



Laboratorio de Sistemas Electrónicos Digitales

Departamento de Ingeniería Electrónica
E.T.S.I. de Telecomunicación
Universidad Politécnica de Madrid

Memoria del proyecto desarrollado en el Laboratorio de Sistemas Electrónicos Digitales (LSED)

Curso 2007/2008

Título del proyecto desarrollado

“Juego de “tres en raya” con efectos musicales basado en el MCF5272”

Autores (orden alfabético): Jesús Javier Alonso Sánchez

Luciano Rubio Romero

Código de la pareja: **MM-20**

ÍNDICE GENERAL

1	INTRODUCCIÓN	5
2	DIAGRAMA DE SUBSISTEMAS	6
3	DESCRIPCIÓN DEL SUBSISTEMA HARDWARE	7
3.1	ESQUEMA CIRCUITAL	7
3.2	FILTRO PASO ALTO (HPF)	7
3.2.1	<i>Descripción del módulo</i>	7
3.2.2	<i>Análisis teórico</i>	7
3.2.3	<i>Obtención y medidas de diagramas de Bode</i>	8
3.3	FILTRO PASO BAJO (LPF)	8
3.3.1	<i>Descripción del módulo</i>	8
3.3.2	<i>Análisis teórico</i>	8
3.3.3	<i>Obtención de medidas y diagramas de Bode</i>	9
3.4	AMPLIFICADOR DE POTENCIA	9
3.4.1	<i>Descripción del módulo</i>	9
3.4.2	<i>Análisis teórico</i>	10
3.4.3	<i>Obtención de medidas y diagramas de Bode</i>	10
4	DESCRIPCIÓN DEL SUBSISTEMA SOFTWARE	11
4.1	PROCESO DEL PROGRAMA PRINCIPAL	11
4.1.1	<i>Rutina setPuntuacion3EnRaya</i>	12
4.1.2	<i>Rutina dosJugadores</i>	12
4.1.3	<i>Rutina finalizarPartida</i>	13
4.1.4	<i>Rutina limpiaTablero</i>	13
4.1.5	<i>Rutina muestraTablero</i>	13
4.1.6	<i>Rutina escribeTablero</i>	13
4.1.7	<i>Rutina getFicha</i>	14
4.1.8	<i>Rutina preparaTurno</i>	14
4.1.9	<i>Rutina movimientoHumano</i>	14
4.1.10	<i>Rutina sleep</i>	15
4.1.11	<i>Rutina hayTresEnRaya</i>	15
4.1.12	<i>Rutina hayTresEnRayaFila</i>	15
4.1.13	<i>Rutina hayTresEnRayaColumna</i>	15
4.1.14	<i>Rutina hayTresEnRayaDiagonal</i>	16
4.1.15	<i>Rutina cambiaTurno</i>	16
4.1.16	<i>Rutina todasLasFichas</i>	16
4.1.17	<i>Rutina mueveFicha</i>	16
4.1.18	<i>Rutina mueveFichaOrigenDestino</i>	17
4.1.19	<i>Rutina sePuedeMover</i>	17
4.1.20	<i>Rutina teclado</i>	17
4.1.21	<i>Rutina cuentaAtras</i>	18
4.1.22	<i>Rutina contraMaquina</i>	18
4.1.23	<i>Rutina movimientoMaquina</i>	19
4.1.24	<i>Rutina mueveMaquina</i>	19
4.1.25	<i>Rutina posibleTresEnRaya</i>	20
4.1.26	<i>Rutina posiciónAleatoria</i>	20
4.1.27	<i>Rutina jugadaFinalMaquina</i>	20
4.1.28	<i>Rutina melodíaFinal</i>	21

<i>4.1.29 Rutina notaMusical</i>	21
<i>4.1.30 Rutina reproduceNota</i>	21
4.2 PROCESO DE LA INTERRUPCIÓN TIMER0	22
<i>4.2.1 Función setTIMER0</i>	22
<i>4.2.2 Función desactivaTIMER0</i>	23
4.3 PROCESO DE LA INTERRUPCIÓN TIMER1	23
<i>4.3.1 Función setTIMER1</i>	24
5 DESCRIPCIÓN DE LAS MEJORAS	26
5.1 JUEGO DEL SUDOKU PARA VARIOS JUGADORES	26
<i>5.1.1 Breve historia del Sudoku</i>	26
<i>5.1.2 Las reglas y objetivo del Sudoku</i>	26
<i>5.1.3 Extensiones que hemos desarrollado para el juego básico</i>	27
<i>5.1.4 Esquema del juego Sudoku desarrollado</i>	28
<i>5.1.5 Breve análisis de cómo jugar al Sudoku</i>	28
<i>5.1.6 Diagrama del programa principal</i>	30
<i>5.1.7 Proceso del programa principal</i>	31
<i>5.1.8 Variables del sistema</i>	32
<i>5.1.8.1 Funcionalidad iniciaSudoku</i>	34
<i>5.1.8.2 Funcionalidad refrescoTablero</i>	34
<i>5.1.8.3 Funcionalidad refrescaTurno</i>	34
<i>5.1.8.4 Funcionalidad incrementoTurno</i>	35
<i>5.1.8.5 Funcionalidad setConfiguracion</i>	35
<i>5.1.8.6 Funcionalidad setConf</i>	36
<i>5.1.8.7 Funcionalidad devuelveSolucion</i>	36
<i>5.1.8.8 Funcionalidad setRed</i>	36
<i>5.1.8.9 Funcionalidad setJugadores</i>	37
<i>5.1.8.10 Funcionalidad setDificultad</i>	37
<i>5.1.8.11 Funcionalidad setAyuda</i>	37
<i>5.1.8.12 Funcionalidad setPuntuacion</i>	38
<i>5.1.8.13 Funcionalidad actualizaPuntos</i>	38
<i>5.1.8.14 Funcionalidad muestraResultados</i>	39
<i>5.1.8.15 Funcionalidad hayRepeticion</i>	40
<i>5.1.8.16 Funcionalidad hayRepeticionFila</i>	41
<i>5.1.8.17 Funcionalidad hayRepeticionColumna</i>	42
<i>5.1.8.18 Funcionalidad hayRepeticionBloque</i>	43
<i>5.1.8.19 Funcionalidad estanTodosNumeros</i>	44
<i>5.1.8.20 Funcionalidad turnoGanador</i>	44
<i>5.1.8.21 Funcionalidad determinaMaximo</i>	44
<i>5.1.8.22 Funcionalidad inicializaPuntuacion</i>	45
<i>5.1.8.23 Funcionalidad inicializaSudoku</i>	45
<i>5.1.8.24 Funcionalidad haySudoku</i>	45
<i>5.1.8.25 Funcionalidad muestraTableroSudoku</i>	46
<i>5.1.8.26 Funcionalidad movimientoSudoku</i>	46
<i>5.1.8.27 Funcionalidad compruebaMovimiento</i>	47
<i>5.1.8.28 Funcionalidad escribeSudoku</i>	47
<i>5.1.9 Niveles de dificultad</i>	49
5.2 DESCARGA DE TABLEROS DE SUDOKU POR SERVIDOR TFTP	52
<i>5.2.1 Cómo funciona TFTP y qué uso podemos darle</i>	52
<i>5.2.2 Modo de funcionamiento en el Sudoku</i>	53

5.2.3 Funcionalidades añadidas	53
5.2.3.1 Funcionalidad leeEthernetInicial	53
5.2.3.2 Funcionalidad leeEthernet	54
5.2.3.3 Funcionalidad escribeEthernet	55
5.2.4 Funcionalidades modificadas	56
5.2.4.1 Funcionalidad devuelveSolucion	56
5.2.4.2 Funcionalidad setDificultad	56
5.3 JUEGO DE SUDOKU POR LAN O POR INTERNET	57
5.3.1 <i>Modo de funcionamiento del modo LAN</i>	57
5.3.2 <i>Diagrama del programa principal</i>	59
5.3.3 <i>Modo de funcionamiento del modo INTERNET</i>	60
5.3.4 <i>Funcionalidades añadidas</i>	61
5.3.4.1 Funcionalidad actualizaSDRAM	61
5.3.4.2 Funcionalidad actualizaHOST	61
5.3.4.3 Funcionalidad pingTurno	62
5.3.4.4 Funcionalidad pingTablero	62
5.3.5 <i>Funcionalidades modificadas</i>	62
5.3.5.1 Funcionalidad sudokuJuego	62
5.3.5.2 Funcionalidad refrescoTablero	63
5.3.5.3 Funcionalidad incrementoTurno	64
5.3.5.4 Funcionalidad setConfiguracion	64
5.3.5.5 Funcionalidad muestraResultados	65
5.4 LSED RALLY	66
5.4.1 <i>Objetivos y descripción del juego</i>	66
5.4.2 <i>Variables del sistema</i>	66
5.4.3 <i>Extensiones que hemos introducido al Rally</i>	66
5.4.4 <i>Proceso de la interrupción TIMER1</i>	67
5.4.5 <i>Diagrama del programa principal</i>	68
5.4.6 <i>Programa principal</i>	69
5.4.6.1 Funcionalidad setConfiguracionRally	69
5.4.6.2 Funcionalidad setVelocidad	69
5.4.6.3 Funcionalidad setCircuito	70
5.4.6.4 Funcionalidad iniciaRally	70
5.4.6.5 Funcionalidad inicializaSuelo	70
5.4.6.6 Funcionalidad situaTramo	71
5.4.6.7 Funcionalidad sigueCircuito	71
5.4.6.8 Funcionalidad decrementaRepeticion	71
5.4.6.9 Funcionalidad actualizaPosicion	72
5.4.6.10 Funcionalidad compruebaMeta	72
5.4.6.11 Funcionalidad muestraTramo	72
5.4.6.12 Funcionalidad detectaColision	73
5.4.6.13 Funcionalidad situaVehiculo	73
5.4.6.14 Funcionalidad borraVehiculo	73
5.4.6.15 Funcionalidad muestraTramoFuturo	74
5.4.6.16 Funcionalidad mensajeFinal	74
5.4.6.17 Funcionalidad disparoSalida	74
5.5 SISTEMA DE PUNTUACIÓN EN EL “3 EN RAYA”	75
5.5.1 <i>Función inicializaPuntos3EnRaya</i>	76
5.5.2 <i>Función setPuntuacion3EnRaya</i>	76
5.5.3 <i>Función sumaPuntos</i>	77

<i>5.5.4 Función decrementaMovimientos</i>	77
<i>5.5.5 Función sumaPuntosPosibleVictoria</i>	77
<i>5.5.6 Función incrementaPuntos</i>	78
<i>Función muestraPuntos</i>	78
<i>5.5.7 Función tablaDePuntos</i>	78
<i>5.5.8 Función determinaMaximaPuntuacion</i>	79
6 PRINCIPALES PROBLEMAS ENCONTRADOS	80
6.1 DIFICULTADES DURANTE EL DESARROLLO DE LA PRÁCTICA	80
6.2 CALIFICACIÓN PERSONAL	80
7 MANUAL DE USUARIO	82
8 DEFINICIÓN DEL PRODUCTO	83
9 BIBLIOGRAFÍA	85
10 ANEXO I: CÓDIGO DEL PROGRAMA DE LA PRIMERA SESIÓN	86
11 ANEXO II: CÓDIGO DEL PROGRAMA DEL PROYECTO FINAL	98

1 Introducción

El objetivo del Laboratorio de Sistemas Electrónicos Digitales para el siguiente curso es el desarrollo, sobre la plataforma *ColdFire MCF5272*, del conocido juego del “**3 en raya**”, con el añadido de poder jugar **contra la máquina** y de tener que **reproducir distintas melodías** según el desarrollo del juego.

Desde el primer momento nos planteamos el uso del **lenguaje de programación C** como el más apropiado. Los fundamentos de esta decisión fueron, en primer lugar, el interés de ambos miembros del grupo en el lenguaje, que conocíamos, pero sobre el que teníamos la motivación de ampliar nuestros conocimientos. El segundo motivo fue la sensación de que el desarrollo sería más rápido, lo que acabaría concluyendo en un mayor tiempo para **implementar mejoras**, y que estas fueran más complejas.

Sin embargo también se plantearon algunos problemas que podrían derivarse del uso de *C*. El principal, que –a priori– tendríamos más dificultades a la hora de configurar los aspectos de *nivel hardware más bajo*, como los temporizadores o el acceso directo a *bytes de memoria*. Dicho inconveniente desapareció mediante el uso de las funciones de las librerías que el entorno de desarrollo *ED Coldfire* proporcionaba.

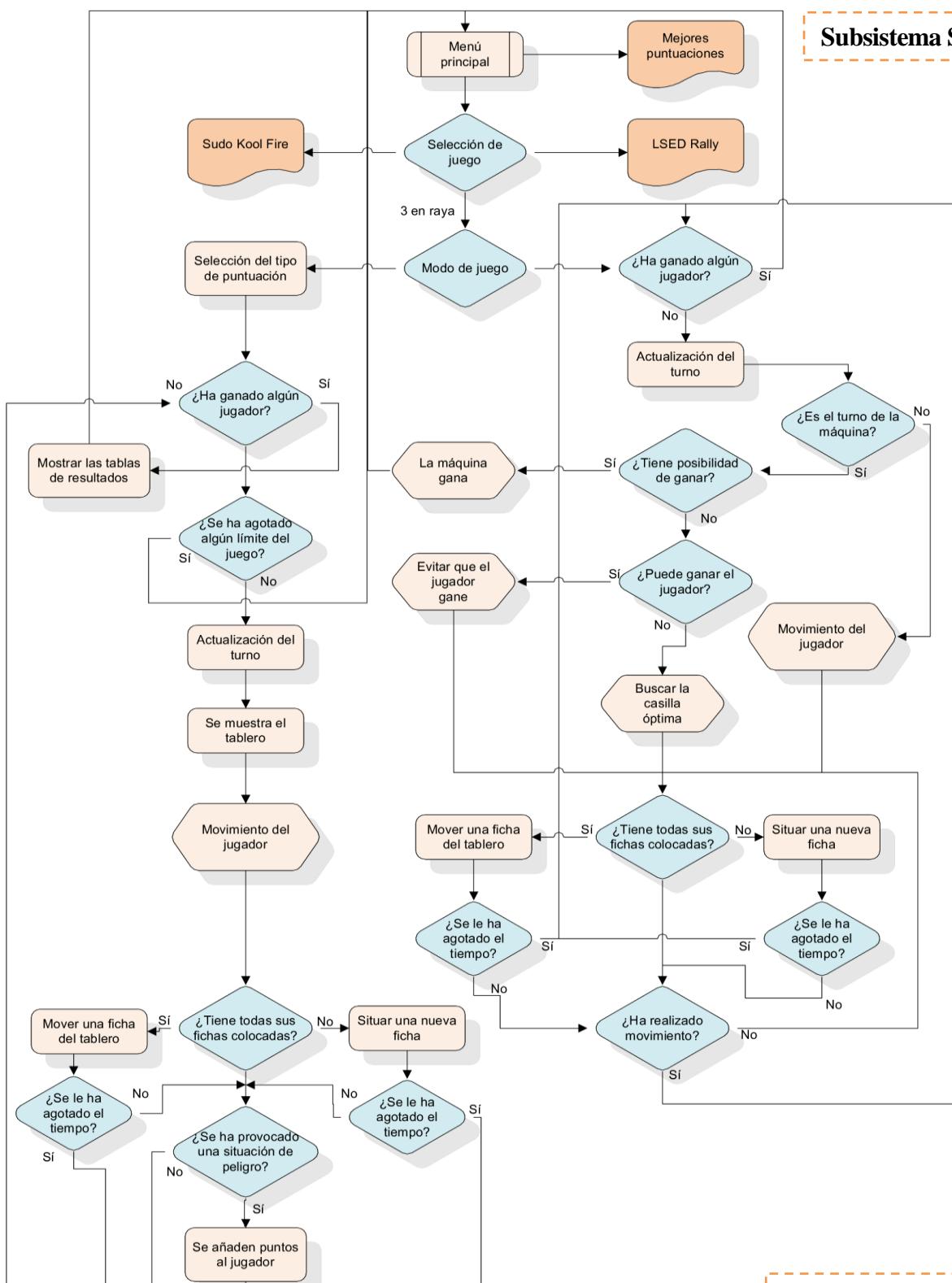
Asimismo, como consecuencia de esta serie de objetivos planteados, consideramos que nuestro *código* presenta una serie de propiedades. La primera que, debido al empleo didáctico de *C*, se ha tratado de expresar mediante la **modularización** (*includes*, *.c*, *.h*), uso de **estructuras complejas** (*vectores*, *punteros*, *enums*, *structs*), control de flujo (*switches*). También creemos que el código presenta una gran reutilización de funciones (mediante el paso de múltiples argumentos para adecuarlas a situaciones concretas) y evasión de algoritmos que comprobaran situaciones concretas predefinidas (se ha intentando usar al máximo algoritmos que detecten propiedades comunes a dichas situaciones, lo más genéricos posibles, lo que aumenta su complejidad y reduce su longitud).

Relación de mejoras implementadas:

- Sistemas de **puntuación** en el juego del “3 en raya”.
- Implementación extendida de un **juego de Rally**
- Juego completo de **Sudoku multijugador** (de 1 a 9 participantes)
- Sistemas de **puntuación** por aciertos y por tiempos en el juego de Sudoku
- Descarga de tableros de Sudoku por **servidor TFTP**
- **Partida en red LAN** o por Internet entre **distintas** plataformas ENT2004CF
- Sistema de **máximas puntuaciones** (*records*)
- Algunas mejoras simples (*ascii-art*, mensajes por pantalla, tecla de escape...)

Palabras clave: Optimización, reutilización, funcionalidad, algoritmo, estructurización.

2 Diagrama de subsistemas



3 Descripción del subsistema Hardware

El subsistema Hardware, objeto de descripción en este apartado, tiene una misión fundamental: adaptar y adecuar las señales cuadradas generadas por el microcontrolador MCF5272, para que sean audibles usando unos auriculares estándar.

Consta de tres etapas, la primera un **filtro paso alto**, que filtra la componente continua. La segunda, un filtro **paso bajo**, cuya misión es reducir el espectro (a 12Khz) de la señal proveniente del microcontrolador. Por último nos encontramos con un amplificador de potencia, para poder suministrar la cantidad de corriente demandada por los auriculares que conectemos al sistema.

A continuación se muestra el esquema eléctrico del Hardware y un análisis detallado de los módulos mencionados.

3.1 Esquema circuitual

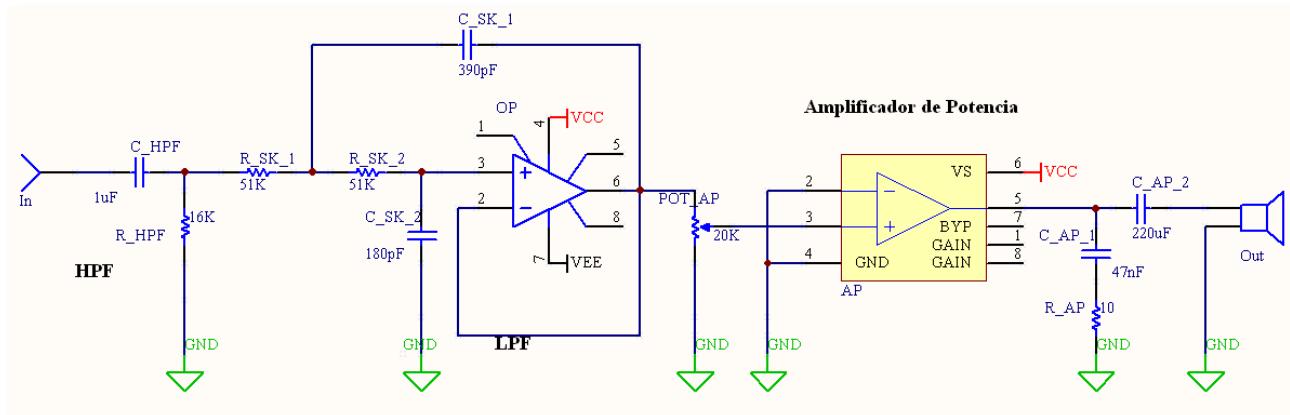


Figura 3.1: Esquema circuital del Hardware

3.2 Filtro paso alto (HPF)

3.2.1 Descripción del módulo

Como ya se ha mencionado, el propósito de este componente consiste en eliminar la componente continua de la señal generada por el MCF5272, la es **cuadrada y unipolar** (entre 0V y 5V). Se trata de una red RC, con unos componentes de valores tales, que sitúan su frecuencia de corte en los pocos Hz.

La necesidad de éste módulo es esencial, puesto que la señal mencionada anteriormente tiene una *componente continua* distinta de cero y el filtro usado para reducir su ancho de banda está alimentado entre -10V y 10V.

3.2.2 Análisis teórico

Para una frecuencia de corte, $F_C \approx 10\text{Khz}$, obtenemos el valor de R, $R=f(C)$:

$$F_C = \frac{1}{2\pi RC}; R = \frac{1}{2 \cdot 10\pi C} = \frac{1}{20\pi C}$$

Llegados a este punto, podemos seleccionar un valor para C que nos resulte conveniente, y hallar el valor de R para la F_C que se ha fijado (recordemos, 10Hz). Escogiendo $C = 1\mu\text{F}$, tenemos que:

$$R = \frac{1}{20\pi \cdot 1 \cdot 10^{-6}} \approx 16K\Omega$$

Como anotación, es interesante comentar que, como la gama de valores comerciales de los condensadores, es menor que la de los resistores, se decidió fijar en primer lugar un valor comercial – disponible en tiendas- para el condensador, y a partir de dicho valor hallar el de R.

3.2.3 Obtención y medidas de diagramas de Bode

A continuación se muestran los resultados de las medidas prácticas realizadas en laboratorio.

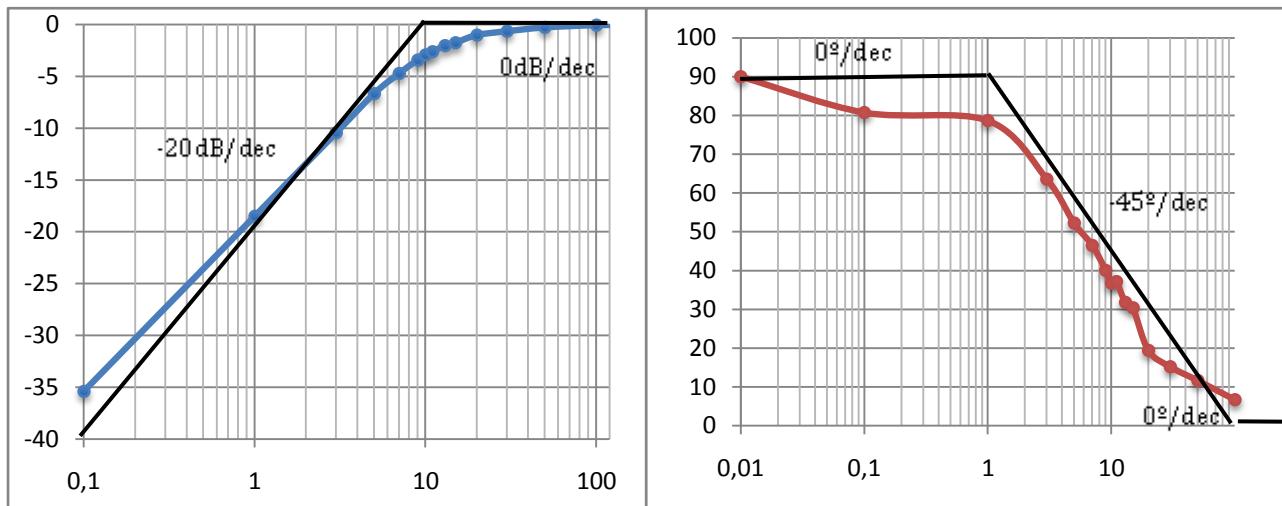


Figura 3.2: Respuesta en módulo y fase del filtro paso alto situado a la entrada.

3.3 Filtro paso bajo (LPF)

3.3.1 Descripción del módulo

Tras eliminar la componente continua de la señal que genera el *microcontrolador*, es necesario reducir su ancho de banda (**infinito**, al ser una señal cuadrada), para disminuir la sensación de distorsión que produce al oírla (debido a sus -teóricamente- infinitos armónicos). Es por esto que recurrimos a una célula **Sallen-Key** de segundo orden, en configuración **Paso Bajo**.

La frecuencia de corte escogida es **fc = 12 KHz**, suficiente para permitir el paso de, al menos, dos armónicos para cada una de las frecuencias reproducidas, manteniendo así, cierta riqueza espectral (sin llegar al extremo de la distorsión dañina original).

Este módulo, junto con el anterior, configuran la banda espectral en la que va a trabajar el subsistema Hardware: entre **10Hz** y **12KHz**.

3.3.2 Análisis teórico

Las resistencias y los condensadores pueden modelarse, respectivamente, como: $R_1 = mR$, $R_2 = R$, $C_1 = nC$ y $C_2 = C$. Los parámetros de importancia vienen dados por:

$$f_0 = \frac{1}{2\pi\sqrt{mn}RC}; Q = \frac{\sqrt{mn}}{m+1}$$

En nuestro caso: $f_0 = 12\text{KHz}$ y $Q = 1/\sqrt{2}$. Para comenzar el diseño, fijamos un valor moderado de $R = 51\text{K}\Omega$. Calculamos C y n :

$$C = \frac{1}{4\pi f_0 R} \Rightarrow C \approx 130\text{pF}; n = 4Q^2 \Rightarrow n = 2.$$

Escogemos unos valores comerciales para C y nC , tales que C sea muy similar al calculado y el valor real de n sea mayor o igual al calculado:

$$C = 180\text{pF}; nC = 390\text{pF}$$

Obtenemos los valores de k , n y la resistencia R :

$$k = \frac{n}{Q^2} - 2 = 2; m = \frac{k + \sqrt{k^2 - 4}}{2} = 1; R = \frac{1}{2\pi\sqrt{mn}f_0C} \approx 52\text{K}\Omega$$

Con lo que ya podemos considerar unos valores comerciales para R y mR :

$$R_1 = 51\text{K}\Omega; R_2 = 51\text{K}\Omega$$

Recalculamos los parámetros fijados f_0 y Q , para comprobar que se mantienen similares:

$$f_0 = \frac{1}{2\pi\sqrt{mn}RC} \approx 12259\text{Hz}; Q = \frac{\sqrt{mn}}{m+1} = \frac{\sqrt{2}}{2} = \frac{1}{\sqrt{2}}$$

3.3.3 Obtención de medidas y diagramas de Bode

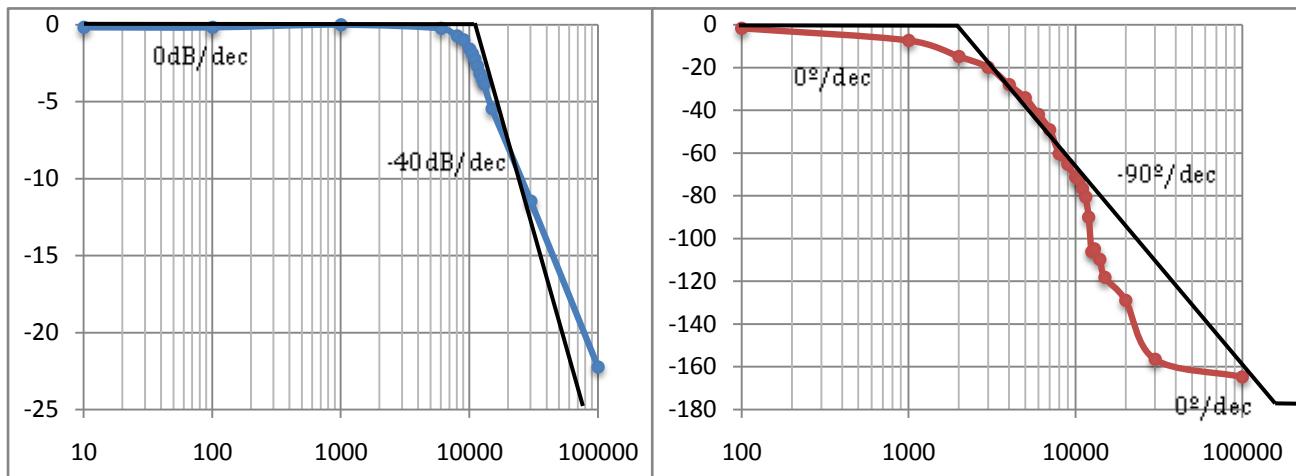


Figura 3.3: Respuesta en módulo y fase del filtro paso bajo encargado de reducir el espectro de la señal de audio.

3.4 Amplificador de potencia

3.4.1 Descripción del módulo

Tras filtrar convenientemente la señal producida por el *microcontrolador* con el objetivo de que posea las cualidades sonoras especificadas, es el momento de *escucharla*. Para ello se recurre a un amplificador de potencia, cuya misión es **suministrar la cantidad de corriente requerida** por los

auriculares para su correcto funcionamiento. Este módulo nos servirá, además, como **adaptador de impedancias** entre la salida de nuestro Hardware y la entrada de los altavoces.

3.4.2 Análisis teórico

Para implementar este amplificador, consultamos la documentación ofrecida por el fabricante para el LM386. Obtuimos un montaje interesante capaz de producir una ganancia máxima de 20dB, controlada por un potenciómetro a la entrada que actúa de divisor de tensión. Además incluye dos condensadores (**50nF** y electrolítico de **250μF**). El primero de ellos es el responsable de **introducir un polo a alta frecuencia** que provocará la caída de la ganancia A_{vmid} , y el segundo es el encargado de acoplar la etapa amplificadora a una resistencia de carga (impedancia de los auriculares).

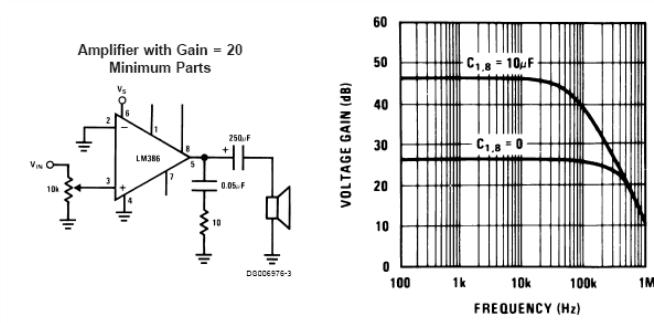


Figura 3.4: Esquema del amplificador de potencia y módulo de la respuesta en frecuencia. Fuente: Datasheet LM386 National Semiconductor.

Finalmente se reguló la ganancia a un valor apropiado según los niveles de amplitud habituales de nuestro sistema, evitando la saturación y el volumen excesivo.

En cualquier caso, las características más interesantes de este integrado, son que **a las frecuencias de trabajo presenta una ganancia constante**, que con el montaje que presentamos, se puede regular entre 0dB y 20B. Y por otra parte, que automáticamente, a la salida nos encontramos una señal con una **tensión de continua** igual a la mitad de la tensión de alimentación. Esto es así a causa de la circuitería interna del amplificador.

3.4.3 Obtención de medidas y diagramas de Bode

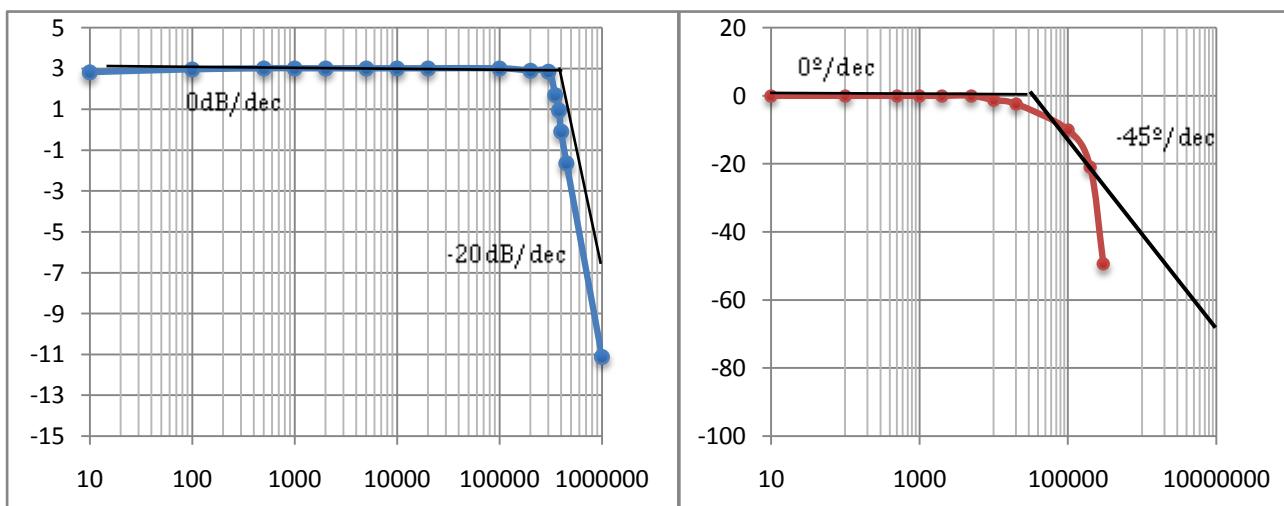


Figura 3.5: Respuesta en módulo y fase del Amplificador de Potencia.

4 Descripción del subsistema Software

Tras ejecutar el programa y ejecutarse el código de la rutina `__init()`, que configura algunos parámetros de los temporizadores, se ejecuta la función `bucleMain()`, que ofrece distintas opciones a elegir (3 en Raya: 2 Jugadores, 3 en Raya: Vs. Máquina, Rally, Sudoku, Récords). Una vez elegida una opción se llama a su correspondiente función (alojada junto a otras, en ficheros destinados a cada opción).

Unidas a las características enunciadas en la Introducción (apdo. 1), cabe mencionar las siguientes:

Se ha tratado de modularizar al máximo el código: las funciones correspondientes a un juego u opción concretas están en su propio archivo. Las funciones comunes a varios juegos están alojadas en su fichero correspondiente (si están relacionadas) o en `main.c` si son de entidad menor.

Todos los *ficheros.c* llevan su correspondiente cabecera, en la que se distinguen las partes de: declaración de variables globales, defines, y declaración de cabeceras de funciones.

En cuanto a la política de inclusiones, cada *fichero.c* incluye a su respectiva *cabecera.h*, siendo especial el caso de *main.c*, fichero principal, que incluye, **en un orden concreto**, el resto de *ficheros.c* del programa.

Es importante mencionar, antes de pasar a explicar las funciones de que se compone la parte del “3 en raya”, que el tablero se almacena es un *vector*, como se puede observar en la fig. 4.1.

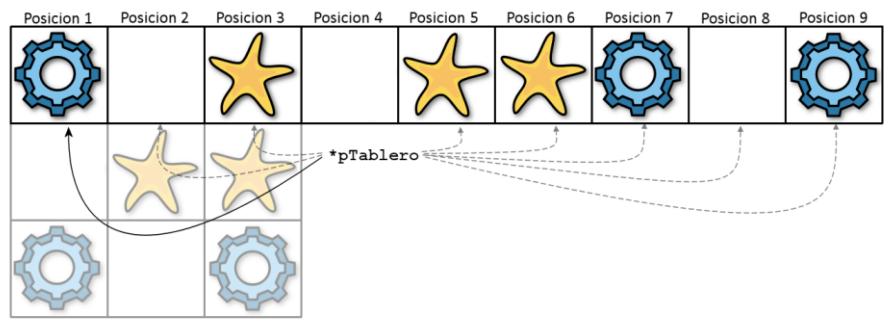


Figura 4.1: Esquema de almacenamiento del tablero del “3 en raya”. Como se puede observar, de la pulsación del teclado se puede obtener directamente la posición en el array.

4.1 Proceso del programa principal

```
bucleMain() {  
    Creación de punteros e inicialización de variables globales  
    SetTIMER1();  
    Selección de un juego por parte del usuario  
    Seteo del sistema para ese juego en concreto  
    Inicialización del juego (llamada a la correspondiente subrutina)  
}
```

```
* Variables de entrada: void  
* Variables de salida: void
```

* **VARIABLES GLOBALES MODIFICADAS:** BOOL turnoPerdido, BOOL pintaTiempo, int estadoJuego

Descripción (ver apdo. 4). Es importante resaltar que las variables globales mencionadas son inicializadas a FALSE y 0 respectivamente.

4.1.1 Rutina setPuntuacion3EnRaya

```
setPuntuacion3EnRaya() {
    ... (Funciones del modo puntuación)
    Inicialización del 3 en Raya para dos jugadores
}

* VARIABLES DE ENTRADA: BOOL *pHayGanador, char *pTurno, char *pTablero, BOOL
*pJugSinTiempo, int *pMovimientosO, int *pMovimientosX, int *pPuntoso, int *pPuntosX
* VARIABLES DE SALIDA: void
* VARIABLES GLOBALES MODIFICADAS: tiempoPartida = TIEMPO_PARTIDA*1000 (se setea el tiempo
total de partida en el modo "límite por tiempo de partida")
```

Función `setPuntuacion3EnRaya()`: Configura el modo de juego “2 jugadores” del 3 en raya según el esquema de puntuación escogido entre los cuatro disponibles: sin puntuación, puntuación con límite de tiempo, puntuación por tiempo de movimiento y puntuación por tiempo restante (ver apdo. mejoras).

4.1.2 Rutina dosJugadores

```
dosJugadores() {
    dosJugadoresTitulo();
    limpiaTablero();
    Mientras no haya ganador {
        Si algún jugador ha agotado sus movimientos, se sale del bucle
        Si el tiempo global se ha agotado, se sale del bucle
        preparaTurno();
        movimientoHumano();
        El contador de tiempo de turno restante se para
        hayTresEnRaya();
        sumaPuntos();
    }
    finalizaPartida();
}
}

* VARIABLES DE ENTRADA: BOOL *pHayGanador, char *pTurno, char *pTablero, TipoPunt3Raya
tipo, int *pPuntoso, int *pPuntosX, int *pMovimientosO, int *pMovimientosX
* VARIABLES DE SALIDA: void
* VARIABLES GLOBALES MODIFICADAS: timeout (se chequea para comprobar si se ha agotado el
tiempo total se juego)
```

Función `dosJugadores()`: Rutina principal del modo 2 jugadores del “3 en raya”. Gestiona turnos, movimientos, comprueba la existencia de un posible tres en raya y gestiona la puntuación, de haberla.

4.1.3 Rutina finalizarPartida

```
finalizarPartida() {  
    Se avisa al sistema de un cambio de melodía  
    muestraTablero();  
    Según el tipo de puntuación actual (movimientos, tiempo global o ninguna) {  
        Se configura al sistema para reproducir la melodía de ganador o perdedor  
        De haberla, se muestra la puntuación (muestraPuntos())  
    }  
}  
  
* Variables de entrada char *pTablero, char *pTurno, TipoPunt3Raya, int *pPuntos0, int  
*pPuntosX  
* Variables de salida: void  
* Variables globales modificadas: estadoJuego
```

Esta función es llamada al término de una partida de “3 en raya” entre dos jugadores. Se encarga de mostrar el tablero final, de configurar el sistema para reproducir la melodía de ganador o perdedor y de mostrar los puntos obtenidos por cada jugador (según el tipo de puntuación escogido).

4.1.4 Rutina limpiaTablero

```
limpiaTablero() {  
    Setea el turno para el jugador X  
    Inicializa un tablero, dejándolo vacío salvo en la posición central que coloca X  
}  
  
* Variables de entrada char *pTurno, char *pTablero  
* Variables de salida: void
```

Función limpiaTablero(): Inicializa el tablero y por tanto se usa cada vez que se inicia un nuevo juego.

4.1.5 Rutina muestraTablero

```
muestraTablero() {  
    Recorre el tablero y saca por pantalla su estado actual  
}  
  
* Variables de entrada char *pTablero  
* Variables de salida: void
```

Esta función recorre el tablero y muestra en la pantalla –con el formato adecuado– su estado. Es llamada al final de cada turno, tras la realización (si no se ha pasado el tiempo) del movimiento correspondiente.

4.1.6 Rutina escribeTablero

```
escribeTablero() {  
    Si la posición dónde quiero escribir está vacía, escribo  
    Si no está vacía y no se ha pasado el turno {
```

```

        Se comprueba si quiero escribir dentro del tablero
        Se comprueba que dónde quiero escribir no hay ya una ficha
    }

    Mientras intente escribir en sitio en el que no se pueda {
        escribeTablero();
    }
}

```

*** Variables de entrada:** char posicion, char *pTurno, char *pTablero
*** Variables de salida:** void

Función que comprueba la validez del sitio en el que se quiere situar una ficha. En caso de no ser válido continúa llamándose hasta que el jugador seleccione una posición adecuada.

4.1.7 Rutina getFicha

```

getFicha() {
    Devuelve la ficha existente en tablero en una posición dada como un carácter
}

```

*** Variables de entrada:** char tecla, char *pTablero
*** Variables de salida:** char (carácter correspondiente a la ficha en la posición dada)

Función que devuelve el `char` que hay en la posición indicada (en la llamada a la función), siendo dicha posición un valor de tipo `char`.

4.1.8 Rutina preparaTurno

```

preparaTurno() {
    Se setea el flag turnoPerdido a False
    muestraTablero();
    cambiaTurno();
    menuJuego();
}

```

*** Variables de entrada:** char *pTurno, char *pTablero, TipoPunt3Raya tipo, int *pMovimientosO, int *pMovimientosX
*** Variables de salida:** void
Variables globales modificadas: turnoPerdido = FALSE, independientemente de que en el turno anterior se haya agotado el tiempo o no

Función `preparaTurno()`: Esta función es usada cada vez que se inicia un nuevo turno. Por ello se especifica que no hay turno perdido (puesto que acaba de empezar). Se muestra el tablero actual, se cambia el turno anterior y se muestra información relativa al juego (turno actual e instrucciones de uso).

4.1.9 Rutina movimientoHumano

```

movimientoHumano() {
    mensajeTurno();
    Se pide al usuario que pulse una tecla
    Si no hay turno perdido {
        Si están todas las fichas (se comprueba mediante todasLasFichas()) {
            mueveFicha();
        } Si no {
            escribeTablero(); (Se pone una nueva ficha)
        }
    }
}

```

```
* Variables de entrada char *pTurno, char *pTablero
* Variables de salida: void
* Variables globales modificadas: Si turnoPerdido = FALSE, no se realiza el movimiento. En caso contrario sí
```

Función movimientoHumano(): Gestiona el movimiento de una persona: comprueba si puede mover o se le ha pasado el turno. En caso de no haber perdido el turno, se comprueba si tiene todas las fichas en el tablero para decidir si ha de colocar una nueva o mover una existente.

4.1.10 Rutina sleep

```
sleep() {
    Setea el tiempo de turno restante (en ms)
}

* Variables de entrada ULONG milisegundos
* Variables de salida: void
* Variables globales modificadas: cont_retardo = milisegundos
```

Función sleep(): Setea la variable global `cont_retardo`, que se va decrementando en una unidad con cada interrupción del timer 1.

4.1.11 Rutina hayTresEnRaya

```
hayTresEnRaya() {
    Se detecta si hay algún tres en raya: fila, columna o diagonal (llamada a funciones hayTresEnRayaFila(), hayTresEnRayaColumna(), hayTresEnRayaDiagonal)
}

* Variables de entrada BOOL *pHayGanador, char *pTablero
* Variables de salida: void
```

Función hayTresEnRaya(): Detecta un posible tres en raya, en caso de que `hayTresEnRayaFila()`, `hayTresEnRayaColumna()` o `hayTresEnRayaDiagonal()` sea TRUE (dichas funciones se describen a continuación).

4.1.12 Rutina hayTresEnRayaFila

```
hayTresEnRayaFila() {
    Bucle que fija el primer elemento de cada fila {
        Bucle: Se verifica que el resto de elementos de la fila sean iguales al fijado
    }
}

* Variables de entrada char *pTablero
* Variables de salida: TRUE o FALSE dependiendo de si se ha encontrado un tres en raya o no
```

Esta función se encarga de verificar la existencia de un tres en raya en cada una de las filas. Realiza una iteración por filas que va fijando como elemento de referencia el primero de cada fila. Una vez fijado cada uno de esos elementos se hace una iteración por columnas, comparando los dos elementos restantes de cada fila con el de referencia. Si en alguna iteración dentro de una fila concreta se detecta un elemento distinto del de referencia, se pasa a comprobar la siguiente fila.

