cuando el botón se pulsa el contacto se cierra y el pin de entrada pasa a nivel bajo, mientras que si no se pulsa, una resistencia hace que la salida lógica se mantenga a nivel alto. Veamos ahora un ejemplo usando estos pulsadores.

Si se considera al switch como un componente ideal la transición entre los dos estados sería inmediata, no obstante esto no es así. Generalmente existe la posibilidad de que ocurra un rebote, este efecto es conveniente eliminarlo ya que estos rebotes se podrían contar como distintas activaciones de la línea de entrada. Este efecto se puede observar en la siguiente gráfica así como en el ejemplo *bounce.c*.

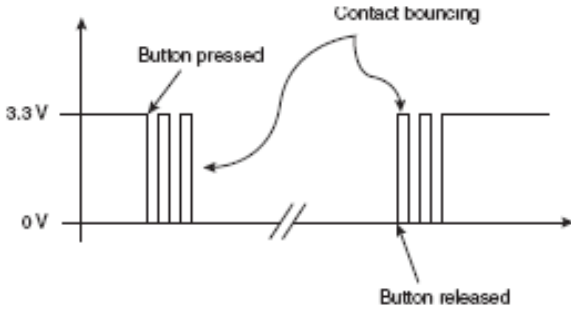


Figura 5.34: Rebotes ocasionados por los pulsadores.

Si ejecutamos el programa *bounce.c*, podremos comprobar este efecto. Añadiendo a la ventana Watch el contador count y presionando el pulsador unas veinte veces nos damos cuenta que este marca alrededor de 23 ya que se ha producido el efecto comentado anteriormente. Código del programa *bounce.c*:

```
main (void){
int count; //contador de rebotes
count=0;
while(1){
        while (_RD6); //esperamos a que el boton sea pulsado
        count++; //contamos las veces que es presionado
        //esperamos a que se deje de pulsar el boton
        while(! _RD6);
        }//Bucle principal
}//main
```

La técnica general para evitar estos rebotes consiste en añadir un pequeño restraso después de detectar la primera conmutación de una entrada y seguidamente verificar que la salida ha alcanzado una condición estable. Cuando el botón se suelta, se añade otro breve retraso, antes de verificar una vez más que se ha alcanzado una condición estable, función getK(). Mientras que la función readK(), nos va a permitir detectar que entrada ha sido pulsada o si se han presionado más de un pulsador simultáneamente.

En el siguiente programa ejemplo (*buttons.c)* mostramos por el LCD que botón se ha presionado codificándolo en hexadecimal. Es decir, si se pulsa el botón más a la izquierda (S4) se corresponderá con 1, y si se pulsa el botón más a la derecha (S3), se corresponderá con 8. Además se pueden presionar varios botones a la vez dando lugar al código hexadecimal desde el 1 hasta F. A continuación mostramos el código completo:

```
int readK( void){ // devuelve 0..F si se presiona, 0 = nada
   int c = 0;
   if ( !_RD6) // De izquierda a derecha
```

```
      c |= 8;
   if ( !_RD7)
      c |= 4;
   if ( !_RA7)
      c |= 2;
   if ( !_RD13)
      c |= 1;
   return c;
} // readK

int getK( void){ // Esperamos a que se pulse un pulsador
   int i=0, r=0, j=0;
   int c;
   // 1. Esperamos a que se pulse un pulsador durante al menos 0.1seg
   do{
      Delayms( 10);
      if ( (c = readK())){
         if ( c>r)       // Si mas de un pulsador
            r = c;      // Generamos un nuevo codigo
         i++;
      } else
         i=0;
   } while ( i<10);
   // 2. Esperamos a que se suelte el pulsador durante al menos .1 sec
   i =0;
   do {
      Delayms( 10);
      if ( (c = readK())){
         if (c>r)        // Si mas de un pulsador
            r = c;      // Generamos un nuevo codigo
         i=0;
         j++;           // Contamos
      } else
         i++;
   } while ( i<10);
   // 3. Chequeamos si un boton ha sido pusado durante mas de 500ms y añadimos
         // si es necesario una bandera que nos indicara de ello.
   if ( j>50)
      r+=0x80;           // Añadimos una bandera en el bit 7 del codigo
   // 4. Regreso del codigo
   return r;
} // getK

main( void) {
   char s[16];
   int b;

   initLCD();         // Inicializamos el LCD
   putsLCD( "Press any button\n");

   // Bucle principal
   while( 1){
      b = getK();  //el mensaje no se muestra hasta que no soltamos todos los pulsadores
                   //si se muestran dos bits, el primero siempre sera 8 e indica que se
                   //ha pulsado el boton durante mas de 500ms.
      sprintf( s, "Code = %X", b);
      clrLCD();
      putsLCD( s);
   } // bucle principal
} // main
```

A continuación mostramos una foto del programa *buttons.c* siendo ejecutado en la Explorer16. La primera imagen se corresponde al haber presionado el pulsador situado más a la izquierda (S3) y la segunda al pulsar los dos botones que están más a la derecha (S5 y S4) simultáneamente.

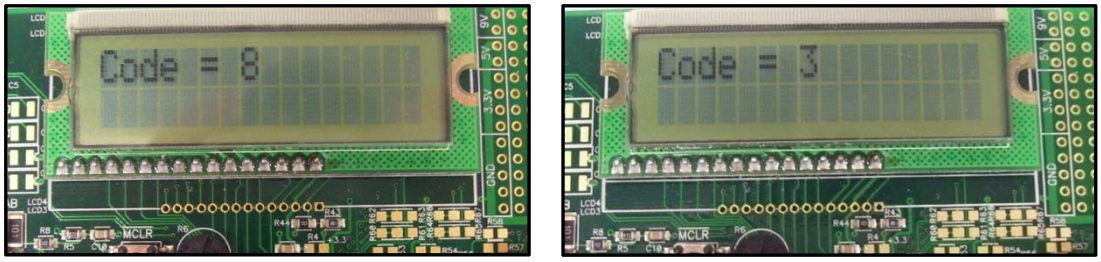


Figura 5.35: Programa "*buttons.c*" ejecutado en la Explorer16.

5.3.6. EJEMPLO 6: ENTRADAS ANALÓGICAS

Como hemos comprobado en el primer apartado de este capítulo, el sistema de desarrollo Explorer16 dispone de un potenciómetro y de un sensor de temperatura (TC1047A). Vamos por tanto a usar la capacidad que ofrece el PIC32 de convertir la información analógica a digital a través de un modulo ADC de 10 bits para poder usar los elementos anteriores.

El potenciómetro (R6) está conectado a través de una resistencia (R12) al canal analógico AN5. Por lo que lo primero que tendremos que realizar es configurar el modulo ADC accediendo a una serie de registros.

Registros a configurar inicialmente:

- AD1PCFG, pasaremos una máscara para seleccionar los canales de entradas analógicos, 0 serán entradas analógicas y 1 entradas digitales.
- AD1CON1: fijaremos la conversión para que empiece de manera automática (existe la opción manual también) y el formato de salida será un valor entero sin signo.
- AD1CSSL, lo pondremos a 0 ya que no usaremos la función de escanear, pues solo usamos una entrada.
- AD1CON2: seleccionaremos el uso de MUXA y conectaremos las entradas de referencia del ADC a los pines de las entradas analógicas AVdd y AVss.
- AD1CON3: seleccionaremos el reloj y el divisor.
- Activaremos ADON en el registro AD1CON1, para activar el modulo ADC y que esté este preparado para su uso.

La conversión Analógico-Digital es un proceso con dos etapas: Primero obtenemos una muestra de la señal de entrada y entonces podemos desconectar la entrada y realizar la conversión de la muestra de voltaje al valor digital. Estas dos etapas son controladas por dos bits del registro AD1CON1: SAMP y DONE. De tal forma que el tiempo de las dos etapas es importante para proporcionar una precisión adecuada de la medida. Este tiempo que hay que dejar se puede realizar de forma manual o automática. No obstante, nosotros la haremos de forma automática ya que de este modo no será necesario utilizar un retraso y el código necesario para la lectura de la entrada será más sencillo. Por tanto el código para inicializar el modulo ADC es el siguiente:

```
void initADC (int amask){
AD1PCFG=amask; //seleccionamos los pins de entrada
AD1CON1=0x00E0;   //secuencia de conversion automatica
AD1CSSL=0;   //no se requiere sccaning
AD1CON2=0;    //uso del MUXA, Avss/Avdd como referencia Vref+/-
AD1CON3=0x1F3F; //Tsamp=32*Tad, el maximo
AD1CON1bits.ADON=1; //activamos el ADC
}//intADC
```

Por otra parte para comenzar la lectura y conversión hay que acceder a los siguientes registros:

- AD1CHS, selecciona el canal de entrada para el MUXA.
- Cuando el bit SAMP del registro AD1CON1 este a 1, comienza la fase de toma de las muestras y seguidamente la conversión.
- El bit DONE se pondrá a 1 en el registro AD1CON1 tan pronto como la secuencia termine y el resultado esté listo.
- Leyendo el registro AD1BUF0 obtendremos el resultado de la conversión.

De tal forma que el código en C para la lectura de una entrada analógica es:

```
int readADC( int ch){
AD1CHSbits.CH0SA=ch;          // Seleccionamos la entrada analogica
AD1CON1bits.SAMP=1;           // Empezamos a tomar muestras.
while(!AD1CON1bits.DONE);     // Esperamos a que termine la conversion
return ADC1BUF0;              // Leemos el resultado.
}//readADC
```

Con ambas funciones anteriores nos creamos una librería para el modulo ADC. Para probarlas nos creamos el siguiente programa, el cual utiliza también las librerías para manejar el modulo LCD. Este programa lo que muestra en el display es una barra la cual muestra la resistencia del potenciómetro en tanto por cierto, es decir, si la resistencia es máxima (10KΩ) 100% y si la resistencia es mínima (0KΩ) 0%. Para ello se han usado las funciones creadas en el programa *ProgressBar.c* (visto en el apartado

anterior) para dotar de más precisión, recordemos que en este programa nos definíamos un carácter nuevo. Además en la segunda línea del display se muestra el valor de la resistencia. El código del programa es el siguiente:

```
main(){
int i, a, p,value;
char s[8],val[4],Aux[1];
//Inicializaciones
initADC( AINPUTS); //inicializacion del ADC
initLCD();

//Bucle principal
while(1){
a =readADC(POT); //sleccionamos la entrada POT y convertimos
//reducimos el resultado de 10bits a un valor de 7 bits (0..127)
//(dividimos por 8 o movemos a derecha 3 veces)
value=a/10;
a>>=3;
clrLCD();
p=((a*100)/127);
sprintf(s, "%2d", p);
putsLCD(s); putLCD('%');
//dibujamos una barra en el display proporcional a la posicion
// del potenciometro
drawProgressBar(a, 127,HLCD-3);
sprintf(Aux, "%d.%d", value);
if(value<10){// Valores menores que 1 Kh
                        Aux[1]=Aux[0];
                        Aux[0]='0';
                }
val[0]=Aux[0];
val[1]='.';
val[2]=Aux[1];
val[3]='K';
val[4]='h';
putsLCD("\nPoten:");
putsLCD(val);

//Parada para evitar parpadeos
Delayms(200);
}//bucle principal
}//main
```

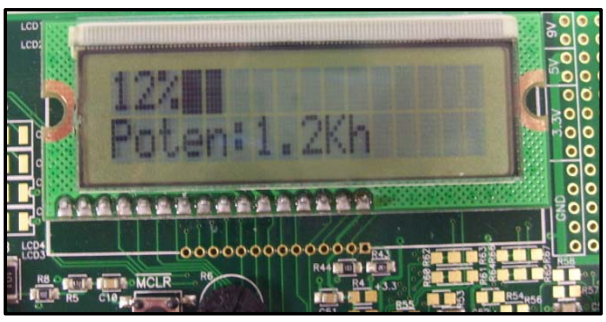


Figura 5.36: Programa "*Pot.c*" ejecutado en la Explorer16.

Como segundo programa utilizando el potenciómetro probamos el ejemplo de la referencia [10], el cual consiste en utilizar el potenciómetro para simular el juego Pac-Man en el LCD, moviendo este último girando el potenciómetro de izquierda a derecha. El juego consiste en mover la "boca" hasta llegar a la comida, simulada por el carácter "*" (asterisco). Tan pronto como lleguemos a esta posición aparecerá otra en una nueva posición para lo cual se usa la función *rand()* definida en la librería *stdlib.h*.

Además se ha modificado este programa ejemplo añadiendo un contador con las veces que "comemos", mostrándolo en la segunda línea del display. También se ha añadido la posibilidad de que podamos comenzar a que cuente de nuevo las veces que hemos conseguido alcanzar el asterisco, para lo cual presionaremos el pulsador S4 (el situado más a la izquierda). A través del LCD se nos mostrara el número total de veces que hemos "comido" e inmediatamente comenzará de nuevo el contador. A continuación mostramos el código del programa así como una imagen del mismo en la figura 5.37.

```
main (){
int a, r, p, n,cont=0;
char Conta[2];
// 1.- Inicializaciones
initADC( AINPUTS); //inicializacion del ADC
initLCD();
// 2.-leemos el primer valor del potenciometro y usamos su posicion para crear un valor aleatorio
srand(readADC(POT));
// 3.- inicializamos PAC
p='<';
// 4.- Generamos la posicion del bit de comida
r=rand()%16;
while(1){ //bucle principal
//5.- Seleccionamos como entrada POT y convertimos
a=readADC( POT);
// 6.- Reducimos el resultado de 10 a 4 bits (0...15)
a>>=6;
// 7.- Giramos a PAC en la direccion del movimiento
if (a<n)//movemos a la izquierda
        p='>';
if (a>n)//movemos a la derecha
        p='<';
// 8.- Cuando Pac coma generamos mas comida
while(a==r){
        r=rand()%16;
        cont++;
}
// 9.- Actualizamos el display
clrLCD();
setLCDC( a);
putLCD( p);
setLCDC( r);
putLCD( '*');
putsLCD("\nNum:");
sprintf(Conta, "%d", cont);
putsLCD(Conta);
putsLCD(" Press S4");
if ( !_RD13){
        clrLCD();
        putsLCD("Comenzamos de 0");
```

```
            putsLCD("\nAnterior: ");
            putsLCD(Conta);
            cont=0;
            Delayms( 5000);
}
Delayms( 200); //limite de velocidad del juego
n=a;
}//bucle principal
}//main
```

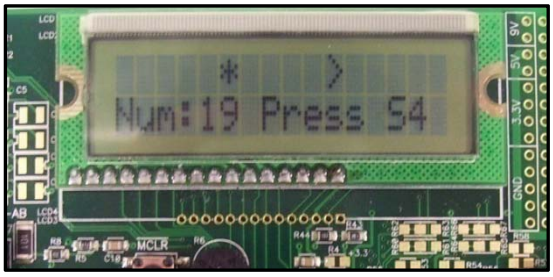


Figura 5.37: Programa "*POT-MAN.c"* ejecutado en la Explorer16.

Por otra parte, como ya comentamos, la placa dispone de un sensor de temperatura con salida analógica (TC1074A) conectado a una de los canales del controlador A/D, concretamente al canal AN4. Para utilizar la librería anterior para este nuevo sensor, lo único que tendremos que modificar de los programas anteriores será la entrada, es decir, en lugar de usar la entrada cinco correspondiente al potenciómetro, la entrada cuatro (TSENS=4). A continuación mostramos una imagen del programa (el código del mismo no se muestra por ser muy parecido al anterior, disponible en el Cd adjunto) del juego del Pac-Man, aplicado al sensor de temperatura. Al igual que en el anterior mostramos en la segunda línea del display la temperatura del sensor así como el número de veces que hemos alcanzado la comida. Así mismo si pulsamos S4, comienza la cuenta desde 0.

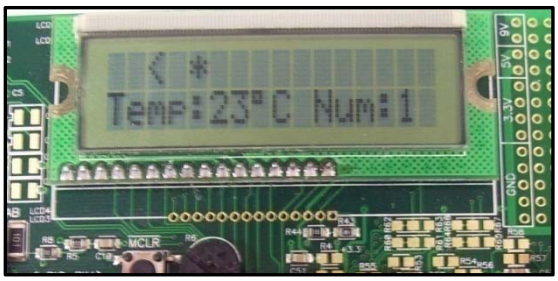


Figura 5.38: Programa "*Temperatura.c"* ejecutado en la Explorer16.

6. PANTALLA TÁCTIL: HARDWARE Y SOFTWARE

CAPÍTULO 6. PANTALLA TÁCTIL: HARDWARE Y SOFTWARE.

6.1. INTRODUCCIÓN

Este capítulo se basa en el estudio Hardware de la pantalla táctil presente en la placa de expansión “Graphics PICtail Plus Daughter Board (versión 2)” la cual se puede conectar al sistema de desarrollo Explorer16 a través del conector PICtail plus. Se trata de una GLCD táctil, en concreto el modelo TFT-G240320UTSW-92W-TP-E con una resolución de 320x240 píxeles y 262k colores.

Además se estudiará el funcionamiento de la librería “Microchip Graphic Library versión 1.6.” la cual nos va a permitir desarrollar programas para el control de la pantalla táctil basándonos en las funciones implementadas en esta librería. Aunque esta librería fue en principio diseñada para los microntroladores PIC24F/24H y dsPIC33F, es posible utilizarla si se emplea un PIC32 sobre la Explorer16.

6.1.1. IMPORTANCIA DE LAS PANTALLAS TÁCILES EN APLICACIONES EMBEBIDAS

Los displays gráficos se están volviendo cada vez más populares ya que se está incrementando el rango de las aplicaciones embebidas tales como displays de navegación, unidades handheld (portables como PDAS), domotica, aplicaciones médicas, etc. Además hay varios beneficios asociados al uso de funciones gráficas para distintas aplicaciones. La más notable es que proporciona una visualización mucho más rica para el usuario y una información más precisa y detallada gracias a las imágenes. Además, el precio cada vez más bajo de las pantallas gráficas así como de la tecnología TFT o STN está haciendo que cada vez se usen más en aplicaciones gráficas embebidas. A continuación mostramos varias imágenes donde se usan pantallas gráficas táctiles.

Figura 6.1: Ejemplos de Aplicaciones con pantallas táctiles.

6.2. ARQUITECTURA HARDWARE

6.2.1. ARQUITECTURA DE LA PANTALLA TÁCTIL

Características generales de la placa de evaluación “Graphics PICtail Plus Daughter Board (versión 2)”:

1.- Modulo LCD QVGA con una resolución de 320 x 240 pixeles.

2.- Controlador gráfico, LG electronics LGDP453.

3.- Memoria flash para almacenamiento de datos adicionales de 4Mbit (512Kx8).

4.- Zumbador (sound buzzer).

Figura 6.2: Graphics PICtail Plus Daughter Board v2.

Además, la tarjeta dispone de una serie de jumpers para configurar la placa. Por ejemplo, el jumper JP3 controla el sound buzzer. Cuando colocamos el jumper el pitido estará habilitado y cuando lo quitamos se deshabilitará. El resto de jumpers con sus posibles configuraciones así como las distintas conexiones de la placa se puede consultar en la siguiente referencia [15].

En la siguiente figura podemos ver el diagrama de bloques del sistema gráfico de Microchip. El sistema consiste en un microcontrolador, un controlador LCD y finalmente el cristal LCD. El controlador LCD incluye la lógica de control digital, un acelerador grafico (opcional), un buffer para las imágenes (RAM) y el gate driver. El microcontrolador normalmente, crea, manipula y dibuja elementos gráficos como botones, menús o imágenes. El acelerador gráfico proporciona una aceleración hardware para algunos elementos gráficos y otras funciones gráficas. Además, el modulo LCD requiere de un buffer de imágenes para almacenar las estructura de estas, de tal forma que la lógica de control digital sirve como un controlador para el buffer de imágenes y para el acelerador gráfico. Finalmente el gate driver convierte la señal digital a analógica y hace funcionar al cristal LCD.

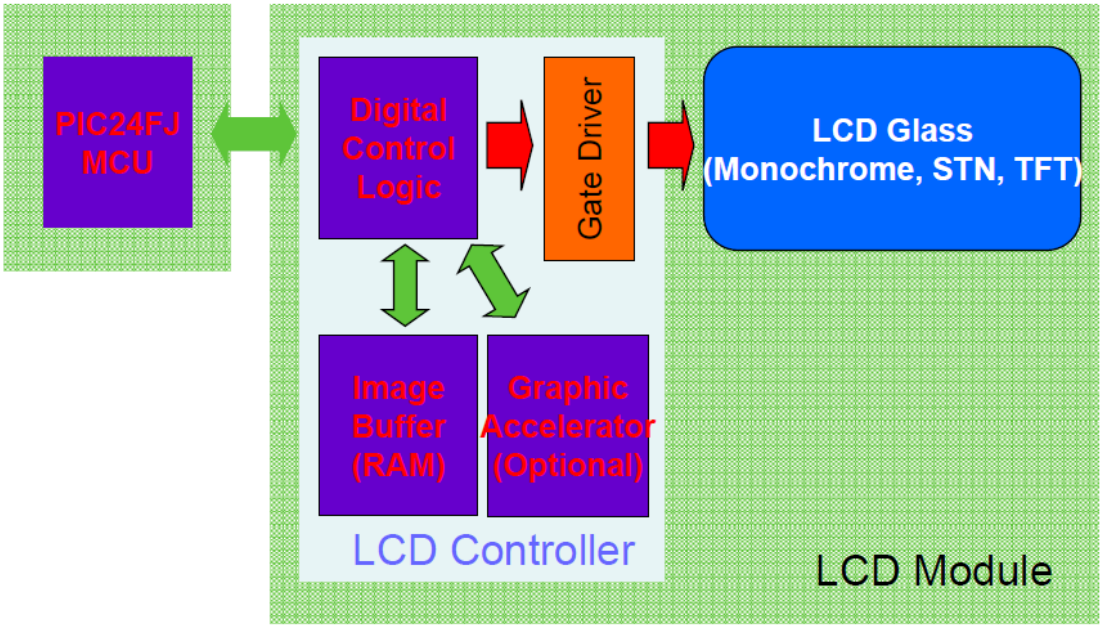


Figura 6.3: Diagrama de Bloques del modulo LCD.

El panel LCD puede venir con el controlador LCD o sin él, en nuestro caso viene con el controlador LCD embebido, diagrama de bloques anterior. Nuestro controlador es concretamente el LGDP453, para más información sobre este consúltese la siguiente referencia [16], disponible en la carpeta de documentos en el Cd adjunto.

6.2.2. FUNCIONAMIENTO DE UNA PANTALLA TÁCTIL

Para ver de qué manera funciona la pantalla táctil primero vamos a ver cómo actúa cada pixel y después veremos cómo funciona todo en conjunto. Sin embargo, hay que distinguir entre dos tipos de pantallas táctiles, las reflectantes o las transmisivas.

Reflectantes:

El LCD, como su nombre indica, está hecho de cristales líquidos. Los cristales líquidos son el corazón del display y la operación del display depende de la manipulación de la luz polarizada. La luz ambiente pasa a través del primer polarizador (front polarizer), el cual elimina los rayos de luz excepto los que son polarizados verticalmente. De tal forma, que la luz atraviesa los cristales líquidos y gira 90 grados. Entonces la luz atraviesa el segundo polarizador y se refleja y continúa el mismo proceso hasta el primer polarizador. Hay que tener en cuenta que si la luz sale del primer polarizador, entonces no veríamos ninguna imagen, por lo que hay que evitar dañar el polarizador haciendo un correcto uso de la pantalla táctil. A continuación mostramos la siguiente imagen que ilustra lo explicado.

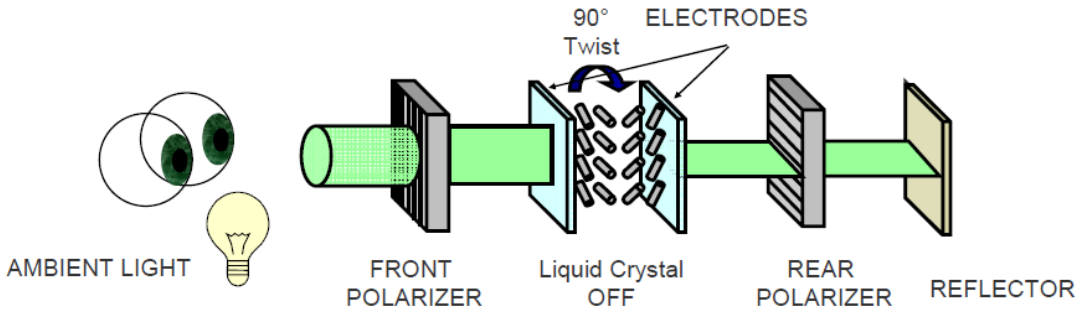


Figura 6.4: Esquema de funcionamiento de un pixel en una pantalla reflectiva.

No obstante, la orientación del cristal líquido puede cambiar cambiando la carga aplicada en él. La carga se aplica a través de unos electrodos en forma de cuadrado, como podemos ver en la figura anterior. Por tanto, para cada pixel del LCD gráfico, habrá un cuadrado.

Cuando cambiamos la carga, el resultado equivale a que los cristales líquidos no giran (ver figura 6.5). Por lo que cuando la luz alcanza el segundo polarizador la luz se "bloquea" debido a la polarización vertical de esta. De tal forma que la luz no se reflejará y este pixel estará oscuro. Este tipo de LCD requiere de luz ambiente para que funcione, por lo que no funcionará en una habitación oscura.

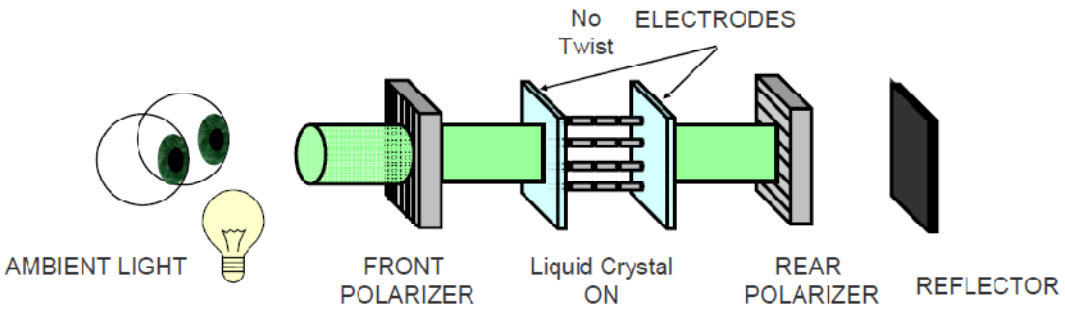


Figura 6.5: Esquema de funcionamiento de un pixel en una pantalla reflectiva II.

Transmisivas:

El otro tipo de tecnología de LCD no requiere luz ambiente. En lugar de la luz ambiente dispone de una luz dentro del display, la cual está situada en la parte posterior del LCD, "backlight". La forma de operación es contraria a la tecnología anterior. Si la luz gira 90 grados (TN, Twisted Nematic), esta se bloquea en el polarizador delantero y no veremos nada, tal y como podemos ver en la siguiente imagen.

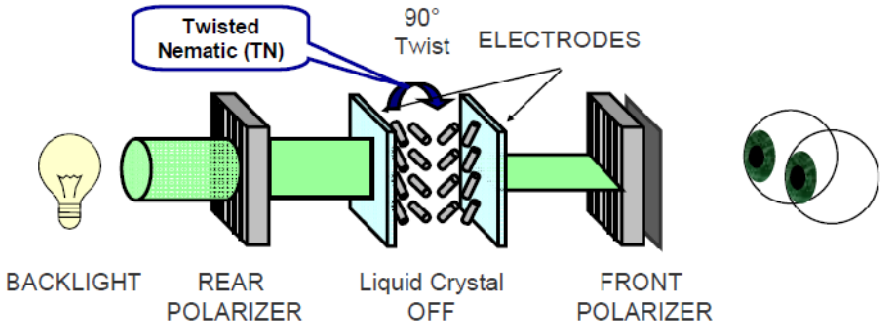


Figura 6.6: Esquema de funcionamiento de un pixel en una pantalla transmisiva.

Por tanto si cambiamos la carga aplicada a los cristales líquidos se eliminará el giro y entonces la luz pasará a través del polarizador delantero y veremos la imagen, es como decir que el pixel estará en ON (ver figura 6.7).

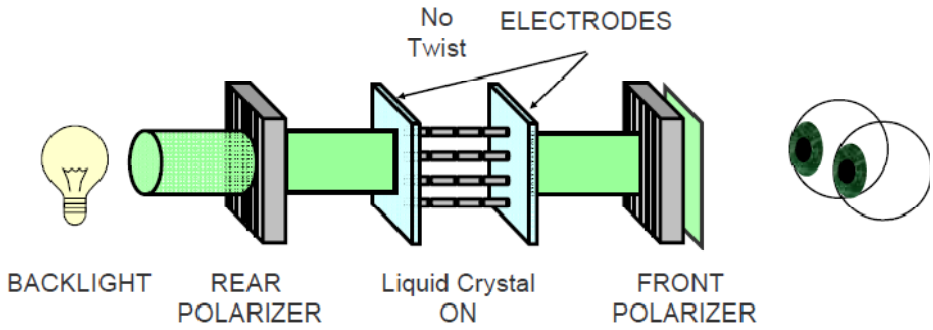


Figura 6.7: Esquema de funcionamiento de un pixel en una pantalla transmisiva II.

La ventaja de esta tecnología respecto a la anterior, es que este tipo de display se puede usar bajo condiciones de poca luz, sin embargo, no funcionará bien si son expuestas directamente al sol. Nuestra pantalla táctil usa esta última tecnología, cuyo backlight son 4 Leds conectados en paralelo [17].

Una vez hemos visto el modo de operación de un pixel, vamos a ver qué tipo de tecnologías se usan para controlar y conectar los pixeles de una LCD gráfica.

La solución más sencilla sería utilizar una matriz de puntos habilitando los pixeles de las columnas y filas que queramos, Passive Matrix. Todos los pixeles están conectados a través de cables formando una matriz. Sin embargo cuando queremos cambiar el estado de un pixel, esta acción no se realiza de forma inmediata, sino que se comporta como un condensador. Por tanto hay que esperar un tiempo para que el pixel pase de un estado a otro.

La ventaja de esta tecnología es que es barata, sin embargo requiere de un tiempo de respuesta alto para que desaparezca el efecto "condensador" al cambiar un pixel de estado. Esto implica, que no son apropiadas para usarse en el movimiento de objetos pues resultaría ser muy lento.

Nuestra pantalla no se basa en la tecnología anterior sino que se basa en la tecnología TFT (Thin-Film Transistor, Display Active Matrix). La ventaja principal respecto a la anterior es que el tiempo de respuesta es mucho más rápido. Esta tecnología se está haciendo cada vez más popular ya que su precio está bajando rápidamente.

6.2.3. CARACTERÍSTICAS MÁS IMPORTANTES DEL LCD GRÁFICO

En la siguiente tabla enumeramos las principales características de la pantalla táctil.

Característica	Tipo
Tipo de LCD	TFT transmisiva
Resolución	240(RGB)x320
Driver	LGDP4531
Colors	262k
Tipo de Backlight	LED
Tipo de interfaz	Puerto paralelo/ Serie
Tiempo de respuesta máximo	55.2ms
Voltaje de entrada	2.8/1.8 V
Consumo	304 mw

Tabla 6.1: Características principales del TFT-G240320UTSW-92W-TP-E.

6.2.4. CONEXIONADO DE LA PANTALLA TÁCTIL

La “Graphics PICtail Plus Daughter Board” ha sido diseñada para utilizarse como herramienta de desarrollo, es decir, para incorporarse a otra placa en la cual se disponga del microcontrolador que la va hacer funcionar. Existen dos opciones, la primera opción es conectarla a la tarjeta Explorer16 a través del bus PICtail Plus y la segunda opción sería conectar la pantalla táctil a la “I/O Expansion Board” [18], también mediante el uso del bus PICtail Plus. Sin embargo, para familiarizarnos con la librería gráfica y con la pantalla táctil resulta mucho más sencillo conectarla a la Explorer16, pues los programas ejemplo disponibles han sido desarrollados para que la pantalla táctil se conecte a esta. La segunda opción la usaremos para desarrollar la aplicación final pues mediante la I/O Expansion board, es muy sencillo acceder a las distintos pines del microcontrolador.

Por tanto, la interconexión del equipo completo para el funcionamiento de todos los dispositivos es la que se muestra en la figura 6.8. Utilizamos el microcontrolador PIC32 que proporciona la tarjeta PIC32 Starter Kit siendo necesario usar el adaptador 100L PIM Adaptor para conectarlo sobre la tarjeta Explorer16 (como ya hicimos en el capitulo anterior) y la tarjeta Explorer16 se conecta a la pantalla táctil mediante el bus PICtail Plus. Por otro lado, como también hemos visto, para la grabación y depuración de los programas editados desde el MPLAB v8.33 usaremos el sistema MPLAB ICD3 in Circuit Debugger.

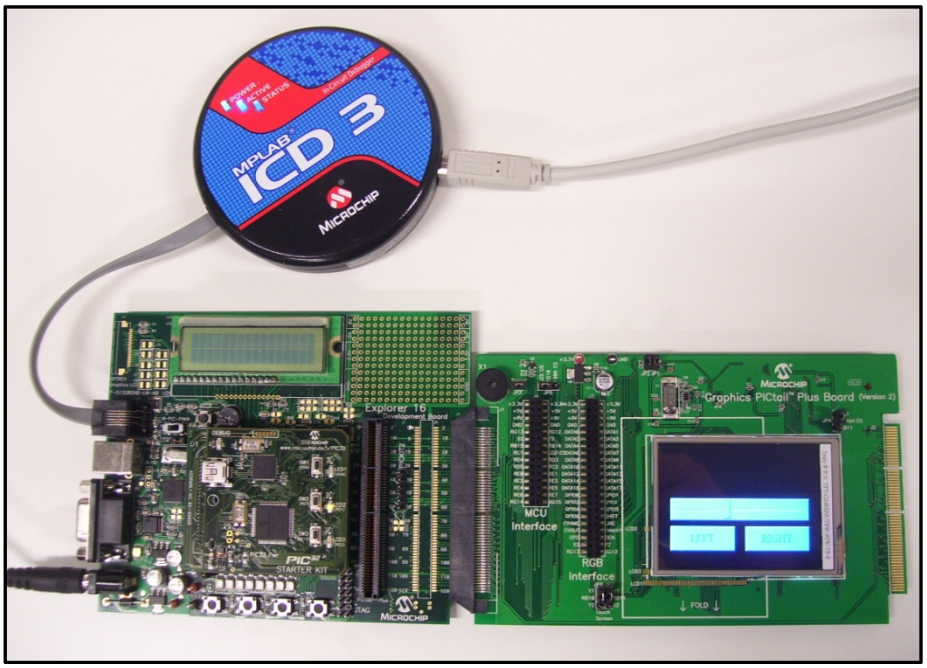


Figura 6.8: Equipo completo en funcionamiento, sistema de desarrollo Explorer16, MPLAB ICD3 y Graphics PICtail Plus Daughter Board.

6.3. MICROCHIP GRAPHIC LIBRARY

En este apartado vamos a estudiar el funcionamiento de la librería “Microchip Graphic Library versión 1.6.” la cual nos va a permitir desarrollar programas para el control de la pantalla táctil basándonos en las funciones implementadas en esta librería.

6.3.1. ESTRUCTURA DE LA LIBRERÍA GRÁFICA

La librería gráfica se puede dividir en 4 capas específicas:

- Un archivo tipo c implementa las funciones más básicas tales como PutPixel, GetPixel, Set Color, necesarias para inicializar y usar el display del dispositivo. Esto está controlado a través del controlador del display. Además el acelerador gráfico proporcionado por el controlador del display va a ejecutar estas funciones de una forma más eficiente. Este acelerador se habilita escribiendo algunos registros especiales en el controlador. Todo esto compone la capa del driver del dispositivo (Device Driver Layer).
- Por encima de esta capa se encuentra la capa de Gráficos primitivos (Graphics primitive layer), la cual llama a las funciones de la capa anterior para construir formas primitivas. Por ejemplo, para la formación de una línea se llamará repetidamente a la función PutPixel de la capa del driver del dispositivo.
- La siguiente capa es la de objetos gráficos (Graphics Objects layer), en esta capa se implementan objetos avanzados como widgets. En esta capa manejaremos el control, la destrucción y creación de estos widgets. De tal forma que esta capa esta realizada para que sea fácilmente integrable con distintas interfaces, siguiente capa.
- Por último tenemos la capa de aplicación (Application layer), la cual contiene los módulos que inicializan y controlan las interfaces del usuario, pantalla táctil.
- Además, a través de la interfaz de mensaje, la aplicación se puede comunicar con los distintos objetos de una forma relativamente sencilla. De tal forma que tendríamos integrado totalmente los objetos en las funciones de la aplicación.

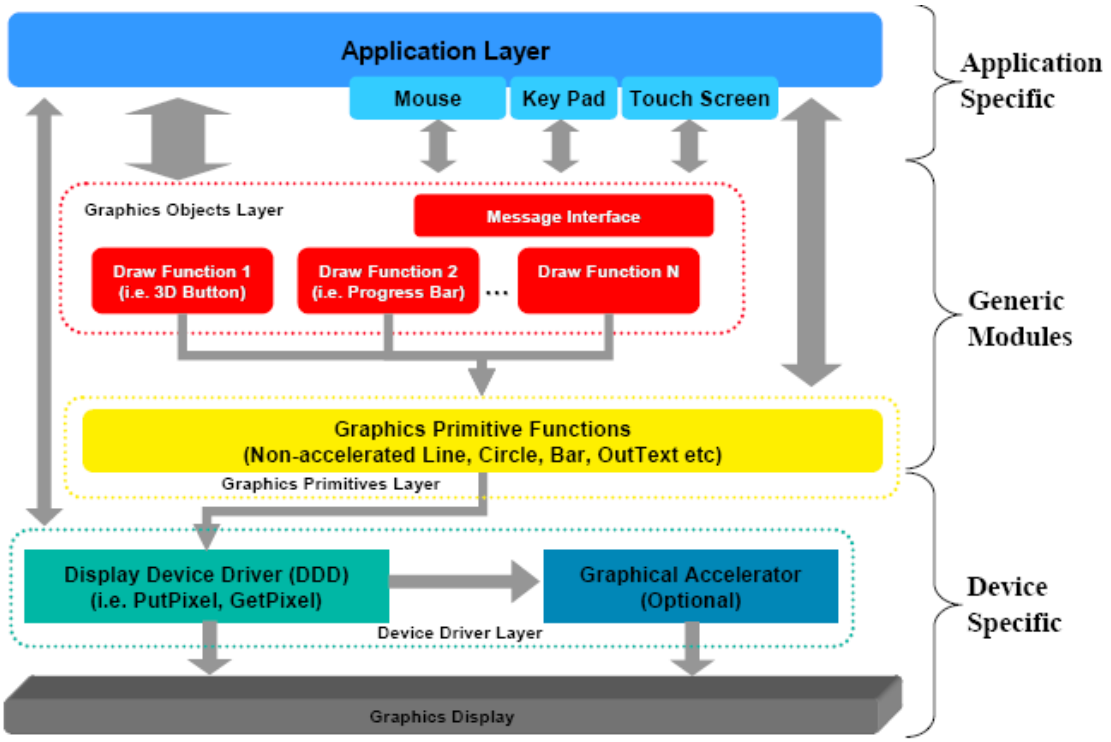


Figura 6.9: Estructura de la librería gráfica v1.6 de microchip.

6.3.2. CARACTERÍSTICAS DE LOS WIDGETS (OBJETOS)

Los widgets (objetos) soportados por la librería son, Button, Chart, Check Box, Dial, Edit Box, Group Box, List Box, Meter, Picture, Progress Bar, Radio Button, Slider, Static Text, Window, Text Entry. Las principales características de estos objetos son:

- Usando la API (Application Programming Interface) de un objeto se puede crear, controlar y eliminar el mismo.
- El comportamiento de los objetos viene determinado por su estado actual y por la acción del usuario en dicho objeto.
- Los objetos se pueden encontrar en un estado activo o inactivo
- El control de los objetos se simplifica mediante una serie de listas vinculadas a estos, de manera que cada objeto nuevo se añade a esta lista. Gracias a esta lista, la librería dibuja de forma automática los objetos en el display.
- Cada objeto se puede implementar usando una combinación de estilos diferente, de tal forma que si no se le asigna una en concreto, se usará una combinación prefijada por defecto. Sin embargo, se puede crear un nuevo estilo y asignarlo al objeto en el momento de la creación de este o en

cualquier otro momento en el que el objeto se encuentre en la memoria. La combinación del estilo determina la tabla fuente y la combinación de colores usada para dibujar el objeto en cuestión.

Además, gracias a que la librería integra el control básico de los objetos basado en las acciones del usuario, se derivan una serie de beneficios debido a esta implementación:

- Desde el punto de vista de la aplicación, simplifica bastante el control de los objetos.
- La capa de la interfaz de mensaje, simplifica la integración de las entradas de nuestro dispositivo.
- Soporta una gran variedad de entradas, ratón, pantalla táctil, etc.
- Proporciona una integración completa de las entradas del dispositivo con los objetos de la pantalla.

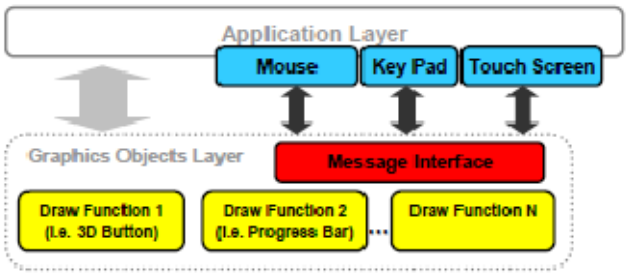


Figura 6.10: Librería gráfica, control de los objetos a través de la interfaz de mensaje.

6.3.3. FUNCIONAMIENTO DE LA LIBRERÍA

Para ver de qué forma funciona la librería, vamos a verlo a través de un ejemplo, de qué forma podemos representar un botón, como lo podemos configurar y a que funciones hace falta llamar para dibujar y actualizar la pantalla táctil.

La primera función a la cual tenemos que llamar es *InitGraph()*; para resetear el controlador del dispositivo del display, mover el cursor a la posición (0,0) e inicializar el dispositivo en negro.

Después llamamos a la función *GOLCrateScheme()*, para definir la combinación de estilos a usar por los objetos. Por defecto, todos los objetos usan una estructura que contiene una combinación de estilos globales, definiendo la tabla fuente y la combinación de colores usada. Veamos la estructura aplicada a un botón:

```
typedef struct {
WORD EmbossDkColor;
WORD EmbossLtColor;
WORD TextColor0;
WORD TextColor1;
WORD TextColorDisabled;
WORD Color0;
WORD Color1;
WORD ColorDisabled;
WORD CommonBkColor;
char *pFont;
} GOL_SCHEME;
```

Como podemos ver en la estructura GOL_SCHEME anterior, todas las variables tipo word seleccionan un color el cual lo podemos elegir nosotros, mientras que pFont es un puntero que selecciona el tipo de fuente de texto a usar. Por otra parte, para emular el efecto de 3D al pulsar el botón, intercambiamos el relieve de este de un color oscuro a otro claro. A continuación podemos ver que representa cada variable:

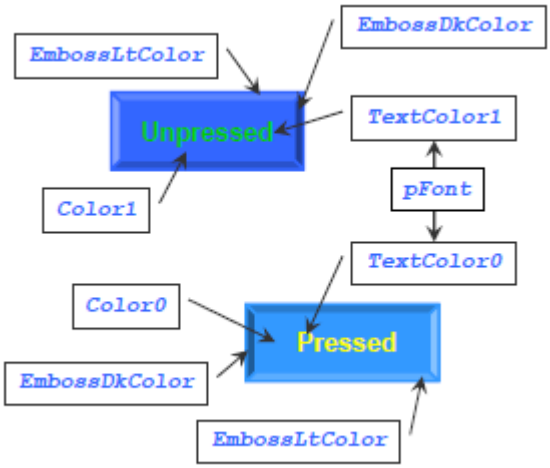


Figura 6.11: Librería gráfica, estructura GOL_SCHEME aplicada a un botón.

También, las dos funciones anteriores de pueden ejecutar usando una única instrucción, llamando a la función *GOL_Init()*.

El siguiente paso es crear los objetos. Para ello cada objeto tiene su función específica en la que se definen los parámetros que hay que asignarles para crear dicho objeto. En el ejemplo del botón, la función a la cual hay que llamar es *BtnCreate().*

```
BUTTON *BtnCreate( WORD ID, SHORT left, SHORT top, SHORT right, SHORT bottom, SHORT radius, WORD state, void *pBitmap, XCHAR *pText, GOL_SCHEME *pScheme)
```

El ID es un número definido por el usuario destinado a identificar dicho objeto. Left, top, right, bottom, son los parámetros usados para definir las dimensiones del objeto, mientras que Radius define el redondeado de la esquinas del botón (ver figura 6.12.)

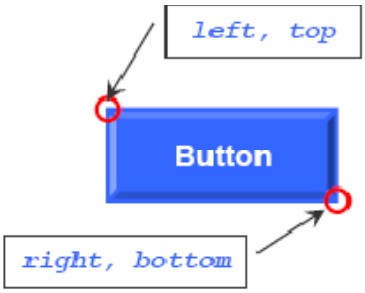


Figura 6.12: Librería gráfica, parámetros para definir las dimensiones de un botón.

Por otra parte, State define el estado actual del objeto. Para cada tipo de objeto se definen una lista de estados posibles, sin embargo, existen una serie de estados comunes a todos los objetos los cuales son:

```
#define BTN_FOCUSED  0x0001        // Bit for focus state.
#define BTN_DISABLED   0x0002      // Bit for disabled state.
#define BTN_DRAW_FOCUS  0x2000     // Bit to indicate focus must be redrawn.
#define BTN_DRAW       0x4000      // Bit to indicate button must be redrawn.
#define BTN_HIDE       0x8000       // Bit to indicate button must be removed from screen.
```

Además el botón tiene una serie de estados propios los cuales son:

```
#define BTN_PRESSED    0x0004      // Bit for press state.
#define BTN_TOGGLE          0x0008 // Bit to indicate button will have a toggle behavior.
#define BTN_TEXTRIGHT  0x0010      // Bit to indicate text is right aligned.
#define BTN_TEXTLEFT   0x0020      // Bit to indicate text is left aligned.
#define BTN_TEXTBOTTOM  0x0040     // Bit to indicate text is top aligned.
#define BTN_TEXTTOP    0x0080      // Bit to indicate text is bottom aligned.
                                   // Note that if bits[7:4] are all zero text is centered.
```

Los estados BTN_DRAW y BTN_HIDE indican que hay que redibujar, actualizar los objetos en el display, para lo cual se basan en los otros estados, BTN_PRESSED, BTN_TOGGLE,etc.

Continuando con la función *BtnCreate(),* si no se usa ningún texto ni un bitmap, las variables pBitmap y pText hay que asignarlas a NULL. Por otra parte, el texto se puede alinear a la derecha, a la izquierda, arriba o abajo, dependiendo del estado seleccionado. Si no se selecciona ninguno de los anteriores, el texto se encontrará centrado en el botón. Por último pSchme, es un puntero que apunta a una variable que contiene la combinación del estilo elegida. Si no se selecciona ninguna se utilizará el estilo por defecto.

Una vez que ya hemos realizado las llamadas a las funciones para crear los objetos, hay que llamar a la función *GOLDraw()* para que los dibuje en el display. Esta función analiza sintácticamente la lista de objetos vinculados y chequea el estado actual de cada uno de los objetos. Si un objeto tiene un estado pendiente de dibujar, el

objeto será redibujado. Una vez que la función *GOLDraw()* haya actualizado el objeto, resetea el estado pendiente anterior.

El siguiente paso es obtener las entradas del usuario. Para ello hacemos uso de la interfaz de mensajes, la cual, cuando ocurre un evento en un dispositivo de entrada, envía a la librería la estructura de mensaje. Es decir, estos mensajes contienen la acción del usuario en los objetos de la pantalla. La estructura de la interfaz de mensajes es la siguiente:

```
typedef struct {
BYTE type;
BYTE event;
SHORT param1;
SHORT param2;
} GOL_MSG;
```

El campo type, define el tipo de dispositivo de entrada usado (en nuestro caso la pantalla táctil). Dependiendo del dispositivo usado param1 y param2 serán interpretados de una manera u otra. Para el caso de la pantalla táctil param1 y param2 son definidos como la coordenada x e y respectivamente. Hay que tener en cuenta que la coordenada (0,0) se sitúa en la parte superior izquierda de la pantalla. Por último, event determina la acción realizada por el usuario, las cuales están predefinidas. Para la pantalla táctil estas son:

```
EVENT_INVALID = 0       // An invalid event.
EVENT_MOVE              // A move event.
EVENT_PRESS             // A press event.
EVENT_RELEASE           // A release event.
```

Cuando un objeto recibe un mensaje, este evalúa si el mensaje es válido o no lo es. Cuando es válido responde con una de las acciones ID definidas. Las acciones ID, son una lista de posibles acciones que un objeto puede aceptar, de tal forma que cada objeto tiene su propia lista de acciones. Para el caso del botón, este tiene dos acciones predefinidas:

```
BTN_MSG_PRESSED,        // Button pressed action ID.
BTN_MSG_RELEASED,       // Button released action ID.
```

Sin embargo, si el mensaje es considerado como invalido, el objeto responderá con un MSG_INVALID. Un ejemplo de esta situación se produce cuando un objeto pasa a estar inactivo, de manera que cualquier mensaje que le llegue a este, el objeto responderá con MSG_INVALID.

De tal forma que cuando se toca la pantalla debe existir una función que procese los mensajes anteriores y que cheque la lista de objetos para determinar que objeto se verá afectado por el mensaje, esta función es GOLMsg(). El objeto que incluya la posición (x, y) cambiará su estado basado en el estado actual y en el evento. Sin embargo, este análisis de los mensajes se pueden realizar de una forma más rápida través de la función GOLMSGCallback(). Cada vez que un mensaje válido sea recibido por algún objeto se llamará a esta función. Además esta función nos va a permitir personalizar el comportamiento de los objetos en función de su estado y el evento ocurrido, tal y como veremos en el siguiente apartado (ejemplos).

Por tanto, una vez los mensajes han sido procesados, se vuelve a llamar otra vez a la función GOLDraw() para actualizar dichos objetos. De este modo se completa el ciclo de funcionamiento de la librería. A continuación mostramos un esquema del funcionamiento de la misma.

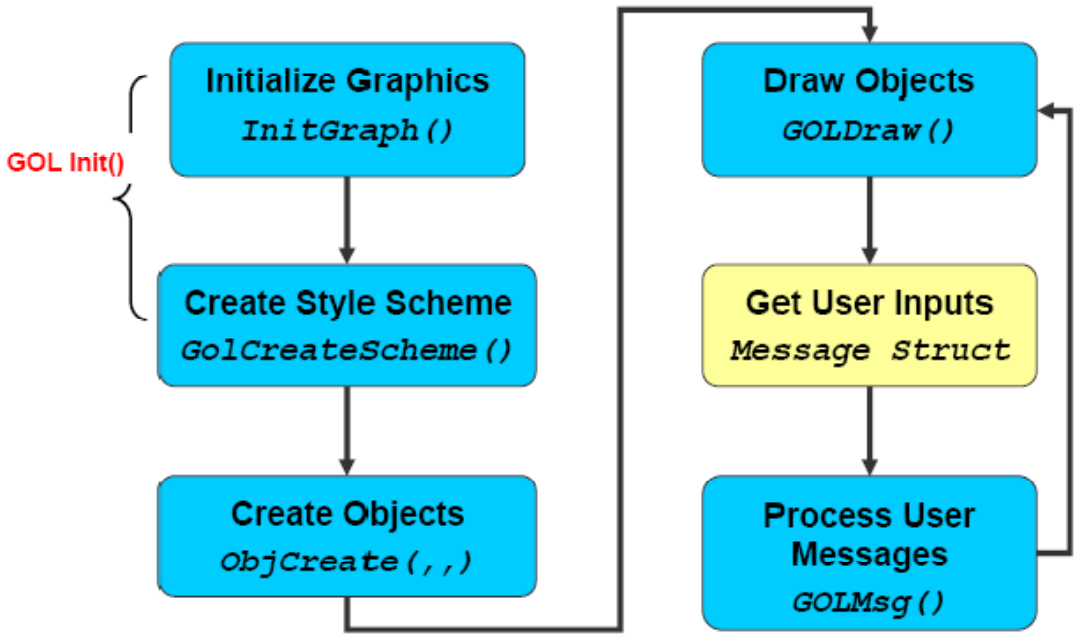


Figura 6.13: Diagrama de flujo básico del funcionamiento de la librería gráfica.