4.1.13 Rutina hayTresEnRayaColumna

```
hayTresEnRayaColumna() {
    Bucle que fija el primer elemento de cada columna {
```

```
Bucle: Verifica que el resto de elementos de la columna sean iguales al
fijado
}
}

* Variables de entrada char *pTablero
* Variables de salida: TRUE o FALSE dependiendo de si se ha encontrado un tres en raya o no
```

Esta función se encarga de verificar la existencia de un tres en raya en cada una de las columnas. Realiza una iteración por columnas que va fijando como elemento de referencia el primero de cada columna. Una vez fijado cada uno de esos elementos se hace una iteración por filas, comparando los dos elementos restantes de cada columna con el de referencia. Si en alguna iteración dentro de una columna concreta se detecta un elemento distinto del de referencia, se pasa a comprobar la siguiente columna.

4.1.14 Rutina hayTresEnRayaDiagonal

```
hayTresEnRayaDiagonal () {
    Se fija como elemento de referencia el central
    Se compara con el resto de elementos de las dos diagonales de forma independiente
}
}

* Variables de entrada char *pTablero
* Variables de salida: TRUE o FALSE dependiendo de si se ha encontrado un tres en raya o no
```

Esta función se encarga de verificar la existencia de un tres en raya en cada una de las diagonales. Fija del elemento central y lo compara, de forma independiente, con los elementos de las dos diagonales existentes.

4.1.15 Rutina cambiaTurno

```
cambiaTurno () {
    Si turno = X, turno = O. Y viceversa
}

* Variables de entrada: char *pTurno
* Variables de salida: void
```

Cuando se llama a esta función, se cambia el turno actual.

4.1.16 Rutina todasLasFichas

```
todasLasFichas () {
    Se recorre el tablero {
        Se cuentan las fichas del turno indicado para ver si están todas
    }
}

* Variables de entrada: char *pTurno, char *pTablero
* Variables de salida: BOOL (TRUE si están todas las fichas)
```

Función que devuelve TRUE O FALSE en función de que un jugador tenga todas sus fichas sobre el tablero o no.

4.1.17 Rutina mueveFicha

```
mueveFicha () {
    Mensaje invitando al jugador a que mueva una ficha
    Mientras la ficha escogida no se pueda mover {
        Se solicita otra (teclado())
    }
}
```

```

        Se muestra el tiempo restante
        Se pide la posición a la cual se debe mover la ficha escogida (teclado())
        Si no se ha perdido el turno {
            Se realiza el movimiento y se pone espacio en blanco en la ficha que se
            decidió mover
        }
    }
}

```

* **Variables de entrada:** char ficha, char *pTurno, char *pTablero

* **Variables de salida:** void

* **Variables globales modificadas:** Se lee la variable turnoPerdido

Función que engloba el movimiento de una ficha ya existente a una nueva posición, comprobando la ficha seleccionada se puede mover a la posición deseada. También se verifica que durante el proceso no se ha perdido el turno, si tal cosa ocurriera, se cancelaría el proceso.

4.1.18 Rutina **mueveFichaOrigenDestino**

```

mueveFichaOrigenDestino() {
    Se vacía la posición original
    Se escribe la ficha correspondiente en la posición deseada (escribeTablero())
}

```

* **Variables de entrada:** char origen, char destino, char *pTurno, char *pTablero

* **Variables de salida:** void

Función que dadas una posición de origen y una de destino, vacía la posición origen del tablero y escribe en la nueva posición la ficha que se ha movido. Se usa sólo en movimientos de la máquina, los cuales son instantáneos y no necesitan comprobaciones de turno perdido.

4.1.19 Rutina **sePuedeMover**

```

sePuedeMover() {
    Se comprueba que no se esté intentando mover la ficha central
    Se comprueba que no se esté seleccionado un hueco en blanco
    Se comprueba que no se esté seleccionado una posición fuera del tablero
    Se comprueba que no se esté intentando mover una ficha del otro jugador
}

```

* **Variables de entrada:** char ficha, char *pTurno, char *pTablero

* **Variables de salida:** BOOL (TRUE en caso de que se cumpla alguna de las cuatro condiciones)

Función a la cual se la llama para comprobar que el jugador no esté seleccionado para ser movida una ficha que no pueda mover, una posición que esté fuera del teclado o un hueco en blanco.

4.1.20 Rutina **teclado**

```

teclado() {
    Mientras no se pulse el teclado {
        Si el tiempo global de partida (modo punt. "3 en raya") se ha agotado,
        devolver el carácter G
        Si el turno ha sido perdido, devolver el carácter G
        Se pinta el tiempo, si t(ms)/1000 es entero (pintoTiempo())
        Se explora el teclado en busca de una pulsación
        retardo(); (1150ns)
    }
}

```

* **Variables de entrada:** void

* **Variables de salida:** char tecla (tecla pulsada)

* Variables globales modificadas: turnoPerdido, pintoTiempo

Estamos ante una de las funciones más importantes del programa, puesto que es la que nos permite interactuar con el Hardware: detecta la tecla pulsada del teclado numérico y la devuelve en forma de char.

Asimismo se usa esta función para explorar repetidamente las variables globales pintoTiempo (cuando está a true, se llama a pintoTiempo, para mostrar el tiempo restante por pantalla) y turnoPerdido (al terminarse el tiempo de turno se interrumpe la iteración para terminar el turno actual y pasar al siguiente).

4.1.21 Rutina cuentaAtras

```
cuentaAtras() {
    Si el tiempo actual es igual al tiempo de turno {
        Se pinta el tiempo de turno
    }
    Si no {
        Se borra el último carácter
        Si el tiempo consta de unidades y decenas se borra un carácter adicional
        Se pinta el tiempo actualizado
    }
    Se setea el flag pintoTiempo a false
}

* Variables de entrada: void
* Variables de salida: void
* Variables globales modificadas: tiempo, pintoTiempo
```

Esta función saca por pantalla el valor del tiempo de turno restante correctamente formateado (borrando el número adecuado de caracteres) para que no se solapen los valores. Por último se setea el flag pintoTiempo a false, para que no se pinten los siguientes 999 milisegundos. (Cuando se llegue al ms 1000, es decir, el siguiente segundo, el TIMER1 seteará pintoTiempo a true para que se pinte dicho segundo.)

4.1.22 Rutina contraMaquina

```
contraMaquina() {
    contraMaquinaTitulo();
    limpiaTablero();
    Mientras no haya ganador {
        preparaTurno();
        Si estamos en turno de máquina: movimientoMaquina();
        Si estamos en turno de jugador: movimientoHumano();
    }
    sleep(); (Se pone el tiempo a cero)
    hayTresEnRaya();
}
melodiaFinal();
muestraTablero();
mensajeFinDeJuego();

* Variables de entrada: BOOL *pHayGanador, char *pTurno, char *pTablero, TipoPunt3Raya
tipo, int *pMovimientos0, int *pMovimientosX
* Variables de salida: void
```

Función principal y que gestiona el juego en el modo “contra la máquina”. Setea los turnos según lo que corresponda, va comprobando la existencia de un tres en raya y en caso de haberlo, se encarga de llamar a los métodos que reproducen la melodía de perdedor y muestra el mensaje de fin de juego.

4.1.23 Rutina movimientoMaquina

```
movimientoMaquina() {
    Selección de una posición libre según criterio:
    Se comprueba que la máquina pueda hacer 3 en raya
        Si no es posible, se busca una posición que evite un 3 en raya del jugador
            Si no es posible, se busca una posición en una esquina
                Si no es posible, se sitúa la ficha en la primera posición libre
                    encontrada
    Si están todas las fichas en el tablero {
        Si en el punto anterior se ha encontrado una posición que de la victoria {
            movimientoFinalMaquina();
        }
        Si no {
            mueveMaquina(); (Se busca para mover una ficha que no comprometa el
                juego para la máquina)
        }
    }
    Si no están todas las fichas {
        escribeTablero(); (Pone una ficha en la posición escogida al principio)
        sleep(); (Se pone el tiempo a cero)
        hayTresEnRaya();
    }
}
```

* Variables de entrada: BOOL *pHayGanador, char *pTurno, char *pTablero

* Variables de salida: void

Esta función es la más importante de las que se encargan de gestionar los movimientos de la máquina. Su funcionamiento se divide en tres fases:

- Busca una posición en la cual conviene situar una ficha (figura 4.2).
- Se decide cuál de las fichas hay que mover, para no facilitar una victoria al contrincante.
- Se escribe la ficha escogida (si ya estaban todas) en la posición elegida en el primer punto.
- Función principal y que gestiona el juego en el modo “contra la máquina”. Setea los turnos según lo que corresponda, va comprobando la existencia de un tres en raya y en caso de haberlo, se encarga de llamar a los métodos que reproducen la melodía de perdedor y muestra el mensaje de fin de juego.

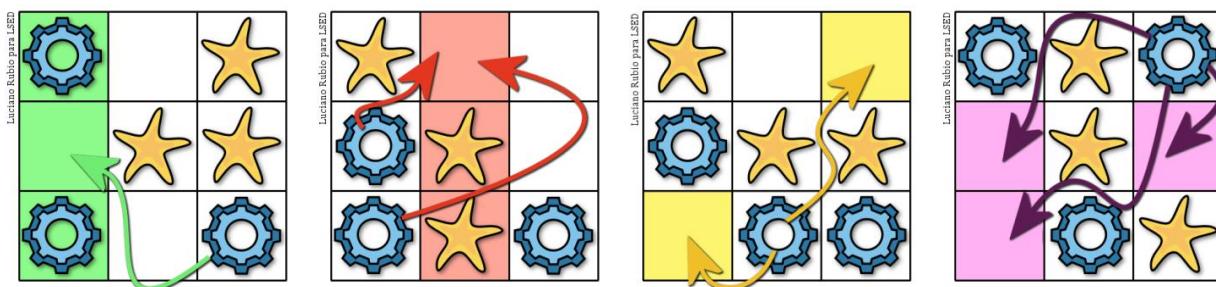


Figura 4.2: Situaciones con las que se puede encontrar la máquina al realizar un movimiento. Ordenadas de más a menos favorables: 1, situación de vistoria para la máquina. 2, situación de bloqueo al contrario. 3, movimiento a las esquinas. 4, ninguna posición preferente.

4.1.24 Rutina mueveMaquina

```
mueveMaquina() {
```

```

    Selección aleatoria de una ficha a mover
    Comprobación de que no deja libre ninguna posición comprometida, en tal caso se
    busca otra ficha que mover
    mueveFichaOrigenDestino();
}

```

* **VARIABLES DE ENTRADA:** BYTE posición, BOOL *pHayGanador, char *pTurno, char *pTablero
* **VARIABLES DE SALIDA:** void

Esta función es llamada desde la función anteriormente descrita sólo en el caso de que todas las fichas estén sobre el tablero y sea necesario mover una de las ya existentes, entre las cuales busca, para decidir cual no deja abiertas opciones de ganar al contrincante.

4.1.25 Rutina posibleTresEnRaya

```

posibleTresEnRaya() {
    Recorro el tablero {
        Si la ficha actual es del tipo que quiero reemplazar, la reemplazo
        Compruebo si hay tres en raya
        Dejo el tablero como estaba antes del reemplazo de ficha
        Si la comprobación de tres en raya ha sido cierta, restablezco *pHayGanador
        A FALSE {
        }
        Si no hay posible 3 en raya, devuelvo -1
    }
    * VARIABLES DE ENTRADA: char ficha_previa, char ficha_post, BYTE indice, BOOL
    *pHayGanador, char *pTablero
    * VARIABLES DE SALIDA: BYTE indice (la posición a la cual si muevo hay tres en raya, o si
    no se puede hacer 3 en raya, -1)
}

```

Función de suma importancia: realiza ensayos. Se le pasa que tipo de ficha (X, O, espacio en blanco) quiero sustituir y cual (X, O). Una vez hecha la sustitución, se ensaya para comprobar si hay habido 3 en raya. En tal caso de devuelve la posición que daría un 3 en raya. Si no se puede hacer 3 en raya, se devuelve -1.

4.1.26 Rutina posiciónAleatoria

```

posicionAleatoria() {
    Recorro el tablero a partir de una posición dada {
        Devuelvo el primer sitio vacío que encuentro a partir de dicha posición
    }
}
* VARIABLES DE ENTRADA: char elemento, BYTE indice, char *pTablero
* VARIABLES DE SALIDA: BYTE (la posición vacía)

```

Realmente se trata de una función pseudoaleatoria, puesto que devuelve la primera posición vacía que encuentra a partir de un índice dado. Las funciones que la llaman comprueban que el elemento que devuelve ni facilita que el contrincante gane. En tal caso se la vuelve a llamar modificando `indice` para que devuelva un nuevo sitio vacío.

4.1.27 Rutina jugadaFinalMaquina

```

jugadaFinalMaquina() {
    Coloco una O en la posición que dará la victoria
    Recorro el tablero {
        Voy comprobando cada una de mis fichas, para ver cuál es la que puedo
        levantar
        Cuando la encuentro la levanto y salgo, en caso contrario sigo buscando
    }
}

```

```

    }
* Variables de entrada: BYTE posicion_ganadora, BOOL *pHayGanador, char *pTurno, char
*pTablero
* Variables de salida: void

```

Función que realiza la jugada final de la máquina (cuando se ha encontrado una posición tal que si se coloca en ella una ficha la máquina gana). Coloca la ficha en la posición ganadora y comprueba que la que va a levantar no acaba con sus opciones de victoria.

4.1.28 Rutina melodíaFinal

```

melodíaFinal() {
    Si hay ganador y si es humano se configura el sistema para que reproduzca la
    melodía de ganador
    En caso contrario se configura el sistema para que reproduzca la melodía de
    perdedor
}

* Variables de entrada: BOOL *pHayGanador, char *pTurno
* Variables de salida: void
* Variables globales modificadas: cambioMelodía = TRUE, estadoJuego (para identificar la
melodía a reproducir)

```

Esta función es llamada cuando se ha terminado una partida del “3 en raya”. Configura el sistema para reproducir la melodía apropiada en función de las circunstancias en las que haya terminado dicha partida.

4.1.29 Rutina notaMusical

```

notaMusical() {
    Configura la variable global duración con la duración indicada
    Configura el TIMER0 con la frecuencia indicada (setTIMER0())
}

* Variables de entrada: int frecuencia, int duración
* Variables de salida: void
* Variables globales modificadas: duración nota

```

Esta función configura el sistema para reproducir una nota concreta. Por una parte de almacena en una variable global la duración de la nota a reproducir (esta duración será leída posteriormente por el TIMER1). La segunda función que cumple es la de configurar el TIMER0 a la frecuencia indicada (ver apdo. TIMER0).

4.1.30 Rutina reproduceNota

```

reproduceNota() {
    Si hay que cambiar la melodía {
        Se resetea el índice de conteo de notas
        Se resetea el flag de cambio de melodía
    }
    Si no se ha llegado al final de la melodía (nota de duración 0) {
        Se reproduce la nota actual (notaMusical())
    }
    Si se ha llegado el final de la melodía actual {
        Si estábamos reproduciendo la melodía de ganador o perdedor {
            Se configura el sistema para que no reproduzca melodía alguna
        }
        Si la melodía se repite {
            Se resetea el índice de conteo de notas, para que la melodía actual se
            pueda volver a reproducir desde el principio
        }
    }
}

```

```

        }
    }

* Variables de entrada struct melodía Música
* Variables de salida: void
* Variables globales modificadas: int indice, estado_juego, BYTE tiempo, BOOL
cambioMelodia, struct melodía Melodia.X

```

Función que comprueba si se ha terminado una melodía. En tal situación resetea los índices que recorren los vectores de notas y duraciones, para que en caso de ser necesario, se vuelva a repetir la melodía que estaba sonando o se comience a reproducir alguna otra.

4.2 Proceso de la interrupción TIMER0

El TIMER0 es el temporizador encargado de generar las frecuencias de las señales audibles. Cada cierto tiempo (duración de una nota), es configurado por el TIMER1 (ver apdo. siguiente) a la frecuencia apropiada, según la frecuencia de la nota que se esté reproduciendo.

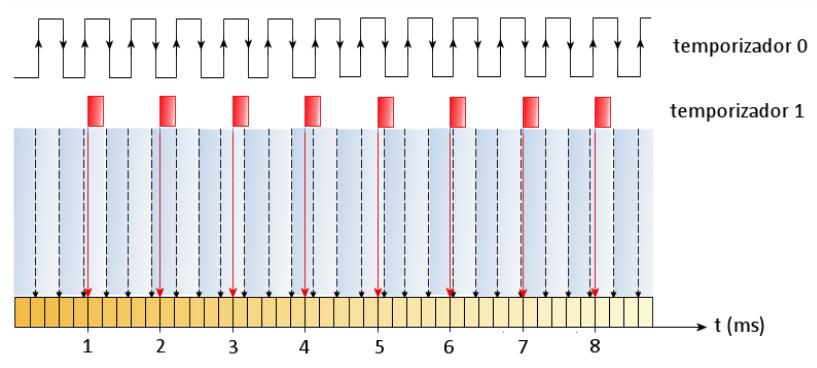


Figura 4.3: Vista de los temporizadores 1 y 0 durante la reproducción de un SOL (6 octava); el TIMER0 genera una señal de frecuencia mayor a la frecuencia de interrupción del timer 1 (que interrumpe cada milsegundo, controlando cuanto tiempo ha de estar el TIMER0 generando la señal actual).

Su manejo y configuración se divide en dos funciones:

4.2.1 Función setTIMER0

```

SetTIMER0() {
    Si la frecuencia deseada es 0 {
        Se configura el registro TMR0 de tal manera que el bit RST sea 0 (timer
        parado)
    }
    En caso contrario {
        Se calcula el valor a introducir en el registro TRR0, para la frecuencia
        que queremos reproducir
        Se configura TMR0 de tal manera que RST1 = 1 (timer habilitado)
        Se configura el registro TCNO
        Se configura el registro TRR0 con el valor ya calculado
        Se configura el registro ICR1 (nivel de interrupción del TIMER0 es )
    }
}

* Variables de entrada: int frecuencia
* Variables de salida: void

```

Esta es la función encargada de configurar el TIMER0 a la frecuencia de la nota actual. Para ello se calcula el valor que ha de tener el registro TRR0 en cada momento, para una frecuencia dada, según la expresión:

$$trr0 = \frac{MCF_CLK/16}{frecuencia \cdot (PS + 1) \cdot 2}$$

En caso de querer reproducir un silencio, se configura el timer de tal manera que el bit de reset del registro TMR0 esté a 0.

4.2.2 Función desactivaTIMER0

```
desactivaTIMER0 () {
    Se configura el registro TMR0 para que el bit RST esté a 0
}

* Variables de entrada: void
* Variables de salida: void
* Variables globales modificadas: duracion nota = 0
```

Respecto a esta función cabe destacar el seteo de `duración_nota` indicado. Se realiza tras detener el registro, para que la rutina TIMER1 no intente seguir usando el TIMER0 para reproducir algún tipo de nota.

4.3 Proceso de la interrupción TIMER1

```
movimientoHumano() {
    Reset del bit de fin de cuenta
    Si la duración de la nota actual > 0 {
        Se decrementa la duración de una unidad
        Si la duración = 0{
            DesactivaTIMER0();
        }
    }
    En caso contrario {
        Se reproduce la melodía adecuada según el estado del sistema
        (reproduceNota())
    }
    Si el tiempo de turno del 3 en raya > 0 (en ms) {
        Si dicho tiempo, dividido por 1000 da como resultado un entero{
            Se guarda dicho tiempo (s) en una variable global
            Se habilita un flag para que dicho tiempo pueda ser leído y mostrado
        }
        Se decrementa en una unidad el tiempo (ms)
        Si dicho tiempo se ha agotado {
            Se configura un flag a TRUE para que el resto del sistema sepa que ha
            habido un evento de "turno perdido"
        }
    }
    Si el tiempo de refresco del Rally > 0 (en ms) {
        Se decrementa
        Si dicho tiempo se ha agotado {
            Se habilita un flag, para que sea pintada una nueva línea del circuito
        }
    }
}

* Variables de entrada char *pTurno, char *pTablero
* Variables de salida: void
* Variables globales modificadas: int duración nota, int índice, estado juego, ULONG
```

```
cont_retardo, BYTE tiempo, BOOL pintoTiempo, ULONG refresco, BOOL turnoPerdido, BOOL finRefresco, BOOL cambioMelodia, struct melodia Musica.X
```

Estamos ante la interrupción temporizada más importante de todo el programa. Realiza una serie de funciones esenciales:

- Decrementa el tiempo de turno del “3 en raya” a razón de una vez por segundo.
- Decrementa el tiempo total de juego del “3 en raya” (si ésta opción es usada).
- Configura el TIMER0 a las frecuencias adecuadas, durante el tiempo adecuado de reproducción. En caso de agotarse la duración de una nota, lee la siguiente y reconfigura el TIMER 0.
- Según la variable global `estado_juego` identifica que melodía se debe reproducir en cada instante.
- Controla el tiempo de turno de la mejora “Sudoku” (ver apdo. mejoras).
- Controla el tiempo de refresco de la mejora “Rally” (ver apdo. mejoras).

Puesto que al tratarse de una rutina de interrupción, no es posible pasarle argumentos, se puede observar que el uso de variables de ámbito global es mucho mayor al que se puede observar en cualquier otra función del sistema.

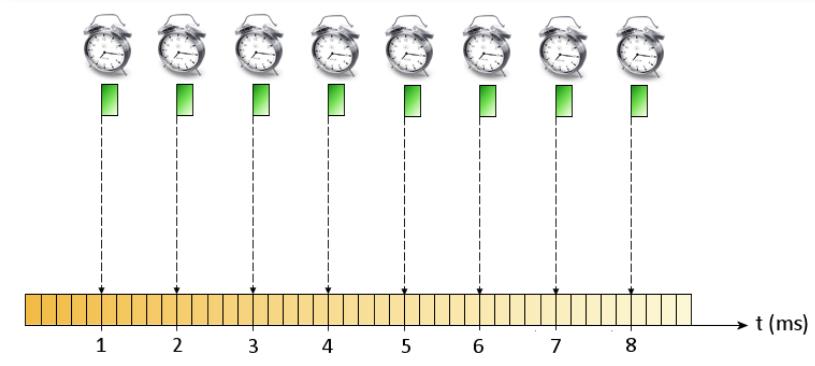


Figura 4.4: Esquema de interrupciones del timer 1, dentro del proceso de ejecución del resto del sistema.

4.3.1 Función setTIMER1

```
SetTIMER1() {
    Se configura TMR1
    Se configura el registro TCN1
    Se configura el registro TRR1
    Se configura el registro ICR1 (nivel de interrupción del TIMER0 es )
}
```

* Variables de entrada: int frecuencia
* Variables de salida: void

Función que contiene la configuración inicial del TIMER1. Está configurado para que interrumpa con una frecuencia de 1 KHz.

Al ser el temporizador más importante, su nivel de interrupción es el más elevado posible. A continuación se muestra una tabla con los valores de configuración de los temporizadores:

- *Configuración del temporizador 1:*

Campo	Valor	Razón
PS	0x4F	Valor de preescalado para que PS+1= 50
CE	0	Captura de entrada inhabilitada
OM	1	Modo de salida (es irrelevante para este caso)
ORI	1	Habilitación de interrupciones en comparación de salida
FRR	1	Activación del modo de reinicio
CLK	0x10	Reloj del sistema con preescalado 16
RST	1	Arranque del temporizador desde la configuración

- *Configuración del temporizador 0:*

Campo	Valor	Razón
PS	0x4F	Valor de preescalado para que PS+1= 50
CE	0	Captura de entrada inhabilitada
OM	1	Se conmuta el modo de salida
ORI	0	Inhabilitación de interrupciones en comparación de salida
FRR	1	Activación del modo de reinicio
CLK	0x10	Reloj del sistema con preescalado 16
RST	1	Arranque del temporizador desde la configuración

5 Descripción de las mejoras

En este apartado recogeremos la especificación de las mejoras realizadas a continuación de la práctica básica.

5.1 Juego del Sudoku para varios jugadores

Siguiendo el esquema de la práctica básica, es decir, la programación de un videojuego simple para la plataforma de desarrollo ENT2004CF, pensamos en desarrollar **otro juego de características similares**. Nos pareció interesante el **Sudoku**, un pasatiempo con un tablero algo más grande que el del tres en raya, pero basado en la lógica y en la colocación adecuada de unos números.

El Sudoku es típicamente un juego para completar en solitario, pero nosotros hemos querido darle un enfoque más divertido y por ello nos hemos inventado una **versión multijugador**. De esta manera todos los participantes estarán atentos a los números que componen el tablero, y la solución se obtendrá más rápidamente, evitando el aburrimiento que pudieran ocasionar los Sudokus de mayor nivel.

5.1.1 Breve historia del Sudoku

Este rompecabezas numérico puede haberse originado en **Nueva York** en 1979. Entonces, la empresa *Dell Magazines* publicó este juego, ideado por Howard Garns, bajo el nombre de *Number Place* (el lugar de los números).

Es muy probable que el Sudoku se crease a partir de los trabajos de **Leonhard Euler**, famoso matemático suizo del siglo XVIII. Dicho matemático no creó el juego en sí, sino que utilizó el sistema llamado del **cuadrado latino** para realizar cálculos de probabilidades.

Posteriormente, la editorial *Nikoli* lo exportó a **Japón**, publicándolo en el periódico *Monthly Nikolist* en abril de 1984 bajo el título "*Sūji wa dokushin ni kagiru*" (数字は独身に限る), que se puede traducir como "los números deben estar solos" (独身 significa literalmente "célibe, soltero"). Fue **Kaji Maki** (鍛治 真起), presidente de *Nikoli*, quien le puso el nombre. Posteriormente, el nombre se abrevió a *Sudoku* (数独; sū = número, doku = solo); ya que es práctica común en japonés tomar el primer kanji de palabras compuestas para abreviarlas. Fuente: *Wikipedia, la enciclopedia libre*.

5.1.2 Las reglas y objetivo del Sudoku

El objetivo de este pasatiempo es **rellenar una cuadrícula** de 9x9 celdas (81 casillas, a las que llamaremos en adelante el tablero) a su vez subdividida en cuadrículas de 3x3 (lo que denominaremos bloques) con las **cifras del 1 al 9** partiendo de algunos números que ya han sido dispuestos en algunas casillas. Para ello, se han de seguir **tres reglas básicas** a la hora de colocar un nuevo número:

- No se puede repetir ninguna cifra en una misma **fila**
- No se puede repetir ninguna cifra en una misma **columna**
- Tampoco se pueden repetir cifras en un mismo **bloque**

5.1.3 Extensiones que hemos desarrollado para el juego básico

Lo que hemos explicado anteriormente son las reglas necesarias e inmutables para jugar al Sudoku básico. Nosotros hemos querido añadirle la **capacidad de multijugador**, rompiendo con el clásico de que en este juego sólo participa una persona y además tarda bastantes minutos en completarlo.

Para ello hemos desarrollado un sistema de **turnos** en que varios jugadores (de 1 a 9, aunque ampliable) van introduciendo los números según les toque. De esta manera, todos los participantes permanecen atentos a la pantalla intentando localizar algún nuevo acierto, y el Sudoku se va **completando a una velocidad mayor** de la que lo hacía una única persona.

Es sabido por todos los aficionados al Sudoku que según la dificultad del tablero, es inevitable que llegue un punto en que haya que **arriesgar** o no tengamos el conocimiento pleno de que nuestra suposición acabe correctamente. En algunos casos esta situación acaba satisfactoriamente y el jugador alcanza la solución, pero otras veces se llega a un bloqueo donde no se puede avanzar y hay que retroceder, lo que provoca que el jugador se canse y abandone la partida en muchos de los casos.

Por ello hemos creado el **modo Ayuda** (que podemos activar o desactivar), que nos permite conocer si los números que vamos colocando **son los correctos o no**. De esta forma el sistema nos impide llenar el tablero con equívocos, y por tanto estamos dando más pistas de cómo llegar a la solución (será especialmente útil para los casos en los que dudemos entre dos números para una determinada casilla). Evidentemente con este modo Ayuda **reducimos la dificultad** del juego, pero a la vez **aumentamos la diversión**, ya que en muchos casos, los jugadores se pueden “aprovechar” de los fallos de otros, y de esta forma se va llenando el tablero, por difícil que sea.

Al acabar la partida, necesitamos añadir un **sistema de puntuación** para determinar el **ganador** del juego o una situación de empate. Por ello hemos desarrollado dos sistemas básicos para anotar puntos a los jugadores.

El primero de ellos se basa en el **número de aciertos**, es decir, cada jugador tiene un tiempo de turno, y en él la posibilidad de escribir un número en el tablero. Si la suposición es correcta, **se anota el tanto** (que puede ser informado según esté o no activado el modo Ayuda), y le toca al siguiente participante. Al completar el Sudoku se hace un recuento de números acertados y gana el que tenga mayor puntuación.

El segundo método se basa en la **rapidez con la que rellenemos** el tablero. Cuando le toca a un jugador, dispone de un **tiempo de turno** (1 minuto, aunque se puede modificar) que se va decrementando. En el momento en que se anota el acierto, se guardan los **segundos restantes** como los puntos obtenidos en ese momento, de manera que se pueden conseguir de **0 a 60 puntos** por vez que nos toque, acumulándose a nuestro contador personal. Según este modelo se valora a los jugadores más rápidos que ya tenían una suposición antes de que les tocara, debido a la atención prestada anteriormente. Es de esperar que cuando un jugador raudo **falla** (si es que tenemos el **modo Ayuda** que nos lo advierte), **pierda el turno** y por tanto no se anote ningún punto a su marcador.

Finalmente hemos querido disponer de una variedad de tableros para jugar, por ello hemos almacenado **9 niveles** en el programa (de menor a mayor dificultad), que pueden ser seleccionados al inicio de la partida. Además, los hemos caracterizado como “niveles”, ya que al finalizar un Sudoku, se pasa al siguiente, manteniendo además el **ciclo de turnos** de los jugadores. De esta forma se evita que los primeros participantes tengan preferencia y encuentren los números más fáciles (que precisamente son los primeros que se colocan en una partida de Sudoku).

5.1.4 Esquema del juego Sudoku desarrollado

Una vez explicadas todas las extensiones que hemos implementado para el juego Sudoku básico, procedemos a resumir en un esquema el funcionamiento normal de una partida, con el programa cargado en la plataforma ENT2004CF:

1. Seleccionamos el juego **Sudo Kuul Fire** en el menú principal
2. Se muestra la pantalla de bienvenida
3. Se pide modo **local** o modo en **red** (se explicará en el *apartado 5.3*)
4. Se pide al usuario el **número** de jugadores (de 1 a 9)
5. Se pregunta por la activación del **modo Ayuda** durante el juego
6. Se pide el sistema de **puntuación** (por aciertos o por rapidez)
7. Se solicita el nivel de **dificultad** inicial (desde el 1º, el más sencillo, al 9º, el más complejo)

A continuación comienza la partida, y el esquema es el siguiente:

1. Comienza el turno del jugador 1 y dispone de **un minuto** para escribir un número
2. **Escribe un número** en el tablero
3. Si está activado el **modo Ayuda**, se le avisa del acierto o el fallo
4. Continúa del jugador 2, y así sucesivamente, hasta que le toca al jugador 1 de nuevo
...
5. Cuando se completa el Sudoku, se muestran los resultados y el **ganador**

5.1.5 Breve análisis de cómo jugar al Sudoku

Una vez estamos dentro de una partida Sudoku, una manera de empezar a llenar las casillas vacías será la siguiente:

1. Primero se buscan los **números triviales**. Se encontrarán en aquellas casillas donde sólo pueda darse esa opción, debido a la posición del resto de números.
2. Una vez se hayan acabado los triviales, se procede al **recuento** en filas, columnas o bloques, buscando los números que faltan.
3. Cuando el recuento no desvela ningún posible número, se procedería al **marcado**, es decir, señalar los posibles números que podrían ocupar cada casilla.

4	7	3		5				2
		1	9	2		3		
		5	4		8	1		
7		6		4				8
	1		7	8	5		6	3
8		4			7			
	4	7	6	8	1			
1			5		9	4		
5				4	3	8		

4	7	3		5				2
			9	2		3		
		5	4		8	1		
7		6		4				8
	1		7	8	5	4	6	3
8		4				7		
	4	7	6	8	1			
1			5		9	4		
5				4	3	8		

Figura 5.1: Localización de números triviales en un tablero de Sudoku. Debido a los números del contorno, podemos deducir por eliminación la única posible opción numérica para una casilla vacía.

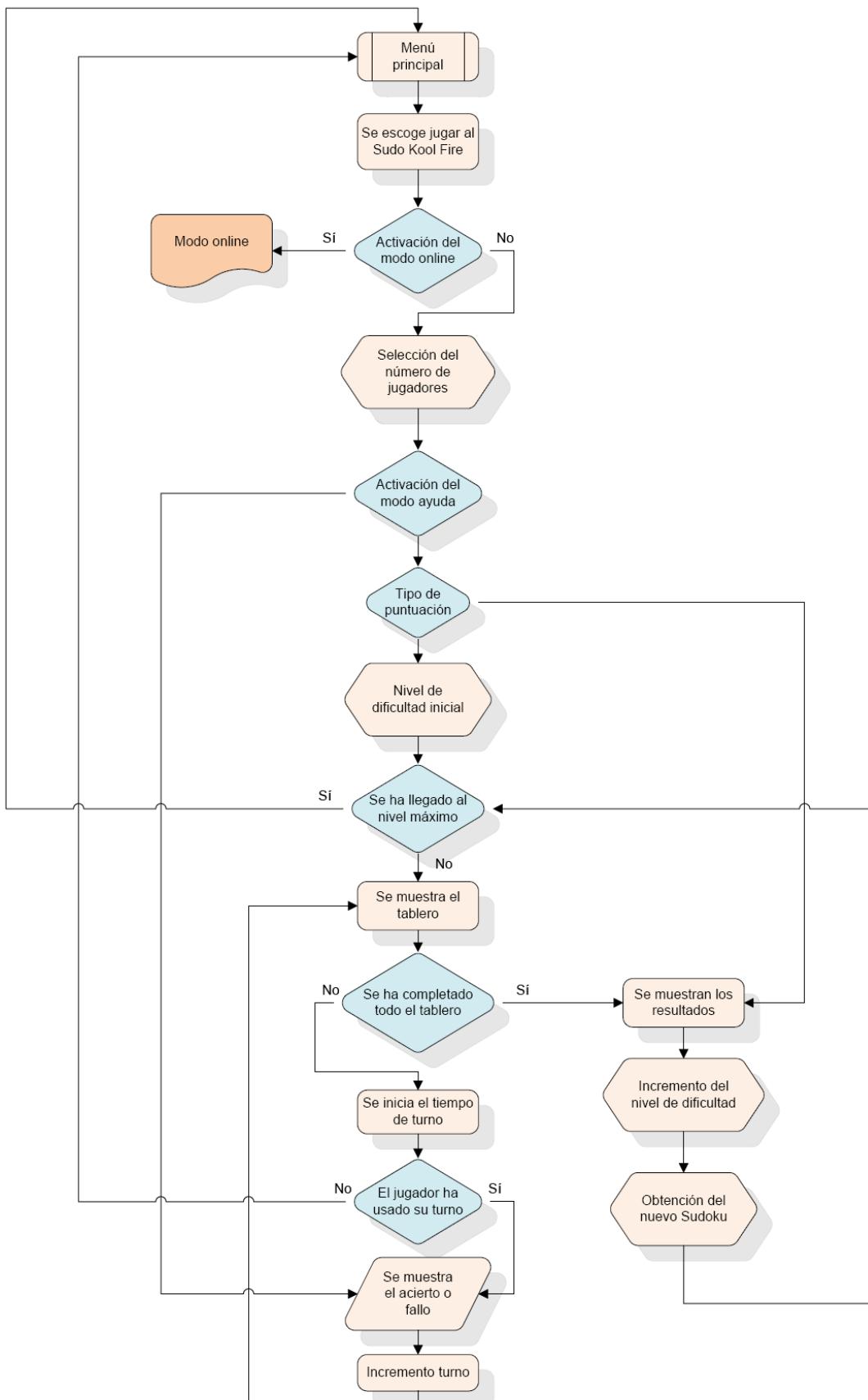
4	7	3	8	5	1	6	9	2
?	?	1	9	7	2	5	3	4
?	?	5	4	?	?	8	7	1
7	?	6		4	?	2	?	8
?	1		7	8	5	4	6	3
8	?	4	?	?	?	7	?	?
?	4	7	6	?	8	1	2	5
1	?	?	5	?	?	9	4	?
5	?	?	?	?	4	3	8	?

Figura 5.2: Cuando se completan todos los números de carácter trivial, se puede hacer un estudio exhaustivo de todas las posibles soluciones para todas las casillas vacías, y dependiendo de la dificultad del tablero, se llegará antes a la solución por descartes.

Este último punto es la **estrategia típica** utilizada por los aficionados a los Sudokus **impresos en papel**, lo que hace que realmente tardemos muchos minutos en completarlo y donde más abandono se produce.

Nosotros hemos **cambiado literalmente** este último paso, ya que la tendencia en nuestro nuevo Sudoku cuando haya dudas será que el jugador **arriesgue**, ya que en el peor de los casos (se equivoque), ni siquiera perderá puntos, sino que perderá el turno y favorecerá a los siguientes jugadores que ahora han recibido nuevas pistas (números que son incorrectos).

5.1.6 Diagrama del programa principal



5.1.7 Proceso del programa principal

```
sudokuJuego() {
    Bucle de nivel de dificultad {
        inicializaPuntuacion();
        Bucle de Turno {
            refrescoTablero();
            haySudoku(); (Comprobación de tablero completado)
            refrescaTurno();
            movimientoSudoku(); (Proceso de escritura en tablero)
            incrementoTurno();
        }
        muestraResultados();
        Aumento del nivel de dificultad
        setConf(); (Mantiene la configuración inicial)
    }
}

* VARIABLES DE ENTRADA: char *pTabSudoku, int *pTabTurnos, int *pRed, int *pNumJugadores,
int *pAciertos, int *pGanador, int *pJugador, int *pDificultad, int *pAyuda, int
*pGanarPorTiempos, char *pSolucion
* VARIABLES DE SALIDA: void
* VARIABLES GLOBALES CONSULTADAS: Reseteo de pintoTiempo (pintoTiempo = FALSE) en caso de
que el usuario pulse "salir"
```

En el pseudo-código anterior se muestra el **bucle principal** del juego de Sudoku. Al principio se le hacen al usuario unas preguntas acerca del funcionamiento del juego, y sus respuestas son guardadas como la **configuración inicial**.

A continuación, se procesa el **bucle de niveles**, que permitirá que se vaya avanzando en dificultad conforme se completen los tableros. Una vez dentro del juego, se hace el **ciclo de turnos** según el número de jugadores indicado, y en cada iteración destacamos los siguientes pasos:

- Se **refresca** el tablero, es decir, se muestra por pantalla su nueva situación numérica
- Se comprueba si el tablero está **completado** para pasar al siguiente nivel
- Si aún quedan números, empieza un turno y se **activa** su marcador de tiempo
- El jugador realiza su movimiento (sitúa un número en el tablero), siempre y cuando no se le haya agotado su tiempo de turno
- Se pasa al siguiente jugador

En caso de que el nivel se complete, se prepara el nuevo tablero y se dispone de la nueva solución manteniendo el resto de parámetros fijos. Es por ello que después de mostrar la tabla de puntuaciones y el ganador, es necesario actualizar las variables del sistema.

Finalmente, cuando se completa el tablero de máximo nivel, el juego del Sudoku termina y se vuelve al menú principal.

En la figura siguiente se muestra el esquema de ciclo de turnos durante una partida de Sudoku. Los jugadores van escribiendo números y disponen de un tiempo limitado cada uno.

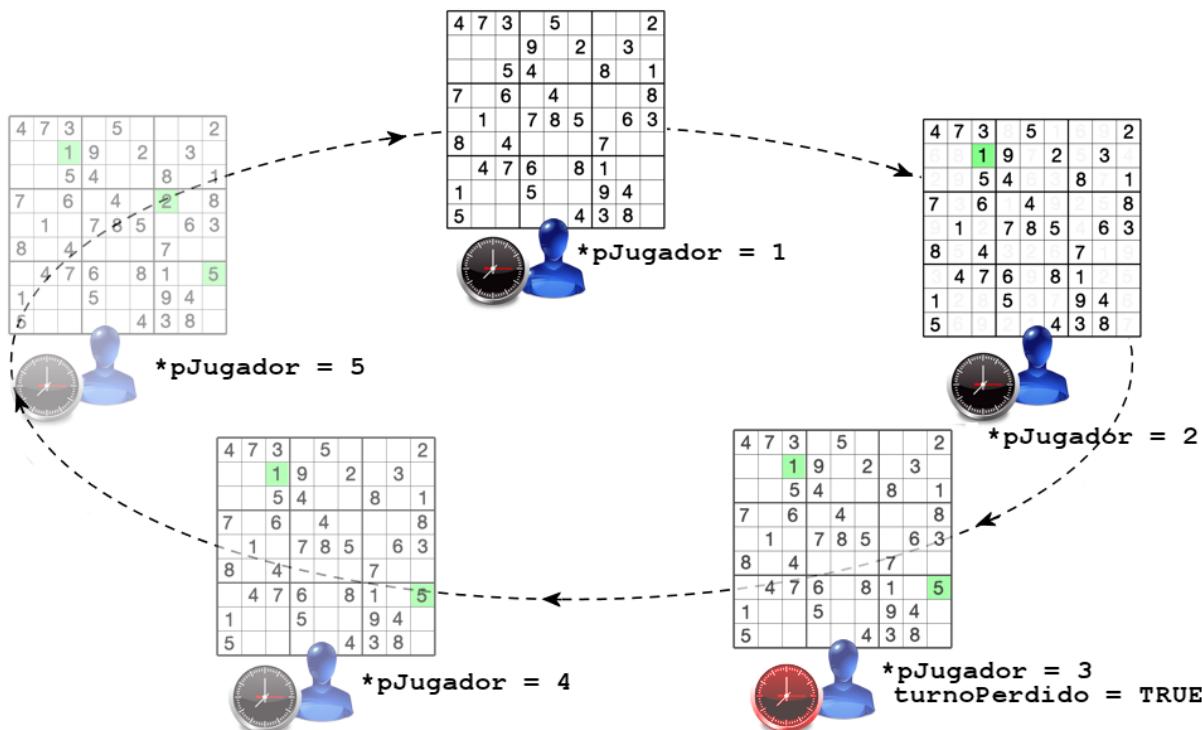


Figura 5.3: Ejemplo de ciclo de turnos en el Sudoku Multijugador (caso de 5 participantes)

5.1.8 Variables del sistema

En el siguiente esquema se caracterizan las variables principales del sistema y los punteros definidos para ellas.