6.3.4. PROGRAMAS EJEMPLO

Los siguientes programas ejemplo han sido extraídos del documento AN1136 "How to use Widgets in Microchip graphics Library" [19], disponible en el Cd adjunto a este proyecto, así como de los programas ejemplos disponibles en la librería una vez que la instalamos en nuestro ordenador. Además los programas extraídos de la referencia anterior se pueden encontrar en el CD adjunto en la carpeta correspondiente a la pantalla táctil.

Hay que tener en cuenta que para el funcionamiento de estos programas es necesario copiar la carpeta que contenga el programa que queramos probar al directorio en el que hemos instalado la librería. Por defecto, tendremos que copiar la carpeta requerida a la siguiente dirección C/Microchip Solutions, para que pueda usar todos los objetos disponibles en la librería.

Carpeta: Graphics Primitives Layer Demo.

Con este primer programa vamos a ejecutar uno de los disponibles en la carpeta creada una vez instalada la librería, también disponible en el Cd adjunto. Este programa sólo va hacer uso de las funciones más básicas de la librería, sin usar objetos, es decir, va a hacer uso de funciones tales como dibujar líneas asignando distintos colores, dibujar círculos rellenándolos de distintos colores, etc. A continuación mostramos un trozo de código extraído del programa *MainDemo.c* con su correspondiente imagen al ser ejecutado en la pantalla táctil.

```
while(1){
   SetColor(WHITE);
   for(counter=0; counter<GetMaxX(); counter+=20){
      Line(counter,0,GetMaxX()-1-counter,GetMaxY()-1);
   }
   DelayMs(4000);
```

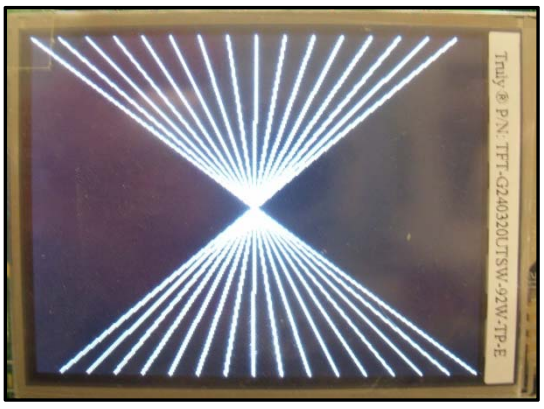


Figura 6.14: Programa *Graphics Primitives Layer Demo* ejecutado en la pantalla táctil.

La función usada en el programa anterior es:

- Void Line(short x1, short y1, short x2, short y2) cuyos parámetros son:
  - x1, Coordenada x del punto de comienzo de la línea.
  - y1, Coordenada y del punto de comienzo de la línea.
  - x2,Coordenada x del punto final de la línea
  - y2, Coordenada y del punto final del la línea.

El programa también utiliza otras funciones básicas como son Circle(), Fillcircle(), Arc(), Bbar(), entre otras.

Programa: *AN1136_v1.0.c*

El siguiente programa ejemplo muestra la interacción entre las acciones del usuario y el estado actual de un objeto. De qué forma podemos modificar el comportamiento de un objeto en función de otro.

El ejemplo consta de 3 objetos, 2 botones y un slider. Cada objeto se ha creado con su correspondiente función de creación del objeto indicando los parámetros necesarios. En los botones se ha escrito las palabras “Left” y “Right”. Una vez ya tenemos creado los 3 objetos vamos a ver de qué forma pueden interactuar entre ellos. Lo que realiza este programa es que dependiendo del botón pulsado por nosotros se mueva en pasos fijos el slider. Concretamente en la creación del slider fijamos el tamaño de la página en 5. Por tanto cada vez que presionemos un botón incrementaremos o reduciremos la posición del Slider en 5 unidades.

Para implementar esta funcionalidad hay que usar, tal y como hemos comentado en el apartado anterior, la función *GOLMsgCallback(),* cuya función se llama cada vez que la función *GOLMsg()* reciba un mensaje válido proveniente de un objeto. Los tres parámetros que necesita la función son:

- Raw GOL message (pMseg), Puntero a la estructura de mensaje con los datos obtenidos al pulsar en la pantalla táctil.
- Pointer to the Object (pObj), este puntero nos permite obtener toda la información sobre el objeto afectado y controlar su estado.
- Translated mesage (objMsg), es un número devuelto por la librería que nos muestra la clase de evento ocurrido por el objeto afectado. Específico para cada objeto, visto anteriormente.

A continuación mostramos el código del programa mediante el cual modificamos el comportamiento del Slider en función de los botones. Si presionamos el botón 1 reducimos la posición del slider mientras que presionando el botón 2 lo incrementamos.

```
//                      OBJECT'S IDs
#define ID_BTN1          10
#define ID_BTN2          11
#define ID_SLD1          20

//                    LOCAL PROTOTYPES
void CheckCalibration(void);       // check if calibration is needed

//                        MAIN
GOL_SCHEME  *altScheme;       // alternative style scheme

int main(void){
        GOL_MSG msg;    // GOL message structure to interact with GOL

/////////////////////////////////////////////////////////////////////////
// ADC Explorer 16 Development Board Errata (work around 2)
// RB15 should be output
        LATBbits.LATB15 = 0;
        TRISBbits.TRISB15 = 0;
/////////////////////////////////////////////////////////////////////////

#ifdef __PIC32MX__
   INTEnableSystemMultiVectoredInt();
   SYSTEMConfigPerformance(GetSystemClock());
#endif
   BeepInit();
   EEPROMInit();                                    // initialize EEPROM
   TouchInit();                                     // initialize touch screen
   GOLInit();                                       // initialize graphics library &
                                                    // create default style scheme for GOL
   // If S3 button on Explorer 16 board is pressed calibrate touch screen
   if(PORTDbits.RD6 == 0){
      TouchCalibration();
      TouchStoreCalibration();
   }
   // If it's a new board (EEPROM_VERSION byte is not programed) calibrate touch screen
   if(GRAPHICS_LIBRARY_VERSION != EEPROMReadWord(EEPROM_VERSION)){
      TouchCalibration();
      TouchStoreCalibration();
      EEPROMWriteWord(GRAPHICS_LIBRARY_VERSION,EEPROM_VERSION);
   }
   // Load touch screen calibration parameters from EEPROM
   TouchLoadCalibration();

   altScheme = GOLCreateScheme();                   // create alternative style scheme
   altScheme->TextColor0 = BLACK;
   altScheme->TextColor1 = BRIGHTBLUE;

BtnCreate(ID_BTN1,                          // object's ID
          20, 160, 150, 210,                // object's dimension
           0,                               // radius of the rounded edge
          BTN_DRAW,                         // draw the object after creation
          NULL,                             // no bitmap used
          "LEFT",                           // use this text
          altScheme);                       // use alternative style scheme

BtnCreate(ID_BTN2,
            170, 160, 300, 210,
            0,
            BTN_DRAW,
```

```
            NULL,
            "RIGHT",
            altScheme);

        SldCreate(ID_SLD1,              // object's ID
                20, 105, 300, 150,      // object's dimension
                SLD_DRAW,               // draw the object after creation
                100,                    // range
                5,                      // page
                50,                     // initial position
                NULL);                  // use default style scheme

        while(1){
if (GOLDraw()) {                        // Draw GOL object
        TouchGetMsg(&msg);              // Get message from touch screen
        GOLMsg(&msg);                   // Process message
      }
   }
}

WORD GOLMsgCallback(WORD objMsg, OBJ_HEADER* pObj, GOL_MSG* pMsg){
        WORD objectID;
        SLIDER *pSldObj;
        objectID = GetObjID(pObj);
        if (objectID == ID_BTN1) {
                if (objMsg == BTN_MSG_PRESSED) {            // check if button is pressed
                        pSldObj = (SLIDER*)GOLFindObject(ID_SLD1);  // find slider pointer
                        SldDecPos(pSldObj);                 // decrement the slider position
                        SetState(pSldObj, SLD_DRAW_THUMB);// redraw only the thumb
                }
        }
        if (objectID == ID_BTN2) {
                if (objMsg == BTN_MSG_PRESSED) {
                pSldObj = (SLIDER*)GOLFindObject(ID_SLD1);  // find slider pointer
                        SldIncPos(pSldObj);                 // increment the slider position
                        SetState(pSldObj, SLD_DRAW_THUMB);// redraw only the thumb
                }
        }
        return 1;
}
//Call Back functions must be defined and return a value of 1 even though they are not used.
WORD GOLDrawCallback(){
   return 1;
}
```

Como podemos observar en el main únicamente llamamos a dos funciones, una para obtener el mensaje cuando tocamos la pantalla táctil y otra para procesarlo. Estas llamarán a su vez a las funciones GOLMsgCallback(), para que realice el comportamiento deseado una vez hemos presionado el botón que queramos. A continuación mostramos una imagen con el programa siendo ejecutado en la pantalla táctil.

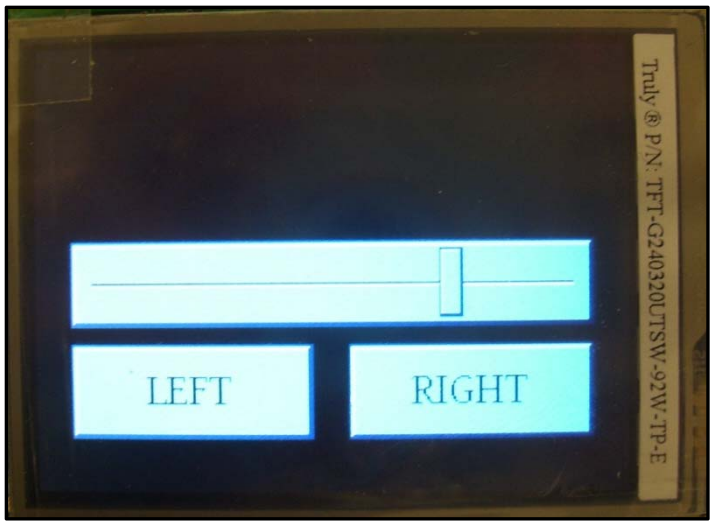


Figura 6.15: Programa "*AN1136_v1.0.c*" ejecutado en la pantalla táctil.

Programa: *AN1136_v2.0.c*

En el siguiente programa se ha realizado una modificación del programa anterior. Esta modificación consiste en que mientras que estamos presionando el botón izquierdo se muestra una imagen (bitmap), una flecha apuntando a la izquierda. Para que se pueda dibujar y no se dibuje encima del texto hay que alinear el texto a la izquierda y una vez soltado el botón que vuelva a su posición original. La misma modificación se realiza en el botón derecho pero al contrario. Para ello hay que modificar la función GOLMsgCallback() tal y como mostramos a continuación:

```
//                      IMAGES USED
extern const BITMAP_FLASH redRightArrow;
extern const BITMAP_FLASH redLeftArrow;

WORD GOLMsgCallback(WORD objMsg, OBJ_HEADER* pObj, GOL_MSG* pMsg){
        WORD objectID;
        SLIDER *pSldObj;

 objectID = GetObjID(pObj);

        if (objectID == ID_BTN1) {
                if (objMsg == BTN_MSG_PRESSED) {                // check if button is pressed
                        BtnSetBitmap(pObj, (void*)&redLeftArrow);    // set bitmap to show
                        SetState(pObj, BTN_TEXTRIGHT);          // move the text to the right

                        pSldObj = (SLIDER*)GOLFindObject(ID_SLD1);   // find slider pointer
                        SldDecPos(pSldObj);                     // decrement the slider position
                        SetState(pSldObj, SLD_DRAW_THUMB);// redraw only the thumb
                }
```

```
                else {
                        BtnSetBitmap(pObj, NULL);               // remove the bitmap
                        ClrState(pObj, BTN_TEXTRIGHT);          // place the text back in the middle
                }
        }
        if (objectID == ID_BTN2) {
                if (objMsg == BTN_MSG_PRESSED) {
                        BtnSetBitmap(pObj, (void*)&redRightArrow);  // set bitmap to show
                        SetState(pObj, BTN_TEXTLEFT);           // move the text to the left
                        pSldObj = (SLIDER*)GOLFindObject(ID_SLD1);   // find slider pointer
                        SldIncPos(pSldObj);                     // increment the slider position
                        SetState(pSldObj, SLD_DRAW_THUMB);// redraw only the thumb
                }
                else {
                        BtnSetBitmap(pObj, NULL);                // remove the bitmap
                        ClrState(pObj, BTN_TEXTLEFT);           // place the text back in the middle
                }
        }
        return 1;
}
```

Además, para la visualización de las imágenes ha sido necesario incluir el archivo *Pictures PIC32.c,* cuyo archivo se ha generado mediante la herramienta *Bitmap and Font converter.exe*. Esta herramienta nos permite, partiendo de una imagen en bitmap, generar un archivo .c con la información requerida para que se puede usar con la librería gráfica. A continuación mostramos una imagen del programa mientras se está pulsando el botón derecho.

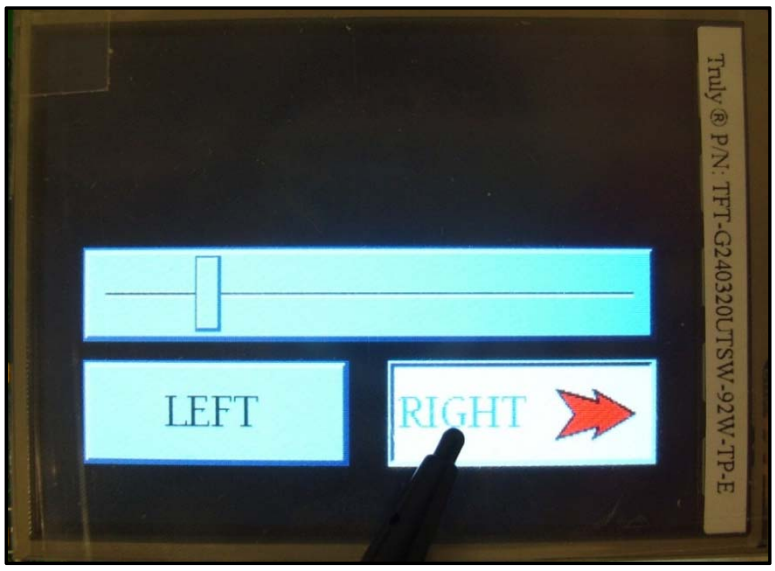


Figura 6.16: Programa "*AN1136_v2.0.c*" ejecutado en la pantalla táctil.

Programa: *AN1136Demo PIC32.c*

Para finalizar probamos el programa completo propuesto en la referencia [19], el cual añade un objeto personalizado mediante una serie de bitmaps. Este objeto se va a comportar como una serie de barras, de tal forma que cuando el valor del Slider se incremente las barras aumentarán en concordancia con su valor. Así mismo cuando el valor del Slider se reduzca las barras correspondientes serán eliminadas. Este objeto personalizado lo tendremos que incluir dentro de la función *GOLDrawCallBack().* Estas barras se pueden desplazar al igual que en los ejemplos anteriores, tocando los botones o bien moviendo el cursor del slider. Debido a la extensíon de este programa no mostramos el código sino una imagen del mismo mientras que esta siendo ejecutado. El programa se encuentra disponible en la misma carpeta que los dos anteriores.

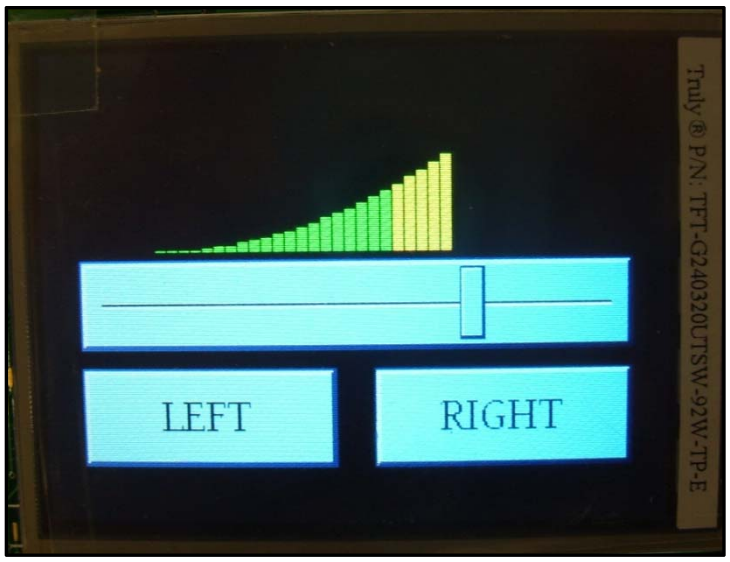


Figura 6.17: Programa "*AN1136Demo.c*" ejecutado en la pantalla táctil.

Para más información acerca de la pantalla táctil, la librería, las distintas soluciones gráficas para aplicaciones embebidas y el software empleado se puede consultar la página oficial de microchip en el apartado Graphics. Desde este apartado se ha descargado y consultado todos los documentos así como el software necesario [20].

7. APLICACIONES DESRROLLADAS

CAPÍTULO 7. APLICACIONES DESARROLLADAS.

7.1. INTRODUCCIÓN

El presente capítulo tiene como finalidad describir cuáles han sido los métodos seguidos y las aplicaciones desarrolladas haciendo uso de todos los componentes explicados en los capítulos anteriores.

El objetivo final buscado es la creación de varios programas que muestren el funcionamiento de dichos módulos, así como las distintas funciones empleadas en la elaboración de las distintas aplicaciones.

7.2. PANTALLA TÁCTIL

Este programa desarrollado para funcionar con la Explorer16+Graphics Pictail nos va a permitir integrar en único programa aspectos tan diversos como son las entradas analógicas tratadas mediante el modulo A/D procedentes del potenciómetro y del sensor de temperatura para su visualización de forma gráfica en la pantalla táctil, así como la utilización del modulo RTCC del PIC32, para configurar de una forma gráfica tanto la fecha como la hora actual.

7.2.1. ASPECTOS A TENER EN CUENTA PARA ELABORAR EL PROGRAMA

Lo primero que tenemos que saber es cómo funciona cada módulo que vamos a necesitar emplear para elaborar nuestro programa, estos son:

- Módulo ADC, visto en el Capitulo 5 apartado 5.3.6.
- Librería Gráfica, funcionamiento visto en el Capítulo 6.
- Módulo RTCC, usamos la librería *rtcc.c* y *rtcc.h.*

De estos tres módulos, el que aún no hemos comentado como funciona es el modulo RTCC (Salvo en el Anexo B en el que se muestra como configurarlo para actuar como interrupciones).

Para usar el Módulo RTCC vamos a usar las librerías *rtcc.c* y *rtcc.h*, las cuales nos van a facilitar la configuración de este módulo. Para usar este módulo se requiere que se disponga de un reloj externo de 32.768kHZ de lo contrarío no lo podríamos utilizar [3]. Sin embargo, el sistema de desarrollo Explorer16 dispone de este reloj incorporado en la placa (Ver Capitulo 5).

Para controlar este módulo vamos a usar las siguientes funciones disponibles en la librería anterior:

- RTCCInit(); Inicializa el modulo RTCC, habilitando el oscilador SOSC, para funcionar bajo el modulo RTCC.
- mRTCCOff(); Deshabilita el modulo RTCC temporalmente, para poder acceder a sus registros, configuración inicial del mismo.
- RTCCSetBinXXX(Value); Asigna el año, el mes, el día, la hora, el minuto y el segundo del valor de “value” a los registros del RTCC, donde XXX se corresponde a las distintas funciones según asignamos el día, hora, etc.
- RTCCCalculateWeekDay(); Esta función lee el día, mes y año de los registros del modulo RTCC y calcula el día de la semana.
- mRTCCSet(); copia los registros asignados mediante las funciones anteriores, a los registros del reloj del módulo RTCC.
- RTCCProcessEvents(); Esta función actualiza las cadenas de caracteres que nos permiten mostrar la hora y fecha, desde los registros del módulo RTCC.
- mRTCCGetBinXXX(); según sea “XXX”, obtiene el valor de los segundos, minutos, horas, día, mes o año actuales desde los registros del RTCC.

A parte de los módulos citados anteriormente, también tenemos que conocer cómo funciona tanto el potenciómetro como el sensor de temperatura para poder trabajar con ellos.

Potenciómetro:

Las principales características del potenciómetro presente en el sistema de desarrollo Explorer16 son:

- Potenciómetro de 10k Ohmios.
- Lectura del potenciómetro a través de R6.
- Está conectado directamente a 3.3V, por lo que su salida estará comprendida entre 3.3V y 0V.
- Conectado a la entrada analógica AN5.

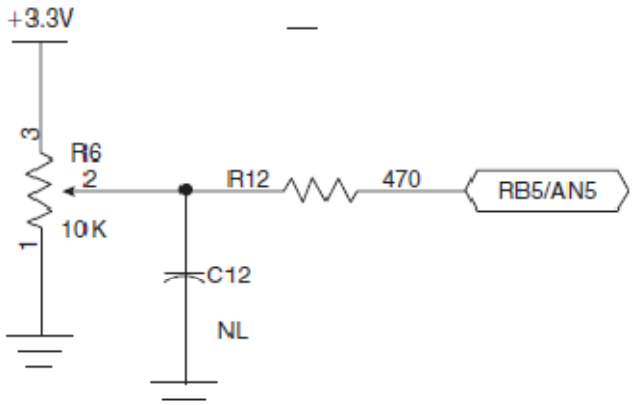


Figura 7.1: Esquema del Potenciómetro R6 de la tarjeta Explorer16.

Sensor de Temperatura:

Las principales características del sensor de temperatura TC1047 presente en el sistema de desarrollo Explorer16 son:

- Pendiente 10mV/ºC
- Conectado a la entrada analógica AN4, en la explorer16.
- Temperatura:

$$T = \frac{Vout - 500mV}{10mV/ºC}$$

Donde Vout=ADC Reading * ADC resolution (mV/bit), y donde resolution= 3.3mV/bit, por tanto:

$$T = \frac{\frac{3.3mV}{bit} * Vout(reading) - 500mV}{10mV/ºC}$$

Para una correcta medición de la temperatura vamos a tener que tomar muestras cada 0.1s durante 1s y vamos a calcular la media de todas ellas, ya que los sensores de temperatura proporcionan un alto nivel de ruido, por lo que de esta forma conseguiremos reducir estos errores.

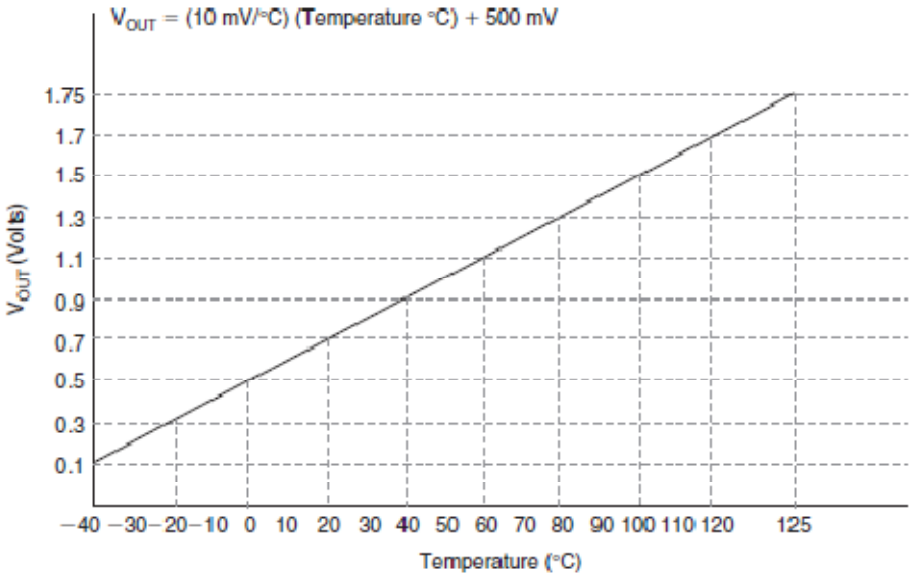
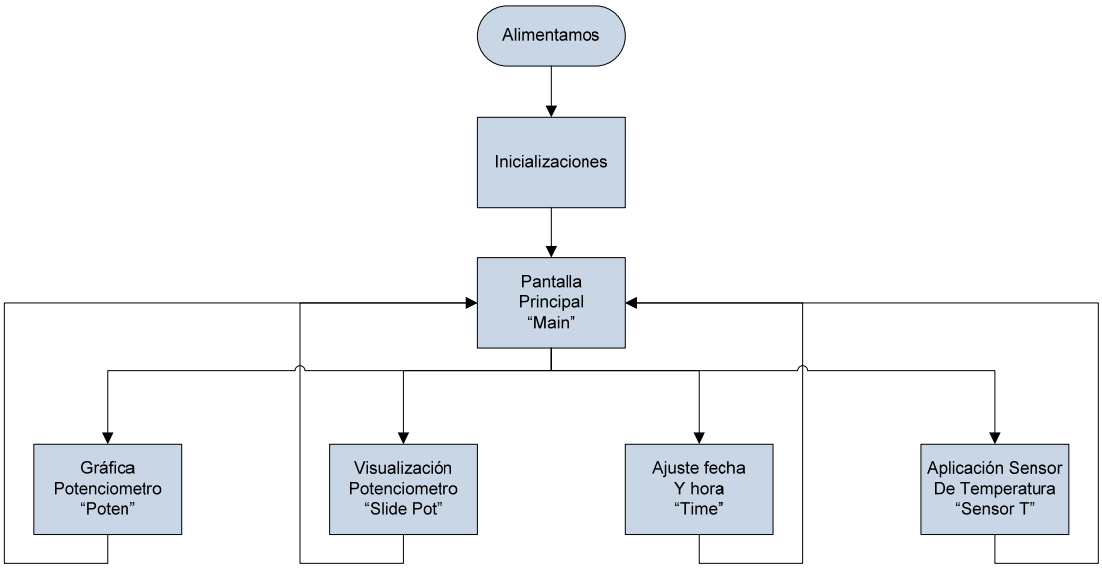


Figura 7.2: Sensor de temperatura TC1047, Voltaje de salida (Vout) respecto temperatura.

7.2.2. ESTRUCTURA DEL PROGRAMA

Una vez conocemos como van a funcionar los módulos empleados para la realización del programa, veamos de qué forma vamos a estructurarlo.

El programa va a constar de una pantalla principal a través de la cual vamos a poder acceder a cada una de las distintas pantallas secundarias con tal de comprobar el funcionamiento del sensor, el potenciómetro y el ajuste de la hora, tal y como podemos observar en el siguiente diagrama:

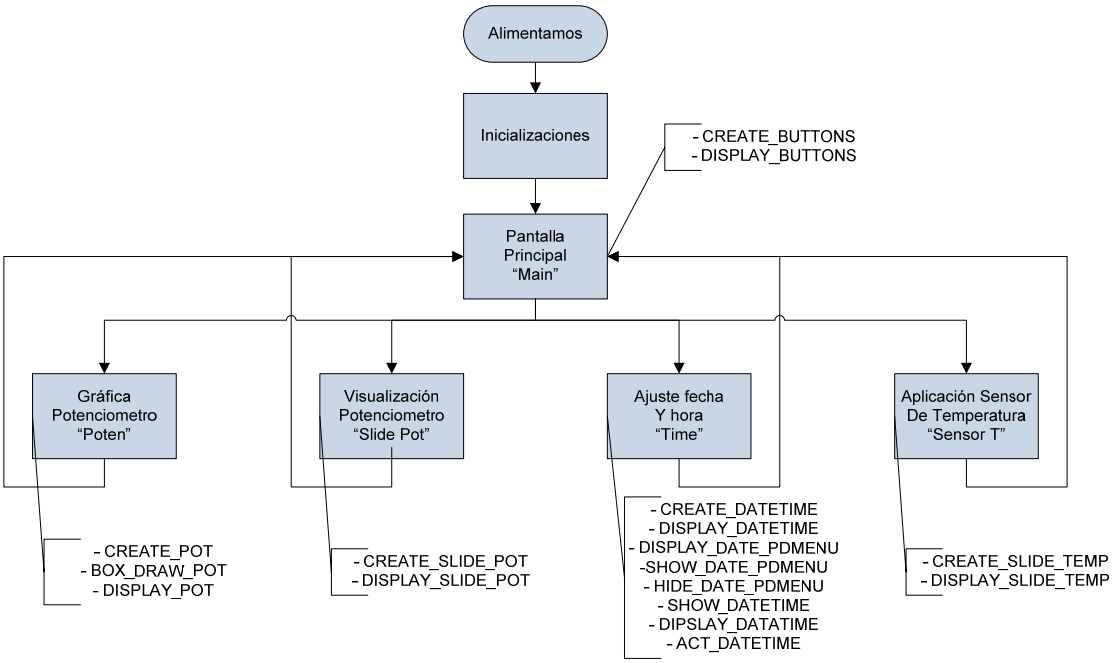


Al igual que en los programas vistos en el Capítulo 6, la función main de nuestro programa solo va a contener dos funciones las cuales llaman a otras dos, estas son:

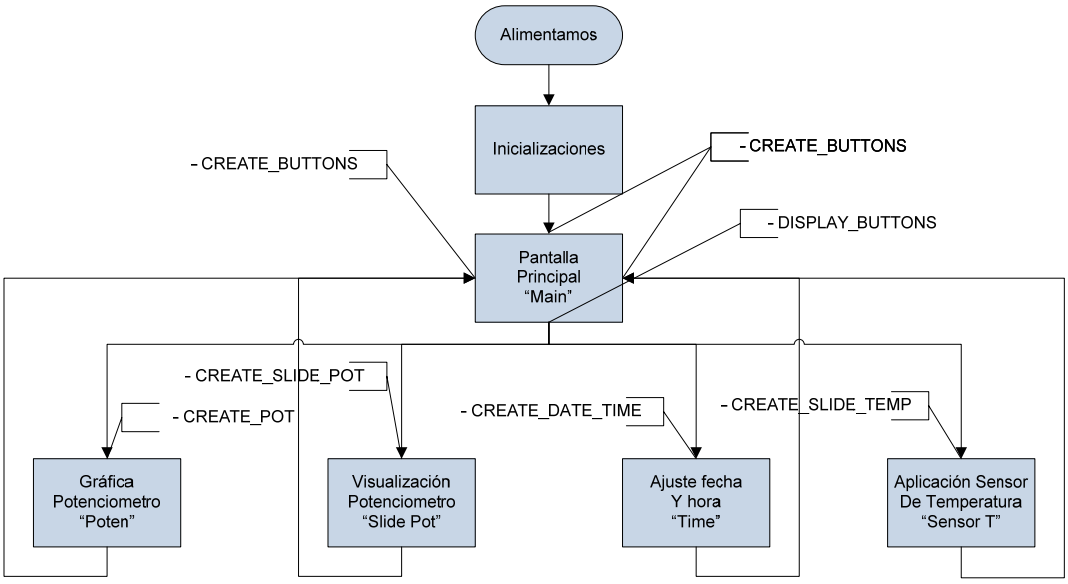
```
while(1){
    if (GOLDraw()) {                    // Draw GOL object
        TouchGetMsg(&msg);              // Get message from touch screen
        GOLMsg(&msg);                   // Process message
    }
}
```

De tal forma que la función GOLMsgCallback será llamada por GOLmsg() cada vez que un mensaje valido sea recibido por un objeto, mientras que la función GOLDrawCallBack(), será llamada por la función GOLDraw() cada vez que un objeto sea dibujado completamente. Por tanto, mediante esta función podremos cambiar el estado de los objetos en la lista activa así como cambiar su configuración, color, tipo de línea, gráficos, etc.

Estas dos funciones, están estructuradas mediante una serie de estados de pantalla. Cada una de las pantallas anteriores, en total vamos a tener cinco (pantalla principal, más las cuatro pantallas secundarias), van a tener una serie de estados asociados a estas, llamados "SCREEN_STATES". Cada uno de estos estados nos va a ayudar a organizar nuestro programa, ya que a través de estos estados vamos a poder saber qué es lo que se está mostrando por pantalla. Los estados asociados a cada una de las pantallas son los siguientes:



El cambio de un estado a otro se puede deber a distintas causas. Por ejemplo, cuando conectamos nuestro sistema a corriente, el primer estado que se asigna es el Create_Buttons. De tal forma que cuando se ejecute la función de actualizar la pantalla y se chequee el estado de la pantalla el seleccionado será este. Este estado llamará a la función *CreateButtons()*, la cual creará el menú inicial de nuestro programa. Una vez finalizada la creación de la pantalla principal, se cambiará el estado por Display_Buttons. Durante este estado únicamente se va a estar chequeando la interfaz de mensaje hasta que se produzca un evento, es decir, seleccionemos alguno de los cuatro botones de la pantalla inicial para que se muestre una secundaria. Dependiendo del botón seleccionado se cambiará a un estado u otro, en el siguiente diagrama de flujo lo mostramos (los estados que no se han mostrado en el siguiente diagrama de flujo, se debe a que todos ellos se utilizan para actualizar cada una de las pantallas secundarias a la que pertenecen).



A continuación mostramos las funciones GOLMsgCallback() y GOLDrawCallback() dada su importancia en el funcionamiento del programa.

```
WORD GOLMsgCallback(WORD objMsg, OBJ_HEADER* pObj, GOL_MSG* pMsg){
   /* beep if button is pressed
   if(objMsg == BTN_MSG_PRESSED)
      Beep();*/
          if ((screenState & 0xF300) != 0xF300) {
                    // check for time setting press, process only when not setting time and date
                    if (objMsg == ST_MSG_SELECTED) {
          if ((GetObjID(pObj) == ID_STATICTEXT1) || (GetObjID(pObj) == ID_STATICTEXT2)) {
                              prevState = screenState - 1;  // save the current create state
                              screenState = CREATE_DATETIME;// go to date and time setting screen
                              return 1;
                         }
              }
   }
   // process messages for screens
   switch(screenState){
   case DISPLAY_BUTTONS:
         return MsgButtons(objMsg, pObj);
   case DISPLAY_POT:
         return MsgPotentiometer(objMsg, pObj);

      // date and time settings display
    case DISPLAY_DATETIME:
         return MsgDateTime(objMsg, pObj);
    case DISPLAY_DATE_PDMENU:
          return MsgSetDate(objMsg, pObj, pMsg);
    case CREATE_DATETIME:
          case SHOW_DATE_PDMENU:
     case HIDE_DATE_PDMENU:
                    return 0;
     case SHOW_DATETIME:
                    return MsgDateTime(objMsg, pObj);
     case ACT_DATETIME:
                    return MsgDateTime(objMsg, pObj);

     case DISPLAY_SLIDE_POT:
         return MsgSlidePOt(objMsg, pObj);

     case DISPLAY_SLIDE_TEMP:
          return MsgSlideTemp(objMsg, pObj);
     default:
           return 1;
                    }
}//GOLMsgCallback


WORD GOLDrawCallback(){

OBJ_HEADER   *pObj;                                   // used to change text in Window
SLIDER       *pSld;                                   // used when updating date and time
LISTBOX      *pLb;                                    // used when updating date and time

static BYTE  pBDelay   = 40;              // progress bar delay variable
static BYTE  direction = 1;               // direction switch for progress bar
static DWORD prevTick  = 0;               // keeps previous value of tick
static DWORD prevTime  = 0;               // keeps previous value of time tick
WORD                    i;
```

```
// update the time display
if ((screenState & 0x0000F300) != 0x0000F300) {// process only when NOT setting time and date
        if ((tick-prevTime) > 1000){
                RTCCProcessEvents();    // update the date and time string variabes
                i = 0;
                while (i < 12) {
                        dateTimeStr[i] = _time_str[i];
                        dateTimeStr[i+13] = _date_str[i];
                        i++;
                }
                dateTimeStr[12] = 0x000A;          // (XCHAR)'\n';
                dateTimeStr[25] = 0x0000;          // (XCHAR)'\0';
        if (pObj = GOLFindObject(ID_STATICTEXT2)) {          // get the time display obj pointer
                StSetText((STATICTEXT *)pObj, dateTimeStr); // now display the new date & time
                SetState(pObj, ST_DRAW);                     // redraw the time display
                StDraw((STATICTEXT *)pObj);
                        }
                prevTime = tick;            // reset tick timer
        }
} else {            // process only when setting time and date
                    // do not update when pull down menus are on
    if ((screenState != DISPLAY_DATE_PDMENU) && (screenState != HIDE_DATE_PDMENU))
    {
                        if ((tick-prevTime) > 1000){
                           updateDateTimeEb();   // update edit boxes for date and time settings
                                prevTime = tick; // reset tick timer
                        }
     }
}

   switch(screenState){
      case CREATE_BUTTONS:
         CreateButtons();                    // create window and buttons
         screenState = DISPLAY_BUTTONS; // switch to next state
         return 1;
      case DISPLAY_BUTTONS:
         return 1;                           // redraw objects if needed

      case CREATE_POT:
         CreatePotentiometer();              // create window
         screenState = BOX_DRAW_POT;   // switch to next state
         return 1;                           // draw objects created
           case BOX_DRAW_POT:
         if(0 == PanelPotentiometer())       // draw box for potentiometer graph
            return 0;                        // drawing is not completed, don't pass
                                             // drawing control to GOL, try it again
         screenState = DISPLAY_POT;          // switch to next state
         return 1;                           // pass drawing control to GOL, redraw objects if needed
      case DISPLAY_POT:
         if((tick-prevTick)>20){
            if(GetPotSamples(POT_MOVE_DELTA))
               GraphPotentiometer();         // redraw graph
            prevTick = tick;
         }
         return 1;

     case CREATE_DATETIME:
         CreateDateTime();                   // create date and time demo
         screenState = DISPLAY_DATETIME;// switch to next state
         return 1;                           // draw objects created
```

```
    case SHOW_DATE_PDMENU:
                ShowPullDownMenu();
                screenState = DISPLAY_DATE_PDMENU;
                return 1;

    case HIDE_DATE_PDMENU:
                if (RemovePullDownMenu())
                 screenState = DISPLAY_DATETIME;       // switch to next state
                return 1;

   case DISPLAY_DATE_PDMENU:
       // this moves the slider and editbox for the date setting to
       // move while the up or down arrow buttons are pressed
       if((tick-prevTick)>100) {
                pLb = (LISTBOX*)GOLFindObject(ID_LISTBOX1);
                        pSld = (SLIDER*)GOLFindObject(ID_SLIDER1);
                                pObj = GOLFindObject(ID_BUTTON_DATE_UP);
       if(GetState(pObj, BTN_PRESSED)) {
                LbSetFocusedItem(pLb,LbGetFocusedItem(pLb)-1);
                SetState(pLb, LB_DRAW_ITEMS);
                SldSetPos(pSld,SldGetPos(pSld)+1);
                SetState(pSld, SLD_DRAW_THUMB);
         }
                                pObj = GOLFindObject(ID_BUTTON_DATE_DN);

       if(GetState(pObj, BTN_PRESSED)) {
                  LbSetFocusedItem(pLb,LbGetFocusedItem(pLb)+1);
              SetState(pLb, LB_DRAW_ITEMS);
            SldSetPos(pSld,SldGetPos(pSld)-1);
          SetState(pSld, SLD_DRAW_THUMB);
       }
       prevTick = tick;
      }
      return 1;

  case DISPLAY_DATETIME:
       // Checks if the pull down menus are to be created or not
                  pObj = GOLFindObject(ID_BUTTON_MO);
                  if (GetState(pObj, BTN_PRESSED)) {
        screenState = SHOW_DATE_PDMENU;  // change state
                                return 1;
                        }
                  pObj = GOLFindObject(ID_BUTTON_YR);
                  if (GetState(pObj, BTN_PRESSED)) {
        screenState = SHOW_DATE_PDMENU;  // change state
                                return 1;
                        }
                  pObj = GOLFindObject(ID_BUTTON_DY);
                  if (GetState(pObj, BTN_PRESSED)) {
        screenState = SHOW_DATE_PDMENU;  // change state
                                return 1;
                        }
       // this increments the values for the time settings
       // while the + or - buttons are pressed
                if((tick-prevTick)>200) {
                        pObj = GOLFindObject(ID_BUTTONHR_P);
       if(GetState(pObj, BTN_PRESSED)) {
                MsgDateTime(BTN_MSG_PRESSED, pObj);
               }
                pObj = GOLFindObject(ID_BUTTONHR_M);
```

```
        if(GetState(pObj, BTN_PRESSED)) {
                MsgDateTime(BTN_MSG_PRESSED, pObj);
               }
                                pObj = GOLFindObject(ID_BUTTONMN_P);
        if(GetState(pObj, BTN_PRESSED)) {
                MsgDateTime(BTN_MSG_PRESSED, pObj);
               }
                                pObj = GOLFindObject(ID_BUTTONMN_M);
        if(GetState(pObj, BTN_PRESSED)) {
                MsgDateTime(BTN_MSG_PRESSED, pObj);
               }
                                pObj = GOLFindObject(ID_BUTTONSC_P);
        if(GetState(pObj, BTN_PRESSED)) {
                MsgDateTime(BTN_MSG_PRESSED, pObj);
               }
                                pObj = GOLFindObject(ID_BUTTONSC_M);
        if(GetState(pObj, BTN_PRESSED)) {
                MsgDateTime(BTN_MSG_PRESSED, pObj);
               }
              prevTick = tick;
            }
            return 1;

    case SHOW_DATETIME:
       CreateDataTime();                    // create window and buttons
       screenState = ACT_DATETIME;    // switch to next state
       return 1;

    case ACT_DATETIME:
       if ((tick-prevTime) > 1000){
                RTCCProcessEvents();// update the date and time string variabes
                i = 0;
                while (i < 12) {
                        dateTimeStr[i] = _time_str[i];
                        dateTimeStr[i+13] = _date_str[i];
                        i++;
                }
                dateTimeStr[12] = 0x000A;          // (XCHAR)'\n';
                dateTimeStr[25] = 0x0000;          // (XCHAR)'\0';

            if (pObj = GOLFindObject(ID_STATICTEXT1)) {    // get the time display obj pointer
                StSetText((STATICTEXT *)pObj, dateTimeStr); // now display the new date & time
                SetState(pObj, ST_DRAW);                    // redraw the time display
                }
                        prevTime = tick;           // reset tick timer
        }
        screenState = ACT_DATETIME;
                return 1;

    case CREATE_SLIDE_POT:
       CreateSlidePot();                        // create window and buttons
       screenState = DISPLAY_SLIDE_POT;         // switch to next state
       return 1;
    case DISPLAY_SLIDE_POT:
       if((tick-prevTick)>20){
                GraphSlidePOT();           // redraw graph
          prevTick = tick;
       }
       return 1;
```

```
      case CREATE_SLIDE_TEMP:
        CreateSlideTemp();                          // create window and buttons
        screenState = DISPLAY_SLIDE_TEMP;     // switch to next state
        return 1;
      case DISPLAY_SLIDE_TEMP:
        if((tick-prevTick)>20){
                    GraphSlideTemp();          // redraw graph
          prevTick = tick;
        }
        return 1;
    }
}//GOLDrawCallback
```

Como podemos observar en el código anterior, los estados de cada pantalla nos ayudan a organizar nuestro programa llamando a la función concreta para crear una serie de objetos, o bien para actualizar estos.

7.2.3. FUNCIONAMIENTO DEL PROGRAMA

Una vez aclarada la estructura, vamos a ver que realiza cada una de las pantallas mencionadas anteriormente.

La pantalla principal es la pantalla mediante la cual vamos a poder acceder a las demás presionando el botón que queramos. Una vez hayamos accedido a una pantalla secundaria y pulsemos el botón de retorno, volveremos a esta pantalla. Además en esta pantalla principal se ha incorporado el logo de la UMH, cuyo archivo en formato c se ha generado mediante la herramienta "*Bitmap and Font converter.exe",* disponible en la librería gráfica usada. A continuación mostramos una imagen de esta pantalla principal.

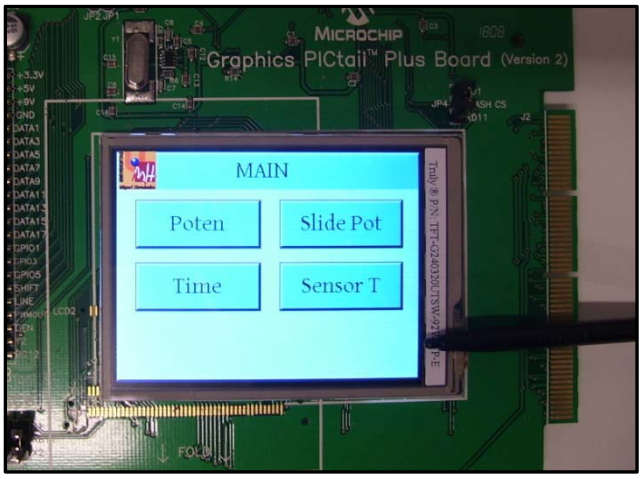


Figura 7.3: Pantalla principal, Programa "*Proyecto_1.c".*

Si presionamos sobre el primer botón (“Poten”) accederemos a una pantalla en la que se nos mostrará una gráfica en la que vamos a poder ver en tiempo real como cambia la resistencia del potenciómetro (la modificamos de forma manual a través de la tarjeta Explorer16) en función del tiempo. Además en todas las pantallas secundarías se ha incorporado un botón de retorno a la pantalla principal situado a la izquierda de la misma. A continuación mostramos una imagen de dicha pantalla en la que se ha ido cambiando el valor del potenciómetro.

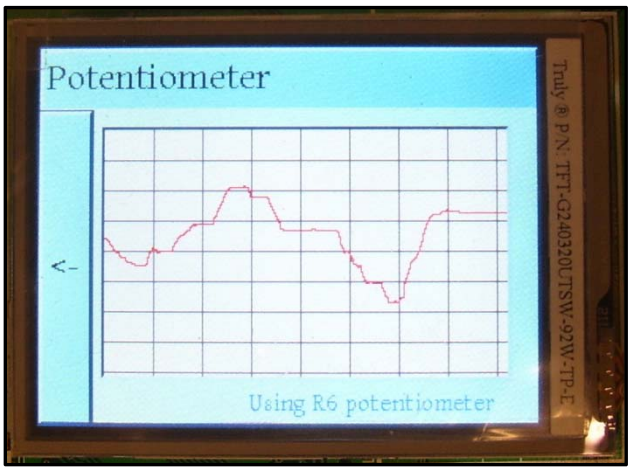


Figura 7.4: Pantalla Secundaria, “Potentiometer” , Programa “*Proyecto_1.c*”.

Si presionamos sobre el segundo botón (“Slide Pot”), vamos a poder ver como varía el potenciómetro mediante el uso de cuatro objetos diferentes. El primero es una barra la cual muestra el valor en tanto por ciento según si la resistencia es máxima (10KΩ, 100%) o mínima (0KΩ, 0%). Esta barra está asociada a un “slide” de tal forma que este último se moverá en la misma medida en la que lo haga la barra en tanto por ciento. Por otra parte se muestra un indicador circular (“meter”) en el que se muestra el valor de la resistencia del potenciómetro de una forma gráfica. Por último a la derecha de la pantalla podemos visualizar el valor de este numéricamente, según la figura7.5.

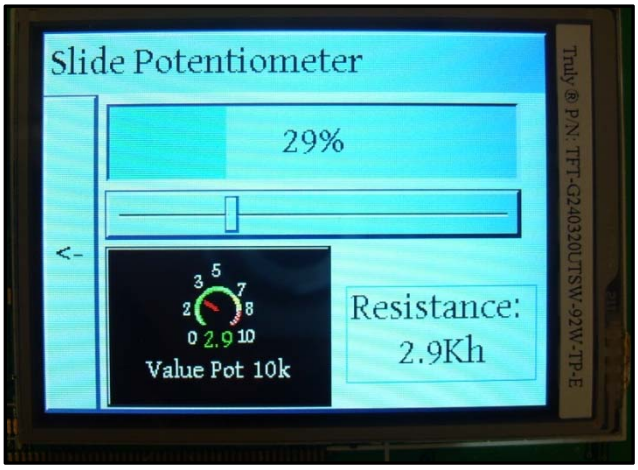
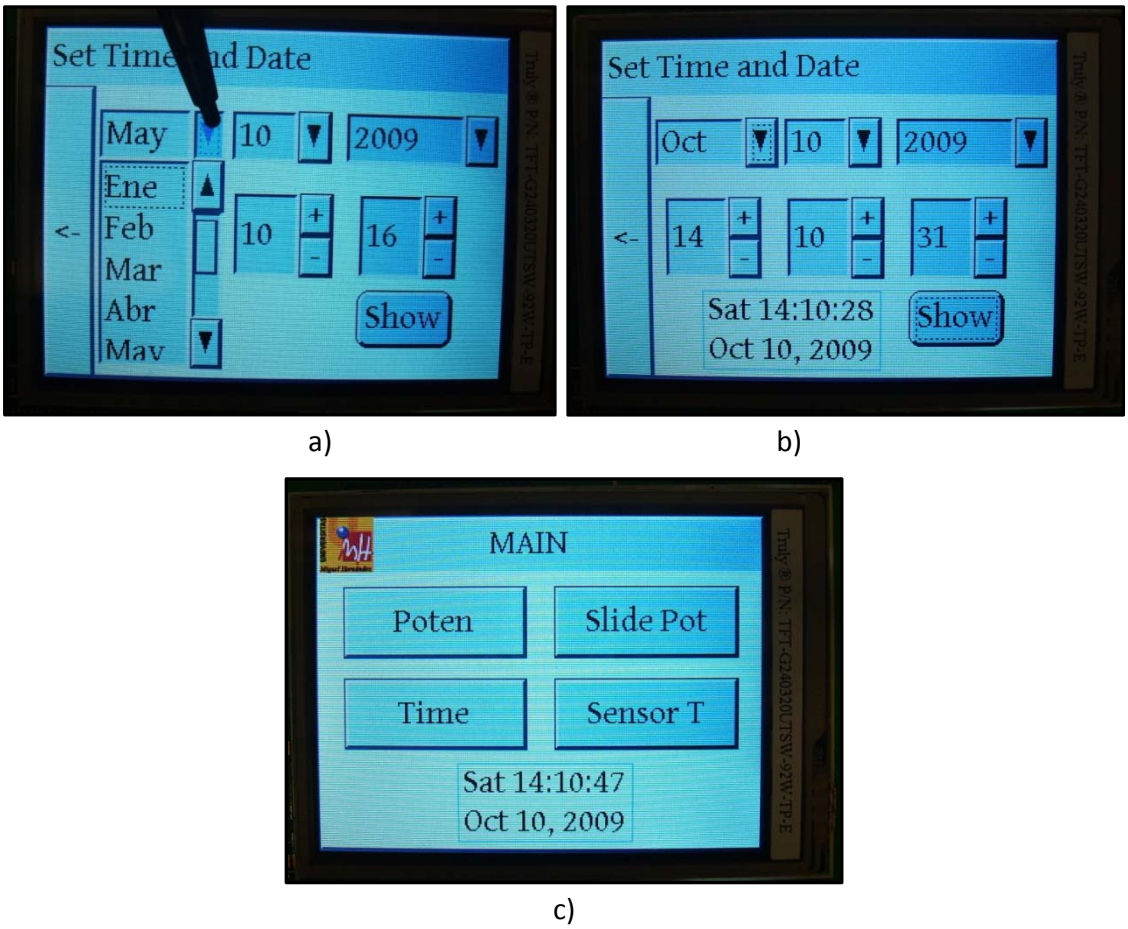


Figura 7.5: Pantalla Secundaria, “Slide Potentiometer”, Programa “*Proyecto_1.c*”.

La tercera pantalla (“Time”) nos permite ajustar la fecha y la hora, para visualizarla posteriormente en la pantalla principal. Primero vamos a indicar la fecha, es decir, mes día y año, y posteriormente la hora, minutos y segundos. Para elaborar estos menús de selección se han utilizado menús desplegables (Figura 7.6.a) para ajustar la fecha y botones de adición o resta para el caso de la hora. Una vez ajustada nuestra hora le damos show y se nos mostrará en esta misma pantalla la fecha y la hora seleccionada (Figura 7.6.b). Una vez que pulsemos el botón de retorno para regresar a la pantalla principal, en esta última, se nos mostrará la fecha y la hora y podremos observar cómo va contando el tiempo (Figura7.6.c). Además si pulsamos sobre este texto una vez configurada la hora también podremos acceder de nuevo al menú de configuración de la fecha y de la hora. A continuación mostramos una imagen de esta pantalla secundaría y de la pantalla principal una vez configurada la hora.