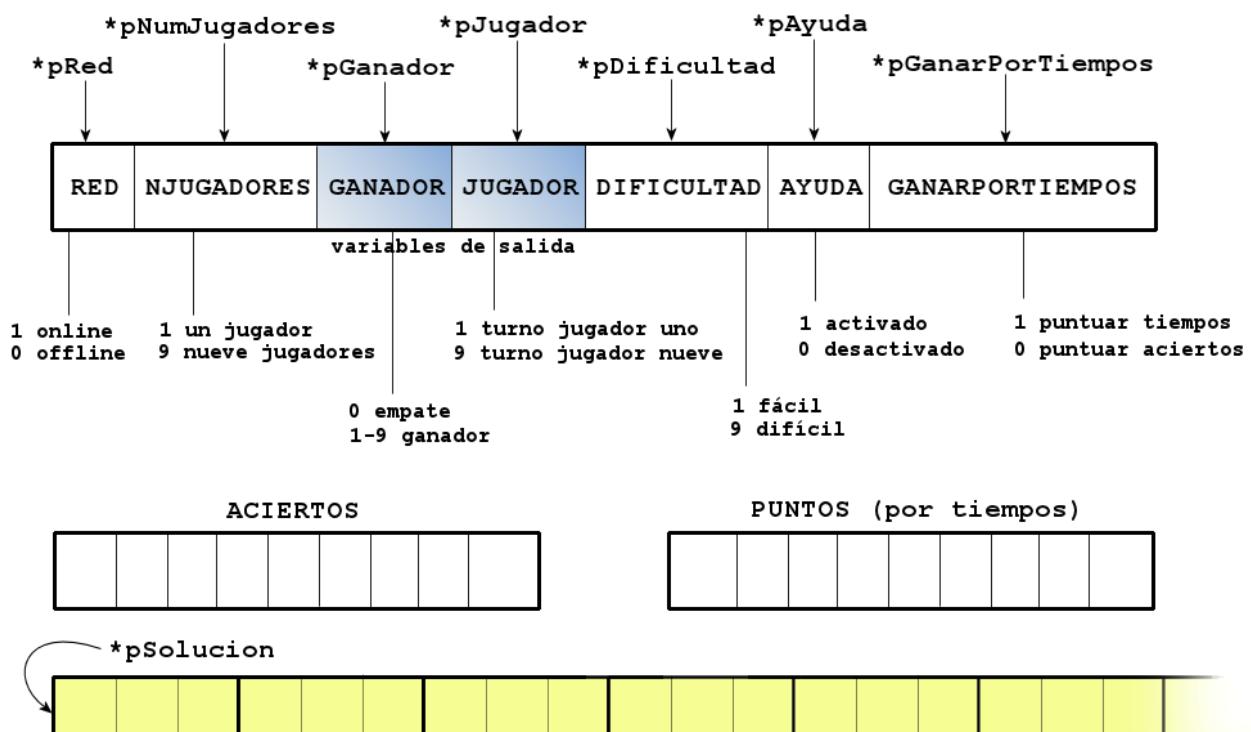


Figura 5.4: Esquemático de las variables y punteros definidos en el desarrollo del Sudoku

Para la definición del tablero Sudoku, hemos optado por escribirlo como **vector** en vez de cómo matriz, pues de esta manera acortamos las funciones que necesiten localizar en qué fila y en qué columna se ha escrito un nuevo número.

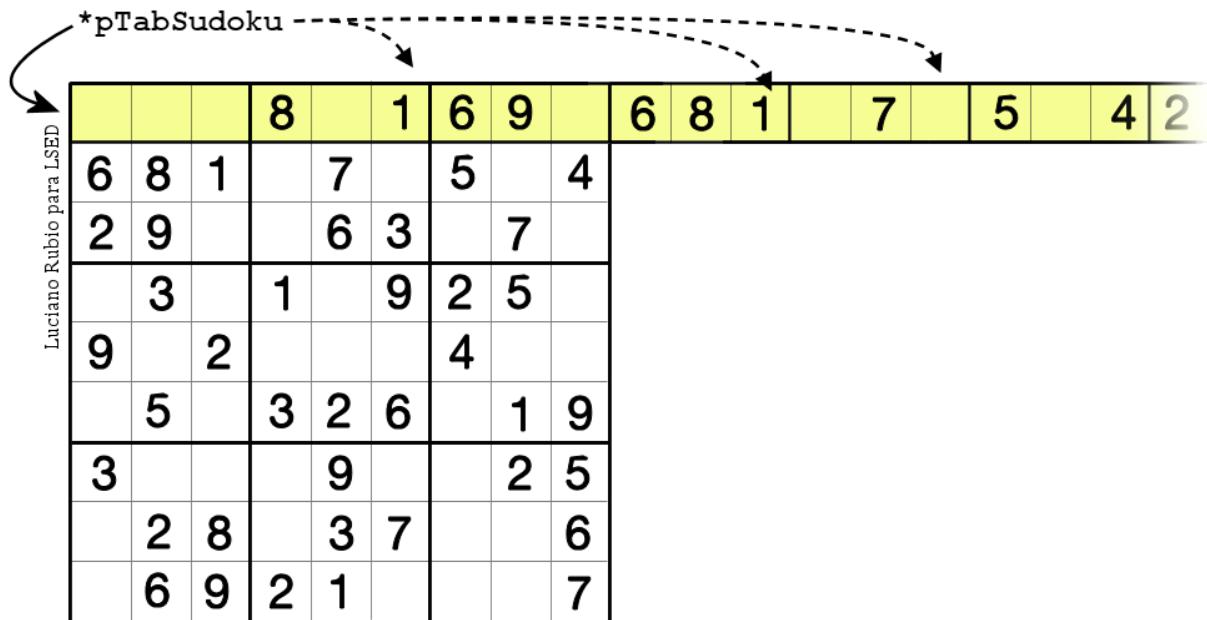


Figura 5.5: Comparativa entre el esquema de matriz 9x9 habitual, a la definición de vector de 81 cifras

A la par que el tablero Sudoku, se define otro **vector** de idéntica medida que almacena el **turno del jugador que ha acertado** el número. De esta forma se facilita más adelante la cuenta total de aciertos, en caso de que se haya elegido ese tipo de puntuación.

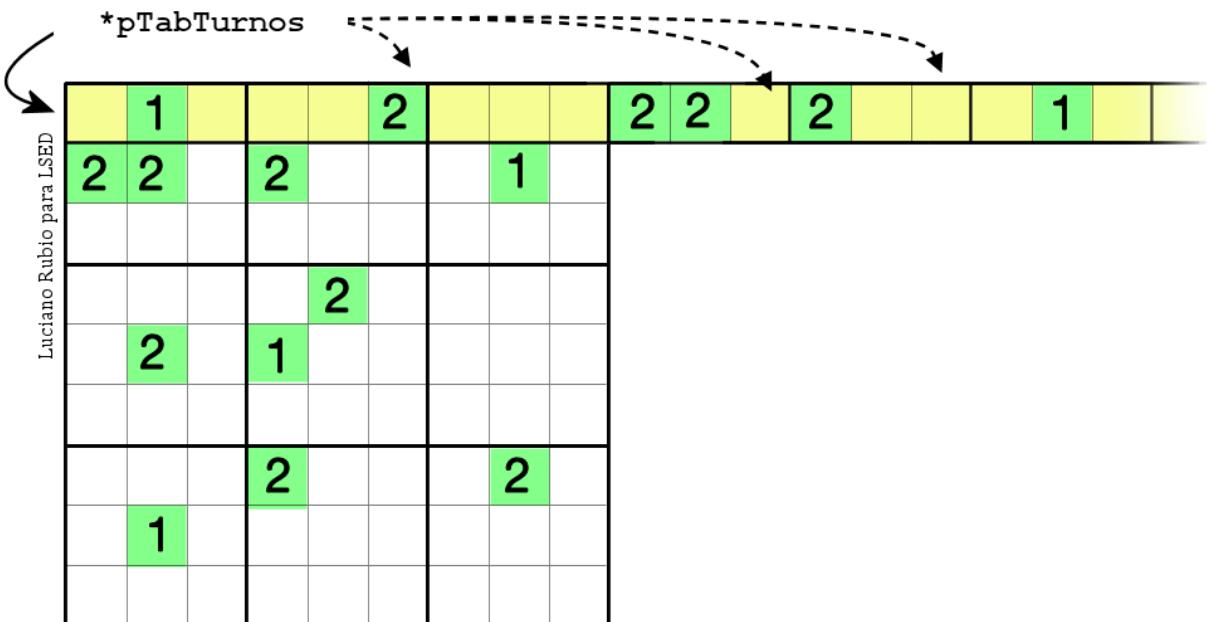


Figura 5.6: Mismo caso que en la figura anterior pero almacenando los aciertos de los jugadores

5.1.8.1 Funcionalidad iniciaSudoku

```
iniciaSudoku() {
    Creación del tablero
    Creación de variables y flags
    Creación de punteros
    setConfiguracion();
    sudokuJuego();
}

* Variables de entrada: void
* Variables de salida: void
```

Esta función define todas las variables y punteros que se utilizarán durante la partida de Sudoku, y recoge la configuración indicada por el usuario. Una vez dispone de todo lo suficiente, es la encargada de invocar a la funcionalidad del juego Sudoku principal.

5.1.8.2 Funcionalidad refrescoTablero

```
refrescoTablero() {
    ... (Funciones de modo online)
    Reseteo de tiempo perdido de jugador
    muestraTableroSudoku();
}

* Variables de entrada: char *pTabSudoku, int *pRed, int *pNumJugadores, int *pJugador
* Variables de salida: void
* Variables globales modificadas: Reseteo de tiempoPerdido (tiempoPerdido = FALSE)
```

Esta función es la que se encarga de representar el nuevo estado del tablero Sudoku por pantalla, de manera que el próximo jugador conozca el estado actual de la partida. Además recoge algunas de las funcionalidades del modo Sudoku online que se explicarán posteriormente, en los apartados 5.2 y 5.3.

5.1.8.3 Funcionalidad refrescaTurno

```
refrescaTurno() {
    Muestra mensaje por pantalla del turno actual
    Arranca el tiempo de turno
}

* Variables de entrada: void
* Variables de salida: void
```

Esta función se encarga de representar por pantalla los datos referidos al turno actual. Muestra el número del jugador a quien le toca rellenar el tablero, y arranca la cuenta atrás de tiempo. Su importancia está en el seteo de segundos disponibles para ese turno.

5.1.8.4 Funcionalidad incrementoTurno

```
incrementoTurno() {
    ... (Funciones de modo online)
    Asigna el turno al jugador siguiente
    Para el tiempo de turno
}

* Variables de entrada: char *pTabSudoku, int *pRed, int *pJugador
* Variables de salida: void
* Variables globales modificadas: Reseteo del flag pintoTiempo (pintoTiempo = FALSE)
```

Esta función se encarga de incrementar el turno para que le toque al siguiente jugador. También maneja funciones del modo online que se explicarán más adelante.

5.1.8.5 Funcionalidad setConfiguracion

```
setConfiguracion() {
    setRed();
    setJugadores();
    setAyuda();
    setPuntuacion();
    setDificultad();
    devuelveSolucion();
}

* Variables de entrada: int *pRed, int *pNumJugadores, char *pTabSudoku, int *pDificultad,
int *pAyuda, int *pGanarPorTiempos
* Variables de salida: char* (puntero a la solución del sudoku)
```

Esta función hace una lista de llamadas a funcionalidades que configurarán el estado inicial de la partida. En todas ellas se le pide al usuario el uso del teclado matricial para escribir los modos de juego que le interesen. En total se le solicita:

- Activación del modo online
- Número de jugadores para la partida
- Activación del modo ayuda durante el juego (permite saber los errores cometidos)
- Sistema de puntuación: por aciertos o por rapidez
- Nivel de dificultad inicial

A continuación, se actualiza la definición del puntero solución del Sudoku según el nivel de dificultad escogido para ser utilizado en el modo ayuda.

```
> ¿Deseas jugar un sudoku en red o local? [1 Red | 0 Local]: 0
> Indica el número de jugadores [de 1 a 9]: 2
> ¿Quieres activa el modo ayuda durante el juego? [1 Sí | 0 No]: 1
> ¿Cómo se puntúa? [0 Por aciertos | 1 Por tiempos] 0
> Indica la dificultad del juego [1 fácil - 9 difícil] 1
> ...
```

5.1.8.6 Funcionalidad setConf

```
setConf() {
    Configuración inicial del número de jugadores
    Configuración inicial del modo ayuda
    Configuración inicial del tipo de puntuación
    devuelveSolucion();
}
```

```
* Variables de entrada: int *pRed, int *pNumJugadores, int numJugadores, char *pTabSudoku,
int *pDificultad, int dificultad, int *pAyuda, int ayuda, int *pGanarPorTiempos, int
ganarPorTiempos
* Variables de salida: char* (puntero a la solución del sudoku)
```

Esta función se encarga de mantener la configuración inicial del juego y de actualizar convenientemente el puntero solución del Sudoku según el nuevo nivel de dificultad.

5.1.8.7 Funcionalidad devuelveSolucion

```
devuelveSolucion() {
    ... (Funciones del modo online)
    Inicializa el tablero de sudoku según dificultad
    Obtención del sudoku solución
}
```

```
* Variables de entrada: char *pTabSudoku, int *pRed, int *pDificultad
* Variables de salida: char* (puntero a la solución del sudoku)
```

Esta función devuelve el puntero solución del Sudoku según se le indique un nivel de dificultad. Es de vital importancia para cuando el modo ayuda se encuentre activo, pues permite comparar los números introducidos en el tablero con su solución.

5.1.8.8 Funcionalidad setRed

```
setRed() {
    Solicitud al usuario de la activación del modo online
    Se imprime por pantalla su respuesta
    Comprobación de respuesta válida
}
```

```
* Variables de entrada: int *pRed
* Variables de salida: void
```

Esta función solicita al usuario la activación o no del modo online, mostrándole por pantalla la pregunta y comprobando que su respuesta es válida dentro de las opciones disponibles (si escribe 1 se activará, 0 para el caso contrario).

5.1.8.9 Funcionalidad setJugadores

```
setJugadores() {
    Solicitud al usuario del número de jugadores
    Se imprime por pantalla su respuesta
    Comprobación de respuesta válida
}

* Variables de entrada: int *pNumJugadores
* Variables de salida: void
```

Esta función solicita al usuario el número de jugadores participantes, mostrándole por pantalla la pregunta y comprobando que su respuesta es válida dentro de las opciones disponibles (ha de escribir un número del 1 al 9 en el teclado matricial).

5.1.8.10 Funcionalidad setDificultad

```
setDificultad () {
    Solicitud al usuario del nivel de dificultad inicial
    Se imprime por pantalla su respuesta
    Comprobación de respuesta válida
}

* Variables de entrada: char *pTabSudoku, int *pRed, int *pDificultad
* Variables de salida: char* (puntero a la solución del sudoku)
```

Esta función solicita al usuario el nivel de dificultad inicial, mostrándole por pantalla la pregunta y comprobando que su respuesta es válida dentro de las opciones disponibles (ha de escribir un número del 1 al 9 en el teclado matricial).

5.1.8.11 Funcionalidad setAyuda

```
setAyuda() {
    Solicitud al usuario de la activación del modo ayuda
    Se imprime por pantalla su respuesta
    Comprobación de respuesta válida
}

* Variables de entrada: int *pAyuda
* Variables de salida: void
```

Esta función solicita al usuario la activación o no del modo ayuda, mostrándole por pantalla la pregunta y comprobando que su respuesta es válida dentro de las opciones disponibles (ha de escribir 1 para activarlo, 0 en caso contrario).

5.1.8.12 Funcionalidad setPuntuacion

```
setPuntuacion() {
    Solicitud al usuario del tipo de puntuación
    Se imprime por pantalla su respuesta
    Comprobación de respuesta válida
}

* Variables de entrada: int *pGanarPorTiempos
* Variables de salida: void
```

Esta función solicita al usuario el sistema de puntuación, mostrándole por pantalla la pregunta y comprobando que su respuesta es válida dentro de las opciones disponibles (ha de escribir 0 para un recuento de aciertos, o 1 para un recuento de tiempos).

5.1.8.13 Funcionalidad actualizaPuntos

```
actualizaPuntos() {
    Comprobación del tipo de puntuación
    Se incrementa el marcador del turno actual
}

* Variables de entrada: int *pJugador, int *pPuntos, int *pTabTurnos, int posición, int
*pGanarPorTiempos
* Variables de salida: void
```

Esta función actualiza los puntos según el sistema de recuento elegido. Si el sistema es por aciertos, únicamente guarda el número de jugador asociado al número acertado (como se muestra en la figura posterior); en cambio si es por tiempos se actualiza el marcador del jugador correspondiente.

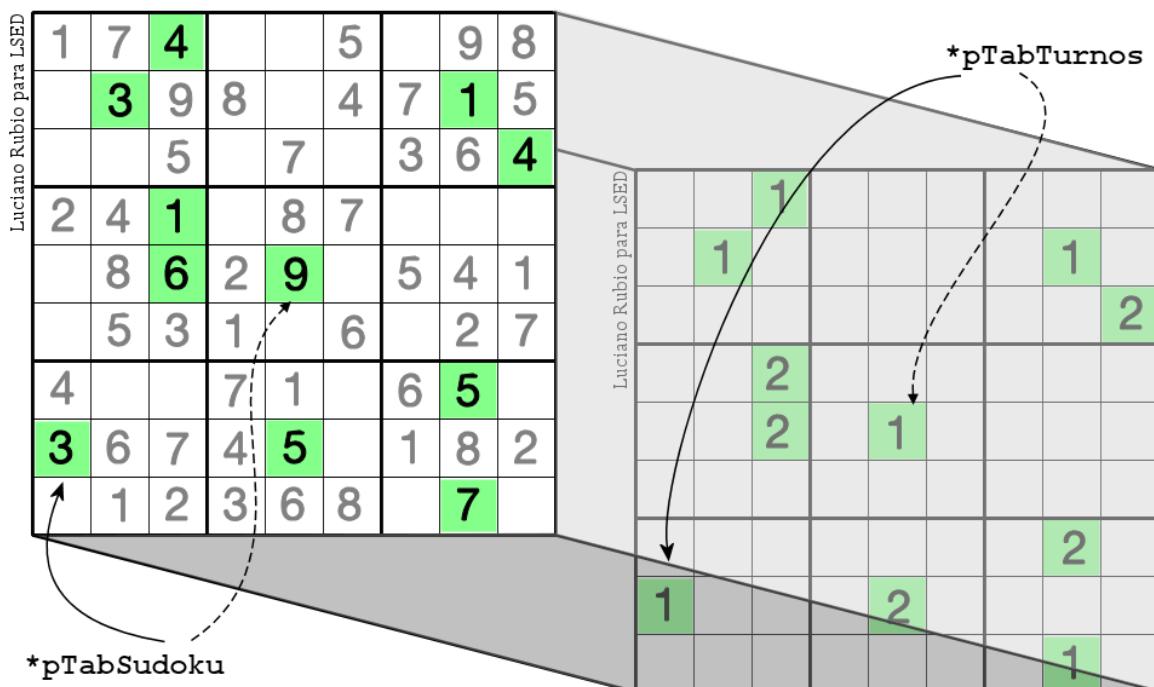


Figura 5.7: Esquema gráfico de cómo se identifican los aciertos en el sudoku con los jugadores acertantes

5.1.8.14 Funcionalidad muestraResultados

```
muestraResultados () {
    Bucle de jugadores {
        Se imprime por pantalla el número del jugador
        Comprobación del tipo de puntuación
        Se imprime por pantalla su puntuación
    }
    Comprobación de empate
    Se imprime por pantalla el jugador ganador
}

* Variables de entrada: int *pNumJugadores, int *pAciertos, int *pPuntos, int *pGanador, int
*pGanarPorTiempos, int *pRed
* Variables de salida: void
```

Esta función imprime por la pantalla una lista simple de puntuaciones finales una vez se ha completado un Sudoku. Para ello recorre los marcadores de los jugadores y muestra la puntuación en función del sistema de recuento elegido. Si es por aciertos, se cuenta en el tablero los números que ha colocado cada usuario; en cambio, si es por tiempos se muestra el marcador de segundos.

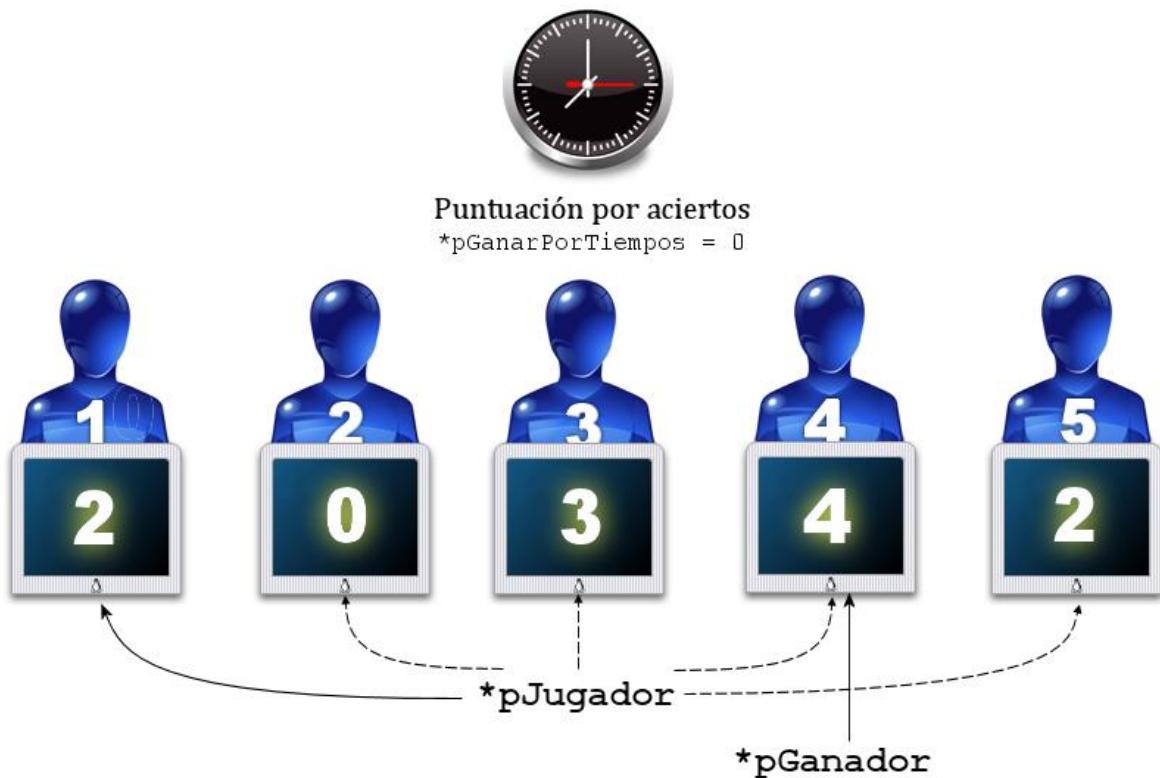


Figura 5.8: Esquema de marcadores (ejemplo de 5 participantes) según la puntuación por aciertos

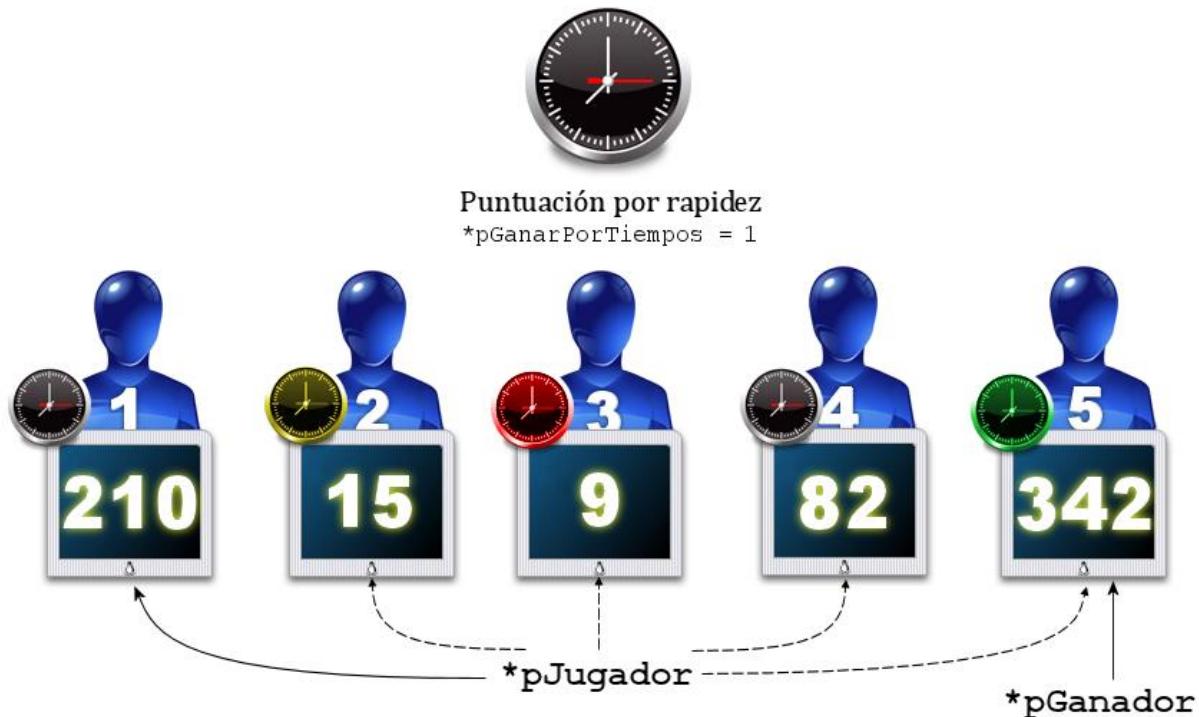


Figura 5.9: Esquema de marcadores (para 5 participantes) según la puntuación por rapidez

5.1.8.15 Funcionalidad hayRepeticion

```
hayRepeticion() {
    Se hace un ensayo con el número que ha puesto el jugador
    Comprobación de número repetido (fila, columna o bloque) {
        hayRepeticionFila();
        hayRepeticionColumna();
        hayRepeticionBloque();
    }
}
```

* Variables de entrada: char *pTabSudoku, int posicion, char numero
* Variables de salida: BOOL (TRUE si hay repetición)

Esta función permite conocer al sistema si el nuevo número introducido en el tablero cumple las tres normas básicas del Sudoku, independientemente de que el número sea correcto o no. Para ello se comprueba si hay alguna repetición de número en la misma fila, columna o bloque. Funciona realizando un ensayo con el nuevo número y detectando una repetición.

5.1.8.16 Funcionalidad hayRepeticionFila

```

hayRepeticionFila() {
    Bucle de Filas a recorrer {
        Bucle de casilla dentro de fila {
            Selecciona el número de la casilla
            Bucle de recorrido de fila {
                Comprobación de repetición
            }
        }
    }
}

* Variables de entrada: char *pTabSudoku
* Variables de salida: BOOL (TRUE si hay repetición en fila)

```

Esta función recorre todas las filas del tablero buscando alguna repetición de número. Para ello, se iteran las filas, y una vez dentro, se van escogiendo uno a uno los números que aparezcan y se comprueba horizontalmente si existe un duplicado.

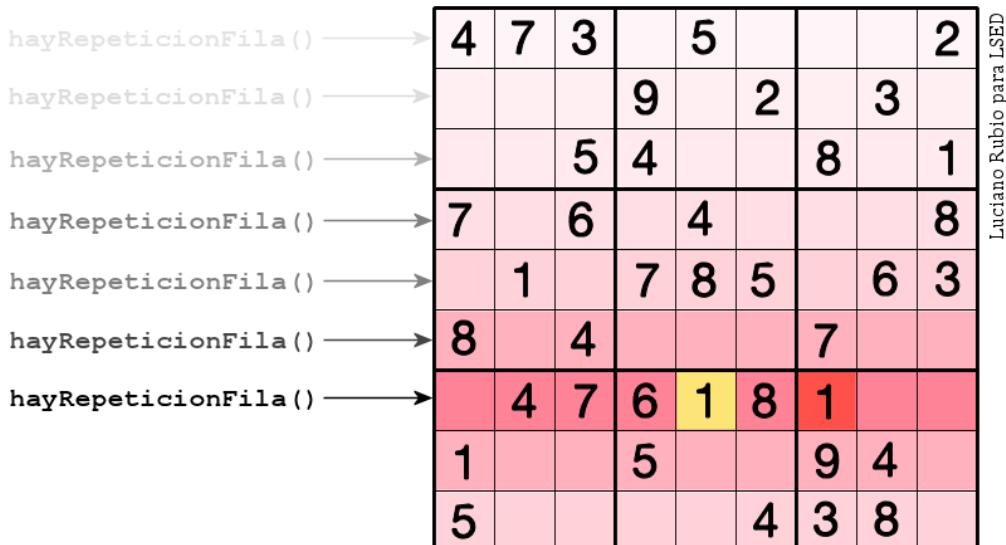


Figura 5.10: Iteración por filas para detectar una repetición de número (1^a regla del Sudoku)

5.1.8.17 Funcionalidad hayRepeticionColumna

```

hayRepeticionColumna() {
    Bucle de Columnas a recorrer {
        Bucle de casilla dentro de columna {
            Selecciona el número de la casilla
            Bucle de recorrido de columna {
                Comprobación de repetición
            }
        }
    }
}

* Variables de entrada: char *pTabSudoku
* Variables de salida: BOOL (TRUE si hay repetición en columna)

```

Esta función recorre todas las columnas del tablero buscando alguna repetición de número. Para ello, se iteran las columnas, y una vez dentro, se van escogiendo uno a uno los números que aparezcan y se comprueba verticalmente si existe un duplicado.

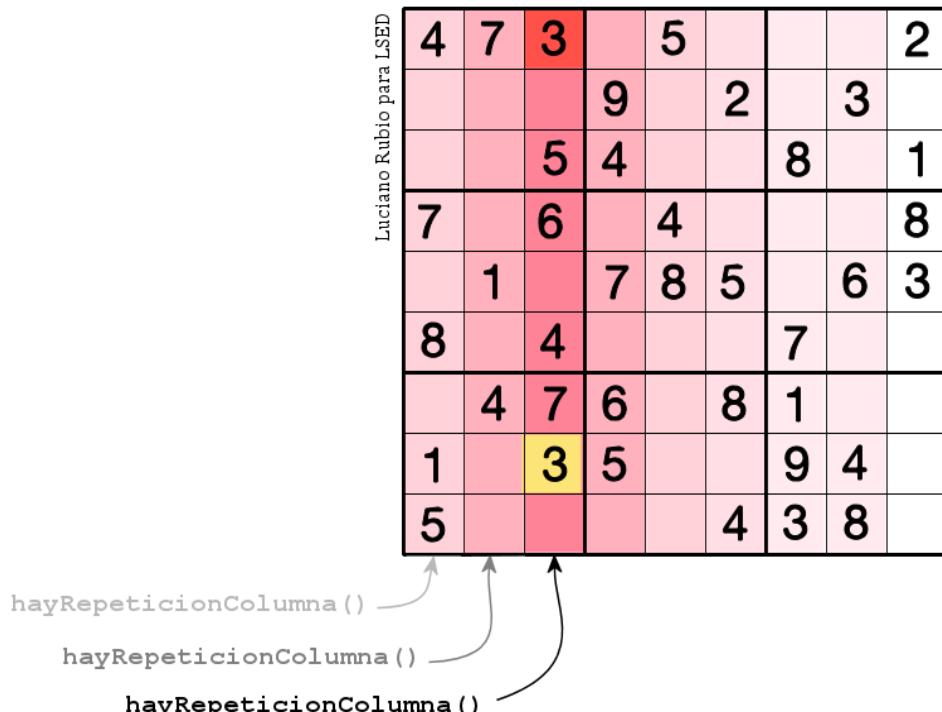


Figura 5.11: Iteración por columnas para detectar una repetición de número (2^a regla del Sudoku)

5.1.8.18 Funcionalidad hayRepeticionBloque

```

hayRepeticionBloque() {
    Bucle de la columna comienzo del bloque {
        Bucle de la fila comienzo del bloque {
            Selecciona el número de la casilla
            Bucle de Filas a recorrer {
                Bucle de casilla dentro de fila {
                    Comprobación de repetición
                }
            }
            Bucle de Columnas a recorrer {
                Bucle de casilla dentro de columna {
                    Comprobación de repetición
                }
            }
        }
    }
}

```

* Variables de entrada: char *pTabSudoku
* Variables de salida: BOOL (TRUE si hay repetición en bloque)

Esta función recorre todos los bloques 3x3 del tablero buscando alguna repetición de número. Para ello, se iteran los bloques (gracias a dos índices que marcan la fila y columna inicial de bloque), y una vez dentro, se van escogiendo uno a uno los números que aparezcan y se comprueba vertical y horizontalmente si existe un duplicado.

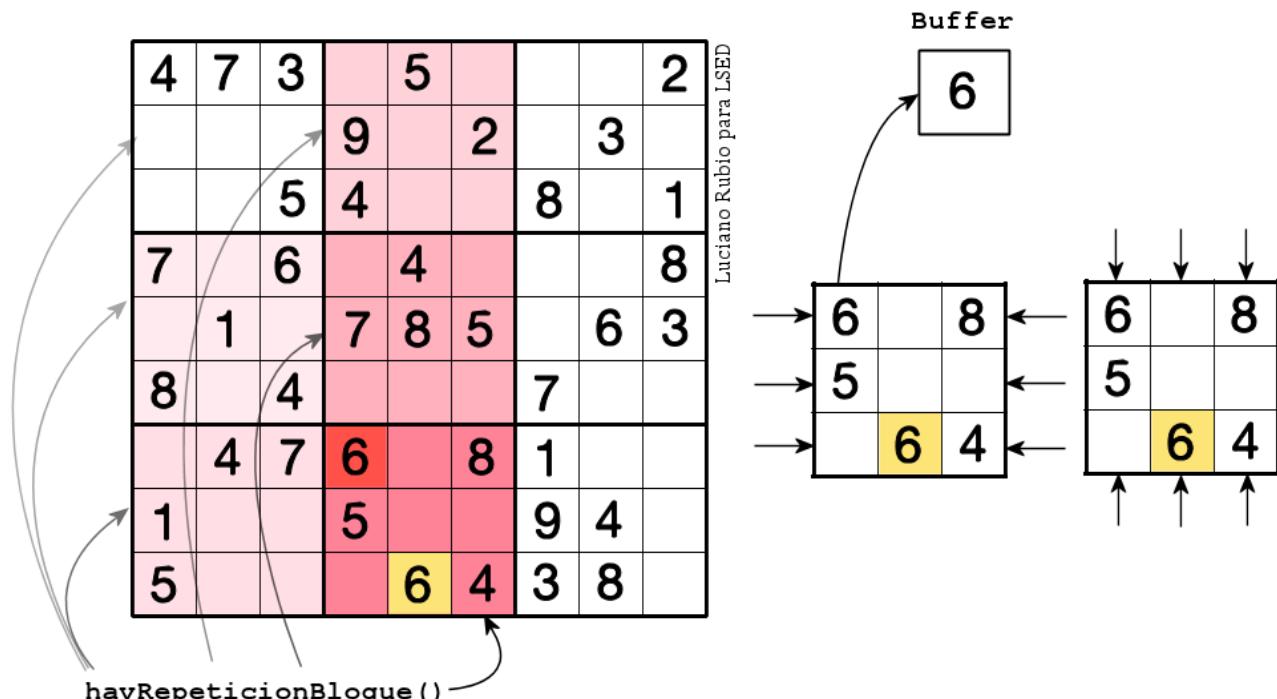


Figura 5.12: Iteración por bloques para detectar una repetición de número (3^a regla del Sudoku)

5.1.8.19 Funcionalidad estanTodosNumeros

```
estanTodosNumeros() {
    Bucle de recorrido del tablero {
        Comprobación de casilla sin número
    }
}

* Variables de entrada: char *pTabSudoku
* Variables de salida: BOOL (TRUE si están todos los números en el tablero)
```

Esta función recorre el tablero Sudoku comprobando que no haya ninguna casilla vacía, sino que todas contengan cualquier número del 1 al 9.

5.1.8.20 Funcionalidad turnoGanador

```
turnoGanador() {
    Comprobación de tipo de puntuación {
        Bucle de recorrido del tablero {
            Recuento del número de aciertos de cada jugador
        }
    determinaMaximo();
    }
}
```

```
* Variables de entrada: int *pTabTurnos, int *pAciertos, int *pNumJugadores, int *pPuntos,
int *pGanarPorTiempos
* Variables de salida: BYTE (número del jugador ganador - 0 si hay empate)
```

Esta función determina el turno del jugador con más puntos según el tipo de puntuación utilizada. Tanto si es por aciertos como por rapidez, se recorren los marcadores de todos los jugadores y se localiza el número mayor, actualizando la variable que proclama al ganador. En caso de que dos o más jugadores tengan una puntuación máxima idéntica, se considerará empate.

5.1.8.21 Funcionalidad determinaMaximo

```
determinaMaximo() {
    Bucle de recorrido de puntuaciones {
        Obtención del valor máximo
    }
}
```

```
* Variables de entrada: int *pLista, int *pNumJugadores
* Variables de salida: BYTE (número del jugador ganador - 0 si hay empate)
```

Esta función recorre una lista de marcadores (que habrá que indicarle si por aciertos o por tiempos) y obtiene el turno con mayor número de puntos.

5.1.8.22 Funcionalidad inicializaPuntuacion

```
inicializaPuntuacion() {
    Resetea las puntuaciones de los jugadores
}

* Variables de entrada: int *pAciertos, int *pPuntos, int *pNumJugadores
* Variables de salida: void
```

Esta función resetea todos los marcadores de puntos de los jugadores para estar en igualdad de condiciones una vez se acabe la partida y se empiece el nivel siguiente.

5.1.8.23 Funcionalidad inicializaSudoku

```
inicializaSudoku() {
    Bucle de recorrido del tablero {
        Rellena los números según el nivel de dificultad
    }
}

* Variables de entrada: char *pTabSudoku, Nivel sudoku)
* Variables de salida: void
```

Esta función inicializa el tablero de Sudoku llenando con números o espacios según el esquema de dificultad indicado. Para ello itera el tablero del nivel adecuado y hace una copia.

5.1.8.24 Funcionalidad haySudoku

```
haySudoku() {
    Comprobación estanTodosNumeros();
    turnoGanador();
}

* Variables de entrada: char *pTabSudoku, int *pTabTurnos, int *pAciertos, int
*pNumJugadores, int *pGanador, int *pPuntos, int *pGanarPorTiempos
* Variables de salida: BOOL (TRUE si están todos los números)
```

Esta función determina si ya se ha finalizado la partida debido a que se hayan situado todos los números sobre el tablero de Sudoku. En caso afirmativo, se determina el turno ganador en función del sistema de puntuación elegido y se sale de la partida (para iniciar el próximo nivel).

5.1.8.25 Funcionalidad muestraTableroSudoku

```
muestraTableroSudoku() {  
    Bucle de recorrido del tablero {  
        Imprime por pantalla las líneas de separación de bloques  
        Imprime por pantalla los números del tablero  
    }  
}  
  
* Variables de entrada: char *pTabSudoku  
* Variables de salida: void
```

Esta función imprime por pantalla el tablero de Sudoku intentando representarlo de la manera habitual: un bloque de 9x9 casillas subdividido en bloques de 3x3, con algunas casillas vacías y otras con números. Por facilidad de juego se han introducido puntos donde hay espacios (aunque es modificable) para localizar mejor la coordenada donde el jugador quiera escribir.

```
>     1 2 3   4 5 6   7 8 9  
>  -----  
>1 | . . 4 | . . . | . . . |  
>2 | . 5 3 | . . 4 | . . . |  
>3 | . 7 . | . 9 . | . . . |  
>  -----  
>4 | . . . | 8 . . | 1 . . |  
>5 | 4 . . | 3 5 . | . . 6 |  
>6 | . . 6 | . 2 . | 8 . 7 |  
>  -----  
>7 | . 2 . | . . . | 6 . . |  
>8 | . . 4 | . 8 . | 7 . . |  
>9 | 3 . 6 | . . 9 | . . 1 |  
>  -----  
> ...
```

5.1.8.26 Funcionalidad movimientoSudoku

```
movimientoSudoku() {  
    Solicitud al usuario de la fila donde quiere escribir  
    Comprobación de respuesta válida  
    Solicitud al usuario de la columna donde quiere escribir  
    Comprobación de respuesta válida  
    Solicitud al usuario del número que quiere escribir  
    Comprobación de respuesta válida  
    compruebaMovimiento();  
}  
  
* Variables de entrada: int *pJugador, char *pTabSudoku, int *pTabTurnos, char *pSolucion,  
int *pAyuda, int *pGanarPorTiempos, int *pPuntos  
* Variables de salida: BOOL (TRUE si ha escrito un número o se le ha pasado el tiempo -  
FALSE si ha pulsado "salir")  
* Variables globales consultadas: turnoPerdido en cada solicitud de dato al usuario (para
```

(saber si se le ha acabado el tiempo de turno)

Esta función solicita al usuario la introducción de la fila y columna en donde quiere escribir el número que a continuación se le pregunta. En cada pregunta se comprueba que la respuesta es válida (siempre será una cifra del 1 al 9), y en caso afirmativo se procede a la escritura del mismo, siempre que siga las reglas del juego.

```
> Indica la fila donde quieras escribir [1-9]: 1
> Indica la columna donde quieras escribir [1-9]: 4
> Indica el número que quieras escribir: 8
```

5.1.8.27 Funcionalidad compruebaMovimiento

```
compruebaMovimiento() {
```

Obtención de la posición de escritura del número

Comprobación de casilla vacía

hayRepetición(); (Comprobación de número repetido)

escribeSudoku();

```
}
```

* **Variables de entrada:** char fila, char columna, char numero, int *pJugador, char *pTabSudoku, int *pTabTurnos, char *pSolucion, int *pAyuda, int *pGanarPorTiempos, int *pPuntos

* **Variables de salida:** void

* **Variables globales consultadas:** turnoPerdido si se le pide otra vez al usuario posición y número, debido a que la casilla no se encontraba vacía o ha detectado repetición.

Esta función analiza la escritura de número que quiere llevar a cabo el jugador. Para ello primero se comprueba que la casilla indicada está vacía y después se analizan las 3 reglas básicas del Sudoku (no haya repetición en fila, columna o bloque). Si todo es correcto se procede a la escritura del número.

5.1.8.28 Funcionalidad escribeSudoku

```
escribeSudoku() {
```

Escritura del número en la casilla

Comprobación de acierto {

Imprime por pantalla un mensaje si modo Ayuda activado

actualizaPuntos();

```
}
```

Comprobación de fallo {

Si modo Ayuda activado, se pierde el turno y se imprime mensaje

```
}
```

```
}
```

* **Variables de entrada:** int posición, char numero, int *pJugador, char *pTabSudoku, int *pTabTurnos, char *pSolucion, int *pAyuda, int *pGanarPorTiempos, int *pPuntos)

* **Variables de salida:** void

* **Variables globales modificadas:** turnoPerdido = TRUE en caso de fallar número con el modo Ayuda activado

Función escribeSudoku(): escribe el número indicado en una casilla del tablero Sudoku. En caso de tener el modo Ayuda activado, imprime un mensaje por pantalla de acierto o fallo (y se actualiza el marcador de puntos correspondiente). Esta función es de vital importancia para ir actualizando el tablero de juego y la cuenta de puntos de cada jugador.