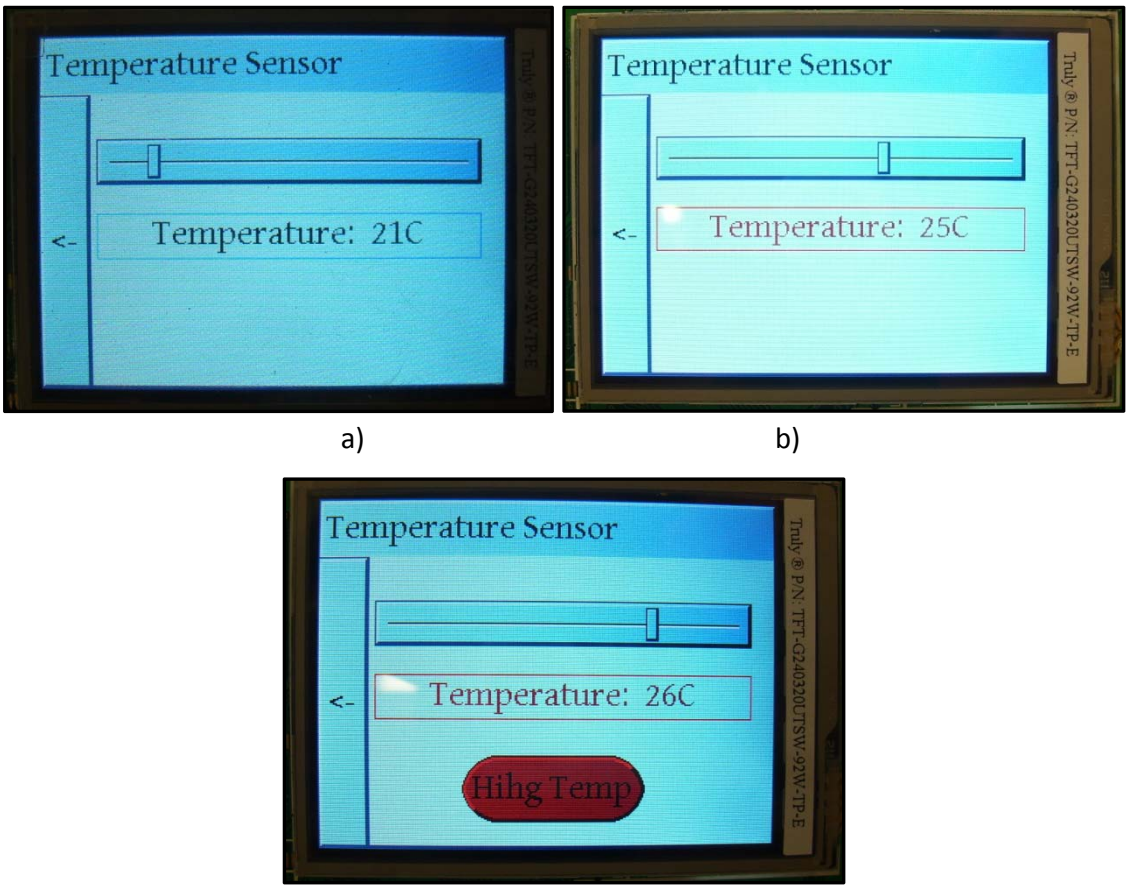


a) b)

c)

Figura 7.6: Programa “*Proyecto_1.c*” a) Pantalla Secundaria, “Set Time and Date”, detalle al presionar el menú desplegable para seleccionar el mes. b) Fecha y hora configurada tras pulsar el botón “Show”. c) Pantalla principal tras configurar la hora y la fecha.

Por último, si presionamos sobre el cuarto botón (“Sensor T”), accederemos a una pantalla en la que vamos a comprobar el funcionamiento del sensor de temperatura TC1047 presente en la tarjeta Explorer16. A través de una barra deslizadora que se mueve en intervalos de un grado vamos a mostrar la temperatura en los alrededores de la placa (Figura7.7.a). El intervalo de temperatura se ha establecido entre 20°C y 27°C, dada la dificultad de enfriar por debajo de 20°C y de calentar por encima de 27°C. Además se han añadido una serie de funcionalidades a este programa. Si la temperatura supera los 24°C, el indicador de la temperatura se mostrará de color rojo (Figura7.7.b), y si esta excede o es igual a 26°C, se mostrará un botón de aviso de alta temperatura (Figura7.7.c). A continuación mostramos las posibles combinaciones de esta pantalla.



a) b)

c)

Figura 7.7: Pantalla Secundaria, “Temperature Sensor”, Programa “*Proyecto_1*.c”
a)Pantalla cuando la temperatura es inferior a 24°C. b) Temperatura del sensor entre 24°C y 25°C. c) Temperatura igual o superior a 26°C.

El código completo de este programa se puede encontrar en la ubicación “APLICACIONES_FINALES/PANTALLA_TACTIL/PROYECTO_1” del Cd adjunto al presente proyecto. Este no se ha imprimido debido a su extensión ya que contiene más de dos mil líneas de código.

7.3. PANTALLA TÁCTIL Y ACELEROMETRO

Una vez visto el programa anterior, vamos a proceder a incorporar un acelerómetro a nuestros programas, calibrándolo primero y realizando una serie de aplicaciones mediante el uso de la pantalla táctil.

7.3.1. ASPECTOS A TENER EN CUENTA PARA ELABORAR EL PROGRAMA

Como dijimos en el capítulo 6, la pantalla táctil se podía conectar bien al sistema de de desarrollo “Explorer16” o bien a la “I/O Expansion Board”, ambas mediante el uso del bus PICtail Plus. Para este programa conectaremos la pantalla gráfica a la “I/O Expansion Board” pues mediante esta tarjeta va a resultar mucho más fácil acceder a los pines del microcontrolador para poder conectar el acelerómetro. Además, como podemos ver en la siguiente figura, colocaremos directamente el “Starter Kit PIC32” en el zócalo reservado para ello, por lo que no será necesario utilizar el adaptador como en los casos anteriores.

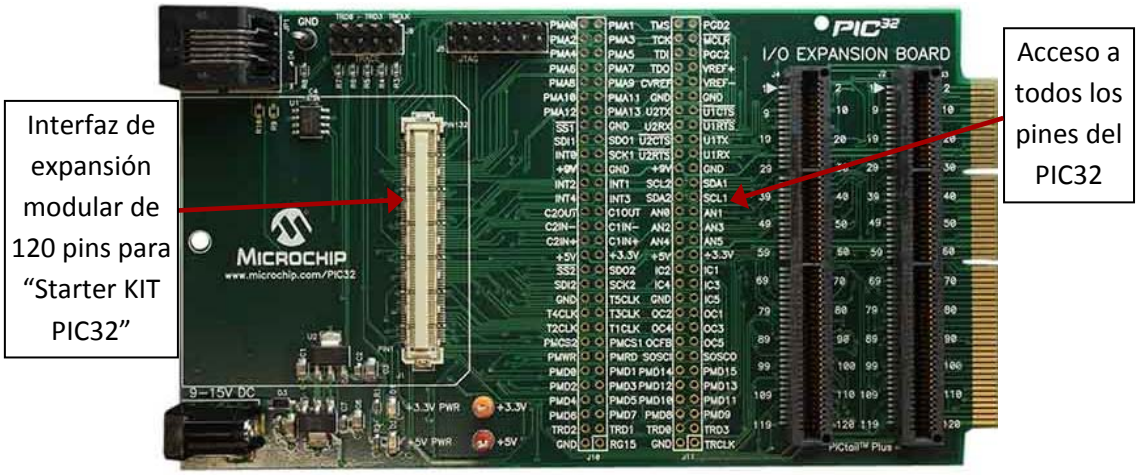


Figura 7.8: I/O Expansion Board.

El acelerómetro, dispone únicamente de 5 pines (ver Anexo C), estos son: alimentación, tierra, y voltaje de salida en los tres ejes. La alimentación la tendremos que colocar a uno de los pines de la tarjeta de expansión que nos proporcione +5V y la tierra a un pin que vaya a masa. Por otra parte, las salidas del acelerómetro las tendremos que conectar a tres canales cualesquiera de entrada del modulo Analógico/Digital. Por tanto, fijándonos en la figura anterior nos damos cuenta que solo vamos a necesitar la hilera J11 (la situada más a la derecha), ya que en esta vamos a tener acceso a todos las señales que necesitamos.

Para poder conectar el sensor a la placa de expansión, lo primero que realizamos fue soldar a la tarjeta en la que se encontraba el acelerómetro 5 cables para poder acceder a ellos de una forma más cómoda. Por otra parte, se soldó en la placa de expansión un conector para poder acceder a los pines comentados en el párrafo anterior, el cual lo colocamos únicamente en la hilera J11, tal y como podemos ver en la siguiente figura (en la figura se han señalado las señales a la que vamos a conectar nuestro acelerómetro).

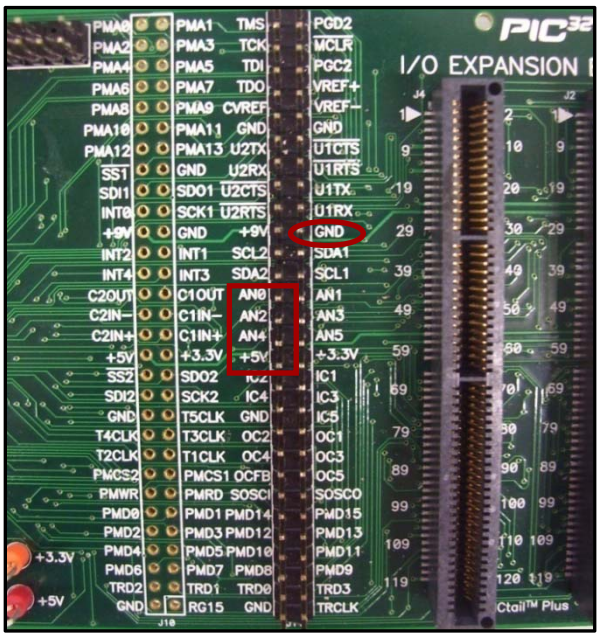


Figura 7.9: I/O Expansion Board con conector soldado.

De tal forma que el conjunto placa de expansión y acelerómetro quedará conectado de la siguiente forma:

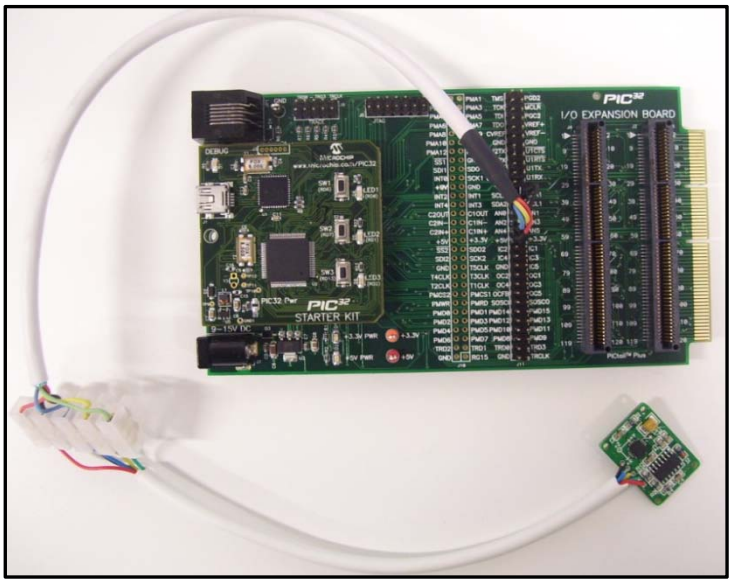
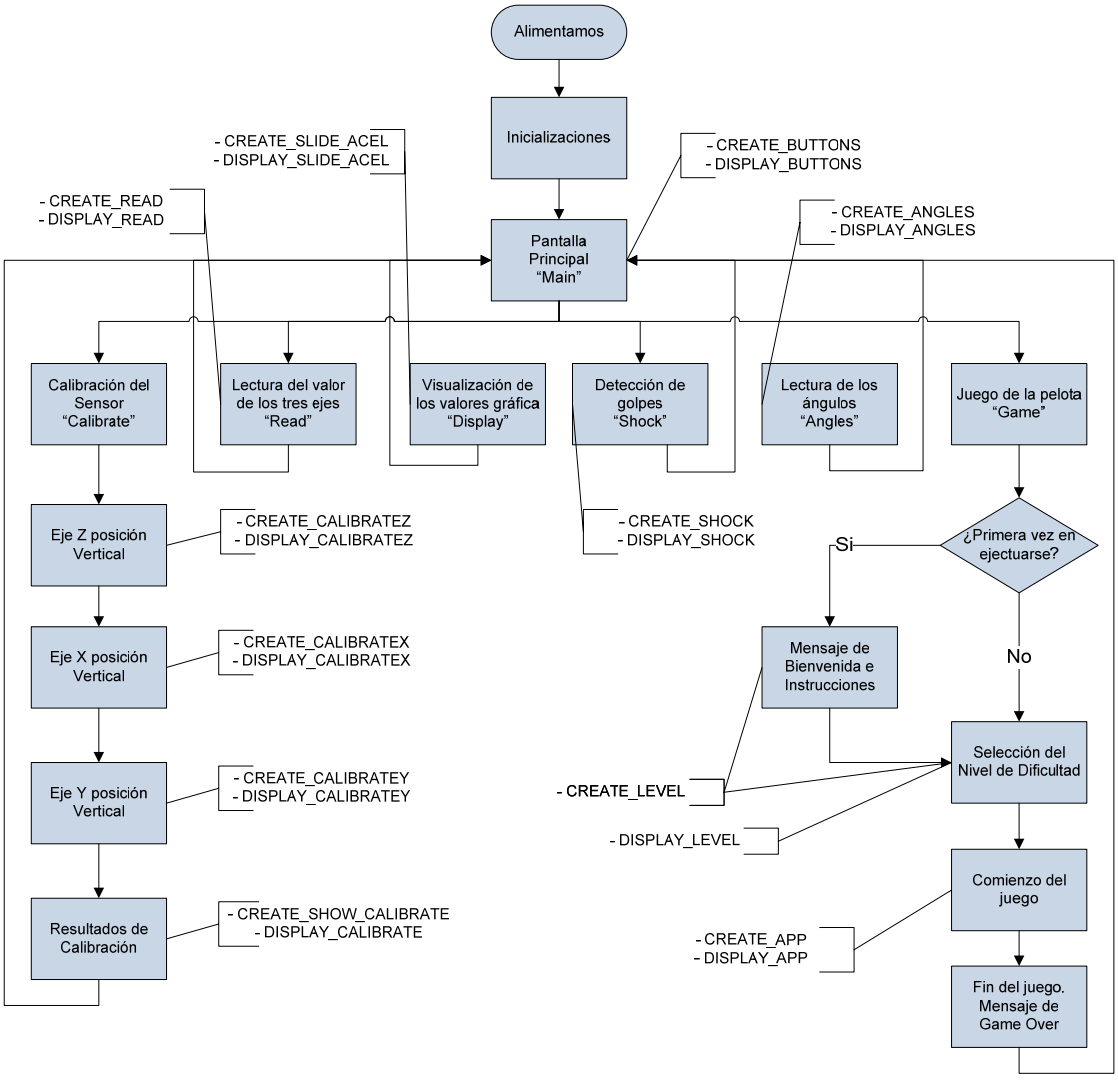


Figura 7.10: Conexión del acelerómetro a la I/O Expansion Board.

7.3.2. ESTRUCTURA DEL PROGRAMA

Al igual que en el programa anterior, este va a constar de una pantalla principal a través de la cual vamos a acceder a las demás pantallas secundarias, calibración del sensor, lectura del sensor en tiempo real, detección de golpes, juego de la pelota, etc. También, vamos a organizar nuestro programa a través de una serie de estados asociados a cada pantalla, de tal manera que los estados asociados a cada uno de las pantallas son los siguientes:



Como podemos ver el diagrama anterior, la estructura y organización del programa es muy similar al anterior. Además tanto la función GOLMsgCallback() como la función GOLDrawCallback() van a estar estructuradas de la misma forma que antes:

```
WORD GOLMsgCallback(WORD objMsg, OBJ_HEADER* pObj, GOL_MSG* pMsg){
   /* beep if button is pressed
   if(objMsg == BTN_MSG_PRESSED)
      Beep();*/

   // process messages for demo screens
   switch(screenState){
      case DISPLAY_BUTTONS:
         return MsgButtons(objMsg, pObj);

                  case DISPLAY_CALIBRATEZ: //Calibrate
         return MsgCalibrationZ(objMsg, pObj);
                  case DISPLAY_CALIBRATEX:
         return MsgCalibrationX(objMsg, pObj);
                  case DISPLAY_CALIBRATEY:
         return MsgCalibrationY(objMsg, pObj);
                  case DISPLAY_CALIBRATE:
         return MsgCalibration(objMsg, pObj);

                  case DISPLAY_READ:
         return MsgRead(objMsg, pObj);

                  case DISPLAY_SLIDE_ACEL:
         return MsgSlideAcel(objMsg, pObj);

                  case DISPLAY_SHOCK:
         return MsgShock(objMsg, pObj);
                  case DISPLAY_ANGLES:
         return MsgAngles(objMsg, pObj);
                  case DISPLAY_APP:
         return MsgAPP(objMsg, pObj);

                  case DISPLAY_LEVEL:
         return MsgLevel(objMsg, pObj);

                  default:
          return 1;
          }
}//GOLMsgCallback

////////////////////////////////////////////////////////////////////////
// Function: WORD GOLDrawCallback()
////////////////////////////////////////////////////////////////////////
WORD GOLDrawCallback(){
OBJ_HEADER   *pObj;                  // used to change text in Window
SLIDER       *pSld;                  // used when updating date and time
LISTBOX      *pLb;                   // used when updating date and time
static DWORD prevTick  = 0;          // keeps previous value of tick
static DWORD prevTime  = 0;          // keeps previous value of time tick
WORD                 i;

   switch(screenState){

      case CREATE_BUTTONS:
         CreateButtons();                        // create window and buttons
         screenState = DISPLAY_BUTTONS;          // switch to next state
         return 1;
      case DISPLAY_BUTTONS:
         return 1;                               // redraw objects if needed
```

```
case CREATE_CALIBRATEZ:
   CreateCalibrationZ();                  // create window and buttons
   screenState = DISPLAY_CALIBRATEZ;      // switch to next state
   return 1;
case DISPLAY_CALIBRATEZ:
   return 1;                              // redraw objects if needed

case CREATE_CALIBRATEX:
   CreateCalibrationX();                  // create window and buttons
   screenState = DISPLAY_CALIBRATEX;      // switch to next state
   return 1;
case DISPLAY_CALIBRATEX:
   return 1;                              // redraw objects if needed

case CREATE_CALIBRATEY:
   CreateCalibrationY();                  // create window and buttons
   screenState = DISPLAY_CALIBRATEY;      // switch to next state
   return 1;
case DISPLAY_CALIBRATEY:
   return 1;                              // redraw objects if needed

case CREATE_SHOW_CALIBRATE:
   CreateCalibration();                   // create window and buttons
   screenState = DISPLAY_CALIBRATE;       // switch to next state
   return 1;
case DISPLAY_CALIBRATE:
   return 1;                              // redraw objects if needed

case CREATE_READ:
   CreateRead();                          // create window and buttons
   screenState = DISPLAY_READ;            // switch to next state
   return 1;
case DISPLAY_READ:
   if((tick-prevTick)>20){
            ActRead();                    // redraw graph
      prevTick = tick;
   }
   return 1;

case CREATE_SLIDE_ACEL:
   CreateSlideAcel();                     // create window and buttons
   screenState = DISPLAY_SLIDE_ACEL;      // switch to next state
   return 1;
case DISPLAY_SLIDE_ACEL:
   if((tick-prevTick)>20){
            ActSlideAcel();               // redraw graph
      prevTick = tick;
   }
   return 1;

case CREATE_SHOCK:
   CreateShock();                         // create window and buttons
   screenState = DISPLAY_SHOCK;           // switch to next state
   return 1;
case DISPLAY_SHOCK:
   if((tick-prevTick)>20){
            ActShock();                   // redraw graph
      prevTick = tick;
   }
   return 1;
```

```
      case CREATE_ANGLES:
        CreateAngles();                          // create window and buttons
        screenState = DISPLAY_ANGLES;            // switch to next state
        return 1;
      case DISPLAY_ANGLES:
        if((tick-prevTick)>20){
                  ActAngles();                   // redraw graph
          prevTick = tick;
        }
        return 1;

      case CREATE_APP:
        CreateAPP();                             // create window and buttons
        screenState = DISPLAY_APP;               // switch to next state
        return 1;
      case DISPLAY_APP:
          ActAPP();                  // redraw graph
        return 1;

  case CREATE_LEVEL:
        CreateLevel();                           // create window and buttons
        screenState = DISPLAY_LEVEL;             // switch to next state
        return 1;
      case DISPLAY_LEVEL:
        return 1;
  }
}//GOLDrawCallback
```

Como podemos observar, estas dos funciones, nos van a permitir organizar nuestro código del programa llamando a la función concreta para crear una serie de objetos o bien para actualizar estos, mediante el uso de los estados de pantalla.

7.3.3. FUNCIONAMIENTO DEL PROGRAMA

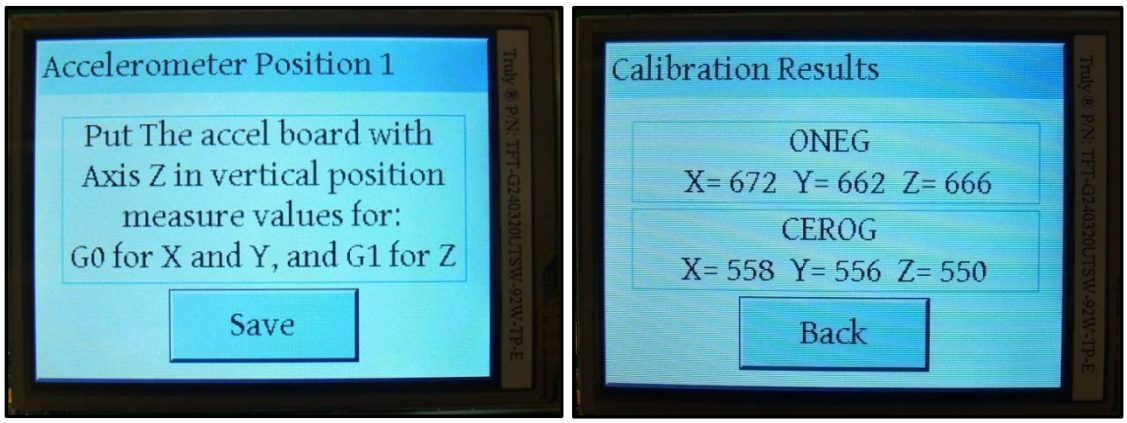
Una vez aclarada la estructura, vamos a ver que realiza cada una de las pantallas mencionadas anteriormente.

La pantalla principal es la pantalla mediante la cual vamos a poder acceder a las demás presionando el botón que queramos, sobre esta, también se ha incorporado el logo de la UMH en la parte superior izquierda de la misma. A continuación mostramos una imagen de esta pantalla principal.

Figura 7.11: Pantalla principal, Programa "*APP.c*"

Lo primero que tenemos que realizar una vez conectamos a corriente el sistema es calibrar el acelerómetro, primer botón ("Calibrate"), tal y como se ha detallado en el Anexo C. Para configurarlo tenemos que colocar el acelerómetro en tres posiciones, las cuales nos las irán indicando por pantalla, de tal forma que una vez colocado en la posición descrita le daremos a "Save", para que guarde los valores que necesitamos obtener en los tres ejes. Esos valores que tenemos que obtener son, el valor del acelerómetro cuando se detecta una aceleración de 1 g, así como el valor cuando la aceleración en cada eje es de 0 g, offset.

Una vez calibrado el sensor se mostrará por pantalla los valores obtenidos para cada eje cuando la aceleración medida es de 1 g y de 0 g, que son los parámetros a calibrar. En la siguiente figura mostramos la pantalla de la primera posición a calibrar así como los resultados obtenidos de la calibración.



a) b)

Figura 7.12: Calibración del sensor, Programa "APP.c" a) Primera posición a calibrar. b) Resultados de Calibración.

Si presionamos sobre el segundo botón (“Read”), vamos a poder leer los valores que obtenemos a través del modulo A/D respecto cada eje en tiempo real. Como podemos observar, al ir variando el acelerómetro de posición, los valores van a variar entre el dato de calibración obtenido para cada eje cuando la aceleración es de 1 g y entre el doble de la diferencia entre el valor para 1 g y el valor obtenido para cada eje en la calibración de 0 g, es decir, el rango de valores estará comprendido entre:

(1g,-1g=1g-2*(1g-0g))

A continuación mostramos esta pantalla en la siguiente figura:

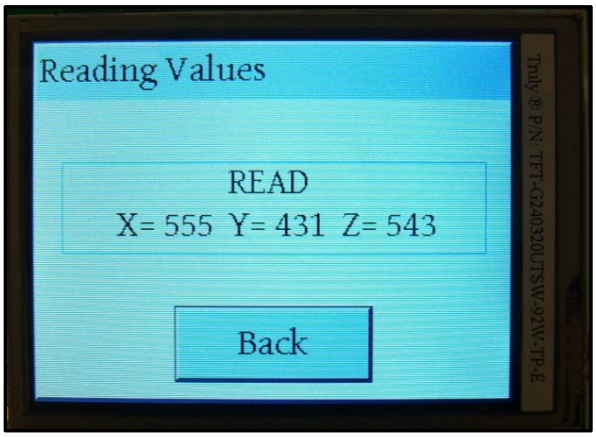


Figura 7.13: Lectura de valores, Programa “*APP.c*”.

La tercera pantalla (“Display”) nos permite visualizar de una forma gráfica el valor en tiempo real de cada eje mientras que observamos el valor de cada uno en la parte superior. El “*slide*” se generará de acuerdo a los parámetros de calibración, estableciendo sus valores máximos y mínimos de acuerdo al rango visto anteriormente (Figura 7.14).

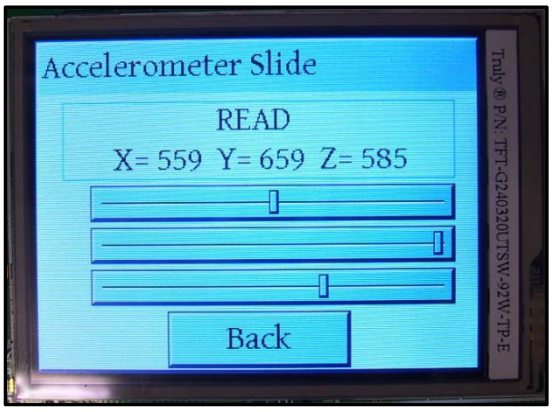
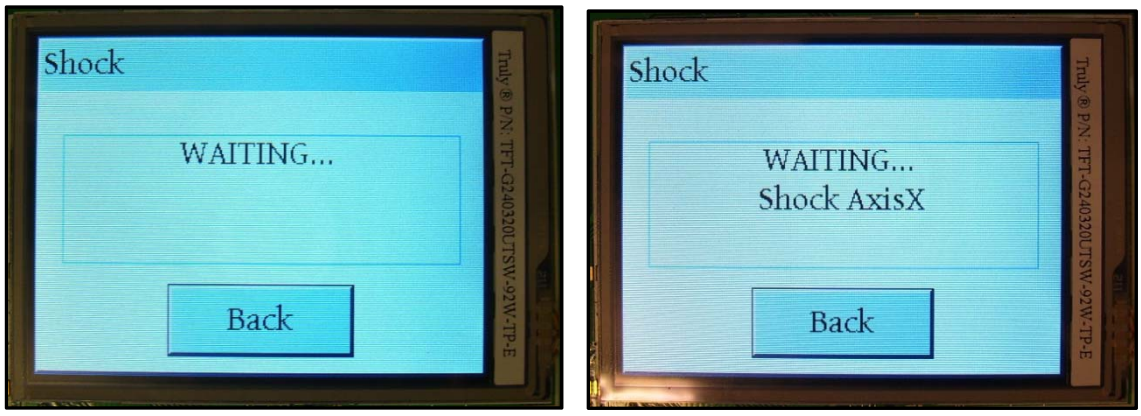


Figura 7.14: Pantalla “display”, Programa “*APP.c*”.

La cuarta pantalla (“Shock”) nos va a detectar golpes en cada uno de los ejes, es decir, el programa está a la espera de que se produzca un cambio brusco en los valores de cada eje una vez que golpeemos alguno de ellos. Una vez existe un shock (golpe), se mostrará por pantalla el eje en el que se ha producido. La mayor o menor sensibilidad ante estos golpes se puede controlar mediante la variable “*umbral”*, un valor mayor se corresponderá con un golpe mayor para que este sea detectado. Si se coloca un valor excesivamente bajo, cualquier mínimo movimiento del sensor, se corresponderá con un golpe mientras que no ha sido así. A continuación mostramos una imagen de esta pantalla.



a) b)

Figura 7.15: Pantalla “shock”, Programa “*APP.c*” a) Esperando a un shock en alguno de los ejes. b) Shock producido en el ejeX.

Por otra parte, si pulsamos sobre el botón (“Angles”), vamos a poder visualizar los ángulos de cada eje respecto a su posición de offset (aceleración igual a 0 g), es decir, cuando el valor de la aceleración sea máxima (1 g) marcará 90 grados y cuando sea mínima (-1 g) -90 grados. Para representar los grados negativos y positivos de cada eje hemos utilizado el objeto “*meter”*, sin embargo, hemos tenido que modificar su librería pues esta estaba realizada para funcionar únicamente con números positivos, creando la nueva librería *“meter_sens.h”* y *“meter_sens.c”.* Por tanto para poder usar este nuevo objeto, tendremos que colocar estos dos archivos anteriores en las carpetas en las que se encuentran los archivos “.c” y “.h” de nuestros objetos. Además, en el archivo *“Graphics.h”,* hay que modificar la librería *“meter.h”* anterior por la nueva que hemos creado, *“meter_sens.h”.* También, en la parte superior de la pantalla, se muestra el ángulo de cada eje mediante un texto dinámico (Figura7.16).

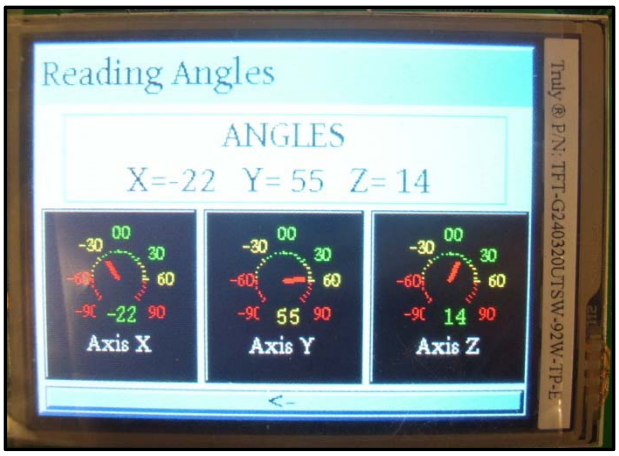
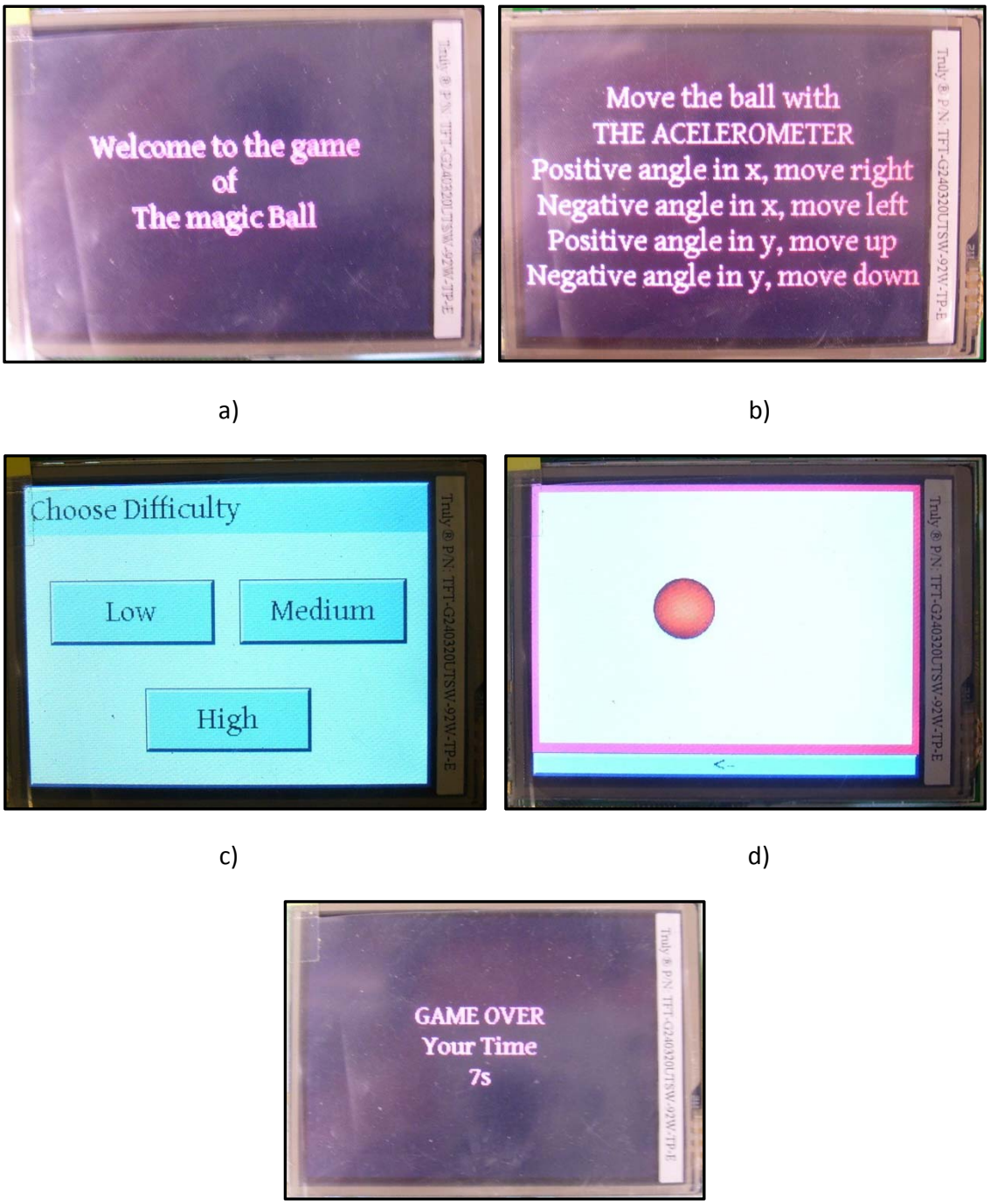


Figura 7.16: Lectura de Ángulos, Programa "*APP.c*".

Por último, si presionamos sobre el último botón ("Game"), vamos a poder ver el juego creado de una pelota controlada por el acelerómetro. El objetivo es mantener la bola dentro de un cuadro rojo, una vez que sobrepasemos este recuadro el juego habrá terminado y podremos comenzar de nuevo.

Por tanto, esta bola se moverá en función de la inclinación de los ejes X e Y, de tal forma que el eje X controlará la posición horizontal de la bola y el eje Y la posición vertical de la misma. Es decir, si el ángulo del eje Y es positivo, la bola se moverá hacia arriba, por el contrario si es negativo hacia abajo. De la misma manera sucederá con el eje X, si este es negativo se moverá hacia la izquierda y si es positivo hacía la derecha. Además, dependiendo de la inclinación del sensor, la bola se moverá a una velocidad u otra, a mayor inclinación con respecto a cualquiera de los dos ejes, la bola se moverá más rápida en esa dirección.

Primeramente, el juego consta de un mensaje de bienvenida y de una explicación de las instrucciones del mismo (Figura 7.17a y Figura 7.17.b respectivamente), como se mueve la pelota con el acelerómetro. Una vez leídas las instrucciones se nos mostrará automáticamente una pantalla para elegir el nivel de dificultad del mismo (Figura 7.17.c). Existen tres niveles de dificultad, nivel alto, medio y bajo, de tal manera que a mayor nivel más difícil será controlar la pelota dentro de nuestro cuadrado. Elegido el nivel de dificultad se nos mostrará por pantalla que el juego está a punto de comenzar, momento a partir del cual podremos comenzar a mover la pelota dentro de nuestra pantalla (Figura 7.17.d). El juego finalizará una vez toquemos una de las paredes de nuestro recuadro, indicándonos el tiempo total que hemos estado jugando en segundos (Figura 7.17.e). A continuación mostramos 5 imágenes del juego correspondiéndose a cada una de las partes del mismo.



a) b)

c) d)

e)

Figura 7.17: Juego de la pelota, Programa "*APP.c*" a) Mensaje de bienvenida al juego. b) Instrucciones del juego. c) Selección del nivel de dificultad. d) Pantalla del juego. e) Mensaje final con el tiempo en ejecución.

Al igual que el programa anterior, el código completo de este programa se puede encontrar en el CD adjunto al presente proyecto en la siguiente ubicación "APLICACIONES_FINALES/PANTALLA_TACTIL+ACELEROMETRO/APP", pues este tampoco se ha imprimido debido a su extensión.

8. CONCLUSIONES Y TRABAJOS FUTUROS

CAPÍTULO 8. CONCLUSIONES Y TRABAJOS FUTUROS

El presente Proyecto Fin de Carrera se ha centrado en el desarrollo de aplicaciones prácticas mediante la programación de microcontroladores PIC de gama alta.

Este proyecto presenta los equipos, herramientas de Microchip, programas y documentos necesarios para implementar el control de una pantalla táctil con un microcontrolador PIC32. Además, hemos pretendido destacar la facilidad con la que es posible trabajar con un microcontrolador de esta potencia.

Por lo expresado anteriormente la elaboración del presente proyecto sirve para comprender un poco más la teoría de los microcontroladores y su aplicación; de ahí la importancia del mismo, donde se puede apreciar lo interesante que es la programación en nuestra vida cotidiana.

Como trabajo futuro o continuación del presente proyecto y tras el resultado bastante satisfactorio de las pruebas realizadas, se abre la puerta para la realización de aplicaciones más sofisticadas.

En concreto se ha pensado en la adquisición de video a través de la placa de expansión USB [21], utilizando el PIC32MX460F512L. La figura 8.1 muestra la placa de expansión USB la cual se puede conectar al sistema de desarrollo Explorer16 a través del conector PICtail plus y la figura 8.2 presenta un esquema del sistema propuesto. Sin embargo, habría que solucionar el problema de incompatibilidades al usar esta placa USB con la tarjeta gráfica [22].

Figura 8.1: USB PICtail Plus Daughter Board.

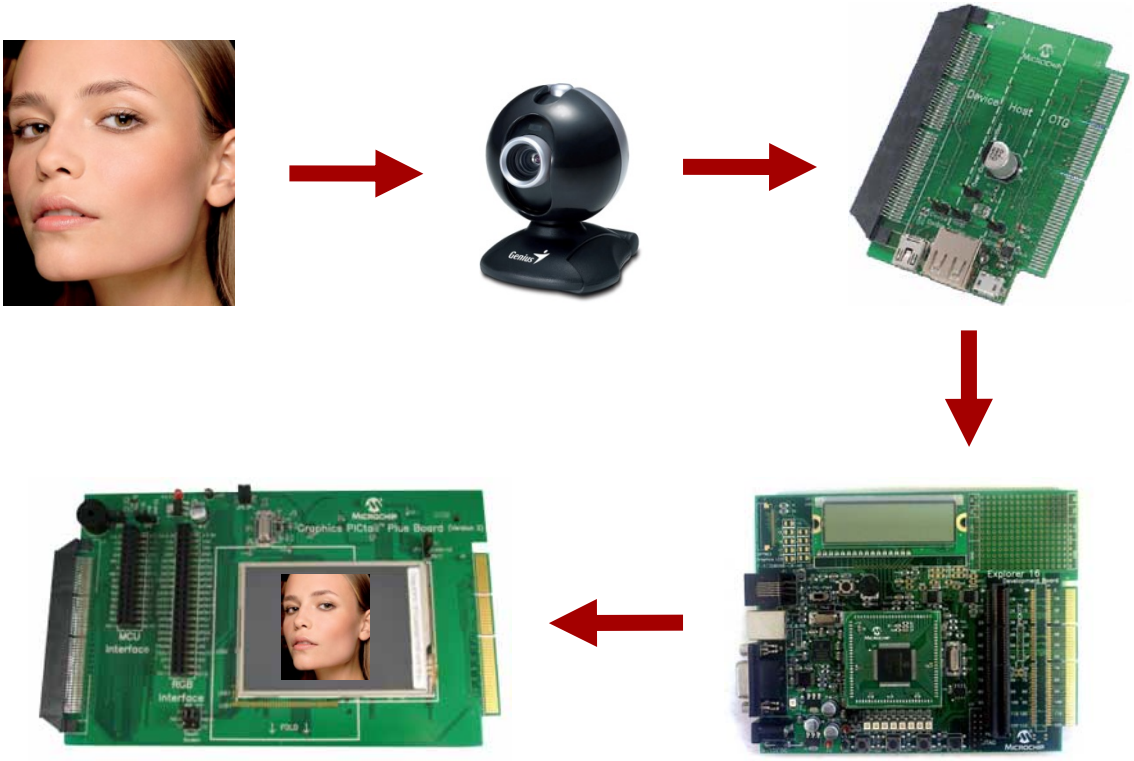


Figura 8.2: Estructura propuesta para la captación de video.

Finalmente, destacar que otra de las posibles ampliaciones es el desarrollo de programas para la teleoperación de un robot móvil (Moway) a través del dispositivo táctil usado en este proyecto.

Para concluir, recalcar que este proyecto fin de carrera puede servir como punto de referencia para el desarrollo de las ampliaciones indicadas, así como de otros trabajos futuros.

ANEXO A. DISEÑO DE LA MEMORIA DEL PIC32

ANEXO A. DISEÑO DE LA MEMORIA DEL PIC32

A.1.INTRODUCCIÓN

En el presente anexo, se describe la manera de configurar la memoria del PIC32.

A.1.1. REGISTROS DE CONTROL

Primero vamos a ver los registros SFRs que hay que usar para configurar la RAM así como las particiones de la memoria flash tanto para datos como código (en ambos modos, user y kernel).

Table 3-1: Memory Organization SFR Summary

Name		Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
BMXCON	31:24	—	—	—	—	—	BMX-CHEDMA	—	—
	23:16	—	—	—	BMXER-RIXI	BMXER-RICD	BMXER-RDMA	BMXER-RDS	BMXERRIS
	15:8	—	—	—	—	—	—	—	—
	7:0	—	BMXWS-DRM	—	—	—	BMXARB		
BMXCONCLR	31:0	Write clears selected bits in BMXCON, read yields undefined value							
BMXCONSET	31:0	Write sets selected bits in BMXCON, read yields undefined value							
BMXCONINV	31:0	Write inverts selected bits in BMXCON, read yields undefined value							
BMXDKPBA	31:24	—	—	—	—	—	—	—	—
	23:16	—	—	—	—	—	—	—	—
	15:8	BMXDKPBA<15:8>							
	7:0	BMXDKPBA<7:0>							
BMXDKPBACLR	31:0	Write clears selected bits in BMXDKPBA, read yields undefined value							
BMXDKPBASET	31:0	Write sets selected bits in BMXDKPBA, read yields undefined value							
BMXDKPBAINV	31:0	Write inverts selected bits in BMXDKPBA, read yields undefined value							
BMXDUDBA	31:24	—	—	—	—	—	—	—	—
	23:16	—	—	—	—	—	—	—	—
	15:8	BMXDUDBA<15:8>							
	7:0	BMXDUDBA<7:0>							
BMXDUDBACLR	31:0	Write clears selected bits in BMXDUDBA, read yields undefined value							
BMXDUDBASET	31:0	Write sets selected bits in BMXDUDBA, read yields undefined value							
BMXDUDBAINV	31:0	Write inverts selected bits in BMXDUDBA, read yields undefined value							
BMXDUPBA	31:24	—	—	—	—	—	—	—	—
	23:16	—	—	—	—	—	—	—	—
	15:8	BMXDUPBA<15:8>							
	7:0	BMXDUPBA<7:0>							
BMXDUPBACLR	31:0	Write clears selected bits in BMXDUPBA, read yields undefined value							
BMXDUPBASET	31:0	Write sets selected bits in BMXDUPBA, read yields undefined value							
BMXDUPBAINV	31:0	Write inverts selected bits in BMXDUPBA, read yields undefined value							

Table 3-1: Memory Organization SFR Summary (Continued)

Name		Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
BMXDRMSZ	31:24	BMXDRMSZ<31:24>							
	23:16	BMXDRMSZ<23:16>							
	15:8	BMXDRMSZ<15:8>							
	7:0	BMXDRMSZ<7:0>							
BMXPUPBA	31:24	—	—	—	—	—	—	—	—
	23:16	—	—	—	—	BMXPUPBA<19:16>			
	15:8	BMXPUPBA<15:8>							
	7:0	BMXPUPBA<7:0>							
BMXPUPBACLR	31:0	Write clears selected bits in BMXPUPBA, read yields undefined value							
BMXPUPBASET	31:0	Write sets selected bits in BMXPUPBA, read yields undefined value							
BMXPUPBAINV	31:0	Write inverts selected bits in BMXPUPBA, read yields undefined value							
BMXPFMSZ	31:24	BMXPFMSZ<31:24>							
	23:16	BMXPFMSZ<23:16>							
	15:8	BMXPFMSZ<15:8>							
	7:0	BMXPFMSZ<7:0>							
BMXBOOTSZ	31:24	BMXBOOTSZ<31:24>							
	23:16	BMXBOOTSZ<23:16>							
	15:8	BMXBOOTSZ<15:8>							
	7:0	BMXBOOTSZ<7:0>							

Figura A.1: Registros SFRs para configurar la memoria del PIC32.

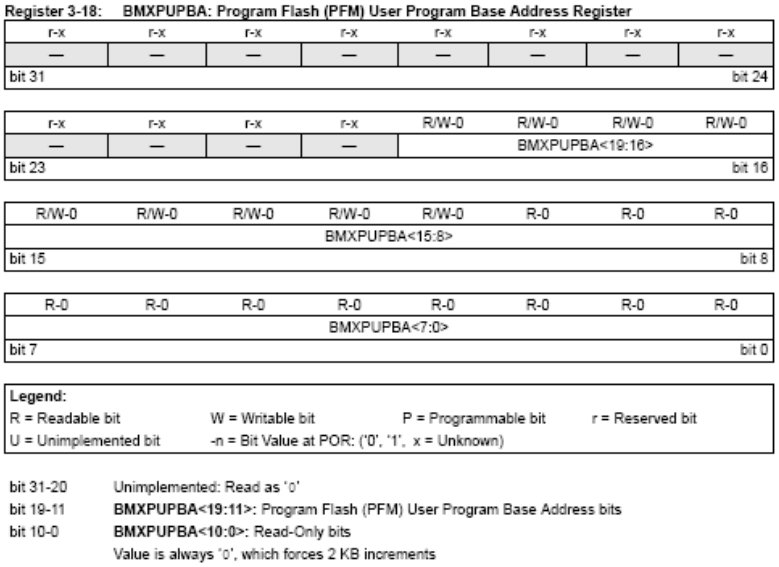
Todos los registros que terminan en CLR, SET o INV, son registros usados para la manipulación de los bits del registro que precede a su nombre. Por ejemplo, si tenemos el registro BMXCONCLR, pondrá a 0 los bits correspondientes al registro BMXCON que hayamos escrito como 1 en los bits de BMXCONCLR. Es decir, BMXCONCLR = 0x00000101 pondrá a 0 los bits 15 y 0 del registro BMXCON.

De igual manera funcionan los registros terminados en SET que pondrán a 1 los bits correspondientes, o los terminados en INV, que invertirán los bits.

Estas funciones nos permiten configurar los siguientes registros BMXCON, BMXDKPBA, BMXDUDBA, BMXDUPBA y BMXPIPBA, mediante el uso de sus correspondientes funciones clear, set o invert.

También existen registros de solo lectura como BMXDRMSZ, el cual indica el tamaño de la RAM de datos en bytes, en nuestro caso 0x00008000, que equivale a que nuestro dispositivo posee 32KB de RAM. Del mismo modo BMXPFMSZ, nos indica el tamaño de la memoria flash de programa (PFM) en bytes, en nuestro caso 0x00080000, dispositivo de 512KB. Por último, el registro BMXBOOTSZ, nos indica el tamaño de la memoria flash boot en bytes, 12KB.

Por otra parte, mediante el registro BMXPUPBA, vamos a ser capaces de definir la dirección de la base para el PFM en modo usuario. Para configurar la dirección tenemos 32 bits, no obstante, desde el bit 31 al 20 no están implementados. Además se ha forzado a que se tomen valores en incrementos de 2KB, ya que sus 11 primeros bits (10-0) son forzados a 0, por lo que solo tenemos del bit 19 al 11 (9 bits) disponibles para configurar nuestra dirección de la base de la memoria de programa flash. En la imagen se muestra el registro BMXPUPBA:



Register 3-18: BMXPUPBA: Program Flash (PFM) User Program Base Address Register

r-x	r-x	r-x	r-x	r-x	r-x	r-x	r-x
—	—	—	—	—	—	—	—
bit 31							bit 24

r-x	r-x	r-x	r-x	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	—	BMXPUPBA<19:16>			
bit 23							bit 16

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R-0
BMXPUPBA<15:8>							
bit 15							bit 8

R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
BMXPUPBA<7:0>							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	P = Programmable bit	r = Reserved bit
U = Unimplemented bit	-n = Bit Value at POR: ('0', '1', x = Unknown)		

bit 31-20 Unimplemented: Read as '0'
bit 19-11 **BMXPUPBA<19:11>**: Program Flash (PFM) User Program Base Address bits
bit 10-0 **BMXPUPBA<10:0>**: Read-Only bits
Value is always '0', which forces 2 KB increments

Figura A.2: Registro BMXPUPBA asociado a la memoria flash de programa.

A.2. DISEÑO DE LA MEMORIA PIC32

El PIC32 implementa dos espacios de direcciones una virtual y otra física. Todos los recursos hardware así como la memoria de programa, memoria de datos y periféricos están localizados con sus respetivas direcciones físicas. Mientras que las direcciones virtuales son exclusivamente usadas por la CPU para cargar, leer y ejecutar las instrucciones. Por tanto, las direcciones físicas son usadas por los periféricos como DMA y los controladores flash que acceden a la memoria independientemente de la CPU.

Los 4GB del espacio de direcciones virtuales se dividen en dos regiones primarias, espacio de usuario y kernel. Los 2GB más bajos del espacio del segmento de memoria del modo usuario, se llama useg/kuseg. Una aplicación en modo usuario debe residir y ejecutarse en este segmento el cual está también accesible para todas las aplicaciones en el modo kernel. Este es el porqué de que se llamen indistintamente useg y kuseg.

Por otra parte, los 2GB más altos ocupan el espacio de direcciones virtuales del kernel. Además, este espacio se divide a su vez en 4 segmentos de 512 MB cada uno, kseg 0, kseg 1, kseg 2 y kseg 3. Solo las aplicaciones en modo kernel pueden acceder a este espacio de memoria, el cual incluye todos los registros periféricos.

También hay que tener en cuenta que solo kseg 0 y kseg 1 señalan a recursos de memoria reales mientras que el segmento kseg 2 solo esta accesible para el EJTAG del debugger. Por tanto, el PIC32 solo usa los segmentos kseg 0 y kseg 1, de manera que, Boot Flash Memory (BFM), Program Flash Memory (PFM), Data RAM Memory (DRM) y los registros periféricos están accesibles desde kseg 0 o bien desde kseg 1.

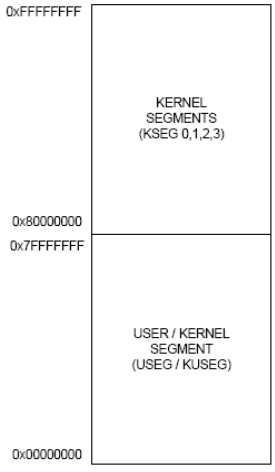


Figura A.3: División de la memoria del PIC32, regiones primarias.