5.1.9 Niveles de dificultad

Nivel Trivial

1	7	4	6	3	5	2	9	8
6		9	8	2	4	7	1	5
8	2	5	9	7	1		6	4
2	4	1	5	8	7	9	3	6
7	8	6	2	9	3	5	4	1
9	5	3		4	6	8	2	7
4	9	8	7	1	2	6	5	3
	6	7	4	5	9	1	8	2
5	1	2	3	6	8	4		9

1	7	4	6	3	5	2	9	8
6	3	9	8	2	4	7	1	5
8	2	5	9	7	1	3	6	4
2	4	1	5	8	7	9	3	6
7	8	6	2	9	3	5	4	1
9	5	3	1	4	6	8	2	7
4	9	8	7	1	2	6	5	3
3	6	7	4	5	9	1	8	2
5	1	2	3	6	8	4	7	9

Nivel Novato

1	7			5		9	8	
		9	8		4	7		5
		5	7		3	6		
2	4			8	7			
	8	2			5	4	1	
5	3	1		6		2	7	
4			7	1		6		
6	7	4			1	8	2	
1	2	3	6	8				

1	7	4	6	3	5	2	9	8
6	3	9	8	2	4	7	1	5
8	2	5	9	7	1	3	6	4
2	4	1	5	8	7	9	3	6
7	8	6	2	9	3	5	4	1
9	5	3	1	4	6	8	2	7
4	9	8	7	1	2	6	5	3
3	6	7	4	5	9	1	8	2
5	1	2	3	6	8	4	7	9

Nivel Sencillo

4	7	3		5			2	
			9	2		3		
		5	4		8		1	
7	6		4			8		
1			7	8	5		6	3
8	4			7				
	4	7	6	8	1			
1			5		9	4		
5				4	3	8		

4	7	3	8	5	1	6	9	2
6	8	1	9	7	2	5	3	4
2	9	5	4	6	3	8	7	1
7	3	6	1	4	9	2	5	8
9	1	2	7	8	5	4	6	3
8	5	4	3	2	6	7	1	9
3	4	7	6	9	8	1	2	5
1	2	8	5	3	7	9	4	6
5	6	9	2	1	4	3	8	7

Nivel Iniciados

3	2		8		1	7	6
6	9		5	3	8		
			6				
5			9		4		
	2		7	3			
3			6			5	
		6					
	9	5	3		1	2	
4	1	6		2	5	3	

Luciano Rubio para LSED

3	2	5	8	4	9	1	7	6
6	9	7	1	5	3	8	2	4
1	4	8	7	2	6	5	3	9
5	6	1	3	9	8	2	4	7
9	8	2	4	7	5	3	6	1
7	3	4	2	6	1	9	8	5
2	5	3	6	1	7	4	9	8
8	7	9	5	3	4	6	1	2
4	1	6	9	8	2	7	5	3

Luciano Rubio para LSED

Nivel Medio

2	8	9	7		3	5		
				2	9		4	
		9	8	1				
7		2			1	4		
		5	1	2				9
	6			7	2			
8			2	4			3	
4				3	9			
9				4		2		

Luciano Rubio para LSED

2	8	9	7	6	4	3	5	1
6	7	1	3	5	2	9	8	4
5	3	4	9	8	1	6	2	7
7	9	2	8	3	5	1	4	6
3	5	5	1	2	6	8	7	9
1	6	8	4	9	7	2	3	5
8	1	7	2	4	9	5	6	3
4	2	6	5	1	3	7	9	8
9	5	3	6	7	8	4	1	2

Luciano Rubio para LSED

Nivel Complejo

	5	8		2	7	1		
2				4		9		
				8				
					9		3	
	3	2		5	1	4		
1		5						
			1					
3		7				6		
5	9	4		2	3			

Luciano Rubio para LSED

4	5	8	3	9	2	7	6	1
2	6	1	5	7	4	3	8	9
9	7	3	6	1	8	4	2	5
8	4	7	2	6	1	9	5	3
6	3	2	9	5	7	1	4	8
1	9	5	8	4	3	6	7	2
7	2	6	1	3	5	8	9	4
3	8	4	7	2	9	5	1	6
5	1	9	4	8	6	2	3	7

Luciano Rubio para LSED

Nivel Difícil

		4					
	5	3		4			
7			9				
			8		1		
4			3	5			6
		6		2	8		7
2					6		
		9		8	7		
3	8			9			1

Luciano Rubio para LSED

9	1	4	7	3	2	5	6	8
8	5	3	1	6	4	9	7	2
6	7	2	5	9	8	3	1	4
2	9	7	8	4	6	1	3	5
4	8	1	3	5	7	2	9	6
5	3	6	9	2	1	8	4	7
7	2	5	4	1	3	6	8	9
1	4	9	6	8	5	7	2	3
3	6	8	2	7	9	4	5	1

Luciano Rubio para LSED

Nivel Chungo

6			5	2			
	7			8			
	9				1	5	2
				6			
7			3		6	1	
						4	9
	1			4			
		8			4		
9	2			5			

Luciano Rubio para LSED

6	1	3	5	2	9	7	8	4
2	7	5	4	8	1	9	3	6
4	8	9	6	7	3	1	5	2
5	9	8	1	4	6	2	7	3
7	4	2	9	3	8	6	1	5
1	3	6	2	5	7	8	4	9
8	6	1	3	9	4	5	2	7
3	5	7	8	6	2	4	9	1
9	2	4	7	1	5	3	6	8

Luciano Rubio para LSED

Nivel Imposible

			4	5			
2		3					9
	3			1			4
1	9						8
			3				9
	7		1				4
8				4			
			4	9			
9				8			

Luciano Rubio para LSED

7	9	1	4	6	5	3	2	8
2	4	8	3	7	1	5	6	9
5	6	3	2	8	9	1	7	4
1	5	9	6	4	2	7	8	3
4	8	2	5	3	7	6	9	1
3	7	6	9	1	8	2	4	5
8	2	7	1	9	3	4	5	6
6	3	5	8	2	4	9	1	7
9	1	4	7	5	6	8	3	2

Luciano Rubio para LSED

5.2 Descarga de tableros de Sudoku por servidor TFTP

Tras realizar la mejora del juego del Sudoku para la plataforma ENT2004CF, quisimos ampliar su funcionalidad mediante el aprendizaje y uso de un servidor TFTP.

5.2.1 Cómo funciona TFTP y qué uso podemos darle

TFTP son las siglas de *Trivial file transfer Protocol*, se trata de un protocolo muy sencillo de **transferencia de archivos**, con una funcionalidad muy básica, de tipo FTP. Fue definido en 1980.

Como es muy simple, es muy fácil de implementar en un espacio muy pequeño de memoria. TFTP es especialmente útil para arrancar en ordenadores o routers que no disponen de dispositivos de almacenamiento de archivos. Se usa principalmente para transferir pequeños ficheros entre terminales de una misma red, como por ejemplo cuando un terminal X-Window o cualquier otro cliente ligero arrancan desde un servidor de red. Algunos detalles técnicos del servidor TFTP son:

- Utiliza **UDP** (puerto 69) como protocolo de **transporte** (a diferencia de FTP que utiliza el puerto 21 TCP).
- No puede **listar** el contenido de los directorios
- No existen mecanismos de **autenticación** o cifrado
- Se utiliza para **leer o escribir** ficheros de un servidor remoto
- Soporta tres modos diferentes de transferencia, “netascii”, “octet” y “mail”, de los que los dos primeros corresponden a los modos “ascii” e “imagen” (binario) del protocolo FTP

Fuente: traducción libre de la página de TFTP de Wikipedia, the Free Encyclopedia

El uso que hemos querido darle para nuestra mejora del juego de Sudoku es la **descarga de más niveles** por medio del servidor TFTP, para poder jugar a más de los 9 tableros que dispone el programa básico.

Un buen motivo para querer descargar tableros del servidor TFTP puede ser el querer crear una partida multijugador con los Sudokus **impresos en los periódicos**. En caso de que nos proporcionasen la solución, podríamos incluirla en el fichero de descarga y jugar con el **modo ayuda activado**. Si no nos la proporcionan (como suele ser el caso habitual), deberemos desactivar el modo ayuda.



Figura 5.13: Ejemplo de inclusión de nuevos tableros de Sudoku en los periódicos

5.2.2 Modo de funcionamiento en el Sudoku

Para la descarga de nuevos tableros necesitaremos definir un fichero `sudoku.txt` en el ordenador que hará de servidor TFTP. Este fichero contendrá una lista de números y espacios (que representaremos por puntos, para mayor facilidad) seguidos unos a continuación de los otros (vector horizontal), formando un total de 81 caracteres ASCII.

Estos caracteres serán leídos al principio del juego de Sudoku, en caso de que el usuario haya seleccionado el modo red, de manera que al empezar la partida, no se preguntará por el nivel de dificultad, sino que se jugará directamente el tablero descargado.

Una vez finalice la partida se saldrá del juego de Sudoku puesto que no hay más niveles a continuación. De aquí en adelante sería responsabilidad del “Servidor TFTP” especificar qué tipo de servicio ofrece, es decir, si la renovación del tablero que almacena será diaria o semanal.

Nuestro plan de pruebas ha sido modificar el fichero `sudoku.txt` con tableros que obteníamos de diarios gratuitos o descargados de Internet.

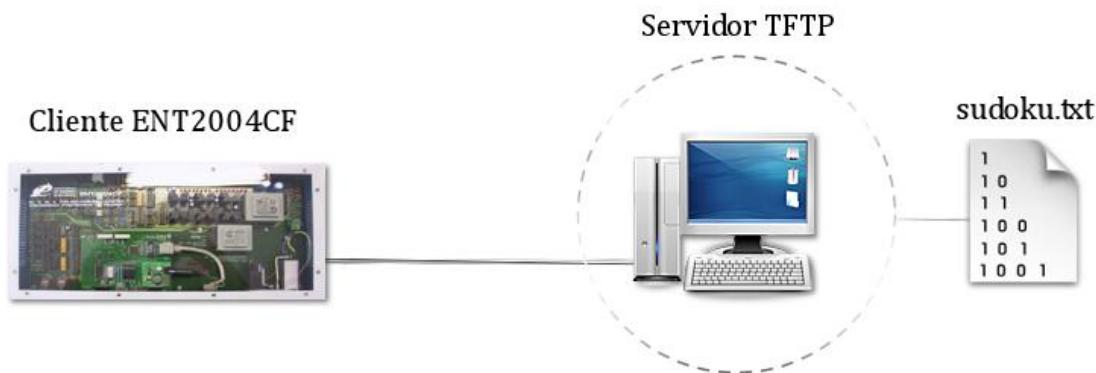


Figura 5.14: Conexión básica entre la plataforma ENT2004CF y el servidor TFTP con el tablero Sudoku

5.2.3 Funcionalidades añadidas

5.2.3.1 Funcionalidad leeEthernetInicial

```
leeEthernetInicial() {  
    Asignación del comienzo de la memoria asignada a Ethernet  
    leeEthernet(); (guarda en memoria la info descargada)  
    Bucle de recorrido del tablero {  
        Copia de la memoria al tablero  
    }  
    Bucle de recorrido de la solución {  
        Copia de la memoria a la solución  
    }  
}
```

```
* Variables de entrada: void
* Variables de salida: void
* Variables globales modificadas: seteo de md_last_address (dirección de comienzo de la memoria SDRAM asignada a Ethernet)
```

Esta función es la función principal para la descarga del fichero de Sudoku a través del servidor TFTP. Para realizar la descarga por el puerto Ethernet, hay que seguir los siguientes pasos:

- Se asigna la dirección de comienzo de memoria asignada a esta tarea. Hemos impuesto el valor de **0x60000 en SDRAM**
- Se procede a la lectura del fichero alojado en el servidor TFTP, y se escribe carácter a carácter en la zona de memoria.
- Leemos los Bytes de esa zona de memoria y con ellos vamos inicializando el tablero de Sudoku. Los primeros bytes corresponderán a la situación inicial del juego, y los siguientes, la solución (necesaria si deseamos activar el modo ayuda).

5.2.3.2 Funcionalidad leeEthernet

```
leeEthernet() {
    Seteo del fichero de lectura
    Inicialización del temporizador de Ethernet
    Habilitación de interrupciones FEC
    Inicialización de FEC
    Escritura de la dirección Ethernet en la estructura NIF
    Inicialización de los búferes de red
    Inicialización de ARP
    Inicialización e IP
    Inicialización de UDP
    Apertura de la conexión TFTP
    Bucle de recorrido del fichero {
        Escritura en memoria carácter por carácter
    }
}
```

```
* Variables de entrada: void
* Variables de salida: void
```

Esta función inicializa todos los protocolos y temporizadores necesarios para la descarga del fichero alojado en el servidor TFTP, y a continuación transfiere todos los bytes a la zona de memoria dedicada a esta tarea. A continuación se muestra un esquema de la pila de protocolos que se han utilizado para la interconexión por cable Ethernet:

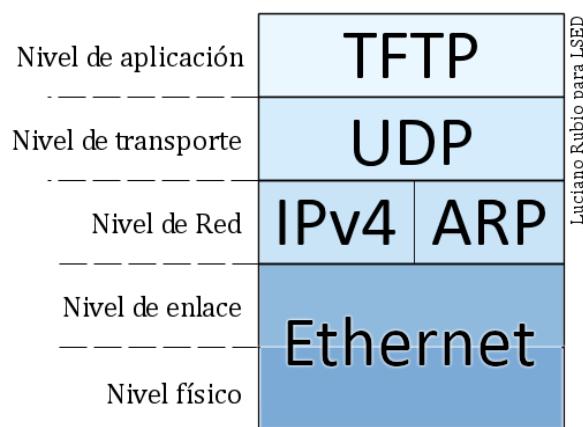


Figura 5.15: Pila de protocolos aplicada en esta mejora

Las direcciones IP se escriben en nomenclatura MAC, a continuación un detalle de la trama Ethernet que se utiliza en esta interconexión:

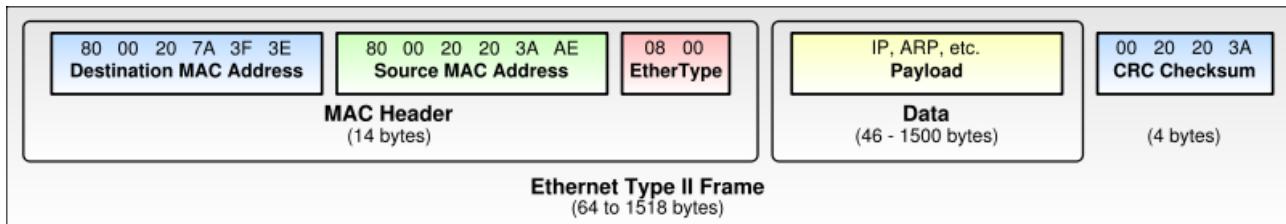


Figura 5.16: Detalle de la trama Ethernet utilizada. Fuente: Wikipedia, the Free encyclopedia

5.2.3.3 Funcionalidad escribeEthernet

```

escribeEthernet() {
    Seteo del fichero de escritura
    Seteo de los bytes a enviar al servidor
    Inicialización del temporizador de Ethernet
    Habilitación de interrupciones FEC
    Inicialización de FEC
    Escritura de la dirección Ethernet en la estructura NIF
    Inicialización de los búferes de red
    Inicialización de ARP
    Inicialización e IP
    Inicialización de UDP
    Apertura de la conexión TFTP para escritura
}

* Variables de entrada: void
* Variables de salida: void
* Variables globales consultadas: lectura desde md_last_address en adelante tantos Bytes
como se vayan a enviar

```

Esta función actúa al igual que **leeEthernet** pero a la inversa: lee byte a byte de la zona de memoria designada para Ethernet y transfiere estos datos al fichero del servidor TFTP. De esta forma estamos actualizando los caracteres de **sudoku.txt**, lo que nos permite ir sincronizando el estado del tablero. Esta rutina será de especial interés para la siguiente mejora desarrollada, que permite un juego en red gracias a que el fichero del servidor se renueva periódicamente.

5.2.4 Funcionalidades modificadas

5.2.4.1 Funcionalidad devuelveSolucion

```
devuelveSolucion() {
    Comprobación de modo online activado {
        Muestra mensaje de conexión al servidor TFTP
        leeEthernetInicial();
        Inicializa el tablero de sudoku con los datos descargados
        Obtención del sudoku solución
    }
    Inicializa el tablero de sudoku según dificultad
    Obtención del sudoku solución
}

* Variables de entrada: char *pTabSudoku, int *pRed, int *pDificultad
* Variables de salida: char* (puntero a la solución del sudoku)
```

En el caso de que escojamos en el menú principal esta opción de descarga de un tablero TFTP, hemos de reconfigurar las variables desde las que leemos el Sudoku, tal y como lo hacíamos en modo local. Esta funcionalidad, encargada de transmitir el puntero que apunta a la solución del juego, tiene que contemplar el caso de que nos hayamos descargado un tablero distinto a los que vienen de serie con el programa.

5.2.4.2 Funcionalidad setDificultad

```
setDificultad () {
    Comprobación de modo online desactivado {
        Solicitud al usuario del nivel de dificultad inicial
        Se imprime por pantalla su respuesta
        Comprobación de respuesta válida
    }
    Se ignora el nivel de dificultad en modo online
}

* Variables de entrada: char *pTabSudoku, int *pRed, int *pDificultad
* Variables de salida: char* (puntero a la solución del sudoku)
```

Esta función se encargaba de configurar el nivel de dificultad, asignando el número de nivel al Sudoku adecuado. Si se juega en modo local, conforme completamos tableros se irá complicando el juego, y se pasa al siguiente Sudoku. Hemos de contemplar el caso de que la descarga desde TFTP no incluye una gama de distintos niveles, luego al acabar la partida no se genera un ciclo de niveles ascendentes.

5.3 Juego de Sudoku por LAN o por INTERNET

5.3.1 Modo de funcionamiento del modo LAN

Tras hacer posible la descarga de un tablero de Sudoku a partir de un servidor TFTP, decidimos darle más uso y desarrollar un modo de juego en red en el que varios jugadores, usando cada uno una plataforma ENT2004CF, mantuvieran el mismo sistema de turnos que hemos descrito en la funcionalidad básica de nuestra mejora de Sudoku.

La idea es **interconectar todas las plataformas** que se deseé utilizando la potencia de la **red LAN** Ethernet que dispone el laboratorio.

Para ello es necesario almacenar en el fichero principal del servidor TFTP un **flag de control** que marcará el ritmo al que los jugadores deberán ir entrando en la partida. Cada plataforma ENT2004CF recibirá un nombre de **HOST** que configuraremos antes de ejecutar el programa, y **HOST_SIGUIENTE**, el nombre del próximo puesto que deberá conectarse al juego. Si se configuran todas las plataformas ENT2004CF correctamente (adjuntamos un ejemplo para 5 jugadores para mayor claridad), se cierra el ciclo de turnos y es posible una partida.

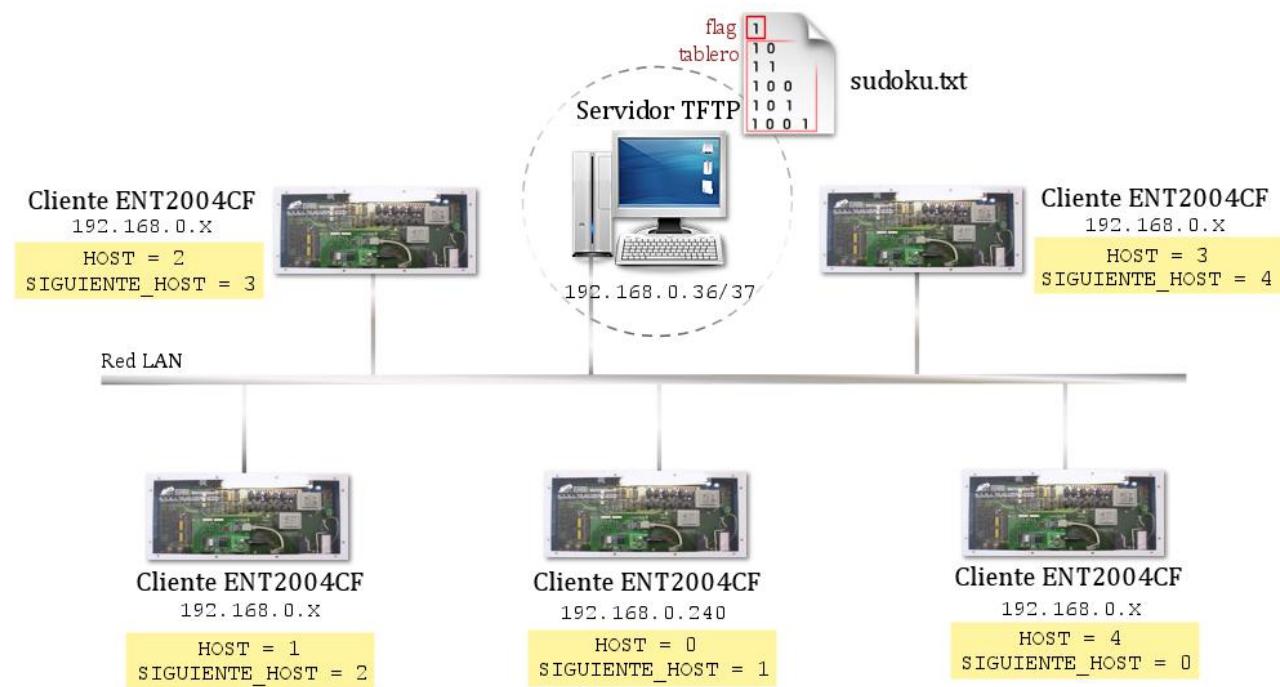


Figura 5.17: Esquema de interconexión y configuración para el modo de juego en red entre varias plataformas del laboratorio

También hay que añadir al sistema básico la capacidad de que cada plataforma **envíe la nueva situación del tablero** al servidor TFTP, de forma que lo primero que hará cada puesto al recibir la oportunidad de turno es descargarlo para sincronizarse con la partida.

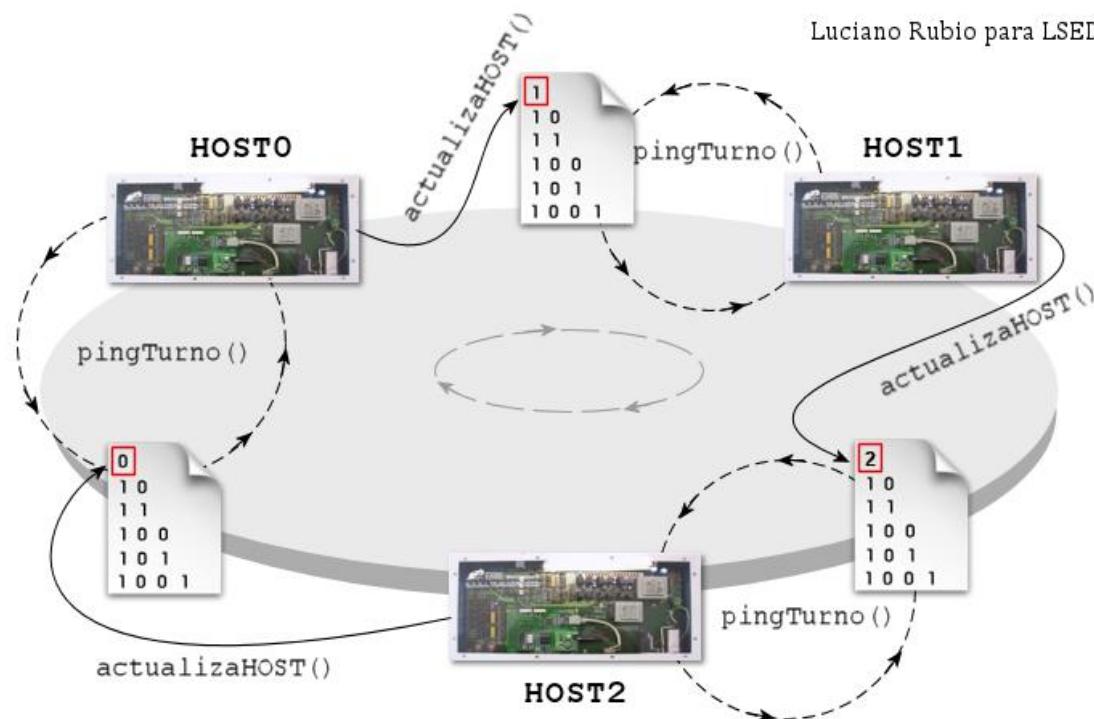


Figura 5.18: Esquema de funcionamiento de la obtención y ciclo de turnos en el modo en Red. Todos los terminales realizan consultas periódicas al servidor TFTP en caso de que no estén jugando. Cuando un HOST realiza su movimiento, actualiza el flag de control y se pasa al siguiente HOST del ciclo.

Además, aparte de configurar el nombre de HOST y SIGUIENTE_HOST, habrá que escribir adecuadamente los parámetros Ethernet, para que la comunicación entre el servidor y los demás puestos del laboratorio se pueda satisfacer. Adjuntamos una tabla con una configuración válida:

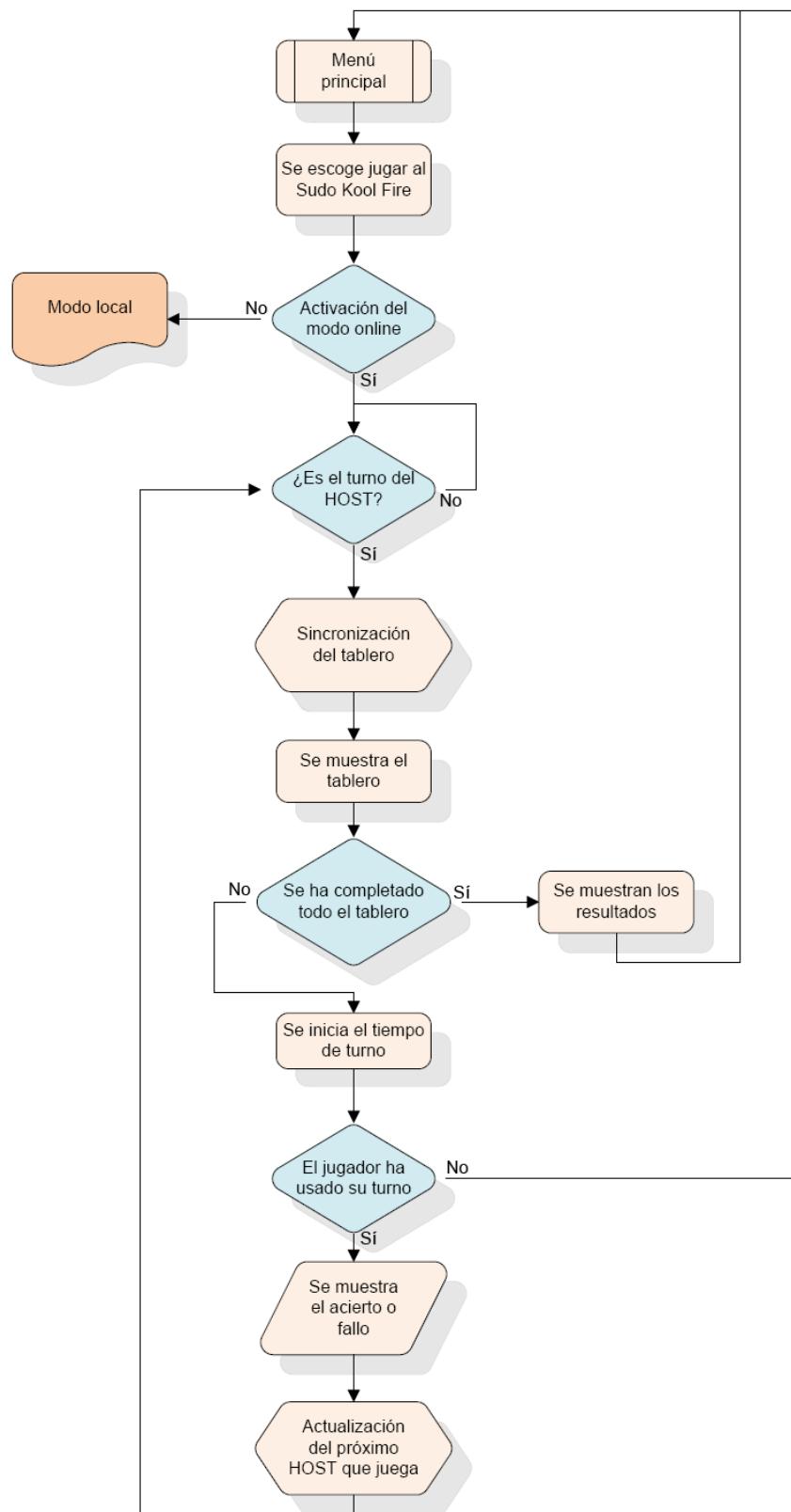
	HOST0	HOST1	...	HOSTn
IP Cliente	192.168.0.X	192.168.0.Y		192.168.0.N
HOST	0	1		n
SIGUIENTE_HOST	1	2		0
IP Servidor		192.168.0.37		
Máscara Subred		255.255.255.0		
Pasarela enlace		192.168.0.1		

Figura 5.19: Tabla de configuración del modo en red.

Vemos que la **IP del servidor**, la **máscara de subred** y la **pasarela de enlace** son parámetros que obligatoriamente han de estar **fijos** para todas las plataformas ENT2004CF que participen. Para el plan de pruebas de esta mejora, tenemos que elegir entre el puesto de laboratorio 36 o 37 (cuyas direcciones son 192.168.0.36 y 192.168.0.37 respectivamente), ya que son los únicos que tienen la aplicación de servidor TFTP instalada.

Los demás puestos clientes pueden ser cualquiera del laboratorio, siempre que se les asigne a cada uno una **IP de cliente distinta** y que los nombres de HOST y SIGUIENTE_HOST sean como la tabla superior indica.

5.3.2 Diagrama del programa principal



5.3.3 Modo de funcionamiento del modo INTERNET

Otra posible extensión de la mejora es el modo de juego a través de internet. El sistema está capacitado para la conexión entre dos plataformas ENT2004CF a larga distancia, pero la **indisponibilidad de clientes** nos impide incluir en el plan de pruebas esta mejora al cien por cien.

Por el contrario, sí podemos interconectar dos puestos de laboratorio a través de la red LAN y obtener los datos de un **servidor TFTP ajeno**, ubicado en cualquier otro lugar. Para ello sólo ha de disponerse de la **IP pública** a través de la cual se va a acceder, y reconfigurar los parámetros Ethernet adecuadamente.

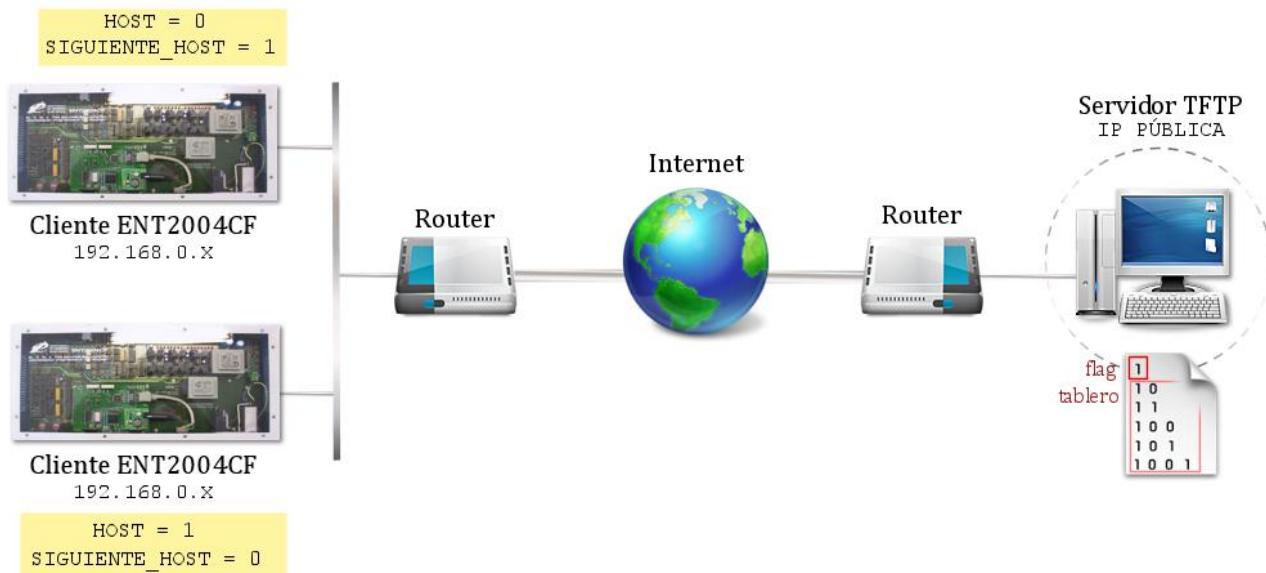


Figura 5.20: Esquema de interconexión de dos plataformas ENT2004CF (en red local) conectándose a un servidor TFTP ajeno.

A continuación mostramos un esquema de la pila de protocolos utilizados en esta interconexión clientes LAN – servidor TFTP ajeno, teniendo en cuenta los routers.

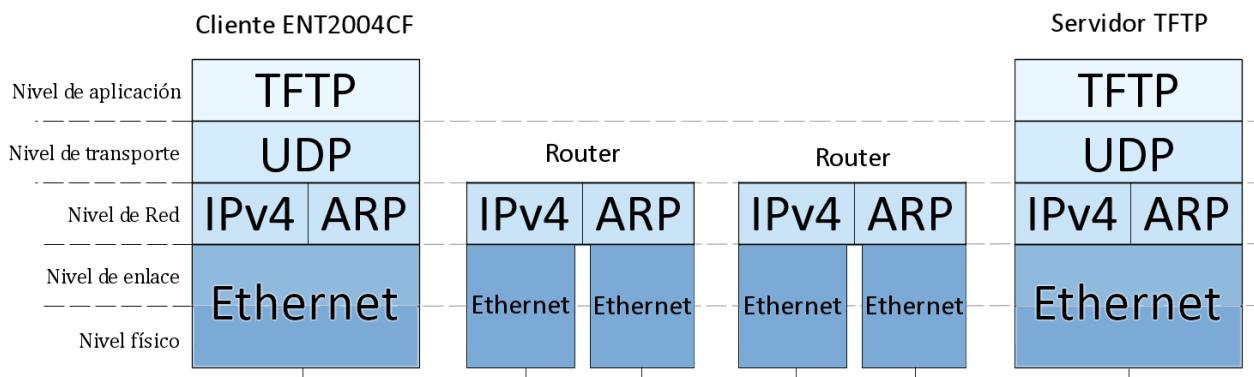


Figura 5.21: Pilas de protocolos utilizadas en esta mejora.

Además, es necesario para que el sistema funcione, una correcta configuración de los parámetros Ethernet tal y como describíamos en la anterior mejora, juego en red de área local.

Las características de la tabla de enrutamiento son semejantes, a diferencia de la IP del servidor, que ahora es de carácter externo y además ha de ser accesible desde el terminal del laboratorio.

	HOST0	HOST1	...	HOSTn
IP Cliente	192.168.0.X	192.168.0.Y		192.168.0.N
HOST	0	1		n
SIGUIENTE_HOST	1	2		0
IP Servidor	IP PÚBLICA			
Máscara Subred	255.255.255.0			
Pasarela enlace	192.168.0.1			

Figura 5.22: Tabla de configuración de todos los HOSTS para el modo Internet

5.3.4 Funcionalidades añadidas

5.3.4.1 Funcionalidad actualizaSDRAM

```
actualizaSDRAM() {
    Obtención del comienzo de la memoria asignada a Ethernet
    Bucle de recorrido del tablero {
        Copia de los valores del tablero a SDRAM
    }
}

* Variables de entrada: char *pTabSudoku
* Variables de salida: void
* Variables globales consultadas: lectura de md_last_address para saber el comienzo de la
zona asignada a Ethernet en SDRAM
```

Esta función actualiza el espacio de memoria de SDRAM con los valores del tablero de Sudoku actuales. La utilidad de estas instrucciones reside en la sincronización de partida con memoria, pues de ahí se transferirán al servidor TFTP para el uso de los siguientes jugadores.

5.3.4.2 Funcionalidad actualizaHOST

```
actualizaHOST() {
    Escribe en memoria el flag SIGUIENTE_HOST
}

* Variables de entrada: void
* Variables de salida: void
* Variables globales consultadas: lectura de md_last_address para saber el comienzo de la
zona asignada a Ethernet en SDRAM
```

Esta función es la encargada de escribir en el espacio de memoria SDRAM el nuevo valor del flag del siguiente host que entrará a la partida. Es de especial importancia ya que permite cerrar el ciclo de

turnos siempre y cuando las variables HOST y SIGUIENTE_HOST de la plataforma ENT2004CF donde se ejecute el programa estén correctamente configuradas.

5.3.4.3 Funcionalidad pingTurno

```
pingTurno() {
    leeEthernet(); (descarga de datos a memoria)
    Lectura del flag de HOST
}

* Variables de entrada: void
* Variables de salida: char (HOST siguiente)
* Variables globales consultadas: lectura de md_last_address para saber el comienzo de la
zona asignada a Ethernet en SDRAM
```

Esta función realiza comprobaciones periódicas del flag de control de host leyendo el primer byte del fichero **sudoku.txt** alojado en el servidor TFTP. Mediante esta función tenemos al host al que no le toca jugar en un *estado de espera*, precisamente atento al movimiento del host que acaba de entrar en juego.

5.3.4.4 Funcionalidad pingTablero

```
pingTablero() {
    leeEthernet(); (descarga de datos a memoria)
    Bucle de recorrido del tablero {
        Copia de los valores de SDRAM al tablero
    }
}

* Variables de entrada: char *pTabSudoku
* Variables de salida: void
* Variables globales consultadas: lectura de md_last_address para saber el comienzo de la
zona asignada a Ethernet en SDRAM
```

Esta función se encarga de descargarse la situación actual del Sudoku, y sólo es llamada una vez ha recibido turno el host. Esto es debido a que recibe del puerto Ethernet los 81 bytes correspondientes a las 81 casillas del tablero, una información mucho más pesada comparada con la lectura del flag de control (1 byte).

5.3.5 Funcionalidades modificadas

5.3.5.1 Funcionalidad sudokuJuego

```
sudokuJuego() {
    Bucle de nivel de dificultad {
        inicializaPuntuacion();
        Bucle de Turno {
            refrescoTablero();
```

```

Comprobación haySudoku() { (Comprobación de tablero completado)
    Si modo online, actualizaHOST();
}

refrescaTurno();

movimientoSudoku(); (Proceso de escritura en tablero)
incrementoTurno();

}

muestraResultados();
Si modo online, se sale del juego
Aumento del nivel de dificultad
setConf(); (Mantiene la configuración inicial)
}

}

* Variables de entrada: char *pTabSudoku, int *pTabTurnos, int *pRed, int *pNumJugadores,
int *pAciertos, int *pGanador, int *pJugador, int *pDificultad, int *pAyuda, int
*pGanarPorTiempos, char *pSolucion
* Variables de salida: void
* Variables globales consultadas: Reseteo de pintoTiempo (pintoTiempo = FALSE) en caso de
que el usuario pulse "salir"

```

Este es el programa principal del juego Sudoku que ya describimos con detalle en el primer apartado de mejoras. En esencia, hemos de tener presente dos cambios que hacen falta para el funcionamiento de una partida en red:

- Despues de obtener la situación actual del tablero, en caso de haberse completado todas las casillas (final de la partida), se debe actualizar el flag de control de host. De esta forma conseguimos que todos los usuarios involucrados se vayan enviando el tablero de Sudoku completado y que se cierre correctamente el ciclo de turnos. Si no se actualizara este flag conseguiríamos que el jugador ganador visualizara correctamente el final de la partida, pero que los demás hosts se quedaran esperando a que este jugador ganador hiciera el siguiente movimiento, cosa que resultaría absurda.
- Por último, ya que se ha planificado que sólo se jugará el tablero Sudoku que se aloja en el servidor TFTP (para utilizar la mejora explicada en el apartado 5.2), en caso de que se termine la partida finalizará el programa, mostrando los resultados.

5.3.5.2 Funcionalidad refrescoTablero

```

refrescoTablero() {
    Comprobación de modo online activado {
        Bucle de espera, turno del otro HOST {
            pingTurno();
            Muestra mensaje de espera
        }
        pingTablero();
    }

    Reseteo de tiempo perdido de jugador
    muestraTableroSudoku();
}

```

```
}
```

```
* Variables de entrada: char *pTabSudoku, int *pRed, int *pNumJugadores, int *pJugador
* Variables de salida: void
* Variables globales modificadas: Reseteo de tiempoPerdido (tiempoPerdido = FALSE)
```

Esta funcionalidad se encargaba de mostrar la situación del tablero por pantalla. En el caso de modo en red es imprescindible la obtención de los últimos números introducidos por otras plataformas conectadas a la LAN, luego se hace la correspondiente descarga de bytes si es que ha llegado el turno del host en el que nos encontramos. El esquema es:

- Bucle de espera, lectura periódica del flag de control de host
- En caso de tener turno, se procede a la descarga del tablero para la sincronización
- Finalmente se imprime por pantalla

5.3.5.3 Funcionalidad incrementoTurno

```
incrementoTurno() {
    Comprobación de modo online activado {
        actualizaHOST();
        actualizaSDRAM();
        escribeEthernet();
    }

    Asigna el turno al jugador siguiente
    Para el tiempo de turno
}

* Variables de entrada: char *pTabSudoku, int *pRed, int *pJugador
* Variables de salida: void
* Variables globales modificadas: Reseteo del flag pintoTiempo (pintoTiempo = FALSE)
```

Esta función en modo local funcionaba de manera simple, incrementando el número del próximo jugador que disfrutaría de su turno. Para el modo en red es necesario la modificación del flag de control, de manera que las tareas son:

- Se escribe en memoria el número del próximo host que participa
- Se actualiza el espacio de memoria con el tablero Sudoku tal y como lo haya dejado el host actual
- Se envían todos los bytes escritos en memoria hacia el servidor TFTP, para renovar flag y tablero

5.3.5.4 Funcionalidad setConfiguracion

```
setConfiguracion() {
    setRed();

    Comprobación de modo online activado {
        Seteo del número de jugadores según HOSTS
    }

    Comprobación de modo online desactivado {
        setJugadores();
    }
}
```

```

    }

    setAyuda();
    setPuntuacion();
    setDificultad();
    devuelveSolucion();
}

/* Variables de entrada: int *pRed, int *pNumJugadores, char *pTabSudoku, int *pDificultad,
int *pAyuda, int *pGanarPorTiempos
* Variables de salida: char* (puntero a la solución del sudoku)

```

Esta función debe tener en cuenta que para la activación del modo red, el ciclo de turnos de los jugadores no se efectúa en un único host, como venía siendo habitual. En este caso sólo se debe mostrar el turno de un jugador en cada plataforma ENT2004CF, y mostrar los mensajes convenientes durante el *estado de espera*. A fin de cuentas, el comportamiento aparente en cada host es la de una partida en solitario, que tras escribir un nuevo número, recibe otros de fuentes exteriores (los otros hosts que participan).

5.3.5.5 Funcionalidad muestraResultados

```

muestraResultados() {
    Bucle de jugadores {
        Se imprime por pantalla el número del jugador
        Comprobación del tipo de puntuación
        Se imprime por pantalla su puntuación
        Comprobación de modo online activa {
            Mostrar puntuaciones del HOST
        }
    }
    Comprobación de empate
    Se imprime por pantalla el jugador ganador
}

/* Variables de entrada: int *pNumJugadores, int *pAciertos, int *pPuntos, int *pGanador, int
*pGanarPorTiempos, int *pRed
* Variables de salida: void

```

Esta función mostraba la tabla de resultados finales una vez se había completado una partida de Sudoku (ninguna casilla vacía). En el caso de modo en red, cada host muestra al final del juego los puntos que ha obtenido.

5.4 LSED Rally

5.4.1 Objetivos y descripción del juego

Siguiendo el esquema de creación de juegos, decidimos añadir al “3 en raya” y al “Sudo Kool Fire” la **mejora del Rally** (representado con caracteres ASCII) que proponía el enunciado de la práctica.

Este juego tiene una dinámica sencilla pero nos pareció muy interesante porque utiliza (de manera completamente distinta a los juegos de tablero) las interrupciones del temporizador 1, en este caso para **refrescar la pantalla** añadiendo los tramos siguientes del circuito.

El objetivo del Rally es **llegar al final** del circuito **sin chocarse** con los límites de la carretera: en este caso lo hemos representado por dos líneas (que representan el arcén) y una fila de neumáticos. El jugador puede escoger la **velocidad** a la que irá el vehículo (cuanto más lento más fácil) y el **circuito** (entre varios disponibles ordenados por dificultad).