Por otra parte, la unidad FMT es la encargada de trasladar los segmentos de memoria a sus correspondientes regiones de memoria física. Un segmento de la memoria virtual puede ser cacheada, lo único que esta solo se puede realizar en los segmentos kseg 0 o useg, ya que el segmento de memoria kuseg 1 no lo es.

Además, los segmentos kseg 0 y kseg 1 son siempre trasladados a la dirección física 0x0. Esto permite que la CPU acceda a las mismas direcciones físicas desde dos direcciones virtuales separadas, una desde kseg 0 y la otra desde kseg 1. Como resultado, la aplicación puede elegir ejecutar el mismo trozo de código desde la memoria cache o no (tal y como hemos visto antes). No obstante hay que tener en cuenta que los periféricos del PIC32 son solo visibles a través del segmento kseg 1, no cache.

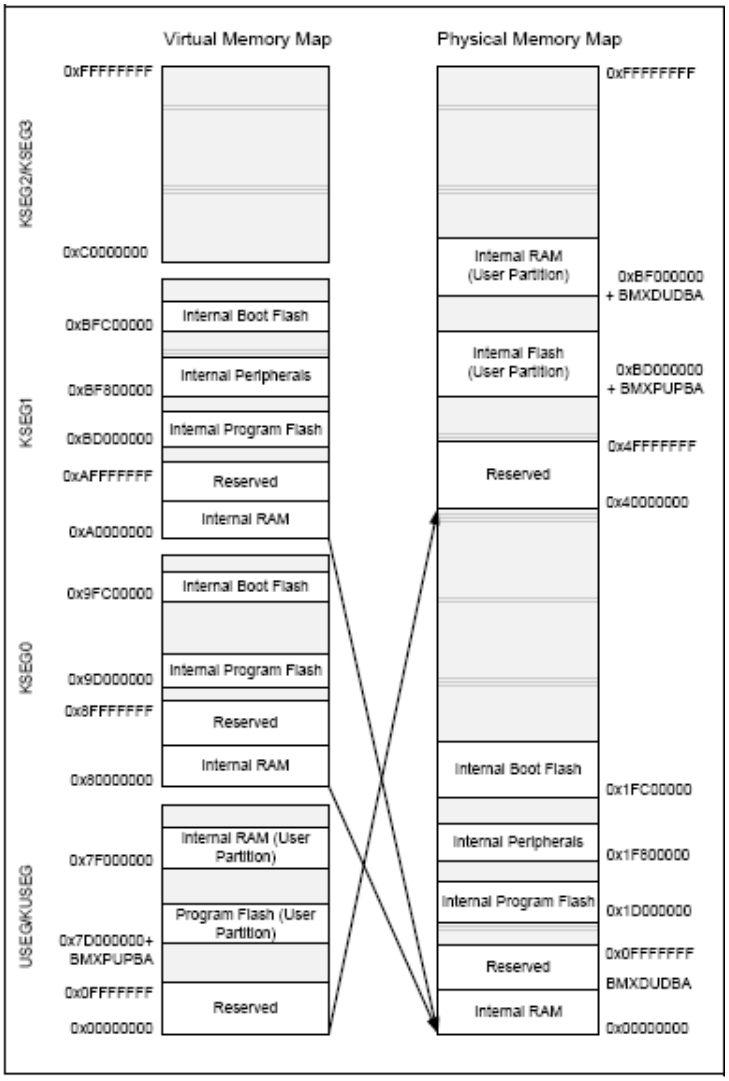


Figura A.4: Mapeo de la memoria virtual a la física.

A.2.1. CÁLCULO DE LA DIRECCIÓN FÍSICA A VIRTUAL Y VICEVERSA

A continuación vamos a describir las operaciones que hay que realizar para conocer una dirección física o virtual en función de la dirección de la cual partamos:

- Para transformar una dirección del kernel (KSEG0 o KSEG1) a una dirección física, hay que realizar una operación AND de la dirección virtual con 0x1FFFFFFF.
- Para pasar una dirección física a una dirección virtual KSEG0, hay que realizar una operación OR con la dirección física y 0x80000000.
- Para pasar una dirección física a KSEG1, hay que realizar una operación OR con la dirección física y 0xA0000000.
- Para pasar una dirección de KSEG0 a KSEG1, hay que realizar una operación OR con la dirección virtual de KSEG0 y 0x20000000.

A.2.2. PARTICIÓN DE LA MEMORIA FLASH DE PROGRAMA

Cuando hay un reset, la partición en modo usuario no existe, por tanto BMXPUPBA es inicializado a 0 y toda la memoria de programa flash es mapeada al modo kernel en el espacio de programa, empezando por la dirección virtual en KSEG1: 0xBD000000 (o KSEG0, 0x9D000000). Para establecer una partición para el modo de usuario, hay que inicializar BMXPUPBA como: BMXPUPBA=BMXPFMSZ-USER_FLASH_PG_SZ. Como hemos visto en nuestro caso BMXPFMSZ0 = 0x00800000, por tanto:

BMXPUPBA=0x00800000-USER_FLASH_PG_SZ, donde USER_FLASH_PG_SZ es el tamaño de la partición del programa en modo usuario.

Ejemplo, queremos diseñar una partición de 20KB (0x5000), por lo que tendremos que realizar la siguiente operación:

BMXPUPBA=0x00800000-0x5000=0x7B000.

Y por tanto la dirección inicial y final de la partición de la memoria flash en modo usuario será:

Mientras que la partición en el modo kernel vendrá condicionada a 512KB-20KB=492KB. Entonces tendremos que para las dos particiones, su rango de direcciones será:

A.2.3. PARTICIÓN DE LA RAM

La memoria RAM se puede dividir en 4 particiones distintas:

- Datos Kernel
- Programa Kernel
- Datos Usuario
- Programa Usuario

Con el fin de ejecutar los datos de la RAM, habrá que definir previamente la partición del programa usuario o kernel. En este caso también, cuando hay un reset, toda la RAM de datos se asigna a la partición de datos del Kernel. Para realizar la partición de la RAM, tendremos que programar los siguientes registros BMXDKPBA, BMXDUDBA y BMXDUPBA. Además, tal y como hemos visto antes, el tamaño de la RAM viene dado por BMXDRMSZ, que en nuestro PIC es de 32KB.

Consideraciones sobre la partición de la memoria RAM:

- La partición de la RAM de programa en modo kernel es requerida siempre que sea necesario ejecutarse el programa desde la RAM de datos en este modo. Hay que tener en cuenta que tras un reset esta parición no existe.
- Por otra parte, habrá que crear una partición de la RAM de datos en modo usuario, para que las aplicaciones se ejecuten en este modo.Tampoco existe tras un reset.

- La partición de la RAM de programa en modo usuario es requerida por el mismo motivo que el primer punto, en caso de que sea necesario ejecutar el programa desde la RAM de datos en modo usuario. Al igual que las anteriores no existe después de un reset.

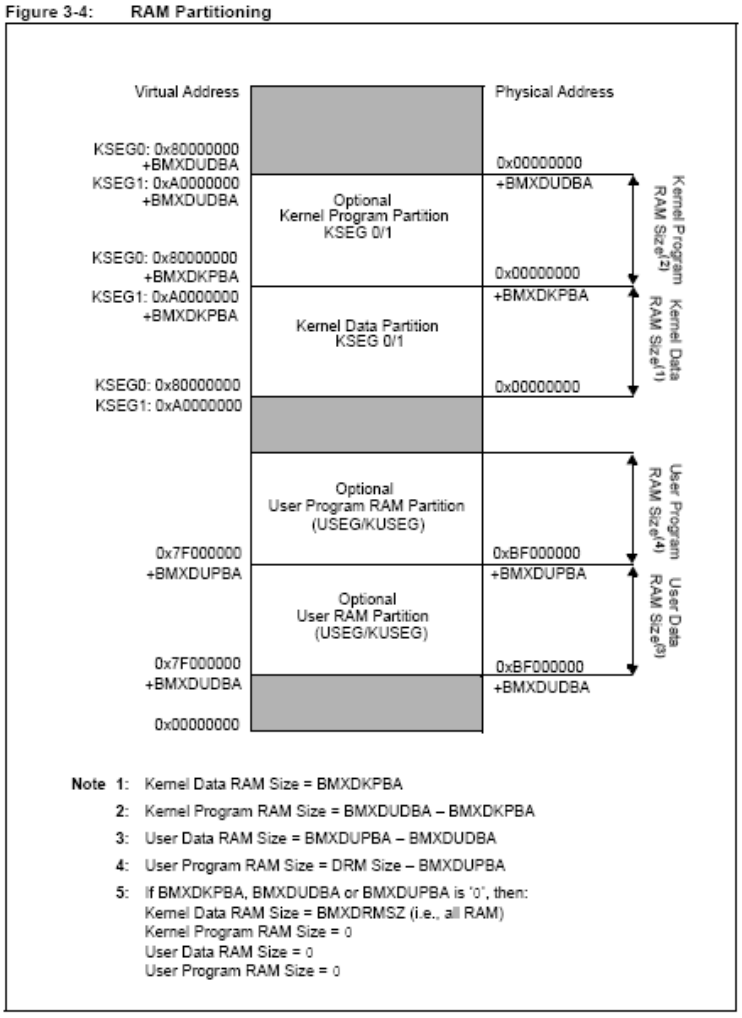


Figura A.5: Esquema de direcciones para la partición de la memoria RAM.

Para más información sobre la memoria del PIC32 se puede consultar el Capítulo 3 "Memory Organization" del documento "*PIC32MX Family Reference Manual*" o bien el Capítulo 6 "Memory Organization" del documento "*PIC32PIC32MX3XX/4XX Family Data Sheet*", disponibles ambos en la página web de Microchip así como en el CD adjunto al presente proyecto.

ANEXO B. CONSIDERACIONES PRÁCTICAS PARA PROGRAMAR EL PIC32

ANEXO B. CONSIDERACIONES PRÁCTICAS PARA PROGRAMAR EL PIC32

B.1. INTRODUCCIÓN

En el presente anexo, se van a describir distintos aspectos a tener en cuenta a la hora de programar sobre el PIC32 que no se han comentado en ninguno de los capítulos del proyecto.

B.2. VARIABLES

En este apartado vamos a ver la importancia que tiene el usar distintas variables para la ejecución de los programas.

A la hora de programar mediante el MPLAB C32, podemos elegir entre 10 tipos de datos enteros diferentes, char(8), short(16), int(32) long(32) y long long(64) con sus correspondientes variantes sin signo ("unsigned"), ver la siguiente tabla:

Tipo	Bits	Min	Max
char, signed char	8	-128	127
unsigned char	8	0	255
short, signed short	16	-32768	32767
unsigned short	16	0	65535
int, signed int, long, signed long	32	-2^{31}	$2^{31}-1$
unsigned int, unsigned long	32	0	$2^{32}-1$
long long, signed long long	64	-2^{63}	$2^{63}-1$
unsigned long long	64	0	$2^{64}-1$

Tabla B.1: Comparación de las variables enteras disponibles en el MPLAB C32.

Tal y como podemos observar en la tabla anterior, cuando el valor es con signo se va a dedicar un bit para evaluar el signo. También podemos comprobar que int y long son sinónimos pues ambos ocupan 32 bits (4 bytes). De hecho, tal y como hemos visto en el capítulo 2, las operaciones con valores 8, 16 bits o 32 bits se procesan en el mismo tiempo por la ALU, sin embargo, cuando las definamos tipo long el tiempo en procesar una operación va a aumentar considerablemente así como la cantidad de RAM ocupada por cada uno de ellos. Veámoslo con un ejemplo:

```
main (){
int i1, i2, i3;
long long ll1, ll2, ll3;

i1=1234; //test para enteros int de 32 bits
i2=5678;
i3=i1*i2;

ll1= 1234; // test para enteros long long de 64 bits
ll2= 5678;
ll3=ll1*ll2;
} //main
```

Una vez compilado el proyecto abrimos el código generado por el compilador ("Disassembly Listing") para observar la diferencia entre el código necesario para ejecutarse con la variable entera tipo int y tipo long long.

```
11:             i1=1234; //test para enteros int de 32 bits
9D000024  240204D2   addiu      v0,zero,1234
9D000028  AFC20000   sw         v0,0(s8)
12:             i2=5678;
9D00002C  2402162E   addiu      v0,zero,5678
9D000030  AFC20004   sw         v0,4(s8)
13:             i3=i1*i2;
9D000034  8FC30000   lw         v1,0(s8)
9D000038  8FC20004   lw         v0,4(s8)
9D00003C  70621002   mul        v0,v1,v0
9D000040  AFC20008   sw         v0,8(s8)
14:
15:             ll1= 1234; // test para enteros long long de 64 bits
9D000044  240204D2   addiu      v0,zero,1234
9D000048  00001821   addu       v1,zero,zero
9D00004C  AFC20010   sw         v0,16(s8)
9D000050  AFC30014   sw         v1,20(s8)
16:             ll2= 5678;
9D000054  2402162E   addiu      v0,zero,5678
9D000058  00001821   addu       v1,zero,zero
9D00005C  AFC20018   sw         v0,24(s8)
9D000060  AFC3001C   sw         v1,28(s8)
17:             ll3=ll1*ll2;
9D000064  8FC30010   lw         v1,16(s8)
9D000068  8FC20018   lw         v0,24(s8)
9D00006C  00620019   multu      v1,v0
9D000070  00002012   mflo       a0
9D000074  00002810   mfhi       a1
9D000078  8FC30010   lw         v1,16(s8)
9D00007C  8FC2001C   lw         v0,28(s8)
9D000080  70621802   mul        v1,v1,v0
9D000084  00A01021   addu       v0,a1,zero
9D000088  00431021   addu       v0,v0,v1
9D00008C  8FC60018   lw         a2,24(s8)
9D000090  8FC30014   lw         v1,20(s8)
9D000094  70C31802   mul        v1,a2,v1
9D000098  00431021   addu       v0,v0,v1
9D00009C  00402821   addu       a1,v0,zero
9D0000A0  AFC40020   sw         a0,32(s8)
9D0000A4  AFC50024   sw         a1,36(s8)
```

Como podemos observar para el caso de las variables tipo long long, el código generado es más grande que para las variables tipo int, ya que para realizar la operación de multiplicación se requiere el uso de más instrucciones para llevarla a cabo. Esto es debido a que, como ya comentamos en el capítulo 2, la ALU puede ejecutar operaciones de 32bits de una sola vez. Sin embargo, para operaciones con datos de 64 bits, en realidad, estas se ejecutan como una secuencia de multiplicaciones y sumas de 32 bits.

Por otra parte, para el caso de una operación de división obtendríamos el mismo resultado. El código para las variables enteras de tipo char, short e int, sería el mismo, mientras que para la variable entera tipo long long, el espacio requerido aumenta considerablemente respecto a las anteriores, pues para realizar dicha operación es necesario llamar a la subrutina jal, la cual está dentro de la librería "*libgcc2.c*".

En cuanto a las variables fraccionales existen tres tipos:

Tipo	Bits
Float	32
Double	64
Long double	64

Tabla B.2: Comparación de las variables fraccionales disponibles en el MPLAB C32.

No existe ninguna diferencia entre estas dos últimas, sin embargo, hay que tener cuidado al declarar una variable float, ya que en el PIC32 no existe una unidad especial que trate los datos tipo float y por tanto tiene que ser compilada usando librerías aritméticas, aumentando el código a usar. Esto se traduce en más memoria necesaria para ejecutar el programa.

Vamos a realizar un análisis temporal de la función de multiplicación dependiendo de qué variable esté implicada en dicha operación, el código a analizar es el siguiente ("*Analisis_Temporal.c*"):

```
main (){
char c1, c2, c3;
short s1, s2, s3;
int i1, i2, i3;
long long ll1, ll2, ll3;
float f1, f2, f3;
long double d1, d2, d3;
c1=12; //test para enteros char de 8 bits
c2=34;
c3=c1*c2;
s1=1234; //test para enteros short de 16 bits
s2=5678;
s3=s1*s2;
i1=1234567; //test para enteros long de 32 bits
i2=3456789;
i3=i1*i2;
```

```
ll1= 1234; // test para enteros long long de 64 bits
ll2= 5678;
ll3=ll1*ll2;
f1= 12,34; //test para single float
f2=56,78;
f3=f1*f2;
d1=12,34; //test para double float
d2=56,78;
d3=d2*d1;
} //main
```

En la tabla B.3 podemos ver de forma más detalla el análisis temporal realizado para una frecuencia de simulación de 64MHz. Las dos últimas columnas nos muestran la diferencia relativa respecto a la variable int o float.

Como podemos observar, las operaciones hasta 32 bits de las variables enteras son hasta 3 veces más rápidas que cuando usamos el entero tipo long long. Evidentemente las operaciones que usan las variables tipo float requieren de más tiempo para ejecutarse, llegando a duplicar el tiempo necesario respecto a esta última para ejecutar una operación con formato long double.

Por tanto podemos decir que la elección de las variables nos va a afectar tanto al tamaño del código del programa como a la velocidad de ejecución de este.

Tipo	Bits	Cycle Count	Tiempo (µs)	Respecto a int	Respecto a float
char	8	6	0.09375	1	-
Short	16	6	0.09375	1	-
Int, long	32	6	0.09375	1	-
Long long	64	21	0.328125	3.5	-
Float	32	51	0.796875	8.5	1
Long double	64	97	1.515625	16.5	2

Tabla B.3: Análisis temporal de las diferentes variables, multiplicación.

También se ha llevado a cabo un analisis temporal del programa anterior sustituyendo la operación multiplicación por una división (“*Analisis_Temporal_II.c”*), cuyos resultado son los siguientes:

Tipo	Bits	Cycle Count	Tiempo (µs)	Respecto a int	Respecto a float
char	8	17	0.265625	0.5	-
Short	16	24	0.375000	0.75	-
Int, long	32	31	0.484375	1	-
Long long	64	67	1.046875	2.2	-
Float	32	80	1.250000	2.6	1
Long double	64	163	2.546875	5.25	2

Tabla B.4: Análisis temporal de las diferentes variables, división.

B.3. INTERRUPCIONES

Tal y como hemos comentado en el apartado 2.2.3.7, las interrupciones proporcionan un mecanismo para el control en tiempo real, permitiendo a las aplicaciones tratar con eventos externos asíncronos, las cuales requieren de una atención rápida por parte de la CPU.

El PIC32 proporciona 64 recursos distintos de interrupciones (Consultar la siguiente referencia para conocer estos recursos [3]). De tal forma, que cada recurso de interrupción puede tener un trozo de código único, llamando a la rutina de servicio de interrupción (ISR) asociada a él, proporcionando de esta manera la respuesta requerida. Por tanto, las interrupciones pueden ser ejecutadas desde cualquier punto en un orden impredecible, cambiando la instrucción actual temporalmente para ejecutarse un procedimiento especial.

Las interrupciones tienen que ser capaces de salvar el contexto del procesador antes de tomar cualquier acción y cargarla después, tal y como estaba antes de que ocurriese la misma. Estas acciones las realiza el compilador MPLAB C32, sin embargo existen una serie de limitaciones que hay que tener en cuenta:

- Las funciones de servicio de la interrupción no devuelven ningún valor, son de tipo void.
- No se pueden pasar parámetros a la función.
- No pueden ser directamente llamadas por otras funciones.
- No deberían llamar a otras funciones, como recomendación para una mayor eficiencia del programa.

Como hemos comentado, existen 64 recursos de interrupción los cuales a su vez pueden generar distintos eventos de interrupción, de tal forma que en total se dispone de 96 eventos diferentes que pueden ser controlados por el PIC32. Evidentemente, cuando varios recursos de interrupción están habilitados, es necesario que la ISR identifique cual ha sido la interrupción que ha ocurrido para que se ejecute el trozo de código correcto. Para lo cual se van a utilizar varios flags en distintos registros.

Cada recurso de interrupción tiene 7 bits de control asociados, agrupados en varios registros especiales, estos son:

- Bit de habilitación de la interrupción, (“Interrupt Enable”,IE). Cuando está a 1, se podrá procesar la interrupción.

- Bit de interrupción (“Interrupt flag”,IF), se activa una vez ocurre el evento esperado por la interrupción. Debe ser puesto a 0 por el usuario antes de salir de la ISR.
- Nivel de prioridad (“Group Priority level”, IP). Existen 7 niveles de prioridad (desde ipl1 a ipl7, 3 bits) de tal forma que si dos interrupciones ocurren al mismo tiempo se ejecutará la que tenga un mayor nivel de prioridad.
- Nivel de subprioridad. Existen 4 niveles más de prioridad (2 bits) para un grupo de prioridad en concreto. De tal forma que si dos eventos ocurren simultáneamente y estos tienen la misma prioridad, el que tenga una subprioridad más alta será el seleccionado. En el caso de no configurar las prioridades existen unas por defecto (consultar el “*Data Sheet*”).

Para configurar y controlar las interrupciones vamos a usar las siguientes librerías, *plib.h* y *int.h.* Entre las principales funciones de estas librerías podemos encontrar:

- INTEnableSystemSingleVectoredInt(); Esta función sigue una secuencia de inicialización del modulo de control de la interrupción para habilitar el manejo de las interrupciones del PIC32.
- mXXSEetIntPriority(x); Asigna un nivel de prioridad al recurso de interrupción seleccionado en XX.
- mXXClearIntFlag(); permite poner a cero el flag IF, bandera que nos detecta que recurso de interrupción ha sido habilitado.

Vamos a ver un ejemplo (“*Interrupciones.c*”) usando las funciones anteriores. Para ello en el primer programa vamos a usar una sola interrupción, la cual va será llamada cada vez que se desborde el timer2. Para ello habilitamos el modulo del Timer2 con una cuenta de 15, de tal forma que cuando se produzca el desbordamiento, será atendida por la rutina de servicio de interrupción (ISR), incrementando el valor de la variable “count”. A continuación mostramos el código del programa empleado:

```
#include <p32xxxx.h>
#include <plib.h>

int count;
#pragma interrupt InterruptHandler ipl1 vector 0

void InterruptHandler ( void){
count++;
mT2ClearIntFlag();
}// Interrupt Handler
```

```
main(){
//Timer
PR2=15;
T2CON= 0x8030;
//Interrupcion
mT2SetIntPriority(1);
INTEnableSystemSingleVectoredInt();
mT2IntEnable(1); (*)
//Bucle
while(1);
}//main
```

A la hora de realizar el programa anterior hay que tener en cuenta dos cosas:

- Antes de habilitar la interrupción hay que declararla completamente (*).
- La prioridad asignada a la interrupción debe coincidir con la sintaxis en la declaración de esta, remarcado en el programa anterior en negrita.

En el caso de no realizar correctamente los dos puntos anteriores nuestro programa no funcionaría correctamente.

Si abrimos la ventana watch del MPLAB y colocamos la variable count, podremos ver como el valor de esta se va incrementando cada vez que el timer2 alcanza el valor seleccionado.

Como segundo programa vamos a usar dos recursos de interrupción asignando a cada uno de ellos diferentes niveles de prioridad. La prioridad decidirá cuál de las dos interrupciones será atendida primero si ocurren de forma simultánea. Sin embargo, cuando una de ellas se esté ejecutando, la otra tendrá que esperar a que esta termine para que se pueda ejecutar.

No obstante, puede darse el caso que se esté ejecutando una interrupción de un nivel de prioridad bajo, pero que una interrupción con una prioridad superior requiera una atención inmediata por parte del programa, interrumpiendo la ejecución de la primera interrupción. Una vez finalizada la ejecución de la interrupción con un nivel de prioridad más alto volverá a la interrupción de nivel más bajo para terminarla, esto es lo que se conoce como nesting.

En el siguiente programa vamos a ver de qué manera funciona el nesting, para lo cual usamos la función del MIPS "*Asm(ei)*" la cual nos va a permitir anidar llamadas a interrupciones, de lo contrario se ejecutarían secuencialmente. El código del programa ("*Interrupciones_II.c*") se muestra a continuación:

```
#include <p32xxxx.h>
#include <plib.h>

int count;
void __ISR(0, ipl1) InterruptHandler ( void){
// 1.- Rehabilitamos las interrupciontes (nesting)
asm("ei");

// 2.- Chequeamos primeramente la prioridad más alta
if(mT3GetIntFlag()){
count++;
//Ponemos a cero el flag
mT3ClearIntFlag();
}// _T3

// 3.- Chequeamos la de más baja prioridad
else if (mT2GetIntFlag()){
//pasamos aquí el tiempo
while(1);
//quitamos la bandera
mT2ClearIntFlag();
}// _T2
}// Manejo de las interrupciones

main(){
// 4.- Inicializamos los timers
PR3=20;
PR2=15;
T3CON=0x8030;
T2CON=0x8030;

// 5.- Inicializamos las interrupciones
mT2SetIntPriority(1);
mT3SetIntPriority(3);
INTEnableSystemSingleVectoredInt();
mT2IntEnable(1);
mT3IntEnable(1);
//main loop
while(1);
}//main
```

Lo que realiza este programa es lo siguiente: una vez inicializado los Timers, asignado las prioridades a las interrupciones (nivel 1 para el timer2 y nivel 3 para el Timer3) y habilitadas las mismas, entra en un bucle hasta que ocurre la primera interrupción, la del Timer2 ya que su cuenta es menor. Por tanto se ejecutará el código de la interrupción del Timer2. Sin embargo, la ejecución de esta equivale a un bucle infinito, por lo que una vez que el Timer3 alcance su cuenta, y al haber empleado el nesting y tener una prioridad mayor, interrumpirá la ejecución de la interrupción del Timer2 para ejecutar la del Timer3, incrementado el valor de la variable count. Una vez finalizada la ejecución de esta interrupción, volverá a ejecutar el código de la del Timer2, y así sucesivamente.

El mecanismo de funcionamiento del servicio de interrupción visto hasta el momento, es muy similar a como funciona en los anteriores microcontroladores de Microchip. Hemos utilizado una función de interrupción para ejecutar una rutina de servicio de la interrupción en función de la que haya sido habilitada. Esto se conoce con el nombre de "Single mode vector". Sin embargo, el PIC32 ofrece la posibilidad de usar "vectored interrupts" y múltiples registros de interrupción, para dotar de una respuesta más rápida a las interrupciones y evitar sobrecargas. En particular existen 64 vectores distintos de interrupción y 2 registros de 32 bits. Los 96 recursos de interrupciones que habíamos comentado que disponía la arquitectura del PIC32 están agrupados en los distintos vectores (consultar el "*Data Sheet*").