5.4.2 Variables del sistema

Incluimos un breve esquema de las variables y punteros que se utilizaron en el desarrollo de este juego. Las más importantes son:

- *pSuelo es un puntero que apunta al comienzo del **suelo**, que puede incluir **terreno** o **carretera**.
- *pOrigen indica el primer **punto de separación** (empezando por la izquierda) entre el terreno y la carretera circulable. Habitualmente tendrá se representará por el carácter de “barra” (representa el límite de calzada o valla), y servirá para detectar las colisiones.
- *pPosCoché apunta a la **localización del coche**, que deberá estar siempre entre los límites circulables de la calzada.
- *pTramo apunta a una variable que indica el tipo de tramo sobre el que estamos (que básicamente puede ser recto, a la izquierda o a la derecha).
- *pNumRep apunta a una variable que indica el **número de repeticiones** de tramo que se producen, y se va decrementando conforme se representa el circuito. Toma su valor inicial del struct Circuito, donde se definen estáticamente las repeticiones de los tramos.
- *pFinRepeticion es un flag que nos indica que se han **acabado las repeticiones** de tramo y que a continuación se produce un **cambio de dirección** en la carretera. Será de especial utilidad al representar por pantalla el tramo futuro.
- El struct Circuito contiene la definición estática del circuito. En el primer array se muestran los **tipos de tramo** (izquierda representado por -1, centro por 0 y derecha por 1), y en el segundo el número de veces que se repetirán.

5.4.3 Extensiones que hemos introducido al Rally

- Hemos querido no sólo mostrar el tramo de carretera actual en el que se encuentra el vehículo, sino también el siguiente (tramo futuro) para que el jugador pueda predecir los giros que se producen en el circuito. Para realizar esta extensión hay que tener en cuenta el gasto de tiempo que se producirá en la escritura de caracteres, más del doble que en el modelo básico.
- Aprovechando la re-escritura de tramos, hemos añadido el “rastro” o derrapes que produce el coche a su paso por la carretera. Para ello borramos los últimos caracteres mostrados en pantalla y los sustituimos por los de las marcas de las ruedas.

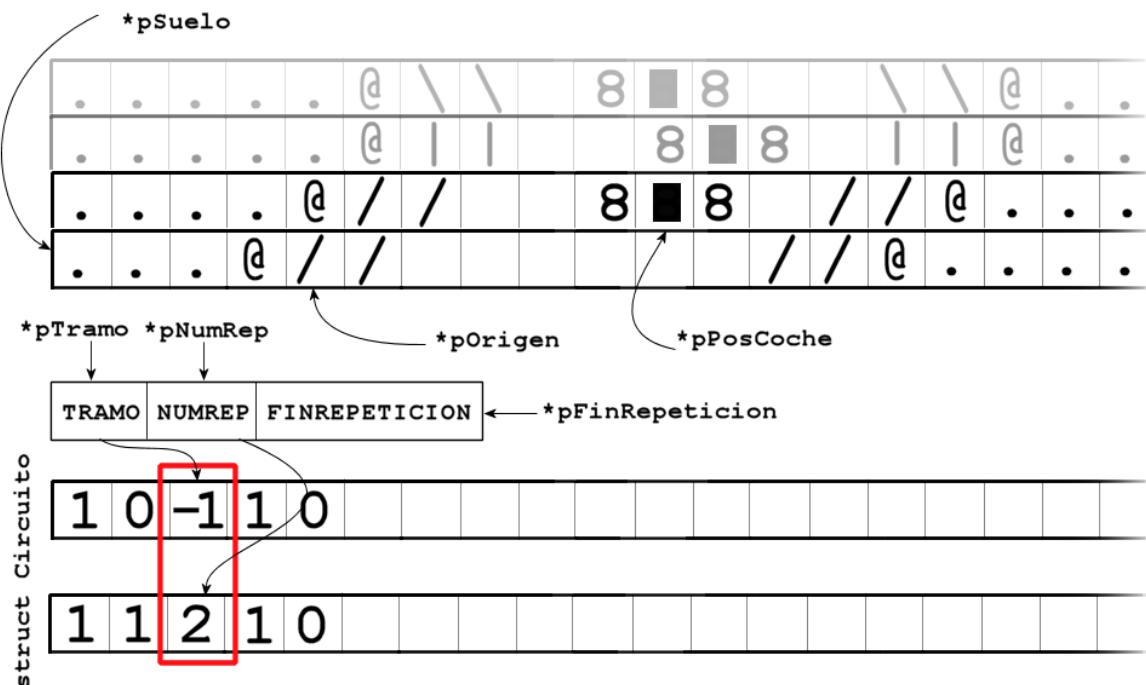


Figura 5.23: Variables del sistema creadas durante el desarrollo del juego de Rally

5.4.4 Proceso de la interrupción TIMER1

Utilizaremos el temporizador 1 que dispone la plataforma ENT2004CF para producir interrupciones periódicas destinadas al refresco de la pantalla (concretamente, para actualizar los tramos del circuito de Rally).

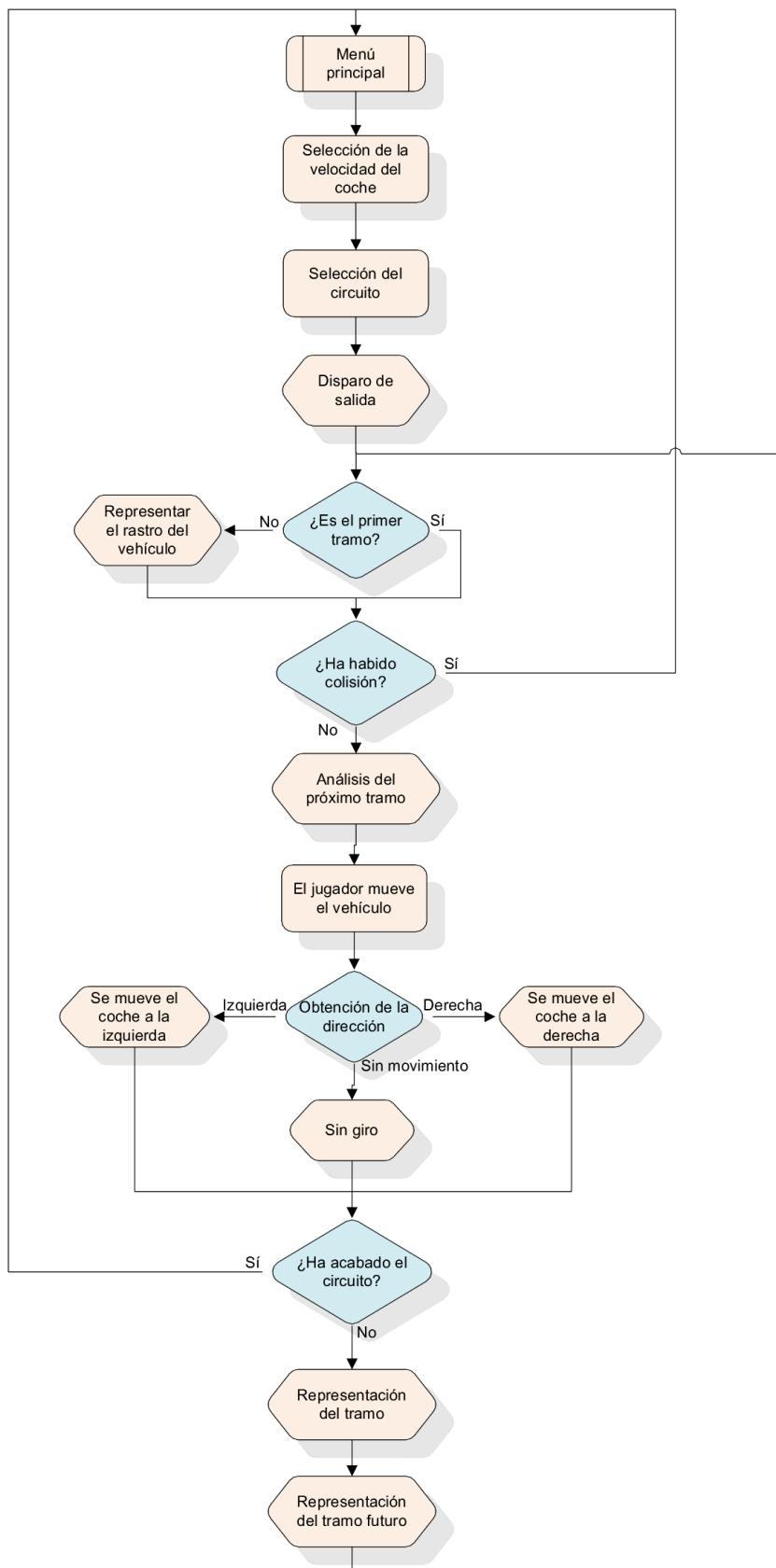
La configuración del TIMER1 será idéntica a la de los juegos “3 en raya” y “Sudo Kool Fire”, que consistía en interrupciones cada milisecondo. En nuestro juego de rally definimos la constante REFRESCO (de valor 100), que contiene el número de milisegundos que se producen entre cada reescritura del circuito por pantalla. Esta constante la modificaremos por una constante multiplicativa que será la velocidad indicada por el usuario (de 1 a 9), de manera que la actualización de pantalla podrá variar entre 1x100 a 9x100 milisegundos.

Sin embargo, hemos de tener en cuenta que la rapidez con que se actualiza la pantalla será mucho más lenta de lo que se podría esperar, ya que hay que tener en cuenta la cantidad de caracteres escritos por pantalla, que producen un retardo no despreciable. Según el caso, los caracteres que se escriben por pantalla son:

- **100 caracteres:** 50 correspondientes al tramo actual + 50 correspondientes al próximo tramo, en caso de que sea el primer tramo del circuito.
- **250 caracteres:** 50 del tramo actual + 50 del próximo tramo + 100 de borrado + 50 del tramo con el rastro del coche, si estamos en cualquier otro tramo del circuito.

Estos datos hay que tenerlos muy en cuenta para justificar el ralentizamiento del juego cuando introducimos la velocidad máxima.

5.4.5 Diagrama del programa principal



5.4.6 Programa principal

```

juegoRally() {
    Bucle de iteración de tramo {
        Reseteo del tiempo de refresco de pantalla
        Obtención de la dirección del vehículo por teclado
        borraVehiculo(); (Se deja el rastro del vehículo)
        detectaColision();
        actualizaPosicion();
        compruebaMeta();
        muestraTramo();
        muestraTramoFuturo();
    }
    Muestra un mensaje de Meta
}

* Variables de entrada: struct circuito Circuito, int velocidad, char *pSuelo, int *pTramo,
int *pNumRep, int *pPosCoche, int *pMeta, int *pFinRepeticion
* Variables de salida: void
* Variables globales modificadas: Reseteo del flag finRefresco al principio de cada tramo

```

Función principal que gestiona el juego “Rally”. En primer lugar se hace un reseteo del tiempo de refresco, para asegurarse de que el siguiente tramo se muestra **como máximo** en el tiempo establecido (a no ser que pulsemos una tecla de dirección). Después se pide al usuario que introduzca la dirección del vehículo para, tras hacer una serie de comprobaciones, mostrar los siguientes tramos del circuito.

5.4.6.1 Funcionalidad setConfiguracionRally

```

setConfiguracionRally() {
    setVelocidad();
    setCircuito();
}

* Variables de entrada: void
* Variables de salida: void

```

Esta función es llamada al iniciar el juego “Rally”. Sirve para configurar la velocidad de juego (velocidad a la que se muestran vehículo y circuito) y escoger un circuito entre los cuatro existentes.

5.4.6.2 Funcionalidad setVelocidad

```

setVelocidad() {
    Solicitud al usuario de la velocidad deseada
    Se imprime por pantalla su respuesta
    Comprobación de respuesta válida
}

* Variables de entrada: void
* Variables de salida: int (velocidad)

```

Función que da la opción de escoger entre las 9 posibles velocidades del juego (9, fácil, a 1, difícil). Tras ser hecha, la elección es mostrada al jugador. Si se pulsa una tecla que representa una opción fuera de rango, se advierte al usuario de que vuelva a hacer la selección.

5.4.6.3 Funcionalidad setCircuito

```
setCircuito() {
    Solicitud al usuario del circuito deseado
    Se imprime por pantalla su respuesta
    Comprobación de respuesta válida
    iniciaRally();
}
```

* Variables de entrada: int velocidad
* Variables de salida: void

Función que da la opción de escoger uno entre los cuatro circuitos disponibles. La elección es mostrada al usuario y si se pulsa una tecla que representa una opción fuera de rango, se advierte al usuario de que vuelva a hacer la selección.

5.4.6.4 Funcionalidad iniciaRally

```
iniciaRally() {
    Creación del suelo
    Creación de variables y flags
    Creación de punteros
    inicializaSuelo();
    situaVehiculo();
    disparaSalida();
    juegoRally();
}
```

* Variables de entrada: struct circuito Circuito, int velocidad
* Variables de salida: void

Esta función realiza la configuración inicial del sistema para poder jugar al juego “Rally” según las opciones escogidas. Se crea la matriz que hará de “suelo” (en la cual se irá escribiendo el circuito) y se sitúa el vehículo en el centro del circuito. Después de esto veremos una cuenta atrás, para posteriormente entrar en el bucle principal del juego “Rally”.

5.4.6.5 Funcionalidad inicializaSuelo

```
inicializaSuelo() {
    Bucle de iteración del suelo {
        Se resetea toda la carretera y se deja el terreno
    }
}
```

* Variables de entrada: char *pSuelo
* Variables de salida: void

Esta función borra cualquier dato (tramos de circuito, coche) que hubiera en la *matriz suelo* y la rellena con el carácter correspondiente al terreno (espacios en blanco, puntos, etc).

5.4.6.6 Funcionalidad situaTramo

```
situaTramo() {
    Obtención de la longitud del tramo
    Bucle de iteración del tramo {
        Se pinta el límite izquierdo
        Se pinta el relleno del tramo
        Se pinta el límite derecho
    }
}

* Variables de entrada: struct tramo Tramo, char *pSuelo, int *pOrigen
* Variables de salida: void
```

Teniendo en cuenta la longitud del tramo y el punto de comienzo de éste, la función actual recorre la matriz de suelo, en busca de la posición comienzo del tramo, posición en la que se escribe el carácter izquierdo. Posteriormente se rellena el interior del tramo con el carácter escogido, y cuando se llega al final del tramo (índice origen + longitud) se pinta el margen derecho.

5.4.6.7 Funcionalidad sigueCircuito

```
sigueCircuito() {
    Obtención del offset del siguiente tramo
    Actualización del margen izquierdo de la carretera
    situaTramo(); (según el tipo de tramo)
    decrementaRepeticion();
}

* Variables de entrada: struct circuito Circuito, char *pSuelo, int *pOrigen, int *pTramo,
int *pNumRep, int *pMeta, int miroFuturo, int *pFinRepeticion
* Variables de salida: void
```

Función que lee las características del siguiente tramo a mostrar. Si es a izquierdas, se decrementa en una unidad el índice que sitúa el comienzo del tramo dentro de la matriz de suelo, si es recto dicho índice se mantiene, y si es a derechas se aumenta en una unidad.

A continuación se sitúa el tramo y se rebaja en una unidad la cuenta de repeticiones de ese tramo.

5.4.6.8 Funcionalidad decrementaRepeticion

```
decrementaRepeticion() {
    Comprobación de tramo no futuro {
        Comprobación de fin de repeticiones de tramo {
            Se decrementan las repeticiones
            Si no se repite más, activación del flag
            Obtención de las repeticiones del siguiente tramo
        }
}
```

```

        }
    }

* Variables de entrada: struct circuito Circuito, int *Tramo, int *pNumRep, int mirofuturo,
int *pFinRepeticion
* Variables de salida: void

```

En caso de que no estemos *observando el futuro* (es decir, mostrando tramos futuros respecto al actual), se decrementa el número de repeticiones. Si resulta que ya se ha agotado las repeticiones para el tramo actual, se obtiene el número de repeticiones del tramo siguiente, para ser usadas en la siguiente iteración.

5.4.6.9 Funcionalidad actualizaPosicion

```

actualizaPosicion() {
    Comprobación de cambio en la dirección {
        situaVehiculo(); (según dirección)
    }
}

* Variables de entrada: struct circuito Circuito, char *pSuelo, int *pOrigen, int *Tramo,
int *pPosCoche, char direccion
* Variables de salida: void

```

Esta función toma como dato la dirección en la que el jugador ha decidido mover el coche, para, en consecuencia, actualizar su posición dentro de la matriz de suelo y que pueda ser mostrado en la posición escogida.

5.4.6.10 Funcionalidad compruebaMeta

```

compruebaMeta() {
    Comprobación de fin de repetición de tramo y último tramo {
        situaTramo(); (de Meta)
        situaVehiculo();
        muestraTramo();
    }
}

* Variables de entrada: struct circuito Circuito, char *pSuelo, int *pOrigen, int *Tramo,
int *pNumRep, int *pPosCoche
* Variables de salida: BOOL (TRUE si se ha llegado a la meta)

```

Esta función comprueba si el circuito ha terminado (tramo recto que se repite cero veces), en cuyo caso, muestra por pantalla una línea de meta con el vehículo en la posición correspondiente.

5.4.6.11 Funcionalidad muestraTramo

```

muestraTramo() {
    Bucle de recorrido del suelo {
        Imprime por pantalla cada carácter que forma el suelo
}

```

```
    }
}

* Variables de entrada: char *pSuelo
* Variables de salida: void
```

Esta función recorre la matriz de suelo, mostrando por pantalla los elementos que contiene: suelo, carretera, límites de la carretera y vehículo.

5.4.6.12 Funcionalidad detectaColision

```
detectaColision() {
    Obtención de la nueva dirección del coche
    Comprobación de que el coche está fuera de la carretera
}

* Variables de entrada: char *pSuelo, int *pPosCoche, char direccion
* Variables de salida: BOOL (TRUE si ha habido colisión)
```

Función que, tomando como dato la posición en la que el usuario desea mover el vehículo, comprueba –antes de mostrar el nuevo tramo con la nueva posición del coche- que no se haya movido el coche a una posición en la que hubiera un límite de la carretera. En tal caso, se muestra un mensaje de aviso de choque y se aborta la partida actual.

5.4.6.13 Funcionalidad situaVehiculo

```
situaVehiculo() {
    Escritura el vehículo en su posición
    Escritura de las ruedas a izquierda y derecha
}

* Variables de entrada: char *pSuelo, int *pPosCoche
* Variables de salida: void
```

Función que muestra el vehículo en la posición adecuada y le añade unas ruedas a ambos lados.

5.4.6.14 Funcionalidad borraVehiculo

```
borraVehiculo() {
    Bucle de recorrido del suelo {
        Borrado del tramo anterior y el tramo futuro
    }
    Se escribe el rastro del coche en la posición antigua
    muestraTramo();
    Se restablece la posición del coche
}

* Variables de entrada: char *pSuelo, int *pPosCoche
* Variables de salida: void
```

Tras representar el tramo actual y el futuro, esta función borra ambos, repintando el actual sin coche (pero con las *marcas* que el vehículo ha dejado en la carretera), aunque ahora será el tramo anterior.

5.4.6.15 Funcionalidad muestraTramoFuturo

```
muestraTramoFuturo() {
    inicializaSuelo();
    sigueCircuito(); (indicando que es mirar tramo futuro)
    muestraTramo();
}

* Variables de entrada: struct circuito Circuito, char *pSuelo, int *pOrigen, int *pTramo,
int *pNumRep, int *pMeta, int *pFinRepeticion
* Variables de salida: void
```

Esta función se encarga de mostrar por la pantalla el tramo siguiente al actual, en el que se encuentra el vehículo, para que el jugador sea capaz de ver *por dónde* ha de mover el coche para no chocarse y perder la partida.

5.4.6.16 Funcionalidad mensajeFinal

```
mensajeFinal() {
    Muestra un mensaje según haya habido colisión o meta
    Reset del flag de fin de refresco de pantalla
}

* Variables de entrada: int meta (flag indicativo de meta)
* Variables de salida: void
Variables globales modificadas: Reseteo del flag finRefresco
```

Función que muestra el correspondiente mensaje de fin de partida, según ésta haya terminado a causa de un choque o de haber llegado a meta.

En cualquier caso es necesario resetear el flag de fin de refresco de la pantalla para que no cause problemas durante la ejecución del resto del software.

5.4.6.17 Funcionalidad disparoSalida

```
disparoSalida() {
    Bucle de cuenta atrás {
        Imprime por pantalla la cuenta atrás
    }
}

* Variables de entrada: void
* Variables de salida: void
```

Esta función muestra por pantalla una cuenta atrás antes del comienzo del juego (salida del coche), que le añade dinamismo.

5.5 Sistema de puntuación en el “3 en raya”

El “3 en raya” es un juego en el que participando dos jugadores humanos, se puede **alargar indefinidamente** debido a sucesivos empates. Y aunque no resulta así en el modo máquina, puesto que podemos suponer que el jugador va a fallar en algún momento, mientras que la máquina siempre va a efectuar el movimiento óptimo, resulta interesante **limitar de alguna manera el juego** y decidir de *alguna forma alternativa* un ganador y un perdedor.

Hay dos formas principales de limitar la duración del juego:

- En primer lugar, establecer un **tiempo de juego máximo**, tras el cual la partida acaba.
- Establecer un **número de movimientos máximos** para cada jugador, si uno de ellos (o ambos) agotan su cantidad de movimientos, la partida acaba.

Una vez terminado el juego mediante una de estas dos posibilidades, es necesario emitir un veredicto sobre quién ha ganado (al menos *parcialmente*). Para ello, tras **cada situación de peligro** creada para el oponente (dos fichas del jugador y un espacio en blanco), se le suma un punto al jugador correspondiente. En caso de situación de *jaque mate* (dos posibles “3 en raya”) se le suman dos puntos al jugador en cuestión.

A continuación un pequeño gráfico del posible estado de una partida con el añadido de la puntuación, y después un diagrama que resume la integración con el **modo clásico** (sin sistema de puntuación, como en la práctica básica de la asignatura)

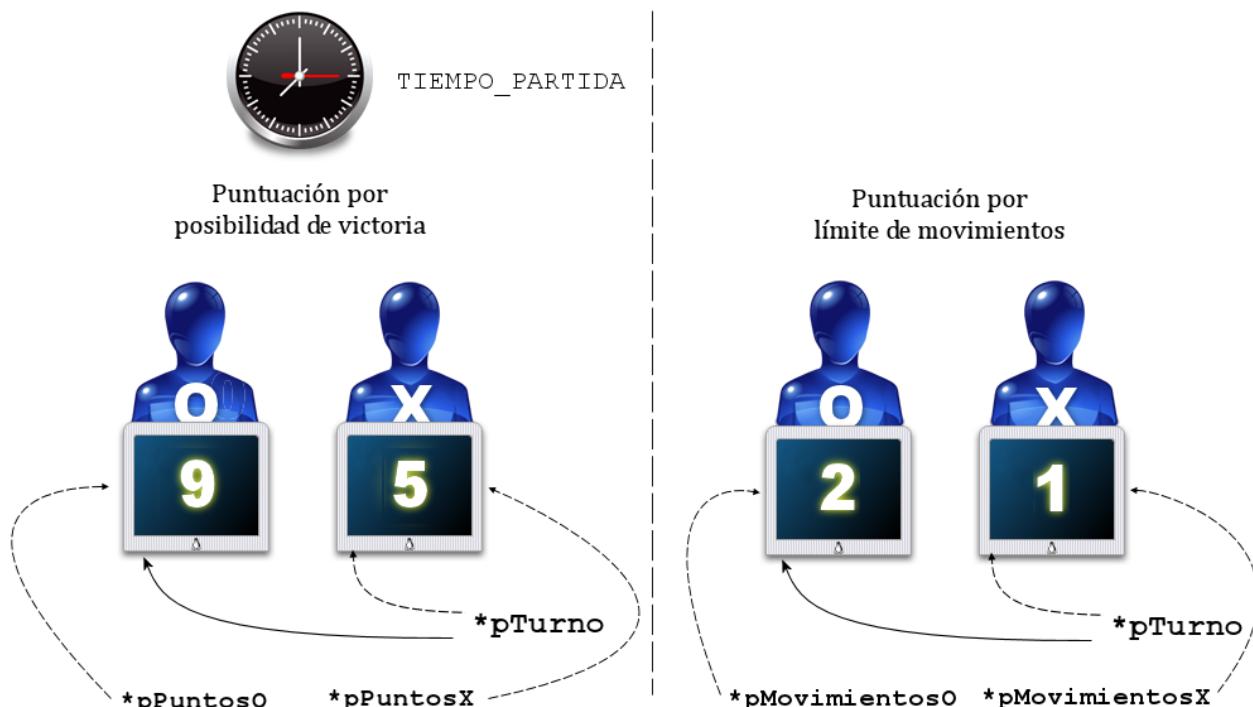
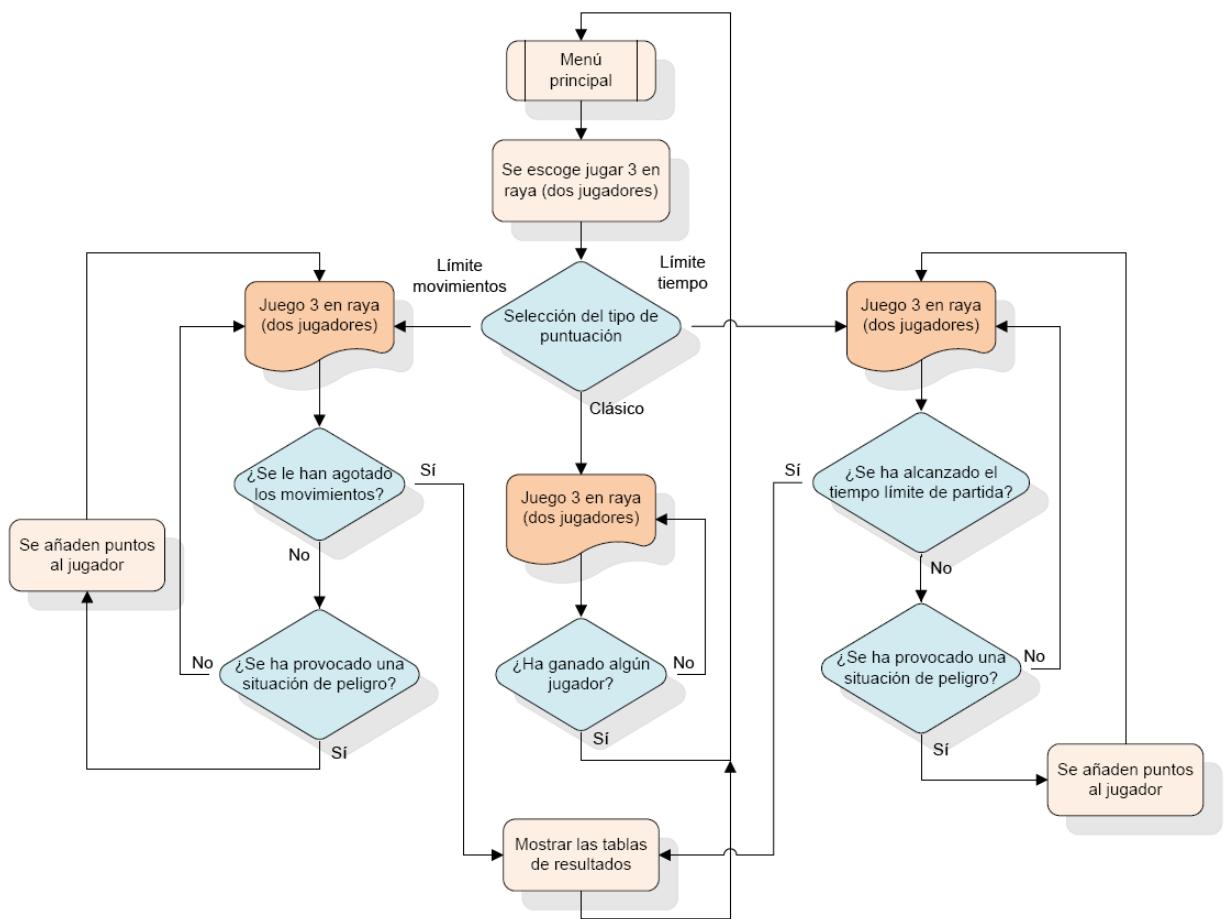


Figura 5.24: Marcadores durante el juego de “3 en raya” para los dos tipos de puntuación definidos.



A continuación se describen las funciones implicadas en esta mejora.

5.5.1 Función inicializaPuntos3EnRaya

```

inicializaPuntos3EnRaya () {
    Creación de punteros para almacenar los puntos de cada jugador
    Inicialización de punteros de puntuación
    Inicialización de los punteros de movimientos
    setPuntuacion3EnRaya ();
}

* Variables de entrada BOOL *pHayGanador, char *pTurno, char *pTablero, int
*pMovimientoO, int *pMovimientosX
* Variables de salida: void

```

Esta función se encarga de crear todas las variables y sus correspondientes punteros, en los que se almacenará la información relativa a puntos y movimientos restantes desarrollada en esta mejora.

5.5.2 Función setPuntuacion3EnRaya

```

setPuntuacion3EnRaya () {
    Selección del tipo de puntuación por parte del usuario
    Puntuación: clásica, por límite de movimientos o por tiempo límite de juego
}

```

```

    Inicialización del 3 en Raya para dos jugadores con dicho tipo de puntuación
}

* Variables de entrada: BOOL *pHayGanador, char *pTurno, char *pTablero, BOOL
*pJugSinTiempo, int *pMovimientosO, int pMovimientosX, int *pPuntosO, int *pPuntosX
* Variables de salida: void

* Variables globales modificadas: tiempoPartida = TIEMPO_PARTIDA*1000 (se setea el tiempo
total de partida en el modo "límite por tiempo de partida")

```

Esta será la principal función de la mejora, pues utilizando los nuevos punteros que se han definido, llamamos a la funcionalidad de juego de dos jugadores con el tipo de puntuación introducido por el usuario. En esta mejora hemos implementado la puntuación por límite máximo de movimientos (configurado a 10, aunque modificable), por tiempo límite de juego (configurado a 5 minutos) o modo clásico, es decir, el sistema de la práctica básica.

5.5.3 Función sumaPuntos

```

sumaPuntos() {
    Según el tipo de puntuación que hayamos elegido {
        Si corresponde, se suman los puntos (salvo que no hayamos escogido sistema
        de puntuación)
        sumaPuntosPosibleVictoria();
    }
}

```

* Variables de entrada TipoPunt3Raya tipo, int *pPuntosO, int *pPuntosX, char *pTurno,
char *pTablero, int *pMovimientosO, int *pMovimientosX, BOOL *pHayGanador

* Variables de salida: void

Esta función se encarga –según el tipo de puntuación escogido- de sumar los puntos correspondientes a cada jugador en caso de que se de una situación de posibilidad de victoria, es decir, dos fichas del mismo jugador en línea con una casilla vacía.

5.5.4 Función decrementaMovimientos

```

decrementaMovimientos() {
    Según el turno actual {
        Se le decrementa en 1 su cuenta de movimientos
    }
}

```

* Variables de entrada char *pTurno, int *pMovimientosO, int *pMovimientosX

* Variables de salida: void

Esta función se encarga de decrementar la cuenta de movimientos del jugador **sólo** si éste ha movido una ficha (criterio que hemos seguido para determinar si no mover afecta a la cuenta de movimientos restantes).

5.5.5 Función sumaPuntosPosibleVictoria

```

sumaPuntosPosibleVictoria() {
    Mientras no hayamos recorrido todo el tablero {

```

```
        Se investigan todas las posibles situaciones que
        puntúan(posibleTresEnRaya()), asignando un único punto a cada una de ellas
        (incrementaPuntos())
    }
}

* Variables de entrada char *pTablero, char *pTurno, int *pPuntos0, int *pPuntosX, BOOL
*pHayGanador
* Variables de salida: void
```

Esta función es de vital importancia para la cuenta de puntos del sistema de puntuación del 3 en raya, ya que investiga las situaciones de peligro para el oponente. En caso de darse esta situación, se suman puntos al jugador que la haya creado (nuestro criterio es un punto por cada situación de posible victoria generada).

5.5.6 Función **incrementaPuntos**

```
incrementaPuntos() {
    Aumenta la cuenta de puntos al jugador correspondiente al turno actual
}

* Variables de entrada char *pTurno, int *pPuntos0, int *pPuntosX, int puntos
* Variables de salida: void
```

Esta función simple incrementa un determinado número de puntos al jugador que acaba de realizar un movimiento de juego.

Función **muestraPuntos**

```
muestraPuntos() {
    Decide cómo hay que mostrar la puntuación
    tablaDePuntos();
}

* Variables de entrada int *pPuntos0, int *pPuntosX, TipoPunt3Raya tipo
* Variables de salida: void
```

Esta función es la encargada de mostrar al final del partido la tabla de resultados en función de que se haya seleccionado un tipo de puntuación para el “3 en raya”.

5.5.7 Función **tablaDePuntos**

```
muestraPuntos() {
    Muestra la cuenta de puntos de cada jugador
    Muestra un mensaje indicando quién ha ganado
}

* Variables de entrada int *pPuntos0, int *pPuntosX
* Variables de salida: void
```

Esta función es la encargada de obtener la puntuación de cada participante del “3 en raya” y valorar la situación final, imprimiendo por pantalla el jugador ganador de la partida.

5.5.8 Función determinaMaximaPuntuacion

```
determinaMaximaPuntuacion() {
```

Dada una puntuación, se devuelve en ganador (máxima puntuación). También se contempla el caso del empate

```
}
```

```
* Variables de entrada int *pPuntos0, int *pPuntosX
```

```
* Variables de salida: void
```

Esta función simple obtiene el valor de máxima puntuación entre los dos jugadores que participan en el “3 en raya” y complementa a la tabla de resultados con el turno del jugador ganador.

6 Principales problemas encontrados

6.1 Dificultades durante el desarrollo de la práctica

En general, el desarrollo de la práctica ha sido fluido, debido a la programación conjunta (dos personas aportando ideas y vigilando la corrección y calidad del código en el momento de escribirlo), salvo en algunos momentos concretos:

- El primero, cuando se empezaron a **sustituir variables globales por punteros** que se pasaban a las funciones como argumentos. Fue necesario reescribir una parte importante de código y debido a la no familiarización con los punteros se cometieron pequeños errores que nos impedían encontrar una fácil solución.
- En el juego del Rally, en ocasiones se intentaba escribir en **posiciones ilegales de memoria** (acceder a un array en la posición -1, por ejemplo), lo que resultaba en errores de bus. Finalmente descubrimos que era problema de los circuitos que habíamos definido, que se salían fuera del vector del escenario.
- Durante la corrección de errores del juego “Sudo Kool Fire”, en especial en la función *hayRepeticionBloque*, que puesto que tenía que analizar **submatrices 3x3** (vectores de 9 elementos) dentro de un vector de 81 caracteres, constaba de una **algorítmica compleja**.
- Durante el diseño del modo multijugador en Red (Sudo Kool Fire): **¿Cómo ha de saber cada ordenador cuando puede y no puede jugar?** Resolvimos estos problemas aplicando una dinámica de consulta periódica al servidor TFTP para conocer el nombre del HOST al que le toca jugar.
- Una vez terminado el Rally, surge la idea de que el circuito vaya **apareciendo antes** que el coche, lo que implica borrar y repintar la pantalla.

6.2 Calificación personal

Los motivos que creemos pueden ser importantes de cara a la calificación de nuestra práctica en el presente curso del *Laboratorio de Sistemas Electrónicos Digitales* son:

- El desarrollo de la práctica ha sido **llevado al día** en todo momento. En las entregas electrónicas se ha cumplido con lo estipulado en todo momento, incluso superando lo exigido (al igual que en las comprobaciones de hitos).
- Nos hemos aventurado en el uso de **herramientas avanzadas de C**, tales como *punteros, enums, structs, arrays de structs*, etc.
- **Creatividad en el desarrollo de mejoras** (juego completo de Sudoku, interconexión de puestos del laboratorio para jugar en red) y abundancia de las mismas (PCB, Puntuación en el 3 en Raya, descarga de tableros de Sudoku, juego de Rally, almacenamiento de Récords, y otras simples de interfaz y presentación).

- **Documentación masiva:** Se ha tratado de documentar de la mejor manera posible el código fuente, y creemos haber superado el 80% de líneas de código comentadas.
- **Memoria exhaustiva pero concisa:** las funciones más importantes han sido comentadas y se han incluido multitud de **gráficos, diagramas de flujo** y de bloques. Asimismo, se ha intentando cuidar al máximo el diseño, para que la lectura del documento fuera lo más amena posible.
- **Nuestra visión del producto:** por un momento nos hemos imaginado que el laboratorio ha sido un procedimiento más de una empresa de software y por ello hemos creado un **manual de usuario muy visual** y una **portada final del juego atractiva** para poder publicitar mejor nuestro trabajo (de momento en la ficción).

Teniendo todos estos aspectos en cuenta, mostramos la tabla con las puntuaciones que esperamos.

Módulos	Puntuación máxima (sobre 100)	Puntuación esperada (sobre 100)
Requisitos mínimos	35	35
Memoria	15	15
Software	25	25
Hardware	5	5
Mejoras	20	20
TOTAL	100	98 ~ 100

7 Manual de usuario

JUEGOS REUNIDOS

[Manual de usuario]

[Menú principal]

¡Bienvenido a "Juegos Reunidos"!

Desde el menú principal puedes elegir tu juego favorito o bien consultar las mejores puntuaciones.

¡Que te diviertas!

[controles]

3 en raya: 2 Jugadores

1	2	3	C	LSED Rally
4	5	6	D	Sudo Kool Fire
7	8	9	E	Mejores Puntuaciones
A	0	B	F	Salir del menú
3 en raya: contra la máquina				



TIC TAC TOE

[2 jugadores]

Reglas del juego:

El jugador 1 sitúa su primera ficha sobre la casilla central y no puede moverla. El primero que consiga alinear sus tres fichas en línea, gana la partida. Si se situán todas las fichas, tendrán que moverlas a casillas vacías. Cada jugador dispone de un turno de 20s para realizar sus movimientos.

Sistemas de puntos:

- | Clásico: sin puntos
- | Límite de 10 movimientos
- | Límite 5 min. de partida

Se obtiene 1 punto por cada posibilidad de ganar. Quien tenga más puntos tras llegar al límite, gana.

[controles]

Posición de la ficha		Posición vacía del tablero	
1	2	3	C
4	5	6	D
7	8	9	E
A	O	B	F



[Jugador 1]



[Jugador 2 | Máquina]

[Contra la máquina]

Reglas del juego:

El jugador humano sitúa su primera ficha sobre la casilla central y no puede moverla. El primero que consiga alinear sus tres fichas en línea, gana la partida. Si se situán todas las fichas, tendrán que moverlas a las casillas vacías. La máquina jugará de forma inteligente y el jugador dispone de 20s por turno.



[modo local]

Reglas del juego:

Se permiten de 1 a 9 jugadores. El objetivo es completar todas las casillas vacías del tablero con los números del 1 al 9, de forma que no se repitan en las filas, columnas o bloques.

Cada jugador dispone de 1min en su turno.

Sistemas de puntos:

- | Por número de aciertos
- | Por rapidez de aciertos (menor tiempo)

El modo ayuda:

El modo ayuda te avisará de tus aciertos o tus fallos. Te será de gran ventaja pero si fallas ¡perderás el turno!

Niveles de dificultad:

Hay 9 niveles disponibles para jugar, además de un tablero extra que se puede descargar en modo online

SUDO KOOL FIRE

[controles]

Elegir fila		Elegir columna		Elegir número
1	2	3	C	
4	5	6	D	
7	8	9	E	
A	O	B	F	

Abandonar

Abandonar

Abandonar

Abandonar

Abandonar

[controles]

Mover vehículo

1	2	3	C
4	5	6	D
7	8	9	E
A	O	B	F

Derecha

Adelante

Izquierda

Reglas del juego

Conduces un rally a toda velocidad, el objetivo es llegar a la meta en el menor tiempo posible a través de todos los circuitos.

Velocidad del vehículo

Puedes elegir la velocidad a la que irá el vehículo. Se disponen 9 marchas desde la más rápida (supersónica), hasta la más lenta (tortuga).

Copas disponibles

Existen una serie de circuitos disponibles ordenados por dificultad:

★ Copas novatos LSED

★★ Copas profes LSED

★★ Copas monitores LSED

★★★ Copas masters LSED

LSED RALLY

[modo online]



Funcionamiento:

Arranca el juego Sudo Kool Fire en el modo online en todas las plataformas que hayas conectado en red.

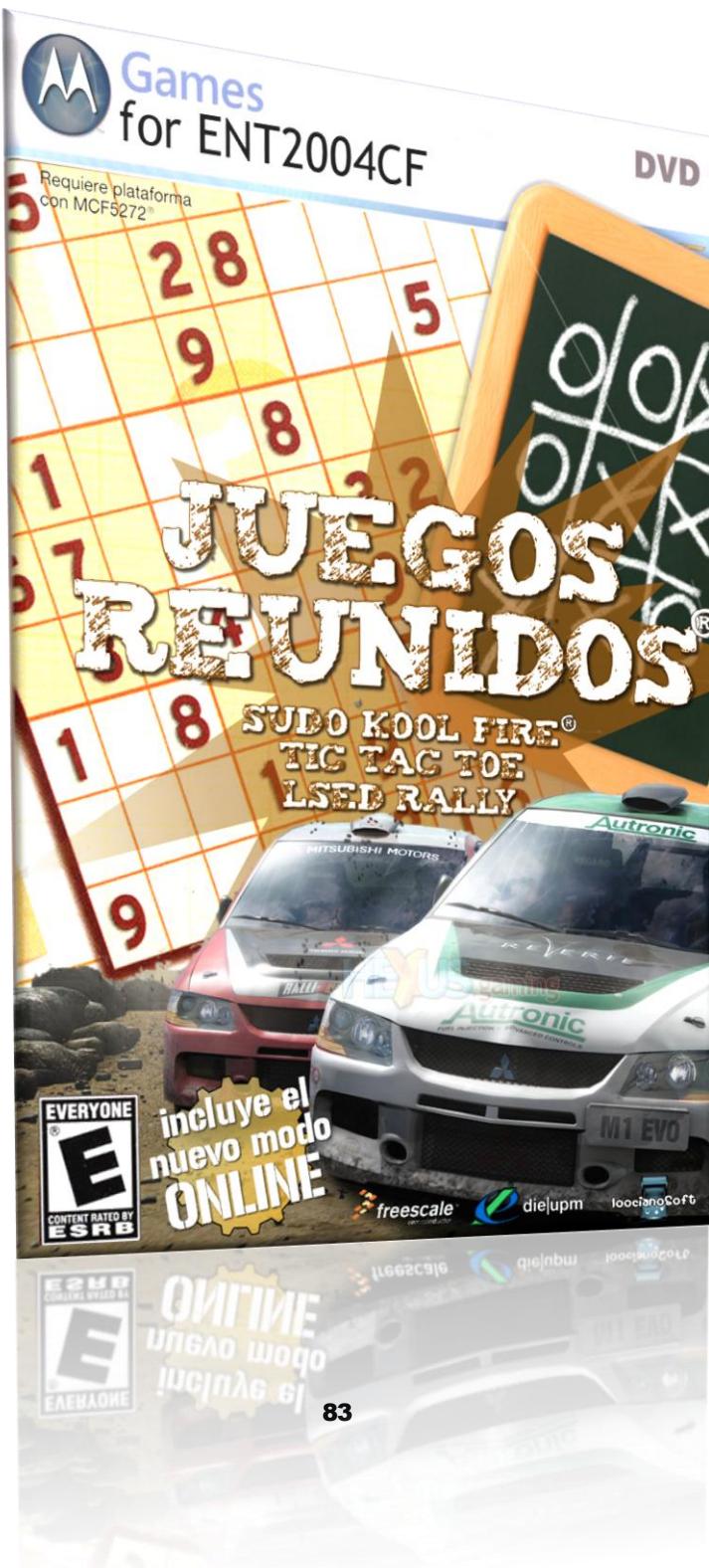
Llama a un adulto para que se lea la info técnica y configure los sistemas adecuadamente.

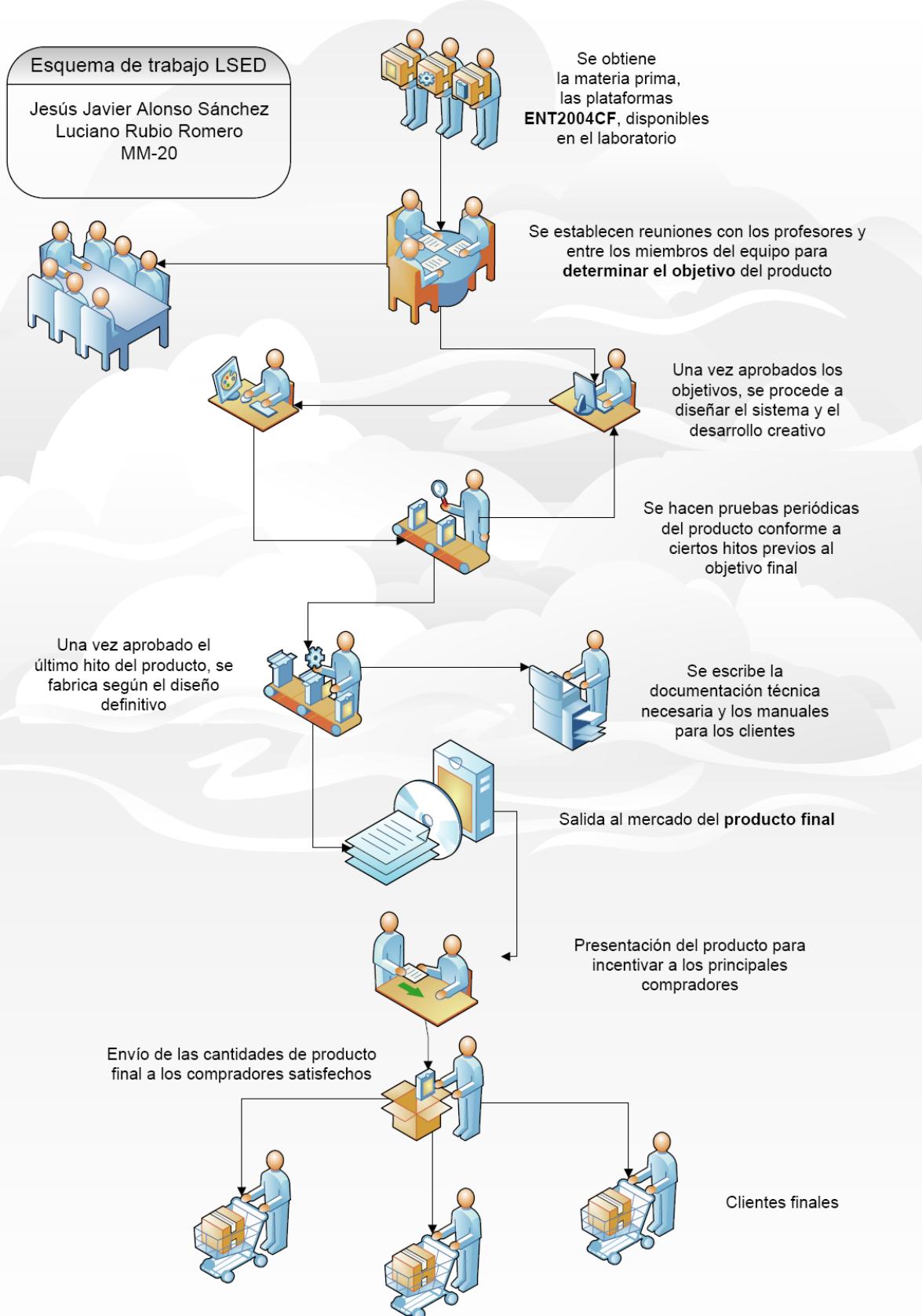
Una vez comience la partida, se descargará el nuevo tablero y empezará la partida en red.

8 Definición del producto

Por un momento nos hemos imaginado como dos ingenieros de una **empresa de software** diseñadora de **juegos de entretenimiento** para controladores semejantes a la plataforma ENT2004CF utilizada en el laboratorio.

En estas condiciones, nuestro objetivo es **ofertar** de la mejor manera posible **nuestro producto** desarrollado en el laboratorio a lo largo de este cuatrimestre. Por ello, hemos echado mano de nuestra creatividad y hemos diseñado la imagen de nuestro programa, un **pack de Juegos Reunidos** que incluye el “**3 en raya**” de la práctica básica, el juego de **Rally** y el **Sudoku** (modo local y **en red**): se trata de darle un enfoque atractivo al cliente (siempre pensando en la ficción).





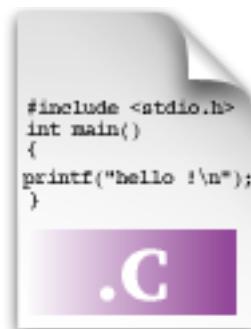
9 Bibliografía

- [1] **Introducción a los Sistemas Digitales con el microcontrolador MCF5272:** Varios, Marcombo, 2006.
- [2] **Entorno de desarrollo EDColdFire 3.0:** Varios, Publicaciones ETSIT-UPM, 2007.
- [3] **Diseño de Sistemas Digitales con el microcontrolador ColdFire 5272:** Varios, Publicaciones ETSIT-UPM, 2007.
- [4] **Diseño con amplificadores operacionales y circuitos integrados analógicos:** Sergio Franco, 3^a Edición, McGraw-Hill, 2002.
- [5] **Circuitos Microelectrónicos:** Sedra/Smithm, Oxford, 1999.
- [6] **The C Programming Language:** Kernighan/Ritchie, 2^a Edición, Prentice Hall, 1988.
- [7] **Beginning C:** Ivor Horton, Apress, 2006.
- [8] **C for Dummies:** Dan Gookin, Paperback.

Enlaces más consultados:

- [1] http://euitio178.ccu.uniovi.es/wiki/index.php/TP:Tres_en_raya_-_Algoritmos_voraces
- [2] Generador de ASCII-Art <http://www.network-science.de/ascii/>
- [3] Página de Sudoku en Wikipedia, <http://en.wikipedia.org/wiki/Sudoku>
- [4] Instrucciones para el LCD: <http://www.myke.com/graphics/lcd-ch1a.gif>
- [5] [Dudas y preguntas frecuentes sobre programación en C](#)
- [6] [Dudas frecuentes](#) de LSED

10 ANEXO I: Código del programa de la primera sesión



lSED.c

```
#include "m5272lib.c"
#include "m5272gpio.c"

#define NUM_FILAS 4           // Número de filas del teclado matricial
#define NUM_COLS 4            // Número de columnas del teclado matricial
#define EXCIT 1

static char tablero[3][3] = {{"   "}, {" X "}, {"   "}};           // Tablero de juego del 3 en raya
static char teclas[4][4] = {{"123C"},                      // Definición de las teclas del teclado matricial
                           {"456D"}, 
                           {"789E"}, 
                           {"A0BF"}};

char tecla;
char turno;
BOOL hayGanador;
```

```
-----  
// char teclado(void)  
//  
// Descripción:  
//   Explora el teclado matricial y devuelve la tecla  
//   pulsada  
-----  
char teclado(void)  
{  
    char tecla;  
    BYTE fila, columna, fila_mask;  
    // Bucle de exploración del teclado  
    while(TRUE){  
  
        // Excitamos una columna  
        for(columna = NUM_COLS - 1; columna >= 0; columna--){  
            set_puertoS(EXCIT << columna);  
            retardo(1150);  
  
            // Se envía la excitación de columna  
            // Esperamos respuesta de optoacopladores  
  
            // Exploramos las filas en busca de respuesta  
            for(fila = NUM_COLS - 1; fila >= 0; fila--){  
                fila_mask = EXCIT << fila;  
                if(lee_puertoE() & fila_mask){  
                    while(lee_puertoE() & fila_mask);  
                    retardo(1150);  
                    return teclas[fila] [columna];  
                }  
            }  
            // Siguiente columna  
        }  
        // Exploración finalizada sin encontrar una tecla pulsada  
    }  
    // Reiniciamos exploración  
}
```

```
void bucleMain(void)
Bucle principal del sistema
-----
void bucleMain(void)
{
    hayGanador = FALSE;
    turno = 'X';
    tecla = menuPresentar();
    if(tecla == 'A') {
        output("Juego contra la máquina\n");
        muestraTablero();
        tecla = teclado();
    }
    if(tecla == 'B') {
        dosJugadores();
    }
    else {
    }
}
```

```
void dosJugadores()
{
    output("Juego entre dos jugadores\n");
    output("Comienza el jugador 'X'\n");

    do {
        muestraTablero();
        cambiaTurno();
        tecla = teclado();
        escribeTablero(tecla);
        hayTresEnRaya();
    } while (!hayGanador);

    muestraTablero();
    output("¡Tres en Raya!\n");
}
```

```
char menuPresentar()
{
    output("Elija el tipo de juego deseado: (A) contra la máquina, (B) entre dos jugadores:\n");
    tecla= teclado();
    outch(tecla);
    output("\n");
    return tecla;
}
```

```
void muestraTablero()
{
    BYTE fila, columna;
    output("Tablero:\n");
    for (fila = 0; fila < 3; fila++) {
        for(columna = 0; columna < 3; columna++) {
            outch(tablero[fila][columna]);
            if (columna != 2) {
                output(" | ");
            }
        }
        if (fila != 2) {
            output("\n-----");
        }
        output("\n");
    }
}
```

```
void escribeTablero(char posicion)
{
    BYTE fila, columna;
    BYTE fila_pos, columna_pos;
    for (fila = 0; fila < 4; fila++) {
        for (columna = 0; columna < 4; columna++) {
            if (posicion == teclas[fila][columna]) {
                fila_pos = fila;
                columna_pos = columna;
            }
        }
    }
    if (tablero[fila_pos][columna_pos] == 0x20) {
        tablero[fila_pos][columna_pos] = turno;
    } else {
        output("Error! Ya hay una ficha en esa posición\n");
        tecla = teclado();
        escribeTablero(tecla);
    }
}
```

```
void cambiaTurno()
{
    if (turno == 'X')
    {
        turno = 'O';
    }
    else
    {
        turno = 'X';
    }
}
```

```
void hayTresEnRaya()
{
    BYTE fila, columna;
    char simbolo_ref;
    for (fila = 0; fila < 3; fila++) {
        simbolo_ref = tablero[fila][0];
        for (columna = 0; columna < 3; columna++) {
            if (tablero[fila][columna] == 0x20) {
                break;
            }
            if (tablero[fila][columna] != simbolo_ref) {
                break;
            } else {
                if (columna == 2) {
                    hayGanador = TRUE;
                    return;
                } else {
                    continue;
                }
            }
        }
    }

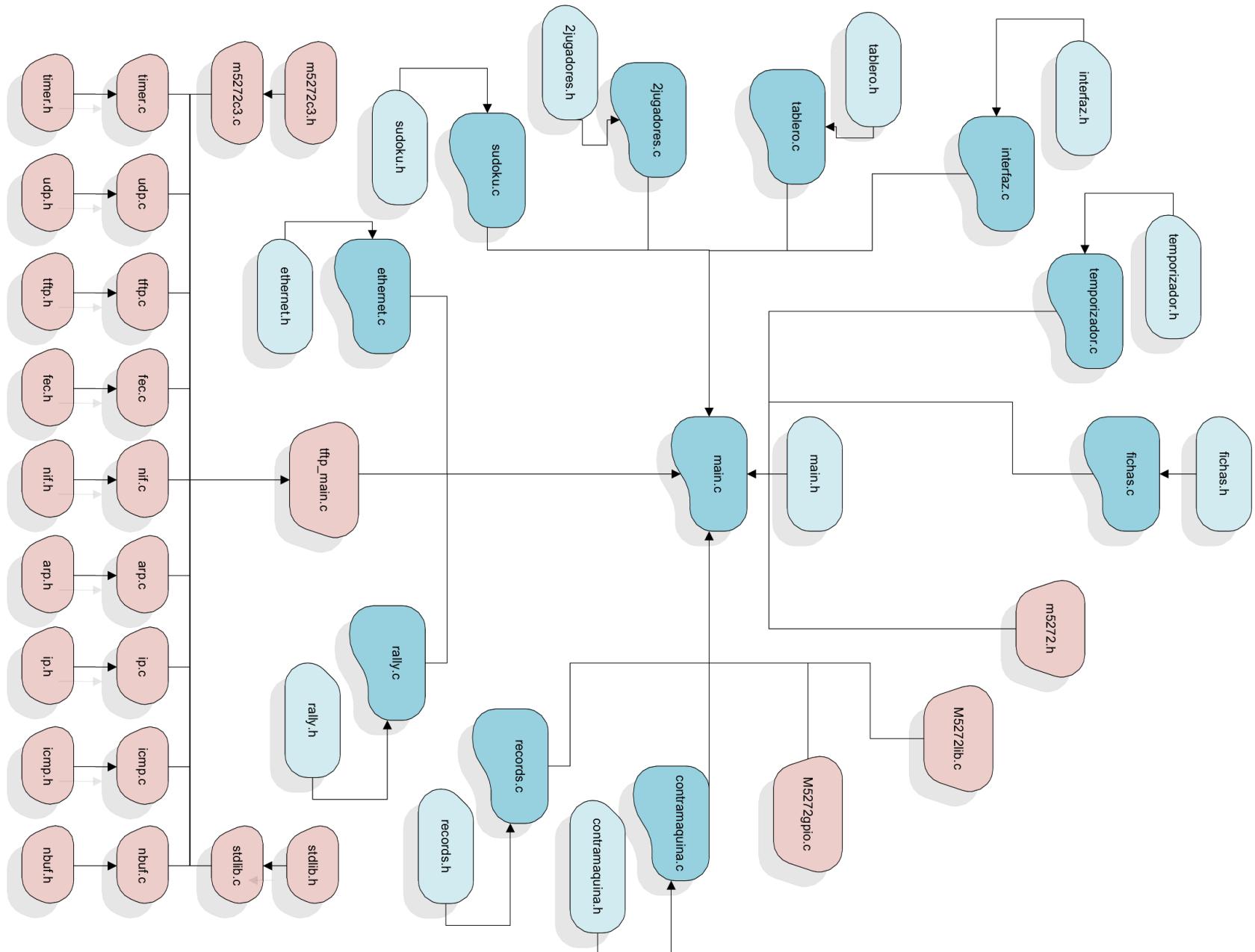
    for (columna = 0; columna < 3; columna++) {
        simbolo_ref = tablero[0][columna];
        for (fila = 0; fila < 3; fila++) {
            if (tablero[fila][columna] == 0x20) {
                break;
            }
            if (tablero[fila][columna] != simbolo_ref) {
                break;
            } else {
                if (fila == 2) {
                    hayGanador = TRUE;
                    return;
                } else {
                    continue;
                }
            }
        }
    }
}
```

```
simbolo_ref = tablero[1][1];
    if ( (tablero[0][0] == simbolo_ref) && (tablero[2][2] == simbolo_ref) ) {
        hayGanador = TRUE;
        return;
    }
    if ( (tablero[2][0] == simbolo_ref) && (tablero[0][2] == simbolo_ref) ) {
        hayGanador = TRUE;
        return;
    }
}
return;
```

```
void __init(void) {}

void rutina_int1(void) {}
void rutina_int2(void) {}
void rutina_int3(void) {}
void rutina_int4(void) {}
void rutina_tout0(void) {}
void rutina_tout1(void) {}
void rutina_tout2(void) {}
void rutina_tout3(void) {}
```

11 ANEXO II: Código del programa del proyecto final





main.h

main.h

Definición de alias y constantes.

```
#define V_BASE 0x40          // Dirección de inicio de la tabla de vectores de interrupción
#define DIR_VTMR0 4*(V_BASE+5) // Dirección del vector de TMR0
#define DIR_VTMR1 4*(V_BASE+6) // Dirección del vector de TMR1
#define DIR_VTMR2 4*(V_BASE+7) // Dirección del vector de TMR2
#define DIR_VTMR3 4*(V_BASE+8) // Dirección del vector de TMR3

#define DIR_VETHRX 4 * (V_BASE + 22) // Dirección del vector de interrupcion de ethernetrx
#define DIR_VETHTX 4 * (V_BASE + 23) // Dirección del vector de interrupcion de ethernettx

#define FREC_INT 1000           // Frec. de interr. de TMR1 = 1000 Hz (cada 1ms)
#define CNT_INT1 MCF_CLK/(FREC_INT*0x4F*16) // Valor de precarga del temporizador de interrupciones TRR1
#define CNT_INT2 MCF_CLK/(FREC_INT*0x4F*16) // Valor de precarga del temporizador de interrupciones TRR2
#define BORRA_REF 0x0002        // Valor de borrado de interr. pendientes de tout1 para TER1

#define TIEMPO_TURNO 20         // Tiempo de cada turno en segundos
#define CASILLA_VACIA 0x20      // Casilla vacía = espacio en blanco
#define TIEMPO_JUEGO_MAX 59     // Definición de tiempo máximo de juego
#define MOVIMIENTOS_MAX 10      // Movimientos máximos para cada jugador

#define NUM_CASILLAS 9          // Definición de número de casillas del tablero de 3 en raya
#define FICHA_CENTRAL 4          // Definición de la posición central del tablero
#define FICHAS_MAX 3            // Definición de número de fichas máximo en 3 en raya
#define TIEMPO_WAIT 1000000     // Retardo de 1s

#define espera() retardo(TIEMPO_WAIT); // Retardo de 1s

#define TIEMPO_PARTIDA 30        // Tiempo de la partida
```

Definición de variables de ámbito global.

```

BOOL finRefresco;                                // Definición del flag de fin de refresco de pantalla
BOOL turnoPerdido;                               // Definición del flag de turno perdido de un jugador
BYTE tiempo;                                     // Definición de la variable que controla el tiempo a presentar por pantalla
BOOL pintoTiempo;                                // Definición del flag que controla los segundos restantes mostrados por pantalla
int tiempoPartida;                             // Definición de la variable que almacena el tiempo de una partida
BOOL timeout;                                    // Definición del flag que controla si el tiempo de una partida ha expirado

volatile ULONG cont_retardo;                     // Contador de milisegundos de tiempo de turno restante
volatile int duracion_nota = 0;                  // Contador de milisegundos de tiempo de nota musical restante
volatile ULONG refresco;                         // Contador de milisegundos de tiempo de refresco de pantalla en el rally

int estadoJuego;                                // Variable de estado, determina el estado del programa:
                                                // 0: no juego
                                                // 1: modo dos jugadores
                                                // 2: modo máquina
                                                // 3: partida finalizada (gana el jugador)
                                                // 4: partida finalizada (gana la máquina)

BOOL cambioMelodia;                            // Flag de cambio de melodía durante el juego
int indice;                                     // Definición del índice que apunta a la nota musical a reproducir

typedef enum {MOVIMIENTOS, TIEMPO_GLOBAL, NINGUNO} TipoPunt3Raya;      // Tipo de puntuación del juego 3 en raya

struct melodia {                                // Definición del tipo melodía
    int frecuencia[100];                         // Frecuencia de las notas a reproducir
    int duracion[100];                           // Duración de las notas a reproducir
    BOOL seRepite;                                // Flag que determina si la sintonía se repite
    char identificador;                          // Identificador de la melodía
};

struct melodia Musica_fondo = {                // Definición de la melodía de fondo durante el juego (canción de Superman)
    {784,784,784,1047,1047,1568,0,1568,1760,1568,1397,1568,0,784,784,784,1047,1047,1568,0,1568,1760,1568,1397,1760,1568,0,1047,1047,1976,1568,1047,1047,1047,1976,1568,1047,1047,1047,1976,1760,1976,2093,1047,1047,1047,1047,1047,1047,1047},  

    {166,166,166,333,166,666,166,166,111,83,166,1333,166,166,166,166,333,166,666,166,166,111,83,166,166,444,333,166,166,166,444},  

    TRUE,  

    'F'  

};

struct melodia Musica_ganador = { // Definición de la melodía de ganador (tres notas ascendentes)

```

```
{0,523,659,784},  
{500,300,300,300},  
FALSE,  
'G'  
};  
  
struct melodía Musica_perdedor = { // Definición de la melodía de perdedor (tres notas descendentes)  
{0,392,330,262},  
{500,500,500,500},  
FALSE,  
'P'  
};
```

Definición de cabeceras de funciones.

```
void bucleMain(void);

void inicializaPuntuacion3EnRaya(BOOL *pHayGanador, char *pTurno, char *pTablero, int *pMovimientosO, int *pMovimientosX);

void setPuntuacion3EnRaya (BOOL *pHayGanador, char *pTurno, char *pTablero, int *pMovimientosO, int *pMovimientosX, int *pPuntosO, int *pPuntosX);

void preparaTurno(char *pTurno, char *pTablero, TipoPunt3Rayas tipo, int *pMovimientosO, int *pMovimientosX);

void movimientoHumano(char *pTurno, char *pTablero);

void setTIMER1 (void);
```



main . c

main.c

```
#include "m5272.h"
#include "m5272lib.c"
#include "m5272gpio.c"
#include "tftp_main.c"
#include "main.h"
#include "interfaz.c"
#include "tablero.c"
#include "fichas.c"
#include "haytresenraya.c"
#include "contramaquina.c"
#include "dosjugadores.c"
#include "temporizador.c"
#include "rally.c"
#include "records.c"
#include "ethernet.c"
#include "sudoku.c"
```

```
void bucleMain(void)

Bucle principal del programa. Se inicializan y crean variables que serán necesarias para cada modo de juego y se presenta al usuario un menú donde seleccionar el tipo de juego deseado.
-----

void bucleMain(void) {

    char turno;                                // Variable turno del 3 en raya, vale X u O
    char *pTurno = &turno;                      // Puntero hacia el turno del 3 en raya

    BOOL hayGanador = FALSE;                    // Flag para saber cuando hay que parar el juego 3 en raya
    BOOL *pHayGanador = &hayGanador;            // Puntero hacia el flag de ganador

    char tablero[NUM_CASILLAS] = {"      X    "}; // Tablero del juego 3 en raya. Se inicializa con X en la posición central
    char *pTablero = &tablero[0];                // Puntero al comienzo del tablero

    int movimientosO;
    int *pMovimientosO = &movimientosO;        // Variable que almacena los movimientos del Jugador O

    int movimientosX;
    int *pMovimientosX = &movimientosX;        // Variable que almacena los movimientos del Jugador X

    turnoPerdido = FALSE;                      // Inicialización de flag de turno perdido
    pintoTiempo = FALSE;                       // Inicialización del flag que permite que el tiempo restante de turno se muestre por pantalla
    timeout = FALSE;
    estadoJuego = 0;                           // Inicialización del estado del sistema

    setTIMER1();                                // Inicialización del TIMER1
    switch(menuPrincipal()) {

        case 'A': // 3 en raya contra la máquina
            tictactoeTitulo();                  // Se muestra el título en ASCII-ART
            estadoJuego = 1;                   // Actualización el estado del sistema
            // Llamada al juego contra la máquina
            contraMaquina(pHayGanador, pTurno, pTablero, NINGUNO, pMovimientosO, pMovimientosX);
            break;

        case 'B': // 3 en raya para dos jugadores
            tictactoeTitulo();                  // Se muestra el título en ASCII-ART
            estadoJuego = 2;                   // Actualización del estado del sistema
            // Llamada al juego de dos jugadores
    }
}
```

```
    inicializaPuntuacion3EnRaya(pHayGanador, pTurno, pTablero, pMovimientosO, pMovimientosX);
    break;

case 'C': // Rally
    setConfiguracionRally();
    break;

case 'D': // Sudo kool fire (en modo online y local)
    sudokuTitulo();                                // Se muestra el título en ASCII-ART
    iniciaSudoku();                               // Llamada al juego de Sudoku
    break;

case 'E': // Lista de mejores puntuaciones (records)
    muestraRecords(TRES_EN_RAYA);
    break;

case 'F': // Salir del programa
    exit(0);                                     // Se sale de la ejecución

default:
    break;                                         // No se hace nada en caso de no elegir una opción correcta
}
```

```
void inicializaPuntuacion3EnRaya (BOOL *pHayGanador, char *pTurno, char *pTablero, int *pMovimientosO, int *pMovimientosX)

Inicia las variables y punteros que van a ser usados en el modo de juego "2 jugadores con puntuación".
-----
```

```
void inicializaPuntuacion3EnRaya(BOOL *pHayGanador, char *pTurno, char *pTablero, int *pMovimientosO, int *pMovimientosX) {

    int puntosO;                                // Variable que guarda los puntos del Jugador O
    int *pPuntosO = &(puntosO);

    int puntosX;                                // Variable que guarda los puntos del Jugador X
    int *pPuntosX = &(puntosX);

    *pPuntosO = 0;                               // Se inicializan los puntos de los Jugadores
    *pPuntosX = 0;

    *pMovimientosX = MOVIMIENTOS_MAX;           // Se setea el número máximo de movimientos de cada Jugador
    *pMovimientosO = MOVIMIENTOS_MAX;

    setPuntuacion3EnRaya(pHayGanador, pTurno, pTablero, pMovimientosO, pMovimientosX, pPuntosO, pPuntosX);
}
```

```
setPuntuacion3EnRaya (BOOL *pHayGanador, char *pTurno, char *pTablero, int *pMovimientosO, int *pMovimientosX, int *pPuntosO,
                      int *pPuntosX)

Función que da la opción de elegir entre los distintos modos de puntuación existentes, y arranca el modo "2 jugadores" con la
configuración escogida.
```

```
void setPuntuacion3EnRaya (BOOL *pHayGanador, char *pTurno, char *pTablero, int *pMovimientosO, int *pMovimientosX, int *pPuntosO, int
* pPuntosX) {

    int opcion;                                // Variable que guarda la opción que escoge el usuario

    do {
        output("\nSelecciona el tipo de puntuación: ");
        output("\n\n[1 - Clásico: sin puntuación]");
        output("\n\n[2 - Por límite de movimientos]");
        output("\n\n[3 - Por límite de tiempo global]\n");
        opcion = teclado() - '0';

    } while (opcion < 1 || opcion > 3); // Comprobación de elección válida

    switch(opcion) {

        case 1: // Clásico (sin límite de movimientos, sin límite de tiempo)
            dosJugadores(pHayGanador, pTurno, pTablero, NINGUNO, pPuntosO, pPuntosX, pMovimientosO, pMovimientosX);
            break;
        case 2:
            // Por límite de movimientos
            dosJugadores(pHayGanador, pTurno, pTablero, MOVIMIENTOS, pPuntosO, pPuntosX, pMovimientosO, pMovimientosX);
            break;

        case 3: // Por límite de tiempo de la partida
            tiempoPartida = TIEMPO_PARTIDA*1000;
            dosJugadores(pHayGanador, pTurno, pTablero, TIEMPO_GLOBAL, pPuntosO, pPuntosX, pMovimientosO,
pMovimientosX);
            break;

        default:
            break;
    }
}
```

```
void preparaTurno(char *pTurno, char *pTablero, TipoPunt3EnRaya tipo, int *pMovimientosO, int *pMovimientosX)

Realiza las operaciones previas a un turno de juego.

-----
void preparaTurno(char *pTurno, char *pTablero, TipoPunt3Rayas tipo, int *pMovimientosO, int *pMovimientosX) {
    turnoPerdido = FALSE;                                // Reseteo del flag de turno perdido (tiempo agotado)
    muestraTablero(pTablero);                            // Se imprime el tablero por pantalla
    cambiaTurno(pTurno);                               // Conmuta el turno de jugador
    menuJuego(pTurno, tipo, pMovimientosO, pMovimientosX); // Se imprime el menú durante el juego por pantalla
}
```

```
void movimientoHumano(char *pTurno, char *pTablero)

Gestiona el movimiento de un jugador real, y determina si ha de poner o mover ficha.
-----

void movimientoHumano(char *pTurno, char *pTablero) {

    char tecla;
    mensajeTurno(pTurno, pTablero);      // Se imprime por pantalla el mensaje del turno correspondiente
    tecla = teclado();
    if (tecla == 'G') {                  // G es un flag que se recibe cuando el tiempo de partida ha expirado y hay que abortar la partida
        return;
    }
    if (!turnoPerdido) {

        if (todasLasFichas(pTurno, pTablero)) {          // Comprobación de que el turno tenga todas sus fichas colocadas
            mueveFicha(tecla, pTurno, pTablero); // Si ha agotado sus fichas, se le pide mover una de ellas
        } else {
            // Si aún no las ha colocado todas, se le pide que coloque la siguiente
            escribeTablero(tecla, pTurno, pTablero);
        }
    }
}
```

```
void __init(void)
{
    Inicialización.
-----
void __init(void)
{
    mbar_writeByte(MCFSIM_PIVR,V_BASE);           // Fijamos el comienzo de vectores de interrupción en V_BASE
    indice = 0;                                  // Inicialización del índice, controla la nota musical a reproducir
    cambioMelodia = FALSE;                      // Flag que detecta un cambio en la melodía
    timeout = FALSE;                            // Flag que detecta si el tiempo de partida (en el 3 en raya) se ha agotado

    ACESO_A_MEMORIA_LONG(DIR_VTMR0)= (ULONG) _prep_TOUT0;      // Escribimos la dirección e la función para TMR0
    ACESO_A_MEMORIA_LONG(DIR_VTMR1)= (ULONG) _prep_TOUT1;      // Escribimos la dirección de la función para TMR0
    ACESO_A_MEMORIA_LONG(DIR_VTMR2) = (ULONG) _prep_TOUT2;      // Escribimos la dirección e la función para TMR2
    ACESO_A_MEMORIA_LONG(DIR_VTMR3) = (ULONG) _prep_TOUT3;      // Escribimos la dirección e la función para TMR3

    setTIMER1();          // Inicialización del TIMER1

    // Escribimos la dirección de la función de atención a la interrupción EthernetRx
    ACESO_A_MEMORIA_LONG(DIR_VETHRX) = (ULONG) _prep_ETHRX;

    // Escribimos la dirección de la función de atención a la interrupción EthernetTx
    ACESO_A_MEMORIA_LONG(DIR_VETHTX) = (ULONG) _prep_ETHTX;

    // Marcamos la interrupción EthernetRx como no pendiente y de nivel 6
    mbar_writeLong(MCFSIM_ICR3, 0x888888E88);

    // Marcamos la interrupción EthernetTx como no pendiente y de nivel 6
    mbar_writeLong(MCFSIM_ICR3, 0x888888E8);

    cont_retardo = 0;          // Inicialización de la cuenta de retardo (milisegundos para TMR1)
    sti();                    // Habilitamos interrupciones
}
```

```
void setTIMER1 (void)  
  
Configura el TIMER1 para generar interrupciones periódicas de 1ms con nivel de interrupción 7.  
-----  
  
void setTIMER1 (void) {  
  
    mbar_writeShort(MCFSIM_TMR1, 0x4F3D);           // TMR0: PS=0x50-1 CE=00 OM=1 ORI=1 FRR=1 CLK=10 RST=1  
    mbar_writeShort(MCFSIM_TCN1, 0x0000);           // Ponemos a 0 el contador del TIMERO0  
    mbar_writeShort(MCFSIM_TRR1, CNT_INT1);         // Fijamos la cuenta final del contador  
    mbar_writeLong(MCFSIM_ICR1, 0x88888F88);        // Marca la interrupción del TIMER1 como no pendiente y de nivel 7  
}
```

```
void rutina_int1(void) {}  
No se utiliza en el programa pero es necesario definirla.  
-----  
void rutina_int1(void) {}
```

```
void rutina_int2(void) {}  
No se utiliza en el programa pero es necesario definirla.  
-----  
void rutina_int2(void) {}
```

```
void rutina_int3(void) {}  
No se utiliza en el programa pero es necesario definirla.  
-----  
void rutina_int3(void) {}
```

```
void rutina_int4(void) {}  
No se utiliza en el programa pero es necesario definirla.  
-----  
void rutina_int4(void) {}
```

```
void rutina_tout0(void)

Rutina de interrupción del TIMER0. Genera ondas cuadradas de frecuencia la indicada por la nota musical a reproducir.
-----
void rutina_tout0(void) {

    extern mcf5272_timer timer[];           // Definiciones necesarias para la mejora TFTP
    timer_default_isr(&timer[0]);
}
```

```
void rutina_tout1(void)

Rutina de interrupción del temporizador 1. Interrumpe cada 1ms y controla el tiempo restante del turno actual para todos los juegos.
También se encarga de controlar la duración de cada nota de una melodía.

-----
void rutina_tout1(void){

    extern mcf5272_timer timer[];           // Definiciones necesarias para la mejora TFTP

    timer_default_isr(&timer[1]);

    mbar_writeShort(MCFSIM_TER1,BORRA_REF);   // Reset del bit de fin de cuenta

    if (refresco > 0) {
        refresco--;
        if (refresco == 0) {
            finRefresco = TRUE;           // Se activa el flag de fin de refresco
        }
    }

    if (tiempoPartida > 0) {                // Se verifica que se ha configurado un tiempo máximo de partida (ms)
        tiempoPartida--;                   // Se decrementa (ms)
        if (tiempoPartida == 0) timeout = TRUE; // Se activa el flag de tiempo de partida agotado
    }

    if (duracion_nota > 0) {                // Comprobación de duración de nota
        duracion_nota--;                   // Decremento de la cuenta de duración de una nota musical.
        if (duracion_nota == 0) {
            desactivaTIMER0();           // En caso de finalizar el tiempo de nota, se desactiva el TIMER0
            indice++;                   // Se incrementa el índice para reproducir la siguiente nota
        }
    } else {

        switch(estadoJuego) {             // En función del estado del sistema, se reproduce la melodía adecuada

            case 1:
                reproduceNota(Musica_fondo); // Reproducción de música de fondo para el estado 1
                break;
            case 2:
                reproduceNota(Musica_fondo); // Reproducción de música de fondo para el estado 2
                break;
        }
    }
}
```

```
    case 3:
        reproduceNota(Musica_ganador);      // Reproducción de música de ganador para el estado 3
        break;
    case 4:
        reproduceNota(Musica_perdedor);    // Reproducción de música de perdedor para el estado 4
        break;
    default:
        break;
    }

    if(cont_retardo > 0){                // Comprobación de duración de turno
        if (cont_retardo % 1000 == 0){      // Comprobación de que han transcurrido 1000ms = 1 s
            tiempo = cont_retardo/1000;    // Seteo de la variable de segundos restantes
            pintoTiempo = TRUE;          // Activación de flag que modifica el valor de tiempo restante por pantalla
        }
        cont_retardo--;                  // Decremento del contador
        if (cont_retardo == 0) {
            turnoPerdido = TRUE;        // Comprobación de tiempo de turno agotado
            // Se activa el flag de turno perdido.
        }
    }
}
```

```
void rutina_tout2(void)

Rutina de atención de la interrupción del temporizador 2, utilizado en la conexión TFTP.
-----
void rutina_tout2(void) {

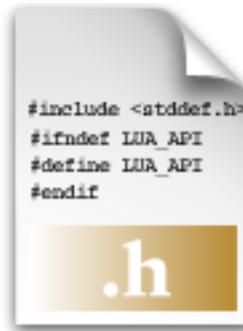
    extern mcf5272_timer timer[];
    timer_default_isr(&timer[2]);
}
```

```
void rutina_tout3(void)  
Rutina de atención de la interrupción del temporizador 2, utilizado en la conexión TFTP.  
-----  
void rutina_tout3(void) {  
  
    extern mcf5272_timer timer[];  
    timer_default_isr(&timer[3]);  
}
```

```
void rutina_ethernetrx(void)
Rutina de atención a la interrupción de recepción de paquetes ethernet. Se utiliza en la mejora de servidor TFTP.
-----
void rutina_ethernetrx(void){
    fec_handler(&fec_nif);
}
```

```
void rutina_etherneTx(void)
Rutina de atención a la interrupción de transmisión de paquetes ethernet. Se utiliza en la mejora de servidor TFTP.
```

```
void rutina_etherneTx(void) {}
```



2jugadores.h

dosjugadores.h

Definición de cabeceras de funciones.

```
void dosJugadores(BOOL *pHayGanador, char *pTurno, char *pTablero, TipoPunt3Raya tipo, int *pPuntosO, int *pPuntosX, int *pMovimientosO, int *pMovimientosX);

void finalizarPartida(char *pTablero, char *pTurno, TipoPunt3Raya tipo, int *pPuntosO, int *pPuntosX);

void sumaPuntos(TipoPunt3Raya tipo, int *pPuntosO, int *pPuntosX, char *pTurno, char *pTablero, int *pMovimientosO, int *pMovimientosX, BOOL *pHayGanador);

void decrementaMovimientos (char *pTurno, int *pMovimientosO, int *pMovimientosX);

void sumaPuntosPosibleVictoria(char *pTablero, char *pTurno, int *pPuntosO, int *pPuntosX, BOOL *pHayGanador);

void incrementaPuntos (char *pTurno, int *pPuntosO, int *pPuntosX, int puntos);

void muestraPuntos(int *pPuntosO, int *pPuntosX, TipoPunt3Raya tipo);

void dosJugadoresTitulo(void);

void tablaDePuntos(int *pPuntosO, int *pPuntosX);

char determinaMaximaPuntuacion(int *pPuntosO, int *pPuntosX);
```



2jugadores.c

```
dosjugadores.c
```

```
#include "dosjugadores.h"
```

```
void dosJugadores(BOOL *pHayGanador, char *pTurno, char *pTablero)

Gestión del modo de juego para dos jugadores.

-----
void dosJugadores(BOOL *pHayGanador, char *pTurno, char *pTablero, TipoPunt3Raya tipo, int *pPuntosO, int *pPuntosX, int
*pMovimientosO, int *pMovimientosX) {

    dosJugadoresTitulo();                                // Se muestra el título del modo 3 en raya por pantalla
    limpiaTablero(pTurno, pTablero);                     // Inicialización de los parámetros de una partida

    while ( !(*pHayGanador) ) { // Comprobación de si algún jugador ha ganado o si alguno se ha quedado sin tiempo, en tal caso.

        if ( tipo == MOVIMIENTOS && (*pMovimientosO == 0 || *pMovimientosX == 0) ) break;
        if ( tipo == TIEMPO_GLOBAL && timeout ) break;

        preparaTurno(pTurno, pTablero, tipo, pMovimientosO, pMovimientosX);      // Preparación del próximo turno
        movimientoHumano(pTurno, pTablero);                                         // Se realiza el movimiento de un jugador
        sleep(0);                                                               // Se pone a cero el tiempo restante
        hayTresEnRaya(pHayGanador, pTablero);                                       // Comprobación de si existe un tres en raya.

        sumaPuntos(tipo, pPuntosO, pPuntosX, pTurno, pTablero, pMovimientosO, pMovimientosX, pHayGanador);
    }
    finalizarPartida(pTablero, pTurno, tipo, pPuntosO, pPuntosX);
}
```

```
void finalizarPartida (char *pTablero, char *pTurno, TipoPunt3EnRaya tipo, int *pPuntosO, int *pPuntosX

Función llamada cuando una partida se termina. Se encarga de reproducir la melodía correspondiente, según como haya terminado el
juego. También muestra los puntos finales de cada jugador (según puntuación escogida).
-----
```

```
void finalizarPartida(char *pTablero, char *pTurno, TipoPunt3Raya tipo, int *pPuntosO, int *pPuntosX) {

    cambioMelodia = TRUE;
    muestraTablero(pTablero);                                     // Se muestra el tablero final

    switch (tipo) {

        case MOVIMIENTOS:
            estadoJuego = 4;                                         // Seteo del estado del sistema
            output("\n;Un jugador ha agotado su número de movimientos!"); // Se imprime por pantalla un mensaje final
            espera();                                                 // Se muestra las puntuaciones finales
            muestraPuntos(pPuntosO, pPuntosX, tipo);
            break;

        case TIEMPO_GLOBAL:
            estadoJuego = 4;                                         // Seteo del estado del sistema
            output("\n;Se ha acabado el tiempo de juego!");          // Se imprime por pantalla un mensaje final
            espera();                                                 // Se muestran las puntuaciones finales
            muestraPuntos(pPuntosO, pPuntosX, tipo);
            break;

        default:
            estadoJuego = 3;                                         // Seteo del estado del sistema
            mensajeFinJuego(pTurno);                                // Se imprime por pantalla un mensaje final
            break;
    }
    espera();
}
```

```
void sumaPuntos (TipoPunt3Raya tipo, int *pPuntosO, int *pPuntosX, char *pTurno, char *pTablero, int *pMovimientosO, int
*pMovimientosX, BOOL *pHayGanador)

Suma los puntos de cada jugador en la forma escogida, dependiendo de la opción que se haya escogido.
-----

void sumaPuntos(TipoPunt3Raya tipo, int *pPuntosO, int *pPuntosX, char *pTurno, char *pTablero, int *pMovimientosO, int
*pMovimientosX, BOOL *pHayGanador) {

    switch (tipo) {          //Se identifica que tipo de puntuación de está usando

        case MOVIMIENTOS:

            sumaPuntosPossibleVictoria(pTablero, pTurno, pPuntosO, pPuntosX, pHayGanador); //Se suman los puntos
            //Si algún jugador no ha movido no se le decrementa su cuenta de movimientos
            if (!turnoPerdido) decrementaMovimientos(pTurno, pMovimientosO, pMovimientosX);
            break;

        case TIEMPO_GLOBAL:

            sumaPuntosPossibleVictoria(pTablero, pTurno, pPuntosO, pPuntosX, pHayGanador); //Se suman los puntos
            break;

        default:                                // Ningún tipo de puntuación
            break;
    }
}
```

```
void decrementaMovimientos (char *pTurno, int *pMovimientosO, int *pMovimientosX)

Función que se encarga de decrementar en una unidad la cuenta de movimientos del jugador que haya movido.
-----

void decrementaMovimientos (char *pTurno, int *pMovimientosO, int *pMovimientosX) {

    switch (*pTurno) {                                //Se comprueba el turno actual

        case 'O':
            *pMovimientosO -= 1;                      //Se le decrementan los movimientos a O
            break;

        case 'X':
            *pMovimientosX -= 1;                      //Se le decrementan los movimientos a X
            break;
        default:
            break;
    }
}
```

```
void sumaPuntosPosibleVictoria (char *pTablero, char *pTurno, int *pPuntosO, int *pPuntosX, BOOL *pHayGanador)
```