Esta nueva posibilidad de asignar un vector separado para cada grupo de recursos de interrupción elimina la necesidad de testear secuencialmente todos los posibles recursos de interrupción para encontrar el que necesita ejecutarse. Es decir:

- Single mode vector: Todas las respuestas de interrupción son ejecutadas desde el mismo vector de interrupción, vector 0.
- Multi-vector mode: Las respuestas de interrupción son ejecutadas desde el vector de interrupción correspondiente.

Por tanto, el prologo de una interrupción (secuencia de comandos que hay que ejecutar antes de pasar a la interrupción propiamente), se ve reducida con esta forma de controlar las interrupciones. Además, podremos seguir usando el nesting para ejecutar las interrupciones de alta prioridad de una forma más rápida.

Para programar las interrupciones usando este nuevo modo, tenemos que tener en cuenta que cada interrupción tiene su propia función, en la cual tendremos que colocar el número del vector, este se puede consultar en el "*Data Sheet*". Además, como ahora ya no hay que comprobar que interrupción ha sido la que ha incrementado su bandera, ya que se realiza implícitamente, cada función será llamada cuando se haya desbordado su bandera correspondiente. Y el último cambio es que para inicializar el "Multi-vector mode" tendremos que usar otra función de inicialización. Vamos a realizar estos cambios respecto al programa anterior, los cuales están remarcados en negrita, programa "*Multivectored.c*".

```
#include <p32xxxx.h>
#include <plib.h>

int count;
void __ISR( _TIMER_3_VECTOR , ipl7) T3InterruptHandler ( void){
// 1.- T3 incrementa la cuenta
count++;
// 2.- Ponemos a cero el flag
mT3ClearIntFlag();
}// _T3

void __ISR( _TIMER_2_VECTOR , ipl1) T2InterruptHandler ( void){
```

```
// 3.- Rehabilitamos las interrupciones (nesting)
asm("ei");

// 4.- Código del T2
while(1);
// 5.- quitamos la bandera y salimos
mT2ClearIntFlag();
}// _T2

main(){
// 6.- Inicializamos los timers
PR3=20;
PR2=15;
T3CON=0x8030;
T2CON=0x8030;
// 7.- Inicializamos las interrupciones
mT2SetIntPriority(1);
mT3SetIntPriority(7);
INTEnableSystemMultiVectoredInt();
mT2IntEnable(1);
mT3IntEnable(1);
//main loop
while(1);
}//main
```

B.3.1. MÓDULO RTCC

La familia de microntroladores PIC32MX dispone de un modulo RTCC (Real-Time clock and Calendar), el cual funciona en modo de bajo consumo y nos va a permitir generar interrupciones en el mes, día, hora, minuto y segundo que queramos una vez al año. Este modulo también dispone de otras características y más funcionalidades que se pueden consultar en el Capitulo 21 de la siguiente referencia [3].

Para configurarlo hay que acceder a una serie de registros en el orden correcto y rellenarlos con los datos adecuados. A continuación mostramos un ejemplo de cómo acceder a estos registros y configurar el modulo RTCC para generar una interrupción en la fecha que queramos:

```
RtccInit();            // inicializar el RTCC
 // Poner en hora
 rtccTime tm; tm.sec=0x15; tm.min=0x30; tm.hour=01;
 // Poner los datos actualese
rtccDate dt;
dt.wday=0; dt.mday=0x15; dt.mon=0x10; dt.year=0x07;
RtccSetTimeDate(tm.l, dt.l);
//  alarma deseada para Feb 29th
dt.wday=0; dt.mday=0x29; dt.mon=0x2;
RtccSetAlarmTimeDate(tm.l, dt.l);
```

B.3.2. OTRAS FUNCIONES ÚTILES

También puede darse el caso de que queramos que un trozo de nuestro código requiera que todas las interrupciones estén temporalmente deshabilitadas, para lo cual podremos usar los siguientes comandos:

```
Asm("di");
// Código protegido aquí
Asm("ei");
```

O bien podemos utilizar las siguientes dos funciones de la librería *plib.h* las cuales nos aportan información adicional respecto a las anteriores:

- *INTDisableInterrupts();* deshabilita las interrupciones y nos devuelve el valor correspondiente a los estados de las interrupciones antes de ejecutar el código que queremos proteger.
- *INTRestoreInterrupts(status*); restaura el estado del sistema original pasando como variable el estado devuelto por la anterior función.

Luego las interrupciones nos proporcionan una herramienta muy flexible para programas de control embebidos, destinadas a ayudar y manejar múltiples tareas mientras existe un control del tiempo así como de los recursos.

B.4. CONFIGURACIÓN DEL PIC32

En este apartado vamos a estudiar los distintos relojes del sistema y como configurarlos y de qué forma sacarle el mayor uso a la memoria cache del sistema, dos aspectos nuevos en la arquitectura hardware del PIC32.

Como se puede observar en el capítulo 2, concretamente en el apartado 2.2.6.3, existen hasta 5 osciladores para proporcionarnos una señal de entrada con la frecuencia, el consumo y la precisión que queramos.

En todas las aplicaciones nos va a interesar controlar tanto el rendimiento como el consumo. El consumo nos determina el tamaño y el coste de la potencia a suministrar. Mientras que el rendimiento nos determina cuanto trabajo puede realizar nuestra aplicación en un periodo de tiempo. Aumentar la velocidad del reloj aumentará el trabajo producido pero aumentara también el consumo del dispositivo.

Para ayudarnos a obtener un buen control del consumo, el modulo del reloj del PIC32 nos ofrece las siguientes características:

- Cambio en tiempo real entre oscilador interno y externo.
- Control en tiempo real de todos los divisores de los relojes
- Control en tiempo real del circuito PLL (multiplicador de reloj).
- Modos IDLE, donde la CPU se para y los periféricos continúan operando.
- Modo SLEEP, donde la CPU y los periféricos se paran y esperan a que un evento los despierte.
- Control separado sobre los relojes periféricos, de tal forma que el consumo de los módulos periféricos puede ser optimizado.

Para los programas que vamos a desarrollar a lo largo del presente proyecto, ejecutados tanto en la placa PC32 STARTER KIT como en la placa de desarrollo Explorer16, ambos tienen en cada una de ellas un reloj principal con una frecuencia de 8MHz. A esta frecuencia, ya que se encuentra por debajo de 10MHZ, se recomienda activar el oscilador primario para que opere en el modo XT.

Posteriormente, para poder usar los circuitos PLL hay que reducir esta frecuencia a 4MHz, ya que es a la máxima frecuencia a la que pueden operar estos circuitos. Una vez asignada esta frecuencia asignaríamos el factor de multiplicación del PLL y por último el factor de división de salida (todos estos valores se controlan a través de distintos bits del registro OSCCON). Dándonos nuestra frecuencia de reloj del sistema, tal y como podemos ver en la siguiente figura:

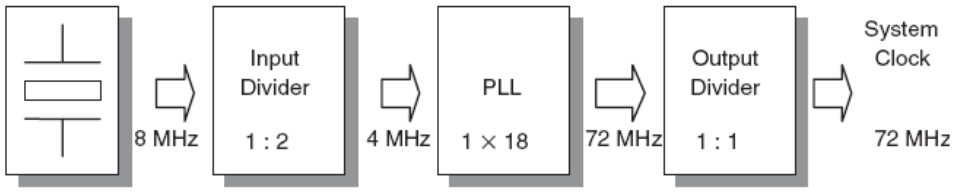


Figura B.1: Esquema de la configuración del reloj del sistema.

Por último, para poder configurar el bus del reloj periférico hay que enviar la señal del reloj del sistema a través de otro circuito divisor, haciendo que el consumo del bus de los periféricos se reduzca, produciendo la señal de reloj PB. Esto se controla mediante el bit PBDIV del registro OSCCON. El valor recomendado para el bus periférico es de 36 MHZ, correspondiéndose a un ratio 1:2 entre el reloj del sistema para una frecuencia de operación de 72MHz y el reloj PB.

Para seleccionar el reloj y las frecuencias a las que vamos a trabajar, existen un grupo de bits, conocidos como bits de configuración (almacenados en la memoria flash del PIC32), los cuales nos van a proporcionar la configuración inicial del dispositivo. El módulo del oscilador utiliza alguno de estos bits para la configuración inicial del registro OSCCON.

De tal forma que para poder configurar estos bits, lo vamos a poder realizar desde el MPLAB en el menú Cofigure->Configuration Bits, cuyas posibles configuraciones se pueden consultar en Help-> Topic-> Config Settings. Sin embargo, usando este método, no se puede modificar el valor del divisor de entrada en tiempo de ejecución. Por lo que para realizar la configuración inicial de estos bits vamos a usar la directiva *#pragma*. A continuación mostramos la configuración representada en la figura anterior y con una frecuencia del bus periférico de 36 MHZ.

```
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF
```

Además, en la última línea de código se deshabilita el watchdog y la protección del código y se habilita la programación de la memoria boot flash.

Una vez realizada la configuración inicial del PIC32 vamos a ver de qué forma podemos ejecutar nuestro código en el menor tiempo posible empleando las características ofrecidas por el PIC32. Para ello vamos a usar el programa "*FFT.c*" (Fast Fourier Transform), disponible en el capítulo 7 de la referencia [10], el cual se encuentra en la carpeta de SIMULACION del CD adjunto, programa "*Analisis_tiempo.c*".

Lo que vamos a realizar es medir el tiempo que tarda en ejecutarse esta función mediante el uso de un Timer de 32 bits formado por la pareja de Timers 4 y 5. A continuación mostramos el main del programa a ejecutar:

```
main(){
   int i, t;
   double f;
    initFFT();
   // test sinusoid
   for (i=0; i<N_FFT; i++)    {
      f =  sin(2 * PI2N * i );
      inB[ i] = 128 + ( unsigned char) (120.0 * f);
   } // for

   // init 32-bit timer4/5
   OpenTimer45( T4_ON | T4_SOURCE_INT, 0);

   // clear the 32-bit timer count
   WriteTimer45( 0);
```

```
    //2.- perform FFT algorithm
    windowFFT( inB);
    FFT();
    powerScale( inB);

    // read the timer count
    t = ReadTimer45();
    f = t/36E6;
     // 3. infinite loop
    while( 1);
} // main
```

Si añadimos un breakpoint al bucle principal y abríamos la ventana wacth asignándole la variable f, podremos ver el tiempo que se ha tardado en ejecutar el algoritmo FFT. En nuestro caso nos sale un tiempo de t=0.17s. Vamos ahora a ver de qué manera podemos hacer que este tiempo se reduzca.

El tiempo de ejecución del programa nos va a depender en gran medida de la velocidad de la memoria Flash. Para mejorar este aspecto, los diseñadores del PIC32 han comprobado que existe un balance perfecto usando un bajo consumo de la memoria flash y desdoblando el bus del núcleo del sistema, proporcionando un número de estados de esperas durante los cuales el procesador espera a que los datos sean leídos desde la memoria flash. Para reducir los estados de espera utilizamos la función: SYSTEMConfigWaitStatesAndPB (freq).

Como vemos hay que pasar la frecuencia del reloj como parámetro, con tal de que se asigne los mínimos estados de espera recomendados para una frecuencia de reloj del sistema dada. Además la función modificará automáticamente la frecuencia de reloj de los periféricos (PB divisor) con tal de mantenerla siempre por debajo de 50MHZ. De tal forma que en nuestro caso:

```
SYSTEMConfigWaitStatesAndPB (72000000L).
```

Si volvemos a ejecutar el programa nos damos cuenta que el tiempo de ejecución del programa se ha reducido a t=0.04s.

Sin embargo nuestro programa aun se puede hacer más rápido si usamos el nuevo modulo entre el bus del núcleo y el de la memoria, la cache. Cada vez que el procesador cargue y lea una instrucción o dato desde la memoria flash, el modulo cache guardará una copia pero también recordará su dirección. De tal forma que cuando el procesador necesite el mismo dato otra vez, la cache devolverá el valor rápidamente, sin tener que acceder de nuevo a la memoria Flash. No obstante, cuando toda la memoria cache esté llena, el contenido será rotado, es decir, la más

vieja será sobrescrita con nueva información. Utilizando el módulo *pcache.h*, añadimos la siguiente línea al programa:

```
CheKseg0CacheOn();
```

Ejecutando de nuevo el programa nos fijamos que el tiempo se ha reducido a t=0.02s.

Además, la cache, posee otra importante herramienta que puede reducir el tiempo de ejecución del programa. Esta tiene la habilidad de ejecutar instrucción pre-fetching, es decir, el módulo de la cache no solo registra instrucciones que han sido leídas y cargadas por el núcleo, también lee todo un bloque de 4 instrucciones (4 palabras de 32bits) a la vez. Por tanto las próximas tres fetches serán ejecutadas sin consumir ningún estado de espera. Para habilitar la cache pre-fetch hay que activar los bits PREFEN en el registro CHECON o bien utilizar la siguiente función desde la librería pcache.h:

```
mCheConfigure (CHECON | 0x30)
```

Realizando esta nueva modificación el tiempo se reduce a t=0.016467s.

No obstante, aún se puede reducir el tiempo, relacionado con el acceso a la memoria RAM. El acceso a la memoria RAM es por defecto lento debido a la presencia de un estado de espera. Este estado se puede deshabilitar utilizando la siguiente función:

```
mBMXDisableDRMWaitState()
```

De tal forma que el tiempo se ha reducido a t=0.016459s. Aunque en este programa no se produce una gran mejora con la introducción de esta función, puede que en otros programas sí que aumente si el tamaño de este es considerable.

Por tanto mediante el uso de estas 4 instrucciones hemos conseguido reducir el tiempo de ejecución del programa de 0.17 segundos a 0.0164 segundos. Además, estas 4 instrucciones utilizadas hasta ahora se pueden resumir utilizando una única función que nos va a permitir llevar a cabo todas las optimizaciones realizadas anteriormente.

```
SYSTEMConfigPerformance(72000000L);
```

A continuación mostramos una tabla resumen con el tiempo exacto tardado en ejecutarse el programa dependiendo de la instrucción añadida:

Instrucción	Tiempo (s)	Respecto Inicial	Respecto Anterior
Ninguna	0.170626	-	-
SYSTEMConfigWaitStatesAndPB (72000000L)	0.042656	4.00	4.00
CheKseg0CacheOn()	0.020231	8.43	2.11
mCheConfigure (CHECON | 0x30)	0.016467	10.36	1.23
mBMXDisableDRMWaitState()	0.016459	10.37	1.00
SYSTEMConfigPerformance(72000000L)	0.016459	10.37	1.00

Tabla B.5: Análisis temporal de las diferentes instrucciones de optimización de cógido.

Por tanto, gracias al mecanismo de la memoria cache y al pre-fetch, el PIC32 puede ejecutar al menos una instrucción por ciclo de reloj incluso cuando se opera a la máxima frecuencia de reloj mientras se usa un bajo consumo de la memoria flash.

ANEXO C. ACELERÓMETRO ADXL330

ANEXO C. ACELERÓMETRO ADXL330.

C.1. INTRODUCCIÓN

En este anexo vamos a estudiar el hardware del acelerómetro ADLX330 para la detección de movimiento en el espacio. El objetivo es incorporarlo a la pantalla táctil para dar vistosidad a los programas ejecutados en esta y ver que su introducción no es muy compleja gracias al uso de la "I/O Expansion Board".

C.1.1. IMPORTANCIA DE LOS SENSORES EN APLICACIONES EMBEBIDAS

Actualmente existen muchas aplicaciones, tanto a nivel industrial como comercial, en las que se utilizan este tipo de sensores. Se puede destacar su actual importancia en sistemas de teléfonos móviles, plataformas de juego, sistemas de seguridad en automóviles, entre otras. Por ejemplo, recientemente se ha incorporado este sensor en teléfonos móviles (iPhone, Blackberry Storm) para la detección de movimiento en el espacio, cuando la pantalla gira de la orientación "retrato" a la posición "paisaje" (Figura C.1). Además de facilitar otras aplicaciones, como reordenar una lista al sacudirlo.

Figura C.1: Uso de un acelerómetro en el iPhone.

C.2. FUNCIONAMIENTO DEL SENSOR

El sensor ADXL330 (Figura C.2) es un acelerómetro que mide la aceleración en los tres ejes, a través de un circuito integrado (CI monolítico, es un tipo de dispositivo electrónico en un circuito integrado el cual contiene dispositivos pasivos y activos como diodos, transistores, etc, los cuales están montados sobre una superficie de una pieza de un solo semiconductor, como una oblea de Silicio) [23].

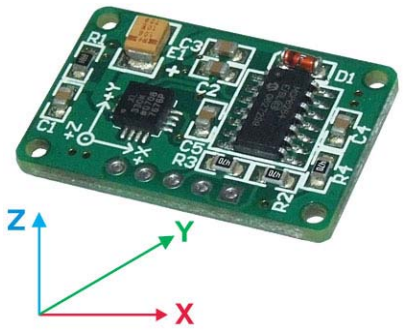


Figura C.2: Acelerómetro ADXL330.

Las señales de salida del sensor son voltajes analógicos los cuales son proporcionales a la aceleración, siendo el rango de este de ±3g. El acelerómetro puede medir la aceleración de la gravedad estática en las aplicaciones de detección de inclinación, así como la aceleración dinámica resultante de movimiento, choque o vibraciones. Además su conexionado es muy sencillo, sólo necesita 5 líneas: alimentación, tierra y 3 líneas de entrada hacia el microcontrolador (aceleración medida en los tres ejes, Figura C.3), tal y como podemos ver en la figura anterior.

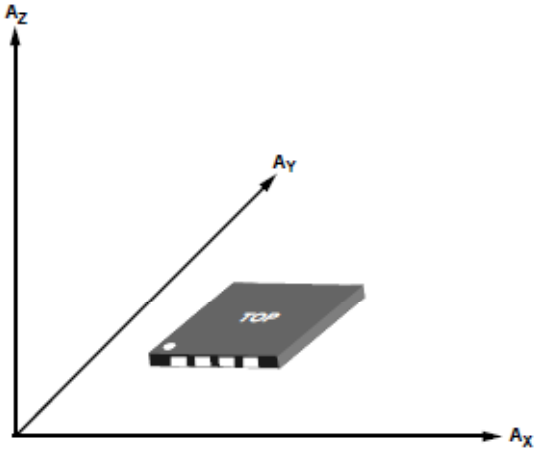
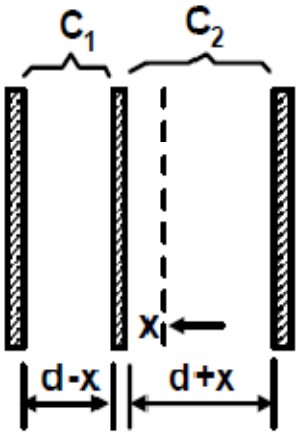


Figura C.3: Sensibilidad del sensor ADLX330 en los tres ejes.

Se trata, de un acelerómetro tipo MEMS (Microelectomechanical Systems, se refiera a la tecnología electromecánica, micrométrica y sus productos) basado en un sensor capacitivo diferencial [24]. Un sensor capacitivo diferencial consiste en dos condensadores variables dispuestos de tal modo que experimentan el mismo cambio pero en sentidos opuestos (Figura C.4).



$$C_2 = \frac{\varepsilon A}{d + x}$$

$$C_1 = \frac{\varepsilon A}{d - x}$$

Figura C.4: Modelo físico de un sensor capacitivo diferencial.

El acelerómetro nos devolverá un voltaje en función de su posición, la cual se podrá evaluar siempre respecto a una aceleración de 0 g, offset. Por tanto habrá que calibrar el sensor para conocer tanto el valor de este offset cuando la aceleración es de 0 g así como el valor del sensor cuando la aceleración es de 1 g (Figura C.5).

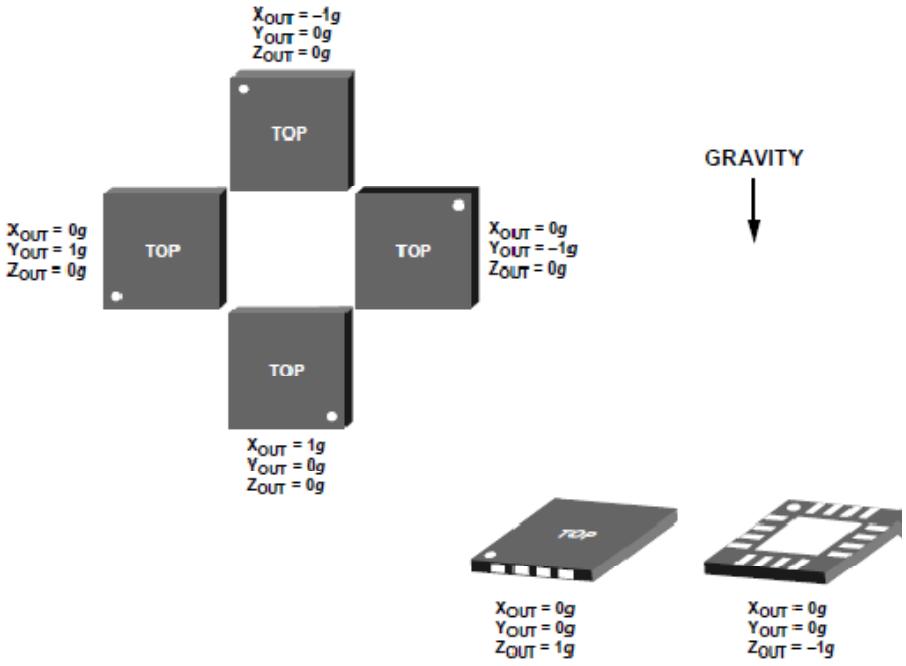


Figura C.5: Salida del sensor ADXL330 en función de la orientación del mismo.

ACRÓNIMOS

ALU	Arithmetic Logic Unit
API	Application Programming Interface
BFM	Boot Flash Memory
CPU	Central Processing Unit
DMA	Direct memory Access
DR	Data Register
DRM	Data RAM Memory
DSP	Digital Signal Processor
EEPROM	Electrical Erasable Programmable Read Only Memory
EPROM	Erasable Programmable Read Only Memory
E/S	Entrada/Salida
FFT	Fast Fourier Transform
FIFO	First In First Out
FMDU	Fast Multiply Divide Unit.
FMT	Fixed Mapping Translation
GOL	Graphics Object Layer
I^2C	Inter-Integrated Circuit
ICD	In-Circuit Debugger
IDE	Integrated Development Environment
IR	Instruction registers

ISR	Interrupt Service Routine
IU	Integer Unit
JTAG	Joint Test Action Group
LCD	Liquid Crystal Display
LED	Light Emitting Diodes
MCU	Microcontroller Unit
MDU	Multiply/Divide Unit
MMU	Memory Management Unit
PFM	Program Flash Memory
PLL	Phase Locked Loops
PMP	Parallel Master Port
PROM	Programmable Read-Only Memory
PWN	Pulse Width Modulator
QVGA	Quarter Video Graphics Array
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RTCC	Real-Time Clock and Calendar
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
TFT	Thin Film Transistor
TN	Twisted Nematic
TQFP	Thin Quad Flat Pack
UART	Universal Asynchronous Receiver Transmitter

BIBLIOGRAFÍA

[1] Información sobre el núcleo del PIC32 M4K CPU basado en la tecnología MIPS32. http://www.mips.com/products/processors/32-64-bit-cores/mips32-m4k/index.cfm#features

[2] "Brief Introduction to MIPS32®M4K® Core Shadow Registers for microcontroller Applications", MIPS Technologies, Inc.

[3] "PIC32MX3XX/4XX Family Data Sheet 64/100-Pin General Purpose and USB 32-Bit Flash Microcontrollers", Microchip Technology Inc.

[4] "PIC32MX Family Reference Manual", Microchip Technology Inc.

[5] Información sobre la familia de de microcontroladores de 32 bits, http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2591

[6] Información y descarga del MPLAB IDE, Microchip Technology Inc. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en019469&part=SW007002

[7] Información y descarga del compilador C32, Microchip Technology Inc http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2615&dDocName=en532454&redirects=c32

[8] Información sobre el MPLAB ICD 3 In-Circuit Debugger en Resource Center, http://edn.resourcecenteronline.com/resource_center/asset/1996-MPLAB_ICD_3_InCircuit_Debugger

[9] "Data Sheet Temperature Sensor TC1047/TC1047A". Microchip Technology Inc.

[10] "Programming 32-bit Microcontrollers in C: Exploring The PIC32 (Embedded Technology)",L. Di Jasio. Newnes Books (2008).

[11] Información y descarga de programas ejemplos, Microchip Technology Inc http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en024858&part=DM240001&redirects=explorer16

[12] “Explorer16 Development Board User’s Guide”, Microchip Technology Inc.

[13] “256K SPI™ Bus Serial EEPROM”, Microchip Technology Inc.

[14] “HD44780U (Dot Matrix Liquid Crystal Display Controller/Driver)”, Hitachi.

[15] “Graphics PICtail™ Plus Daughter Board 2”, Microchip Technology Inc.

[16] Hoja de características del controlador LCD, LGDP453, LG Electronics.

[17] Hoja de características de la pantalla táctil TFT-G240320UTSW-92W-TP-E, TRULY SEMICONDUCTORS LTD.

[18] Información sobre la tarjeta “I/O Expansion Board”, Microchip Technology Inc http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2615&dDocName=en535444

[19] “How to use Widgets in Microchip graphics Library”, Microchip Technology Inc.

[20] Información sobre la pantalla táctil, Microchip Technology Inc http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2608&page=1&param=en532061&redirects=Graphics

[21] Información sobre la placa de expansión USB, Microchip Technology Inc http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2651&param=en534494

[22] “TB3012, Conversion of Graphics PICtail™ Plus Board 2 for Compatibility with USB PICtail Plus and Firmware Modification”, Microchip Technology Inc.

[23] Manual del Acelerómetro ADXL330 iMEMS, Analog Devices http://www.analog.com/static/imported-files/data_sheets/ADXL330.pdf

[24] Información sobre la tecnología MEMs en el ADXL330, Electroiq http://www.electroiq.com/index/display/semiconductors-article-display/328028/articles/solid-state-technology/chip-forensics/2008/05/two-different-approaches-to-integrated-mems.html