Función que suma puntos al jugador que haya creado una situación de peligro. Estas se identifican comprobando que haya dos fichas de un jugador y un espacio en blanco.

```
void sumaPuntosPosibleVictoria(char *pTablero, char *pTurno, int *pPuntosO, int *pPuntosX, BOOL *pHayGanador) {
```

```
    BYTE casillaVictoria = 0; //Variable que almacena la posición de la casilla que puede dar el tres en raya  
    BYTE buffer = 0; //Variable que almacena la posición a partir de la cual hay que seguir buscando situaciones de peligro
```

```
    while (buffer < NUM_CASILLAS) { //Condición para no iterar más allá de las posiciones de memoria correspondientes al tablero
```

```
        //Variable que almacena la posición que puede dar un posible 3 en raya (-1 si no existe tal posición)  
        casillaVictoria = posibleTresEnRaya(CASILLA_VACIA, *pTurno, buffer, pHayGanador, pTablero);
```

```
        //Ya se ha encontrado una casilla ganadora, se buscan más
```

```
        if (casillaVictoria > -1) incrementaPuntos(pTurno, pPuntosO, pPuntosX, 1);
```

```
        else break; //Si no se ha encontrado ninguna posición ganadora salimos
```

```
        //Cuando se ha encontrado una posición ganadora, se itera para buscar otra a partir de la siguiente posición  
        buffer = casillaVictoria+1;
```

```
}
```

```
return;
```

```
}
```

```
void incrementaPuntos (char *pTurno, int *pPuntosO, int *pPuntosX, int puntos)
Incrementa los puntos al jugador correspondiente, según el turno actual.
-----
void incrementaPuntos (char *pTurno, int *pPuntosO, int *pPuntosX, int puntos) {
    switch(*pTurno) {
        case 'O':
            *pPuntosO += puntos; //Se suman los puntos obtenidos al jugador correspondiente
            break;
        case 'X':
            *pPuntosX += puntos;
            break;
        default:
            break;
    }
}
```

```
void muestraPuntos (int *pPuntosO, int *pPuntosX, TipoPunt3EnRaya tipo).  
-----  
void muestraPuntos(int *pPuntosO, int *pPuntosX, TipoPunt3Raya tipo){  
    char mensaje[] = {"Puntuación final\n"}; // Mensaje de fin de juego  
    output("\n");  
    switch(tipo) {  
        case MOVIMIENTOS:           // Para ambos tipos de puntuación, mismo esquema para mostrar los puntos  
        case TIEMPO_GLOBAL:  
            formateaCadena(mensaje); // Se muestra el mensaje de puntuación  
            tablaDePuntos(pPuntosO, pPuntosX); // Se llama a la función que muestra la tabla de puntuación  
            break;  
        default:  
            break;  
    }  
}
```

```
void tablaDePuntos (int *pPuntosO, int *pPuntosX)

Muestra en el formato adecuado los puntos de cada jugador y anuncia el ganador.
-----

void tablaDePuntos(int *pPuntosO, int *pPuntosX) {

    char jugadorGanador;           // Variable que almacena el Jugador ganador

    output("\[Jugador O]: ");
    outNum(10, *pPuntosO, SIN_SIGNO); // Mostrar puntos de O
    output(" puntos\n");
    espera();                      // Retardo en la aparición de los puntos (1s)
    output("\[Jugador X]: ");
    outNum(10, *pPuntosX, SIN_SIGNO); // Mostrar puntos de X
    output(" puntos\n");
    espera();                      // Espera (1s)

    // Llamada a la función que determina quién ha obtenido la máxima puntuación. En caso de empate se devuelve 'E'
    jugadorGanador = determinaMaximaPuntuacion(pPuntosO, pPuntosX);

    switch (jugadorGanador)          // Según el jugador que haya ganado se muestra el mensaje correspondiente

        case 'O':
            output("\n;Ha ganado el jugador O!");
            break;
        case 'X':
            output("\n;Ha ganado el jugador X!");
            break;
        default:
            output("\n;Ha habido un empate!");
            break;
    }
    output("\n");
}
```

```
char determinaMaximaPuntuación (int *pPuntoso, int *pPuntosX)

Función que determina qué jugador ha obtenido la máxima puntuación. También detecta si ha habido empate.
```

```
-----  
char determinaMaximaPuntuacion(int *pPuntoso, int *pPuntosX) {  
  
    if (*pPuntoso == *pPuntosX) {  
        return 'E';  
    }  
    if (*pPuntoso > *pPuntosX) {  
        return 'O';  
    } else {  
        return 'X';  
    }  
}
```

```
void dosJugadoresTitulo(void)  
Muestra en ASCII-ART el modo de juego del 3 en raya para dos jugadores.  
-----  
  
void dosJugadoresTitulo(void) {  
  
    output("\n      \\" );           " ) ;           // Definición de los caracteres ASCII  
    output("\n|  \\" );           " ) ;  
    output("\n ) |" );           " ) ;  
    output("\n / /" );           " ) ;  
    output("\n / /" );           " ) ;  
    output("\n|____|" );           " ) ;  
    output("\n      /" );           " ) ;  
    output("\n      /" );           "\n" );  
}
```



contramaquina.h

```
contramaquina.h
```

```
Definición de cabeceras de funciones.
```

```
void contraMaquina(BOOL *pHayGanador, char *pTurno, char *pTablero, TipoPunt3Raya tipo, int *pMovimientosO, int *pMovimientosX);
void movimientoMaquina(BOOL *pHayGanador, char *pTurno, char *pTablero);
void mueveMaquina(BYTE posicion, BOOL *pHayGanador, char *pTurno, char *pTablero);
BYTE posibleTresEnRaya(char ficha_previa, char ficha_post, BYTE indice, BOOL *pHayGanador, char *pTablero);
BYTE detectaEsquinasLibres(char *pTablero);
BYTE posicionAleatoria(char elemento, BYTE indice, char *pTablero);
void jugadaFinalMaquina(BYTE posicion_ganadora, BOOL *pHayGanador, char *pTurno, char *pTablero);
void melodíaFinal(BOOL *pHayGanador, char *pTurno);
void contraMaquinaTitulo(void);
```



contramaquina.c

```
contramaquina.c
```

```
#include "contramaquina.h"
```

```
void contraMaquina(BOOL *pHayGanador, char *pTurno, char *pTablero, TipoPunt3Raya tipo, int *pMovimientosO, int *pMovimientosX)
* Gestión del modo contra la máquina.
-----
void contraMaquina(BOOL *pHayGanador, char *pTurno, char *pTablero, TipoPunt3Raya tipo, int *pMovimientosO, int *pMovimientosX) {
    void contraMaquinaTitulo();                                // Título del modo contraMaquina
    limpiaTablero(pTurno, pTablero);                          // Inicialización de tablero
    while (!(*pHayGanador)) {                                 // Comprobación de si algún jugador ha ganado
        // Inicialización de variables involucradas en el turno
        preparaTurno(pTurno, pTablero, tipo, pMovimientosO, pMovimientosX);

        // Se realiza un movimiento dependiendo del turno actual
        switch (*pTurno) {
            case 'O':
                movimientoMaquina(pHayGanador, pTurno, pTablero); // La máquina mueve
                break;
            case 'X':
                movimientoHumano(pTurno, pTablero);                 // El Jugador mueve
                break;
            default:
                break;
        }
        sleep(0);                                              // Se pone a cero el tiempo restante
        hayTresEnRaya(pHayGanador, pTablero);                  // Comprobación de si existe un tres en raya.
    }
    melodíaFinal(pHayGanador, pTurno);                      // Reproducción de melodía final (el jugador es ganador o perdedor)
    muestraTablero(pTablero);                                // Se muestra el tablero final
    mensajeFinJuego(pTurno);                                // Mensaje por pantalla que muestre el ganador
}
```

```
void movimientoMaquina(BOOL *pHayGanador, char *pTurno, char *pTablero)

Evalúa los tres tipos posibles de movimiento que puede realizar la máquina.
-----

void movimientoMaquina(BOOL *pHayGanador, char *pTurno, char *pTablero) {

    // Lugar donde la ficha situará su ficha
    BYTE posicion;
    // Flag de que la máquina puede ganar
    BOOL victoria = FALSE;

    // Caso 1: Se comprueba que la máquina pueda realizar un 3 en raya
    // Para ello se hacen supuestos rellenando espacios con fichas de la máquina.
    posicion = posibleTresEnRaya(0x20,'O',0, pHayGanador, pTablero);

    if (posicion == -1) {

        // Caso 2: Se evita que el otro jugador realice un 3 en raya
        // Para ello se hacen supuestos rellenando espacios con fichas del otro jugador
        posicion = posibleTresEnRaya(0x20,'X',0, pHayGanador, pTablero);

        if (posicion == -1) {

            // Caso 3: Si es posible se realiza nu movimiento a una esquina
            posicion = detectaEsquinasLibres(pTablero);

            if (posicion == -1) {
                // Caso 4: Si no se da el caso 1 ni el caso 2, la máquina escoge una posición aleatoria
                posicion = posicionAleatoria(0x20,0, pTablero);
            }
        }
    } else {
        victoria = TRUE; // Si se encuentra una posición ganadora
    }

    // Se comprueba si la máquina tiene todas sus fichas en el tablero
    if (todasLasFichas(pTurno, pTablero)) {

        if (victoria) {
            // Si la máquina puede ganar, realiza la jugada ganadora
            jugadaFinalMaquina(posicion, pHayGanador, pTurno, pTablero);
        } else {
            mueveMaquina(posicion, pHayGanador, pTurno, pTablero); //La máquina ha de cambiar una ficha al lugar escogido
        }
    }
}
```

```
        }
    } else {
        escribeTablero(posicion+'1', pTurno, pTablero);           // Sitúa ficha en una posición del tablero
    }
}
```

```
void mueveMaquina(BYTE posicion, BOOL *pHayGanador, char *pTurno, char *pTablero) *
Dada una posición, se evalúa qué ficha tiene que mover la máquina.
-----
void mueveMaquina(BYTE posicion, BOOL *pHayGanador, char *pTurno, char *pTablero) {

    // Posición de la ficha que vamos a mover
    BYTE posicion_vieja;
    BYTE contador = FICHAS_MAX;
    // Se obtiene la posición de la primera ficha de la máquina
    posicion_vieja = posicionAleatoria('O',0,pTablero);

    // Se comprueba que moviendo esa ficha no permitimos al jugador que haga 3 en raya
    while ( posicion_vieja == posibleTresEnRaya('O','X',posicion_vieja, pHayGanador, pTablero) ) {

        if (contador == 0) break;
        // Obtenemos la posición de la siguiente ficha de la máquina
        posicion_vieja = posicionAleatoria('O',posicion_vieja+1, pTablero);
        contador--;
    }
    // Se mueve la ficha de la máquina (pasándole las posiciones como char
    mueveFichaOrigenDestino(posicion_vieja+'1',posicion+'1', pTurno, pTablero);
}
```

```
BYTE posibleTresEnRaya(char ficha_previa, char ficha_post, BYTE indice, BOOL *pHayGanador, char *pTablero)
```

Función que comprueba si hay tres en raya sustituyendo una ficha (o espacio en blanco) por otra ficha.

```
BYTE posibleTresEnRaya(char ficha_previa, char ficha_post, BYTE indice,
    BOOL *pHayGanador, char *pTablero) {

    // Se itera el tablero para realizar los supuestos empezando según indique "indice"
    for ( ; indice < NUM_CASILLAS; indice++) {

        // Se localiza la ficha a sustituir
        if (*pTablero+indice) == ficha_previa {
            *(pTablero+indice) = ficha_post;           // Se sustituye por la ficha del supuesto
            hayTresEnRaya(pHayGanador, pTablero);      // Comprobación de 3 en raya
            *(pTablero+indice) = ficha_previa;         // Se restablece el estado anterior

            if (*pHayGanador) {
                *pHayGanador = FALSE;                  // Se restablece el flag
                return indice;
            }
        }
    }
    return -1;           // Si no se encuentra ninguna posición ganadora, devolvemos -1
}
```

```
BYTE detectaEsquinasLibre (char *pTablero)
Detecta si las esquinas del tablero están libres, para colocar una ficha en alguna de ellas.
-----
BYTE detectaEsquinasLibres(char *pTablero) {
    BYTE i;
    for (i = 0; i < NUM_CASILLAS; i+=2) { // Se recorren las posiciones pares del tablero
        if (i == FICHA_CENTRAL) break; // Obviando la ficha central
        if (*pTablero+i) == CASILLA_VACIA) return i; //Si encontramos alguna esquina vacía, devolvemos su posición
    }
    return -1; // Si no hay esquinas vacías
}
```

```
BYTE posicionAleatoria (char elemento, BYTE indice, char *pTablero)
Devuelve la posición de un espacio en blanco de manera aleatoria.
-----
BYTE posicionAleatoria(char elemento, BYTE indice, char *pTablero) {
    for ( ; indice < NUM_CASILLAS; indice++) {           // Iteración del tablero empezando por el valor "indice"
        if (* (pTablero+indice) == elemento) {           // Devuelve la posición del primer elemento encontrado
            return indice;                             // Esta función no es estrictamente aleatoria
        }
    }
    return -1;
}
```

```
void jugadaFinalMaquina(BYTE posicion_ganadora, BOOL *pHayGanador, char *pTurno, char *pTablero)

Realiza la jugada ganadora de la máquina.
-----

void jugadaFinalMaquina(BYTE posicion_ganadora, BOOL *pHayGanador,
                        char *pTurno, char *pTablero) {

    BYTE indice;
    *(pTablero+posicion_ganadora) = *pTurno;                                // Se sitúa una ficha de la máquina en la posición ganadora
    for (indice = 0; indice < NUM_CASILLAS; indice++) {
        if (* (pTablero+indice) == 'O') {                                       // Para determinar qué ficha ha de mover la máquina para
            *(pTablero+indice) = 0x20;                                            // realizar el 3 en raya, se van levantando las fichas
            hayTresEnRaya(pHayGanador, pTablero);                                // de la máquina y comprobando si seguiría ganando
            if (*pHayGanador) {
                return;
            } else {
                // Si la ficha que se levanta no es la adecuada, se sigue iterando en busca de la adecuada
                *(pTablero+indice) = 'O';
            }
        }
    }
    return;
}
```

```
void melodíaFinal(BOOL *pHayGanador, char *pTurno)

Reproducción de la melodía final en función de que el jugador gane o pierda.
-----
void melodíaFinal(BOOL *pHayGanador, char *pTurno) {

    cambioMelodía = TRUE;                                // Si ha acabado el juego, forzamos el cambio a la melodía final
    if (*pHayGanador && *pTurno == 'X') {                // Gana el jugador
        estadoJuego = 3;
    } else {                                              // Gana la máquina (melodía de perdedor)
        estadoJuego = 4;
    }
}
```




haytresenraya.h

```
haytresenraya.h
```

```
Definición de cabeceras de funciones.
```

```
void hayTresEnRaya(BOOL *pHayGanador, char *pTablero);  
BOOL tresEnRayaFila(char *pTablero);  
BOOL tresEnRayaColumna(char *pTablero);  
BOOL tresEnRayaDiagonal(char *pTablero);
```



haytresenraya . c

```
haytresenraya.c
```

```
#include "haytresenraya.h"
```

```
void hayTresEnRaya(BOOL *pHayGanador, char *pTablero)
Función principal que setea la variable.
hayGanador en función de si hay 3 en raya.
-----
void hayTresEnRaya(BOOL *pHayGanador, char *pTablero) {
    // Hay ganador si alguno de los dos jugadores consigue alguna de las tres modalidades de tres en raya
    *pHayGanador = (tresEnRayaFila(pTablero) || tresEnRayaColumna(pTablero) || tresEnRayaDiagonal(pTablero));
}
```

```
BOOL tresEnRayaFila(char *pTablero)

Comprueba si hay alguna fila con 3 en raya.
-----

BOOL tresEnRayaFila(char *pTablero) {

    BYTE i;
    char simbolo_ref; // Variable para almacenar el símbolo de referencia

    for (i = 0; i < 7; i+=3) { // Recorremos las filas (cuyo principio está en las posiciones 0, 3 y 6 del array)
        BYTE indice = i;
        simbolo_ref = *(pTablero+i);           // Fijamos como elemento de referencia el primero de cada fila

        for ( ; indice < 3 + i; indice++) { // Recorremos, para cada fila las dos columnas restantes (0,1,2 ; 3,4,5 ; 6,7,8)

            // Si alguna de las restante posiciones no es el símbolo de referencia, en esa fila no puede haber un 3 en raya
            if (*(pTablero+indice) == 0x20 || *(pTablero+indice) != simbolo_ref) {
                break;
            }
            // Si hemos llegado al final de la fila sin entrar en la condición anterior, hay un 3 en raya
            if (indice == 2+i ) {
                return TRUE;
            }
        }
    }
    return FALSE;      // Si no encontramos nada
}
```

```
BOOL tresEnRayaColumna(char *pTablero)
Comprueba si hay alguna columna con 3 en raya.
-----
BOOL tresEnRayaColumna(char *pTablero) {
    BYTE i;
    char simbolo_ref; // Variable para almacenar el símbolo de referencia

    for (i = 0; i < 3; i++) { // Recorremos las columnas (cuyo principio está en las posiciones 0, 1 y 2 del array)
        BYTE indice = i;
        simbolo_ref = *(pTablero + i); // Fijamos como elemento de referencia el primero de cada columna

        // Recorremos, para cada columna las dos filas restantes (0,3,6 ; 1,4,7 ; 2,5,8)
        for ( ; indice < 7 + i; indice += 3) {

            // Si alguna de las restante posiciones no es el símbolo de referencia, en esa fila no puede haber un 3 en raya
            if (*(pTablero + indice) == 0x20 || *(pTablero + indice) != simbolo_ref) {
                break;
            }
            // Si hemos llegado al final de la columna sin entrar en la condición anterior, hay un 3 en raya
            if (indice == 6 + i) {
                return TRUE;
            }
        }
    }
    return FALSE; // Si no encontramos nada
}
```

```
BOOL tresEnRayaDiagonal(char *pTablero)
Comprueba si hay alguna diagonal con 3 en raya.
-----
BOOL tresEnRayaDiagonal(char *pTablero) {
    char simbolo_ref = *(pTablero+4); // El simbolo de referencia es la ficha central
    // Se compara el simbolo de referencia con ambos extremos de la diagonal principal
    if (*pTablero == simbolo_ref && *(pTablero+8) == simbolo_ref) {
        return TRUE;
    }
    // Se compara el simbolo de referencia con ambos extremos de la diagonal secundaria
    if (*(pTablero+2) == simbolo_ref && *(pTablero+6) == simbolo_ref) {
        return TRUE;
    }
    return FALSE; // Si no encontramos 3 en raya
}
```



interfaz.h

```
interfaz.h
-----
Definición de alias y constantes.
-----
#define NUM_FILAS 4           // Definición del número de filas + 1
#define NUM_COLS 4            // Definición del número de comlumnas + 1
#define EXCIT 1                // Definición de la rutina teclado

Definición de cabeceras de funciones
-----
char menuPrincipal();
char teclado(void);
void mensajeMueveFicha();
void mensajeFinJuego(char *pTurno);
void tictactoeTitulo();
void borraPantalla(BYTE numCaracteres);
void formateaCadena(char* cadena);
void menuJuego(char *pTurno, TipoPunt3Raya tipo, int *pMovimientosO, int *pMovimientosX);
void cuentaAtras(void);
```



interfaz.c

```
interfaz.c
```

```
#include "interfaz.h"
#include "fichas.h"
#include "temporizador.h"
```

```
char menuPresentar()

Se muestra el menú principal del juego y se solicita al usuario la pulsación de tecla.
-----

char menuPrincipal()
{
    char tecla;
    output("\n*****\n");
    output("*      Menú Principal      *\n");
    output("*****\n");
    output("* A Juego contra la máquina *\n");
    output("* B Juego multijugador     *\n");
    output("* C Rally                  *\n");
    output("* D Sudoku                 *\n");
    output("* E Ayuda                  *\n");
    output("* F Salir                  *\n");
    output("*****\n");
    tecla = teclado()          // Se pide una tecla para elegir una opción
    output("\nOpción elegida: ")   // Se muestra dicha opción
    outch(tecla);
    output("\n");
    return tecla;
}
```

```
char teclado(void)

Explora el teclado matricial y devuelve la tecla pulsada.
-----

char teclado(void)
{
    BYTE fila, columna, fila_mask;
    char teclas[4][4] = {{"123C"}, {"456D"}, {"789E"}, {"AOBF"}};
    // Bucle de exploración del teclado
    while(TRUE) {

        if (timeout) {           // Si se ha acabado el tiempo de partida devolvemos un indicador
            // para avisar a las funciones superiores
            return 'G';
        }
        /*if(finRefresco) {
            return '5';
        }*/
        if(turnoPerdido) {       // Si se ha perdido el turno, se interrumpe la búsqueda de
            // pulsación y se devuelve un indicador
            output("\b0");
            return 'G';
        }
        if(pintoTiempo) { // Si se permite pintar el tiempo, se pinta
            cuentaAtras();
        }

        // Excitamos una columna
        for(columna = NUM_COLS - 1; columna >= 0; columna--) {
            set_puertoS(EXCIT << columna);           // Se envía la excitación de columna
            retardo(1150);                            // Esperamos respuesta de optoacopladores

            // Exploramos las filas en busca de respuesta
            for(fila = NUM_COLS - 1; fila >= 0; fila--) {
                fila_mask = EXCIT << fila;           // Máscara para leer el bit de la fila actual
                if(lee_puertoE() & fila_mask){          // Si encuentra tecla pulsada,
                    while(lee_puertoE() & fila_mask); // Esperamos a que se suelte
                    retardo(1150);                  // Retardo antirrebotes
                    return teclas[fila][columna];      // Devolvemos la tecla pulsada
                }
            }
        }
    }
}
```

```
        }
        // Siguiente columna
    }
    // Exploración finalizada sin encontrar una tecla pulsada
}
// Reiniciamos exploración
}
```

```
void menuJuego (char *pTurno, TipoPunt3EnRaya tipo, int *pMovimientosO, int *pMovimientosX)

Muestra las opciones del jugador durante su turno.
-----

void menuJuego(char *pTurno, TipoPunt3Raya tipo, int *pMovimientosO, int *pMovimientosX)
{
    output("*****\n");
    output("* Turno del jugador: ");
    outch(*pTurno);
    output(" *\n");
    output("*****\n");
    output("* 1-9  Situar ficha *\n");
    output("*****\n");

    if (tipo == MOVIMIENTOS) {

        output("\n Te quedan "); // Se muestran los movimientos restantes para cada jugador (si dicho modo de
                               // puntuación está activo)
        switch (*pTurno) {

            case 'O':
                outNum(10, *pMovimientosO, SIN_SIGNO);
                break;
            case 'X':
                outNum(10, *pMovimientosX, SIN_SIGNO);
                break;
        }
        output(" movimientos\n");
    }
}
```

```
void mensajeMueveFicha()  
Muestra un mensaje de que el jugador ha situado todas sus fichas sobre el tablero.  
-----  
  
void mensajeMueveFicha()  
{  
    output("*****\n");  
    output("*   Has puesto todas tus fichas.      *\n");  
    output("*   Selecciona la ficha que deseas mover *\n");  
    output("*****\n");  
}
```

```
void mensajeFinJuego(char *pTurno)
Muestra el mensaje de 3 en raya y el jugador que ha ganado.
-----
void mensajeFinJuego(char *pTurno)
{
    char tecla;
    output("*****\n");
    output("*           ;3 en raya!\n");
    output("*           Ha ganado el jugador: ");
    outch(*pTurno);
    output("\n");
    output("*****\n");
    output("Pulsa cualquier tecla para continuar...\n");
    tecla = teclado();
}
```



```
void formateaCadena(char* cadena)
Escribe una frase con reborde de asteriscos.
-----
void formateaCadena(char* cadena)
{
    BYTE i = 0;
    output("\n**");
    // Se escribe un salto de línea y esquina de asteriscos
    do {
        output("*");
        // Se escriben asteriscos (línea superior) conforme a la longitud de la frase
        i++;
    } while(cadena[i] != '\n');
    // Condición de dejar de pintar asteriscos: la frase tiene un retorno de carro

    output("**\n* ");
    // Se pinta asteriscos del lateral izquierdo
    i = 0;
    do {
        outch(cadena[i]);
        // Se escribe la cadena sin alterarla
        i++;
    } while(cadena[i] != '\n');
    // Condición de final de cadena: la frase contiene un retorno de carro
    output(" *\n");
    i = 0;
    output("**");
    // Se pintan asteriscos del lateral derecho
    do {
        output("*");
        // Se escriben asteriscos (línea inferior) conforme a la longitud de la frase
        i++;
    } while(cadena[i] != '\n');
    // Condición de dejar de pintar asteriscos: la frase tiene un retorno de carro.
    output("**\n");
    // Esquina inferior derecha de asteriscos
    return;
}
```

```
void mensajeTurno(char *pTurno, char *pTablero)
Se escriben mensajes de tiempo y movimiento de ficha para cada turno.
-----
void mensajeTurno(char *pTurno, char *pTablero) {
    if (todasLasFichas(pTurno, pTablero)) {
        mensajeMueveFicha();
    }
    output("Tiempo restante: ");
    sleep(TIEMPO_TURNO*1000); // Se setea el tiempo de que dispone el jugador en un turno
}
```

```
void cuentaAtras(void)

Escribe el tiempo restante del turno.
-----
void cuentaAtras(void) {

    if (tiempo == TIEMPO_TURNO) {
        outNum(10,tiempo,0);           // Se muestra el tiempo de turno inicial
    } else {
        outch(0x08);
        if (tiempo > 8) { // Se borra un caracter más cuando se pasa
                            // de decenas a unidades.
            outch(0x08);
        }
        outNum(10,tiempo,0);           // Se muestra el timempo
    }
    pintoTiempo = FALSE;             // Se resetea la variable pintoTiempo
}
```



tablero.h

```
tablero.h
```

```
Definición de cabeceras de funciones.
```

```
void muestraTablero(char *pTablero);  
void escribeTablero(char posicion, char *pTurno, char *pTablero);  
void limpiaTablero(char *pTurno, char *pTablero);  
char getFicha(char tecla, char *pTablero);
```



tablero.c

```
tablero.c
```

```
#include "tablero.h"
```

```
void muestraTablero(char *pTablero)

Muestra la situación actual del tablero en forma de cuadrícula 3 x 3.
-----

void muestraTablero(char *pTablero)
{
    BYTE indice;
    output("\nTablero:\n");
    for (indice = 0; indice < NUM_CASILLAS; indice++) { // Se recorre el tablero
        if (indice % (NUM_CASILLAS/3) == 0) {
            output("\n ----- \n"); // Entre filas se coloca una línea horizontal
            output(" | ");
        }
        outch(* (pTablero+indice)); // Se muestra el carácter con un separador
        output(" | ");
    }
    output("\n ----- \n");
}
```

```
void escribeTablero(char posición, char *pTurno, char *pTablero)
Escribe la ficha del turno actual en una posición del tablero.
-----
void escribeTablero(char posición, char *pTurno, char *pTablero)
{
    if (getFicha(posición, pTablero) == 0x20) {
        // Si la posición en la se quiere escribir esta vacía, se escribe directamente
        *(pTablero+(posición-'1')) = *pTurno;
        return;
    }
    if (!turnoPerdido) {
        if (getFicha(posición, pTablero) == 0) {      // Se está intentando escribir fuera del tablero
            char mensaje[] = {"Selecciona una posición válida del tablero\n"};
            formateaCadena(mensaje);
            output("Tiempo restante:    ");           // Se muestra el tiempo restante
        } else {                                     // Se está intentando escribir en una posición no vacía
            char mensaje[] = {"Por favor, selecciona una posición libre\n"};
            formateaCadena(mensaje);
            output("Tiempo restante:    ");           // Se muestra el tiempo restante
        }
        escribeTablero(teclado(), pTurno, pTablero); // Si se quiere escribir en una posición incorrecta
                                                    // (las anteriores), se sigue pidiendo nueva posición
    }
    return;
}
```

```
void limpiaTablero(char *pTurno, char *pTablero)

Resetea el tablero poniendo todas sus posiciones con espacio en blanco y la ficha X en la posición central.
-----
void limpiaTablero(char *pTurno, char *pTablero)
{
    BYTE indice;
    *pTurno = 'X';
    for (indice = 0; indice < NUM_CASILLAS; indice++) {           // Se recorre el tablero
        if (indice == FICHA_CENTRAL) {
            *(pTablero+indice) = *pTurno;                            // Se rellena la posición central con una X
            // (el turno está inicializado a X)
        }
        else {
            *(pTablero+indice) = 0x20;                                // El resto se rellena con espacios en blanco
        }
    }
}
```

```
char getFicha(char tecla, char *pTablero)
Dada una posición del tablero, devuelve la ficha que haya colocada.
-----
char getFicha(char tecla, char *pTablero) {
    return *(pTablero+(tecla-'1')); // Se devuelve el char existente en una posición dada (como char)
                                    // en el tablero
}
```



temporizador.h

```
temporizador.h
```

```
Definición de variables de ámbito global
```

```
void sleep(ULONG milisegundos);  
void notaMusical(int frecuencia, int duracion);  
void setTIMER0(int frecuencia);  
void desactivaTIMER0();  
void reproduceNota(struct melodía Musica);
```



temporizador.c

```
temporizador.c
```

```
#include "temporizador.h"
```

```
void sleep(ULONG milisegundos)
Setea la variable cont_retardo, que representa el tiempo restante del turno actual.
-----
void sleep(ULONG milisegundos) {
    cont_retardo = milisegundos;           // Inicializa el contador
    return;
}
```

```
void seteaRefresco(ULONG milisegundos)
Setea la variable refresco, que representa el tiempo que tarda en aparecer un nuevo tramo del circuito en el modo Rally.
-----
void seteaRefresco(ULONG milisegundos) {
    refresco = milisegundos; // Inicializa el contador
    return;
}
```

```
void notaMusical(int frecuencia, int duracion)
    Inicializa el contador que controla la duración de una nota y establece la frecuencia de la onda cuadrada generada por el TIMER0
-----
void notaMusical(int frecuencia, int duracion) {
    duracion_nota = duracion; // Se seta la variable global duración con la duración de la nota a reproducir
    setTIMER0(frecuencia); // Se configura el TIMER0 con la frecuencia de la nota a reproducir
    return;
}
```

```
void setTIMER0(int frecuencia)

Configura los registros del TIMER0 para generar una onda cuadrada de una determinada frecuencia.
-----

void setTIMER0(int frecuencia) {
    // Comprobamos el caso en que haya silencios en la melodía
    if (frecuencia == 0) {
        mbar_writeShort(MCFSIM_TMR0, 0x4F2C);                      // Si la frecuencia es 0, inhabilitamos el timer
    } else {
        LONG trr0 = MCF_CLK/(frecuencia*0x4F*2*16);              // Calculamos el valor de TRR0 apropiado para la frecuencia
                                                                // que queremos reproducir
        ACCESO_A_MEMORIA_LONG(DIR_VTMR0)= (ULONG)_prep_TOUT0;      // Escribimos la dirección de la función para TMR0
        mbar_writeShort(MCFSIM_TMR0, 0x4F2D);                      // TMR0: PS=0x50-1 CE=00 OM=1 ORI=0 FRR=1 CLK=10 RST=1
        mbar_writeShort(MCFSIM_TCNO, 0x0000);                      // Ponemos a 0 el contador del TIMER0
        mbar_writeShort(MCFSIM_TRR0, trr0);                        // Fijamos la cuenta final del contador
        mbar_writeLong(MCFSIM_ICR1, 0x8888EF88);                  // Fijamos el nivel de interrupción de los timers
    }
}
```

```
desactivaTIMER0 ()  
  
Para la generación de onda cuadrada y pone  
-----  
  
void desactivaTIMER0 () {  
    mbar_writeShort(MCF5SIM_TMR0, 0x4F2C);           // Inhabilitamos el TIMERO (bit de RST a 0)  
    duracion_nota = 0;                                // Para que el sistema no reproduzca más notas.  
                                                    // De ser necesario, se reproducirá otra melodía.  
}
```

```
reproduceNota(struct melodía Musica)

Setea la frecuencia y duración de una determinada nota musical.
-----

void reproduceNota(struct melodía Musica) {
    if (cambioMelodia) {
        indice = 0;                      // Si hay que cambiar la melodía se resetea el índice que recorre el array de
        // duraciones de notas de una melodía
        cambioMelodia = FALSE;           // Se resetea el flag cambioMelodia
    }

    if (Musica.duracion[indice] != 0) { // Se comprueba que, para la melodía actual, haya siguiente nota a reproducir
        notaMusical(Musica.frecuencia[indice],Musica.duracion[indice]); // Se setea el sistema para reproducir otra nota
    } else {                           // Si una melodía se ha acabado
        if (Musica.identificador == 'G' || Musica.identificador == 'P') {
            estadoJuego = 0;           // Si es la de ganador o perdedor, se deja de reproducir música
        }
        if (Musica.seRepite) {
            indice = 0;              // Si es una melodía que se repite se resetea el índice de duraciones
        }
    }
}
```



ethernet.h

ethernet.c

Definición de alias y constantes

```
#define HOST 0x30          // Valor en hexadecimal del nombre asignado al host donde se ejecute el programa
#define SIGUIENTE_HOST 0x31 // Valor en hexadecimal del nombre asignando al host externo

#define TAM_SUDOKU_FILA 9    // Definición del tamaño de fila y columna del tablero sudoku
#define TAM_SUDOKU 81        // Definición del tamaño total del tablero sudoku
```

Definición de variables de ámbito global

```
typedef struct {
    char inicial[81];           // Definición de tablero de un nivel de juego sudoku
    char solucion[81];          // Situación inicial de los números
} Nivel;                         // Situación final de los números (solución del tablero)

Nivel SudokuRed = {               // Definición del tablero sudoku a rellenar por el modo online
    {".....", ".....", ".....", ".....", ".....", ".....", ".....", ".....", "....."},  

    {".....", ".....", ".....", ".....", ".....", ".....", ".....", ".....", "....."}  

};
```

Definición de cabeceras de funciones

```
void leeEthernetInicial(void);  

void leeEthernet (void);  

void escribeEthernet();  

void actualizaSDRAM (char *pTabSudoku);  

void actualizaHOST(void);  

char pingTurno(void);  

void pingTablero(char *pTabSudoku);
```



ethernet.c

```
ethernet.c
```

```
#include "ethernet.h"
```

```
void leeEthernetInicial(void)

Obtiene los datos iniciales del tablero sudoku que se va a jugar y su correspondiente solución, necesaria para comprobar la solución en caso de activa el modo ayuda.
-----
void leeEthernetInicial(void) {

    int i;                                // Variable índice
    md_last_address = 0x00060000;           // Definición del comienzo de la zona de memoria SDRAM destinada a ethernet

    leeEthernet();                          // Se leen los datos del fichero del servidor TFTP escribiéndose en memoria

    for (i = 1; i < (TAM_SUDOKU+1); i++) {    // Recorremos los 81 dígitos para el tablero de sudoku

        SudokuRed.inicial[i-1] = lee_byte((md_last_address+i));      // Escribimos de la memoria a la definición del sudoku

    }

    for (i = (TAM_SUDOKU+1); i < (TAM_SUDOKU*2 + 1); i++) {        // Recorremos los 81 dígitos siguientes para la solución sudoku

        // Escribimos de la memoria a la solución sudoku
        SudokuRed.solucion[i-(TAM_SUDOKU+1)] = lee_byte((md_last_address+i));

    }
    return;
}
```

```
void leeEthernet (void)

Se leen datos del servidor TFTP a través del puerto ethernet y se escriben en la memoria asignada.
-----

void leeEthernet (void) {

    uint32 addr = USERSPACE;                      // Definición del espacio de usuario en memoria
    char *fn;                                     // Definición el puntero de fichero
    int ch;                                       // Definición del carácter a leer

    fn = filename;                                // Seteo del fichero de lectura (predeterminado)

    // Inicialización del Temporizador 2 destinado a ethernet
    timer_init(TIMER_NETWORK, TIMER_NETWORK_PERIOD, SYSTEM_CLOCK, TIMER_NETWORK_LEVEL);

    // Habilitación de interrupciones FEC para el núcleo del ColdFire
    mbar_writeLong(MCFSIM_ICR3,MCF5272_SIM_ICR_ERX_IL(FEC_LEVEL));

    // Inicialización del dispositivo de red
    fec_init(&fec_nif);

    // Escritura de la dirección ethernet en la estructura NIF
    fec_nif.hwa[0] = mac[0];
    fec_nif.hwa[1] = mac[1];
    fec_nif.hwa[2] = mac[2];
    fec_nif.hwa[3] = mac[3];
    fec_nif.hwa[4] = mac[4];
    fec_nif.hwa[5] = mac[5];

    nbuf_init();        // Inicialización de los búferes de red

    // Inicialización de ARP
    arp_init(&arp_info);
    nif_bind_protocol(&fec_nif,FRAME_ARP,(void *)arp_handler,(void *)&arp_info);

    // Inicialización de IP
    ip_init(&ip_info,client,gateway,netmask);
    nif_bind_protocol(&fec_nif,FRAME_IP,(void *)ip_handler,(void *)&ip_info);

    // Inicialización de UDP
    udp_init();

    // Se abre la conexión TFTP
```

```
if (tftp_read(&fec_nif, fn, server))
{
    while (1)
    {
        ch = tftp_in_char();           // Obtenemos el caracter

        if (ch == -1)
        {
            break;                  // Se ha terminado la lectura
        }
        else
        {
            *(unsigned char *)addr++ = (unsigned char)ch;
        }
    }
    tftp_end(TRUE);                // Se cierra la conexión TFTP
}
```

```
void escribeEthernet()

Se escriben datos en el servidor TFTP a través del puerto ethernet a partir del espacio de memoria asignado en SDRAM
-----

void escribeEthernet() {

    uint32 bytes;           // Definición de bytes a escribir
    char *fn;               // Definición del puntero de fichero

    bytes = TAM_SUDOKU*2 + 1; // Transmitimos los 81 números del sudoku, 81 de la solución, y flag de control

    fn = filename;          // Asignación de fichero (predeterminado)

    // Inicialización del Temporizador 2 destinado a ethernet
    timer_init(TIMER_NETWORK, TIMER_NETWORK_PERIOD, SYSTEM_CLOCK, TIMER_NETWORK_LEVEL);

    // Habilitación de interrupciones FEC para el núcleo del ColdFire
    mbar_writeLong(MCFSIM_ICR3,MCF5272_SIM_ICR_ERX_IL(FEC_LEVEL));

    // Inicialización del dispositivo de red
    fec_init(&fec_nif);

    // Escritura de la dirección ethernet en la estructura NIF
    fec_nif.hwa[0] = mac[0];
    fec_nif.hwa[1] = mac[1];
    fec_nif.hwa[2] = mac[2];
    fec_nif.hwa[3] = mac[3];
    fec_nif.hwa[4] = mac[4];
    fec_nif.hwa[5] = mac[5];

    nbuf_init();           // Inicialización de los búferes de red

    // Inicialización de ARP
    arp_init(&arp_info);
    nif_bind_protocol(&fec_nif,FRAME_ARP,(void *)arp_handler,(void *)&arp_info);

    // Inicialización de IP
    ip_init(&ip_info,client,gateway,netmask);
    nif_bind_protocol(&fec_nif,FRAME_IP,(void *)ip_handler,(void *)&ip_info);

    // Inicialización de UDP
    udp_init();
```

```
// Llamada a la función de escritura de datos
tftp_write(&fec_nif, fn, server, md_last_address, md_last_address + bytes);
}
```

```
void actualizaSDRAM (char *pTabSudoku)

Actualiza el espacio de SDRAM destinado a ethernet con la colocación actual de números del tablero sudoku.
-----

void actualizaSDRAM (char *pTabSudoku) {

    BYTE i;                                // Definición de índice
    BYTE dato;                             // Definición de dato a escribir
    uint32 direccion;                      // Dirección de memoria para introducir dato

    for (i = 1; i < (TAM_SUDOKU+1); i++) {      // Se recorre el tablero sudoku

        direccion = md_last_address + i;      // Se obtiene la dirección en la que introducir el número
        dato = *(pTabSudoku+i-1);            // Se obtiene el número a escribir en la dirección
        escribe_byte(direccion, dato);        // Se actualiza el espacio de SDRAM
    }
    return;
}
```

```
void actualizaHOST(void)  
Modifica el flag de control de jugador en el espacio de memoria SDRAM con el número del otro host.  
-----  
void actualizaHOST(void) {  
    escribe_byte(md_last_address, SIGUIENTE_HOST);           // El flag de control se ubica al comienzo del espacio de SDRAM  
} // Dirección 0x60000
```

```
char pingTurno(void)  
Hace una consulta de turno en el servidor TFTP leyendo el flag de control.  
-----  
char pingTurno(void) {  
    leeEthernet();           // Escritura en memoria de los datos del servidor TFTP  
    return lee_byte(md_last_address); // Lectura del flag de control ubicado al comienzo del espacio SDRAM  
}
```

```
void pingTablero(char *pTabSudoku)

Hace una consulta de la situación del tablero de sudoku en el servidor TFTP leyendo los 81 bytes siguientes al flag de control.
-----
void pingTablero(char *pTabSudoku) {

    BYTE i;                                // Definición de índice

    leeEthernet();                          // Escritura en memoria de los datos del servidor TFTP

    for (i = 1; i < (TAM_SUDOKU+1); i++) {      // Recorrido de los 81 bytes siguientes al flag de control

        *(pTabSudoku+i-1) = lee_byte((md_last_address+i)); // Actualización del tablero sudoku según los valores
                                                               // guardados en el espacio de memoria SDRAM
    }
}
```



sudoku . h

Definición de alias y constantes

```
#define COORD_MIN 0x31
#define COORD_MAX 0x39
#define NUM_MIN 0x31
#define NUM_MAX 0x39
#define NIVEL_MAX 10
#define VACIO 0x2E
#define TIEMPO_JUGADOR 60
#define SALIR 0x46
```

Definición de variables de ámbito global

```
// Sudoku trivial para las pruebas de juego
Nivel SudokuTrivial = {
    {"1746352986.9824715825971.64241587936786293541953.46827498712653.674591825123684.9"}, // Solución del sudoku
    {"174635298639824715825971364241587936786293541953146827498712653367459182512368479"}
};

Nivel SudokuNovato = {
    {"17...5.98..98.47.5..5.7.36.24..87....8.2..541.531.6.274..71.6...674..182.12368..."}, // 44 números iniciales
    {"174635298639824715825971364241587936786293541953146827498712653367459182512368479"} // Solución del sudoku
};

Nivel SudokuSencillo = {
    {"473.5...2...9.2.3...54..8.17.6.4...8.1.785.638.4...7...476.81..1..5..94.5....438."}, // 38 números iniciales
    {"473851692681972534295463871736149258912785463854326719347698125128537946569214387"} // Solución del sudoku
};

Nivel SudokuIniciados = {
    {"32.8..17669..538.....6...5...9..4...2.7.3...3..6...5...6.....953..12416..2.53"}, // 33 números iniciales
    {"325849176697153824148726539561398247982475361734261985253617498879534612416982753"} // Solución del sudoku
};

Nivel SudokuMedio = {
    {"2897..35.....29.4...981...7.2...14...512...9.6...72..8..24...34....3.9.9.....4.2"}, // 33 números iniciales
    {"289764351671352984534981627792835146345126879168497235817249563426513798953678412"} // Solución del sudoku
};

Nivel SudokuComplejo = {
    {"58..27.12....4..9.....8.....9.3.32.5.14.1.5.....1.....3..7....65.94..23."}, // 27 números iniciales
    {"458392761261574389973618425847261953632957148195843672726135894384729516519486237"} // Solución del sudoku
};
```

```

Nivel SudokuDificil = {
    {"..4.....53..4....7..9.....8..1..4..35....6..6.2.8.7.2....6....9.8.7..3.8..9..1"},           // 25 números iniciales
    {"914732568853164972672598314297846135481357296536921847725413689149685723368279451"}           // Solución del sudoku
};

Nivel SudokuChungo = {
    {"6..52.....7..8.....9...152.....6...7...3.61.....49..1..4.....8..4..92...5..."},                  // 23 números iniciales
    {"613529784275481936489673152598146273742938615136257849861394527357862491924715368"}           // Solución del sudoku
};

Nivel SudokuImposible = {
    {"...4.5...2..3...9..3...1.41.9....8....3..9..7..1..4.8....4.....49..9....8.."},                      // 22 números iniciales
    {"791465328248371569563289174159642783482537691376918245827193456635824917914756832"}           // Solución del sudoku
};

Definición de cabeceras de funciones
-----
void iniciaSudoku(void);

void sudokuJuego (char *pTabSudoku, int *pTabTurnos, int *pRed, int *pNumJugadores, int *pAciertos, int *pPuntos, int *pGanador, int
*pJugador, int *pDificultad, int *pAyuda, int *pGanarPorTiempo, char *pSolucion);

void refrescoTablero(char *pTabSudoku, int *pRed, int *pNumJugadores, int *pJugador);

void refrescaTurno(int *pJugador);

void incrementoTurno(char *pTabSudoku, int *pRed, int *pJugador);

char* setConfiguracion(int *pRed, int *pNumJugadores, char *pTabSudoku, int *pDificultad, int *pAyuda, int *pGanarPorTiempo);

char* setConf(int *pRed, int *pNumJugadores, int numJugadores, char *pTabSudoku, int *pDificultad, int dificultad, int *pAyuda, int
ayuda, int *pGanarPorTiempo, int ganarPorTiempo);

char* devuelveSolucion(char *pTabSudoku, int *pRed, int *pDificultad);

void setRed(int *pRed);

void setJugadores(int *pNumJugadores);

char* setDificultad(char *pTabSudoku, int *pRed, int *pDificultad);

void setAyuda(int *pAyuda);

void setPuntuacion(int *pGanarPorTiempo);

```

```
void actualizaPuntos(int *pJugador, int *pPuntos, int *pTabTurnos, int posicion, int *pGanarPorTiempos);
void muestraResultados(int *pNumJugadores, int *pAciertos, int *pPuntos, int *pGanador, int *pGanarPorTiempos, int *pRed);
BOOL hayRepeticion(char *pTabSudoku, int posicion, char numero);
BOOL hayRepeticionFila(char *pTabSudoku);
BOOL hayRepeticionColumna(char *pTabSudoku);
```



sudoku . c

Fichero principal del juego sudoku

```
#include "sudoku.h"
```



```
int ganarPorTiempos = 0;           // Flag que indica si se gana por número de aciertos o por rapidez.
int *pGanarPorTiempos;           // Creación del puntero

char *pSolucion;                // Creación del puntero que apunta a los números solución

// Configuración inicial del juego y obtención del puntero de solución en función del tablero elegido
pSolucion = setConfiguracion(pRed, pNumJugadores, pTabSudoku, pDificultad, pAyuda, pGanarPorTiempos);

// Llamada al comienzo del juego sudoku
sudokuJuego(pTabSudoku, pTabTurnos, pRed, pNumJugadores, pAciertos, pPuntos, pGanador, pJugador, pDificultad, pAyuda,
pGanarPorTiempos, pSolucion);
    return;
}
```

```
void sudokuJuego (char *pTabSudoku, int *pTabTurnos, int *pRed, int *pNumJugadores,int *pAciertos, int *pPuntos, int *pGanador, int
*pJugador, int *pDificultad, int *pAyuda,int *pGanarPorTiempos, char *pSolucion)

Función principal del juego sudoku. Controla el correcto funcionamiento de la partida y la cadena de niveles. Muestra las puntuaciones
y el ganador al acabar.

-----
void sudokuJuego (char *pTabSudoku, int *pTabTurnos, int *pRed, int *pNumJugadores, int *pAciertos, int *pPuntos, int *pGanador, int
*pJugador, int *pDificultad, int *pAyuda, int *pGanarPorTiempos, char *pSolucion) {

    while (*pDificultad < NIVEL_MAX) { // Ciclo de niveles: cuando finalizamos un nivel, se pasa al siguiente

        inicializaPuntuacion(pAcieritos, pPuntos, pNumJugadores);           // Inicializa las tablas de puntuaciones
        muestraNivel(pDificultad);                                              // Se muestra el nivel en el que estamos

        while (TRUE) { // Bucle de turno de jugador

            // Se muestra la situación actual del tablero sudoku
            refrescoTablero(pTabSudoku, pRed, pNumJugadores, pJugador);
            // Comprobación de partida finalizada (sudoku completado)
            if (haySudoku(pTabSudoku, pTabTurnos, pAcieritos, pNumJugadores, pGanador, pPuntos, pGanarPorTiempos)) {
                if (*pRed == 1) actualizaHOST();
                // En caso de estar en modo online, forzamos a que el otro jugador obtenga el tablero acabado
                break;
            }
            refrescaTurno(pJugador); // Se muestran los mensajes correspondientes al turno actual

            // Realización del movimiento sudoku. Se devuelve FALSE si el jugador abandona la partida
            if (!movimientoSudoku(pJugador, pTabSudoku, pTabTurnos, pSolucion, pAyuda, pGanarPorTiempos, pPuntos)) {
                pintoTiempo = FALSE;           // Reseteo del flag de escribir el tiempo restante en pantalla
                sleep(0);                     // Reseteo del contador de milisegundos restantes
                return;
            }
            incrementoTurno(pTabSudoku, pRed, pJugador); // Incremento del turno de jugador
        }

        // Partida finalizada, se muestra la tabla de resultados
        muestraResultados(pNumJugadores, pAcieritos, pPuntos, pGanador, pGanarPorTiempos, pRed);
        *pGanador = 0;                  // Reset del jugador ganador
        if (*pRed == 1) { // En caso de finalizar partida del modo online, se cierra el sudoku por completo
            timer_init(TIMER_NETWORK, TIMER_NETWORK_PERIOD, SYSTEM_CLOCK, 0); // Cambio en el nivel del timer 2
            return;
        }
        *pDificultad += 1;             // Incremento del nivel de dificultad (sólo modo local)
        // Reconfiguración de los parámetros principales (se mantienen las elecciones del usuario)
```

```
    pSolucion = setConf(pRed, pNumJugadores, *pNumJugadores, pTabSudoku, pDificultad, *pDificultad, pAyuda, *pAyuda,
pGanarPorTiempos, *pGanarPorTiempos);
}
return;
}
```

```
void refrescoTablero(char *pTabSudoku, int *pRed, int *pNumJugadores, int *pJugador)

Obtiene y muestra por pantalla la situación actual del tablero sudoku.
-----
void refrescoTablero(char *pTabSudoku, int *pRed, int *pNumJugadores, int *pJugador) {

    if (*pRed == 1) { // Comprobación de modo online
        while ( pingTurno() == SIGUIENTE_HOST) { // Bucle de espera de movimiento del jugador en el otro HOST
            output("\n --- Esperando el movimiento del otro jugador"); // Mensaje de espera
            loading(); // Animación de espera
            espera(); // Tiempo de guarda entre mensajes
            output("\n");
            continue;
        }
        pingTablero(pTabSudoku); // Una vez ha movido el jugador del otro HOST, se obtiene la nueva situación del tablero
    } else {
        if(*pJugador > *pNumJugadores) *pJugador = 1; // Se hace el ciclo de turnos
    }
    turnoPerdido = FALSE; // Reseteo del turno perdido
    muestraTableroSudoku(pTabSudoku); // Se muestra el tablero inicial
    return;
}
```

```
void refrescaTurno(int *pJugador)  
Muestra en pantalla el mensaje del turno actual y setea el tiempo de turno del jugador.  
-----  
void refrescaTurno(int *pJugador) {  
    output("\nTurno del jugador ");           // Mensaje de turno  
    outNum(10,*pJugador,0);                 // Número de jugador  
    output(":\\n-----");  
    sleep(TIEMPO_JUGADOR*1000);            // Seteo del tiempo de turno para el jugador  
}
```

```
void incrementoTurno(char *pTabSudoku, int *pRed, int *pJugador)

Incremento del número del próximo jugador cuyo turno comienza.
-----
void incrementoTurno(char *pTabSudoku, int *pRed, int *pJugador) {

    if (*pRed == 1)                                // Comprobación de modo en red

        actualizaHOST();                           // Commuta el turno al jugador situado en el otro HOST
        actualizaSDRAM(pTabSudoku);                // Actualiza en el espacio de memoria SDRAM la situación nueva del tablero
        escribeEthernet();                         // Escribe los nuevos datos en el servidor TFTP

    } else {
        *pJugador += 1;                            // Se pasa el turno al siguiente jugador (sólo modo local)
    }
    pintoTiempo = FALSE;                          // Reseteo del flag de escritura de tiempo en pantalla
    sleep(0);                                    // Se pone a cero el tiempo restante
    return;
}
```

```
char* setConfiguracion(int *pRed, int *pNumJugadores, char *pTabSudoku, int *pDificultad, int *pAyuda, int *pGanarPorTiempos)
Seteo de la configuración inicial de una partida de sudoku: modo online/local, número de jugadores, activación del modo ayuda, tipo de puntuaciones y dificultad inicial.
-----
char* setConfiguracion(int *pRed, int *pNumJugadores, char *pTabSudoku, int *pDificultad, int *pAyuda, int *pGanarPorTiempos) {
    setRed(pRed);                                // Seteo del modo online por parte del usuario
    if (*pRed == 0) {                            // Comprobación de modo local
        setJugadores(pNumJugadores);            // Seteo del número de jugadores
    } else {                                     // Seteo de 2 jugadores para modo online
        *pNumJugadores = 2;
    }
    setAyuda(pAyuda);                          // Seteo del modo ayuda: el sistema te avisa de los equívocos
    setPuntuacion(pGanarPorTiempos);          // Seteo del modo de puntuar: por aciertos o por rapidez
    return setDificultad(pTabSudoku, pRed, pDificultad); // Seteo del nivel de dificultad inicial
                                                    // (devuelve el puntero con la solución del tablero)
}
```

```
char* setConf(int *pRed, int *pNumJugadores, int numJugadores, char *pTabSudoku, int *pDificultad, int dificultad, int *pAyuda, int ayuda, int *pGanarPorTiempos, int ganarPorTiempos)

Reconfigura la partida de sudoku manteniendo los parámetros iniciales solicitados al usuario.

-----
char* setConf(int *pRed, int *pNumJugadores, int numJugadores, char *pTabSudoku, int *pDificultad, int dificultad, int *pAyuda, int ayuda, int *pGanarPorTiempos, int ganarPorTiempos) {

    *pNumJugadores = numJugadores;                                // Reconfiguración del número de jugadores
    *pDificultad = dificultad;                                     // Reconfiguración de la dificultad
    *pAyuda = ayuda;                                              // Reconfiguración de la activación del modo ayuda
    *pGanarPorTiempos = ganarPorTiempos;                            // Reconfiguración del flag que determina el tipo de puntuación
    return devuelveSolucion(pTabSudoku, pRed, pDificultad);      // Devolución del puntero de la solución del sudoku
}
```

```
char* devuelveSolucion(char *pTabSudoku, int *pRed, int *pDificultad) *
Devuelve el puntero con dirección a la solución del tablero sudoku en función de la dificultad del juego.
-----
char* devuelveSolucion(char *pTabSudoku, int *pRed, int *pDificultad) {

    char *pSolucion;           // Creación del puntero de la solución del tablero

    if (*pRed == 1) {          // Comprobación del modo online
        output("\n\nConectando al servidor de Sudoku online");           // Mensajes de conexión al servidor TFTP
        output("\nPor favor permanece a la espera");
        loading();
        leeEthernetInicial();                                         // Lectura de los datos del sudoku del servidor TFTP
        inicializaSudoku(pTabSudoku, SudokuRed);                      // Inicialización del tablero sudoku con los números escritos en SDRAM
        pSolucion = &(SudokuRed.solucion[0]);                         // Se actualiza la referencia del puntero a la solución

    } else {

        switch(*pDificultad) {                                     // Se inicializa el tablero del sudoku según la dificultad elegida

            case 1:
                inicializaSudoku(pTabSudoku, SudokuTrivial); // Inicialización del tablero sudoku
                pSolucion = &(SudokuTrivial.solucion[0]);   // Se actualiza la referencia del puntero a la solución
                break;
            case 2:
                inicializaSudoku(pTabSudoku, SudokuNovato); // Inicialización del tablero sudoku
                pSolucion = &(SudokuNovato.solucion[0]);   // Se actualiza la referencia del puntero a la solución
                break;
            case 3:
                inicializaSudoku(pTabSudoku, SudokuSencillo); // Inicialización del tablero sudoku
                pSolucion = &(SudokuSencillo.solucion[0]);  // Se actualiza la referencia del puntero a la solución
                break;
            case 4:
                inicializaSudoku(pTabSudoku, SudokuIniciados); // Inicialización del tablero sudoku
                pSolucion = &(SudokuIniciados.solucion[0]); // Se actualiza la referencia del puntero a la solución
                break;
            case 5:
                inicializaSudoku(pTabSudoku, SudokuMedio);   // Inicialización del tablero sudoku
                pSolucion = &(SudokuMedio.solucion[0]);     // Se actualiza la referencia del puntero a la solución
                break;
            case 6:
                inicializaSudoku(pTabSudoku, SudokuComplejo); // Inicialización del tablero sudoku
                pSolucion = &(SudokuComplejo.solucion[0]);  // Se actualiza la referencia del puntero a la solución
                break;
        }
    }
}
```

```
    case 7:
        inicializaSudoku(pTabSudoku, SudokuDificil); // Inicialización del tablero sudoku
        pSolucion = &(SudokuDificil.solucion[0]);      // Se actualiza la referencia del puntero a la solución
        break;
    case 8:
        inicializaSudoku(pTabSudoku, SudokuChungo); // Inicialización del tablero sudoku
        pSolucion = &(SudokuChungo.solucion[0]);      // Se actualiza la referencia del puntero a la solución
        break;
    default:
        inicializaSudoku(pTabSudoku, SudokuImposible); // Inicialización del tablero sudoku
        pSolucion = &(SudokuImposible.solucion[0]);      // Se actualiza la referencia del puntero a la solución
        break;
    }
}
return pSolucion;
}
```

```
void setRed(int *pRed)

Configuración del modo online mediante solicitud al usuario.
-----
void setRed(int *pRed) {

    do {
        output("\n¿Deseas jugar un sudoku en red o local? [1 Red | 0 Local]: ");
        // Solicitud de modo online
        *pRed = (teclado()-'0');
        // Se captura la elección del usuario
        outNum(10, *pRed, SIN_SIGNO);
        // Se muestra la elección
    } while (*pRed != 0 && *pRed != 1);
    // Comprobación de elección válida
}
```

```
void setJugadores(int *pNumJugadores)

Configuración del número de jugadores mediante solicitud al usuario (de 1 a 9 participantes)
-----
void setJugadores(int *pNumJugadores) {

    do {
        output("\nIndica el número de jugadores [de 1 a 9]: "); // Asignación por parte del usuario del número de jugadores
        *pNumJugadores = (teclado() - '0');                      // Se captura la elección del usuario
        outNum(10, *pNumJugadores, SIN_SIGNO);                  // Se muestra por pantalla

    } while (*pNumJugadores < 1 || *pNumJugadores > 9);          // Comprobación de elección válida
}
```

```
char* setDificultad(char *pTabSudoku, int *pRed, int *pDificultad)

Configuración de la dificultad inicial mediante solicitud al usuario (de nivel 1 al 9)
-----
char* setDificultad(char *pTabSudoku, int *pRed, int *pDificultad) {

    if (*pRed == 0) { // Comprobación del modo online

        do {
            output("\nIndica la dificultad de juego [1 fácil - 9 difícil]: ");           // Asignación del nivel de dificultad
            *pDificultad = (teclado()-'0');                                                 // Seteo de la dificultad
            outNum(10, *pDificultad, SIN_SIGNO);                                         // Se muestra por pantalla

        } while (*pDificultad < 1 || *pDificultad > 9 );                         // Comprobación de nivel de dificultad válido
    } else {
        *pDificultad = 0;                                                               // Se define dificultad cero el modo online
    }

    return devuelveSolucion(pTabSudoku, pRed, pDificultad);                         // Devuelve el puntero con la solución del tablero
}
```

```
void setAyuda(int *pAyuda)

Configuración del modo ayuda (te avisa de los errores durante la partida) mediante solicitud al usuario
-----
void setAyuda(int *pAyuda) {

    do {
        output("\n¿Quieres activar el modo ayuda durante el juego? [1 Sí | 0 No]: "); // Solicitud de modo Ayuda
        *pAyuda = (teclado()-'0'); // Se captura la elección del usuario
        outNum(10, *pAyuda, SIN_SIGNO); // Se muestra la elección por pantalla
    } while (*pAyuda != 0 && *pAyuda != 1); // Comprobación de elección válida
}
```

```
void setPuntuacion(int *pGanarPorTiempos)

Configuración del tipo de puntuación mediante solicitud al usuario. Puede ser por suma de tiempos restantes en el momento del acierto o por número de aciertos.

-----
void setPuntuacion(int *pGanarPorTiempos) {

    do {
        output("\n¿Cómo se puntúa? [0 Por aciertos | 1 Por tiempos]: ");
        // Solicitud del modo de puntuar
        *pGanarPorTiempos = (teclado()-'0');
        // Seteo del flag
        outNum(10, *pGanarPorTiempos, SIN_SIGNO);
        // Se muestra la elección por pantalla
    } while (*pGanarPorTiempos != 0 && *pGanarPorTiempos != 1);
    // Comprobación de elección válida
}
```

```
void actualizaPuntos(int *pJugador, int *pPuntos, int *pTabTurnos, int posicion, int *pGanarPorTiempos)

Se actualiza la cuenta de puntos del jugador que ha finalizado su turno, teniendo en cuenta el tipo de puntuación.
-----
void actualizaPuntos(int *pJugador, int *pPuntos, int *pTabTurnos, int posicion, int *pGanarPorTiempos) {

    if (*pGanarPorTiempos) {                                // Comprobación de puntuación basada en tiempos

        *(pPuntos+*pJugador-1) += tiempo; // Se añaden a la cuenta de puntos del jugador los segundos
    } else {                                                 // restantes tras el acierto

        *(pTabTurnos+posicion) = *pJugador; // Se anota el acierto al jugador correspondiente
    }
    return;
}
```

```
void muestraResultados(int *pNumJugadores, int *pAciertos, int *pPuntos, int *pGanador, int *pGanarPorTiempos, int *pRed)

Imprime por pantalla la tabla de puntuaciones finales, en cada fila el número de jugador y sus puntos, teniendo en cuenta el número de
participantes y el sistema de puntuación elegido.
-----
void muestraResultados(int *pNumJugadores, int *pAciertos, int *pPuntos, int *pGanador, int *pGanarPorTiempos, int *pRed) {

    BYTE turno;          // Índice para recorrer el array de cuenta de aciertos
    char mensaje[] = {"¡Sudoku completado! Puntuación final\n\n"}; // Mensaje de fin de juego
    formateaCadena(mensaje);

    for (turno = 0; turno < *pNumJugadores; turno++) {      // Se recorren las puntuaciones de los jugadores
        output("\n[Jugador ");                                // Se imprime por pantalla la puntuación del jugador
        outNum(10, turno+1, 0);
        output("] ");

        if (*pGanarPorTiempos) {                            // Condición de tipo de puntuación
            outNum(10, *(pPuntos+turno), 0);                // Se muestran por pantalla los puntos por rapidez
        } else {
            outNum(10, *(pAciertos+turno), 0);              // Se muestran por pantalla los puntos por aciertos
        }
        output(" puntos\n");
        espera();                                         // Se realiza una espera después de cada fila mostrada
        if (*pRed == 1) break;                            // En modo online, sólo se muestran los puntos del jugador en el primer host
    }
    if (*pGanador != 0) {                                // Comprobación de existencia de jugador ganador
        if (*pNumJugadores == 1) {                        // Comprobación de partida en solitario
            output("\n¡Enhorabuena campeón!");           // Se muestra un mensaje final
        } else if (*pRed == 0) {                          // Comprobación de modo local
            output("\n\n¡Ha ganado el jugador ");
            outNum(10,*pGanador,0);                      // Se muestra por pantalla el jugador ganador
            output("!");
        }
    } else {
        output("\n\n¡Ha habido empate!");                 // Si se produce empate, se muestra por pantalla
    }
    output("\n\nPulsa cualquier tecla para continuar...\n"); // Mensaje de introducción de cualquier tecla para finalizar partida
    teclado();                                         // Solicitud de pulsar cualquier tecla
    return;
}
```

```
BOOL hayRepeticion(char *pTabSudoku, int posicion, char numero)

Comprueba si hay un número repetido en fila, columna o bloque (reglas del sudoku)
-----
BOOL hayRepeticion(char *pTabSudoku, int posicion, char numero) {

    BOOL hayRepeticion = FALSE;

    // Se hace el supuesto de que en esa posición el jugador escriba el número indicado
    *(pTabSudoku+posicion) = numero;

    // Se comprueba que ese número no esté duplicado en fila, columna o bloque
    //hayRepeticion = (hayRepeticionFila(pTabSudoku) || hayRepeticionColumna(pTabSudoku));
    hayRepeticion = (hayRepeticionFila(pTabSudoku) || hayRepeticionColumna(pTabSudoku) || hayRepeticionBloque(pTabSudoku));

    *(pTabSudoku+posicion) = VACIO;      // Se restablece el tablero sudoku

    return hayRepeticion;              // Se devuelve la comprobación final de si existe algún tipo de comprobación
}
```

```
BOOL hayRepeticionFila(char *pTabSudoku)

Comprueba si hay un número repetido en una fila del sudoku
-----
BOOL hayRepeticionFila(char *pTabSudoku) {

    BYTE i;
    char simbolo_ref;

    for (i = 0; i < TAM_SUDOKU-TAM_SUDOKU_FILA+1; i+=TAM_SUDOKU_FILA) { // El índice i apunta al comienzo de cada fila

        BYTE indice = i;

        for ( ; indice < TAM_SUDOKU_FILA+i; indice++) {           // indice recorre cada posición de la fila

            BYTE j = indice;                                     // El índice j recuerda el primer número de la fila y se incrementa
            simbolo_ref = *(pTabSudoku+indice); // Se coge como referencia

            if (simbolo_ref == VACIO) {             // Si no hay número, pasamos a la siguiente posición
                continue;
            }

            for (j = i; j < TAM_SUDOKU_FILA+i; j++) { // Se recorre la fila

                if (j == indice) {                   // Si estamos en la posición de referencia,
                    continue;                      // no tendremos en cuenta repetición.
                }

                if (* (pTabSudoku+j) == simbolo_ref) {
                    return TRUE;                  // En caso de que haya un número repetido, se sale
                }
            }
        }
    }
    return FALSE;
}
```

```
BOOL hayRepeticionColumna(char *pTabSudoku)

Comprueba si hay un número repetido en una columna del sudoku
-----
BOOL hayRepeticionColumna(char *pTabSudoku) {

    BYTE i;                      // Índice para recorrer el sudoku
    char simbolo_ref;            // Se crea un carácter de referencia para hacer las comprobaciones de repetición

    for (i = 0; i < TAM_SUDOKU_FILA; i++) {      // El índice i apunta al comienzo de cada columna

        BYTE indice = i;
        // indice recorre cada posición de la columna
        for ( ; indice < TAM_SUDOKU-TAM_SUDOKU_FILA+i+1; indice+=TAM_SUDOKU_FILA) {

            BYTE j = indice;                  // El índice j recuerda el primer número de la fila y se incrementa
            simbolo_ref = *(pTabSudoku+indice); // Se coge como referencia

            if (simbolo_ref == VACIO) {          // Si no hay número, pasamos a la siguiente posición
                continue;
            }

            for (j = indice ; j < TAM_SUDOKU-TAM_SUDOKU_FILA+i+1; j+=TAM_SUDOKU_FILA) {      // Se recorre la fila

                if (j == indice) {                // Si estamos en la posición de referencia,
                    continue;                  // no tendremos en cuenta repetición.
                }

                if (* (pTabSudoku+j) == simbolo_ref) {
                    return TRUE;               // En caso de que haya un número repetido, se sale
                }
            }
        }
    }
    return FALSE;
}
```

```
BOOL hayRepeticionBloque(char *pTabSudoku)

Comprueba si hay un número repetido en un bloque 3x3 del sudoku
-----
BOOL hayRepeticionBloque(char *pTabSudoku) {

    BYTE i;                                // Creación de índice para recorrer bloques
    BYTE j;                                // Creación de índice para recorrer bloques
    char simbolo_ref;                      // Se crea un carácter de referencia para hacer las comprobaciones de repetición
    BYTE contador = 0;

    for (i = 0; i < TAM_SUDOKU_FILA; i+=(TAM_SUDOKU_FILA/3)) {      // El índice i apunta la columna comienzo del bloque
        for (j = 0; j < TAM_SUDOKU; j+=(TAM_SUDOKU_FILA*3)) { // El índice j apunta a la fila comienzo del bloque

            BYTE l = (j + i);                                // El índice l apunta al número que tomaremos de referencia
                                                               // para detectar si se encuentra duplicado.

            contador = 0;                                    // Un contador nos ayudará a recorrer mejor las filas de un bloque,
                                                               // que en el vector de Sudoku no se encuentran contiguas.

            // Ahora hacemos la comprobación para cada uno de los bloques 3x3

            // Bucle de iteración de símbolo de referencia
            for ( ; l < ((j+ TAM_SUDOKU_FILA*2) + (i+TAM_SUDOKU_FILA/3)); l++) {

                BYTE f = j;
                BYTE g = i;

                if (contador == (TAM_SUDOKU_FILA/3) ) { // Comprobación de que hemos acabado la fila de un bloque
                                                               // Para pasar a la siguiente, hay que sumar al índice dos filas de bloque (3 + 3)
                    l += 2*TAM_SUDOKU_FILA/3;
                    contador = 0;
                }

                simbolo_ref = *(pTabSudoku+l);          // Tomamos la referencia para localizar duplicado

                if (simbolo_ref == VACIO) {             // Si no hay número en esa posición, buscamos el siguiente
                    contador++;
                    continue;
                }
                // El índice f apunta al principio de una fila
                for ( ; f < j + (TAM_SUDOKU_FILA*2 + 1); f+=TAM_SUDOKU_FILA) {
```

```
    BYTE columna = i;
    // Con un índice columna, recorremos la fila.
    for ( ; columna < TAM_SUDOKU_FILA/3 + i; columna++) {
        // Si estamos en la posición de referencia,
        if (f+columna == 1) {
            continue;                                // no tendremos en cuenta repetición.
        }
        if (*pTabSudoku+f+columna) == simbolo_ref) {
            return TRUE;                            // Caso en que detectamos repetición de número
        }
    }
    // El índice g apunta al principio de una columna
    for ( ; g < i+ (TAM_SUDOKU_FILA/3); g++) {

        BYTE fila = j;
        // Con un índice fila, recorremos la columna
        for ( ; fila < j + (TAM_SUDOKU_FILA*2 + 1); fila+=TAM_SUDOKU_FILA) {

            if (g+fila == 1) {                      // Si estamos en la posición de referencia,
                continue;                            // no tendremos en cuenta repetición.
            }
            if (*pTabSudoku+g+fila) == simbolo_ref) {
                return TRUE;                        // Caso en que detectamos repetición de número
            }
        }
        contador++;                         // Incrementamos el contador que detecta si se ha acabado una fila de bloque.
    }
}
return FALSE;
}
```

```
BOOL estanTodosNumeros(char *pTabSudoku)

Comprueba si ya están todos los números en el tablero del sudoku
-----
BOOL estanTodosNumeros(char *pTabSudoku) {

    BYTE indice;          // Creación de un índice para iterar el tablero

    for (indice = 0; indice < TAM_SUDOKU; indice++) {      // Se recorre el tablero del sudoku
        if (*pTabSudoku+indice) == VACIO) {
            return FALSE;                                // Se detecta si queda algún hueco vacío (sin número)
        }
    }
    return TRUE;
}
```

```
char turnoGanador(char *pTabSudoku, char *pTabTurnos, char *pTurno, int *pAciertos, int *pNumJugadores)

Realiza la cuenta de qué turno ha conseguido mayor número de aciertos. En caso de empate no se declara ganador.
-----
BYTE turnoGanador(int *pTabTurnos, int *pAciertos, int *pNumJugadores, int *pPuntos, int *pGanarPorTiempos) {

    BYTE turno;           // Índice que apunta al turno de jugador
    BYTE i;               // Índice para recorrer el array de aciertos del sudoku

    if (*pGanarPorTiempos) { // Condición para determinar ganador: por puntos o por aciertos
        return determinaMaximo(pPuntos, pNumJugadores); // Se determina el máximo de puntos y se devuelve el turno ganador
    } else {

        for (turno = 0; turno < *pNumJugadores; turno++) {
            for (i = 0; i < TAM_SUDOKU; i++) {
                if (*(pTabTurnos+i) == (turno+1) ) {
                    *(pAciertos+turno)+=1;
                }
            }
        }
        return determinaMaximo(pAciertos, pNumJugadores); // Se determina el jugador con máximo de aciertos
    }
}
```

```
BYTE determinaMaximo(int *pLista, int *pNumJugadores) {  
  
    Determina una máxima puntuación para una lista de puntuaciones recibida.  
-----  
BYTE determinaMaximo(int *pLista, int *pNumJugadores) {  
  
    BYTE turno;                                // Índice que apunta al turno de jugador  
    BYTE puntosGanador = 0;                     // Variable que almacena la mayor cantidad de puntos o aciertos  
    BYTE turnoGanador = 0;                       // Variable que almacena el turno ganador (0 si ha habido empate)  
    int empate = 0;                             // Flag de empate  
  
    for (turno = 0; turno < *pNumJugadores; turno++) { // Se recorre la lista buscando el máximo  
  
        if (*pLista+turno) > puntosGanador) {      // En caso de que haya una puntuación mejor, se actualiza el ganador  
            puntosGanador = *(pLista+turno);  
            turnoGanador = turno+1;  
        } else {  
            // En caso de que haya una puntuación igual a la del ganador, se produce empate  
            if (*(pLista+turno) == puntosGanador) {  
                empate = 1;                         // Se activa el flag  
                turnoGanador = 0;                      // Se actualiza el ganador  
            }  
        }  
    }  
    return turnoGanador;                         // Devolución del turno que se proclama ganador  
}
```

```
void inicializaPuntuacion(int *pAciertos, int *pPuntos, int *pNumJugadores)
Pone a cero los marcadores de puntuación de todos los usuarios.
-----
void inicializaPuntuacion(int *pAciertos, int *pPuntos, int *pNumJugadores) {
    BYTE i;                                // Creación del índice para iterar las puntuaciones de los jugadores
    for (i = 0; i < *pNumJugadores; i++) {    // Bucle de iteración de jugadores
        *(pPuntos+i) = 0;                   // Reseteo del marcador de puntos por tiempo
        *(pAciertos+i) = 0;                 // Reseteo del marcador de puntos por aciertos
    }
}
```

```
void inicializaSudoku(char *pTabSudoku, Nivel Sudoku)
    Inicializa un tablero de sudoku según un esquema prefijado de dificultad
-----
void inicializaSudoku(char *pTabSudoku, Nivel Sudoku) {
    BYTE indice;      // Creación del índice para iterar el tablero
    for (indice = 0; indice < TAM_SUDOKU; indice++) {      // Recorrido del tablero sudoku
        *(pTabSudoku+indice) = Sudoku.inicial[indice]; // Se inicializa el tablero según el esquema de dificultad elegido
    }
    return;
}
```

```
BOOL haySudoku(char *pTabSudoku, int *pTabTurnos, int *pAciertos, int *pNumJugadores, int *pGanador, int *pPuntos, int
*pGanarPorTiempos)

Comprueba si el juego ha finalizado. Para ello se mira si están todos los números sobre el tablero y si es así se calcula qué jugador
ha acertado más números.

-----
BOOL haySudoku(char *pTabSudoku, int *pTabTurnos, int *pAciertos, int *pNumJugadores, int *pGanador, int *pPuntos, int
*pGanarPorTiempos) {

    // Para que haya sudoku el tablero tiene que estar lleno
    if (estanTodosNumeros(pTabSudoku)) {

        // En caso afirmativo, se calcula el jugador ganador (mayor número de aciertos)
        *pGanador = turnoGanador(pTabTurnos, pAciertos, pNumJugadores, pPuntos, pGanarPorTiempos);
        return TRUE;

    }
    return FALSE;      // Caso en que no se ha completado el tablero
}
```

```
void muestraTableroSudoku(char *pTabSudoku)

Muestra el tablero sudoku con sus coordenadas horizontales y verticales.
-----
void muestraTableroSudoku(char *pTabSudoku) {

    BYTE indice;
    BYTE indice2 = 1;
    output("\n    1 2 3    4 5 6    7 8 9    ");
                                // Coordenadas de columnas

    for (indice = 0; indice < TAM_SUDOKU; indice++) {

        if (indice % (TAM_SUDOKU/3) == 0) {
            output("\n    -----");
        }
        if (indice % TAM_SUDOKU_FILA == 0) {
            output("\n");
            outNum(10, indice2, 0);
            indice2++;
        }
        if (indice % (TAM_SUDOKU_FILA/3) == 0) {
            output(" |");
        }
        output(" ");
        outch(*pTabSudoku+indice));
        if ( (indice+1) % TAM_SUDOKU_FILA == 0) {
            output(" |");
        }
    }
    output("\n    ----- \n");
                                // Línea horizontal del tablero
}
                                // Se pinta la línea vertical a la derecha del tablero
                                // Se escribe el número del sudoku
                                // En cada fila se escribe en número de fila
                                // Cada tres columnas se pinta una línea vertical
                                // Cada 3 filas se pinta una línea horizontal
                                // Coordenadas de columnas
```

```

void movimientoSudoku(char *pJugador, char *pTabSudoku, int *pTabTurnos, char *pSolucion, int *pAyuda, int *pGanarPorTiempos, int
*pPuntos)

Solicita al usuario una fila y columna del tablero sudoku y el número con el que rellenar.
-----

BOOL movimientoSudoku(int *pJugador, char *pTabSudoku, int *pTabTurnos, char *pSolucion, int *pAyuda, int *pGanarPorTiempos, int
*pPuntos) {

    char fila = '0';           // Coordenada de fila
    char columna = '0';        // Coordenada de columna
    char numero = '0';         // Número a situar en fila y columna

    while ( fila < COORD_MIN || fila > COORD_MAX ) {                                // Comprobación de fila válida
        if (turnoPerdido) return TRUE;                                                 // Comprobación de que no ha perdido su turno
        output("\nIndica la fila donde quieras escribir \[1-9]: |   "); // Solicitud de coordenada fila del tablero
        fila = teclado();                                                       // Seteo de la coordenada fila
        if (fila == SALIR) return FALSE;      // Comprobación de que se abandone la partida (FALSE porque no juega)
        output(" ");
        if (!turnoPerdido) outch(fila);     // Se muestra por pantalla la fila
    }

    while ( columna < COORD_MIN || columna > COORD_MAX ) {                         // Comprobación de columna válida
        if (turnoPerdido) return TRUE;                                              // Comprobación de que no ha perdido su turno
        output("\nIndica la columna donde quieras escribir \[1-9]: |   "); // Solicitud de coordenada columna del tablero
        columna = teclado();                                                       // Seteo de la coordenada columna
        if (columna == SALIR) return FALSE;      // Comprobación de que se abandone la partida (FALSE porque no juega)
        output(" ");
        if (!turnoPerdido) outch(columna);    // Se muestra por pantalla la columna
    }

    while ( numero < NUM_MIN || numero > NUM_MAX ) {                           // Comprobación de número válido
        if (turnoPerdido) return TRUE;                                              // Comprobación de que no ha perdido su turno
        // Solicitud del número a acertar según la posición indicada
        output("\nIndica el número que quieras escribir \[1-9]: |   ");
        numero = teclado();                                                       // Seteo del número elegido
        output(" ");
        if (numero == SALIR) return FALSE; // Comprobación de que se abandone la partida (FALSE porque no juega)
        if (!turnoPerdido) outch(numero);   // Se muestra por pantalla
        output("\n");
    }
    if (!turnoPerdido) {          // Si no hemos perdido el turno, se procede a escribir el número en la posición [fila-columna]
        compruebaMovimiento(fila, columna, numero, pJugador, pTabSudoku, pTabTurnos, pSolucion, pAyuda, pGanarPorTiempos, pPuntos);
    }
}

```

```
void compruebaMovimiento(char fila, char columna, char numero, char *pJugador, char *pTabSudoku, int *pTabTurnos, char *pSolucion, int *pAyuda, int *pGanarPorTiempos, int *pPuntos)

Comprueba que el número y la posición indicada el usuario cumplen las reglas del juego. En caso de que el movimiento no sea válido, se pide al jugador que lo intente de nuevo.

-----
void compruebaMovimiento(char fila, char columna, char numero, int *pJugador, char *pTabSudoku, int *pTabTurnos, char *pSolucion, int *pAyuda, int *pGanarPorTiempos, int *pPuntos)
{
    // A partir de la fila y columna se determina el offset (posición) del vector del tablero sudoku
    int posicion = (fila-'1')*TAM_SUDOKU_FILA + (columna-'1');

    if (*pTabSudoku+posicion != VACIO) {                                // Comprobación de que la posición a escribir se encuentra vacía
        char mensaje[] = {"Esa posición no está vacía.\n"};           // Se muestra mensaje por pantalla
        output("\n");
        formateaCadena(mensaje);
        if (turnoPerdido) return;                                         // Comprobación de que no se ha perdido el turno
        // Se repite el movimiento
        movimientoSudoku(pJugador, pTabSudoku, pTabTurnos, pSolucion, pAyuda, pGanarPorTiempos, pPuntos);
        return;
    }

    // Comprobación de que esa posición puede tomar ese valor según las reglas sudoku
    if (hayRepeticion(pTabSudoku, posicion, numero)) {
        char mensaje[] = {"Has puesto un número repetido. Vuelve a intentarlo.\n"}; // Se muestra mensaje por pantalla
        output("\n");
        formateaCadena(mensaje);
        if (turnoPerdido) return;                                         // Comprobación de que no se ha perdido el turno
        // Se repite el movimiento
        movimientoSudoku(pJugador, pTabSudoku, pTabTurnos, pSolucion, pAyuda, pGanarPorTiempos, pPuntos);
        return;
    } else {
        // Si todo ha ido bien, se procede a escribir en el tablero según posición y número
        escribeSudoku (posicion, numero, pJugador, pTabSudoku, pTabTurnos, pSolucion, pAyuda, pGanarPorTiempos, pPuntos);
    }
    return;
}
```

```
void escribeSudoku(char fila, char columna, char numero, char *pJugador, char *pTabSudoku, int *pTabTurnos, char *pSolucion, int *pAyuda, int *pGanarPorTiempos, int *pPuntos)

Escribe un número en el tablero sudoku según la fila y columna indicada.
-----
void escribeSudoku (int posicion, char numero, int *pJugador, char *pTabSudoku, int *pTabTurnos, char *pSolucion, int *pAyuda, int *pGanarPorTiempos, int *pPuntos) {

    BOOL hayAcierto = ((*pSolucion+posicion)-'0') == (numero-'0');           // Variable que indica si el número corresponde con la
solución
    BOOL haySolucion = ((*pSolucion+posicion)-'0') != (VACIO-'0');           // Variable que indica si existe la solución del
sudoku

    // Si está activado el modo ayuda (y el número coincide con la solución)
    // o bien el modo ayuda está desactivado, se rellena la posición.

    // Se comprueba la activación del modo ayuda y la relación de acierto o fallo
    if ( (*pAyuda == 1 && hayAcierto) || *pAyuda == 0 || (*pAyuda == 1 && !haySolucion)) {

        *(pTabSudoku+posicion) = numero;                                // Se escribe el número en el tablero sudoku

        if (*pAyuda == 1) {                                              // Comprobación de modo ayuda activado
            if (hayAcierto) { // Comprobación de que ha habido acierto
                char mensaje[] = {";Has acertado el número!\n"};          // Se muestra mensaje por pantalla
                formateaCadena(mensaje);
                actualizaPuntos(pJugador, pPuntos, pTabTurnos, posicion, pGanarPorTiempos); // En caso de
acierto, se suman los puntos al jugador
                espera();
            }
        }

    } else {
        // Caso en que se haya activado el modo ayuda y el número colocado sea erróneo
        if (*pAyuda == 1) {
            char mensaje[] = {";Has fallado el número!\n"};          // Se muestra mensaje por pantalla
            formateaCadena(mensaje);
            espera();
        }
        turnoPerdido = TRUE;                                         // En caso de que el modo ayuda esté activado y el número no coincida
        // con la solución, se pierde el turno.
    }
    return;
}
```

```
void loading(void)

Representa un proceso de carga mediante la escritura periódica de tres puntos. Será útil para reducir los mensajes escritos durante la espera.

-----
void loading(void) {

    BYTE i;                                // Índice de iteración
    BYTE numRepeticiones = 3;                // Definimos el número de repeticiones del esquema de escritura/borrado

    for (i = 0; i < numRepeticiones; i++) {   // Bucle de recorrido de cada repetición

        if (i != 0) output("\b\b\b");          // Mientras no sea la primera escritura, borramos los puntos anteriores
        output(".");
        retardo(100000);
        output("\b..");
        retardo(100000);
        output("\b\b...");
        retardo(100000);
    }
}
```

```
void sudokuTitulo(void)  
Muestra el titulo del juego de Sudo Kool Fire en ASCII-ART  
-----  
void sudokuTitulo(void) {  
  
output("\n /_____|      |____|      |____|      |____|      |____|      |____|\n");  
output("\n| \\(|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\n");  
output("\n\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\n");  
output("\n_____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\n");  
output("\n|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\\|____|\n");  
}  
}
```



```

        case 7:
            output("\n      _      _      \n      \(_)\n      | | |__| |\n      _ _ _ _ _ | | / /\n");
            output("\n| '_ \| \| \| / _ \| / / \n| | | | | \| v / __/ | / /\n|_|_|_|_|\ \|/\ \|_|_|\n");
            break;
        case 8:
            output("\n      _      _      \n      \(_)\n      | | / _ \| \n      _ _ _ _ _ | | | \(_)\n");
            output("\n| '_ \| \| \| / _ \| >_ < \n| | | | | \| v / __/ | | \(_)\n|_|_|_|_|\ \|/\ \|_|_\n");
            break;
        default:
            output("\n      _      _      \n      \(_)\n      | | / _ \| \n      _ _ _ _ _ | | | \(_)\n");
            output("\n| '_ \| \| \| / _ \| \\\_, |\n| | | | | \| v / __/ | / /\n|_|_|_|_|\ \|/\ \|_|_\n");
            break;
    }
    output("\n");
}

```



rally.h

Rally.h

Definición de alias y constantes

```
#define REFRESCO 100          // Refresco de pantalla en milisegundos
#define COCHE 0x06             // Definición del carácter de la carrocería del vehículo
#define RUEDAS 0x38            // Definición del carácter de las ruedas del vehículo
#define RASTRO 0x22            // Definición del carácter del rastro
#define ADELANTE 0x35          // Tecla de seguir recto
#define IZQUIERDA 0x34         // Tecla de girar a la izquierda
#define DERECHA 0x36           // Tecla de girar a la derecha
#define TAM_SUELO 50           // Tamaño del terreno donde se dibuja el circuito
#define ESPACIO 0x2E            // Definición del carácter del terreno
#define GRAVILLA 0x20           // Definición del carácter de calzada
#define NEUMATICO 0x40          // Definición del carácter de neumático
#define LINEA_META 0x3D          // Definición del carácter de línea de meta
#define LONG_CIRCUITO_MAX 200   // Longitud máxima del circuito
#define LONG_TRAMO 10           // Definición de la longitud de tramo (mayor de 5 recomendado)

#define LIMITE_RECTO 0x7C        // Definición del carácter de límite de tramo recto
#define LIMITE_IZQUIERDA 0x2F    // Definición del carácter de límite de tramo izquierdo
#define LIMITE_DERECHA 0x5C      // Definición del carácter de límite de tramo derecho
```

Definición de variables de ámbito global

```
struct tramo {                // Definición de la estructura de tramo de circuito
    int anchura;              // Anchura del tramo
    char limite;               // Límites izquierdo y derecho para representarlo
    char relleno;              // Carácter de relleno para diferenciar el tramo del terreno
};

struct tramo Recto = {        // Definición del tramo recto
    LONG_TRAMO,
    LIMITE_RECTO,
    GRAVILLA
};

struct tramo Izquierda = {    // Definición del tramo izquierdo
    LONG_TRAMO,
    LIMITE_IZQUIERDA,
    GRAVILLA
};
```

```
struct tramo Derecha = {           // Definición del tramo derecho
    LONG_TRAMO,
    LIMITE_DERECHA,
    GRAVILLA
};

struct tramo Meta = {             // Definición del tramo de meta
    LONG_TRAMO,
    LIMITE_RECTO,
    LINEA_META
};

struct circuito {                 // Definición de la estructura de circuito
    BYTE tramos[LONG_CIRCUITO_MAX+1]; // Definición del array que contiene la sucesión del tipo de tramos
    BYTE repeticiones[LONG_CIRCUITO_MAX+1]; // Definición del array que contiene el número de repeticiones de cada tipo de tramo
};

struct circuito Novatos = {       // Definición del circuito de Novatos
    {0,1,-1,1,0,0},
    {5,5,3,4,7,0}
};
struct circuito Monitores = {     // Definición del circuito de Monitores
    {0,1,0,1,0,-1,1,0,1,-1,1,0},
    {2,3,1,1,3,4,3,2,2,3,6,0}
};
struct circuito Profes = {        // Definición del circuito de Profes
    {0,1,-1,1,0,-1,1,0,-1,1,-1,0,-1,0,1,0,1,-1,0,-1,0,1,-1,0,-1},
    {2,3,2,4,3,6,3,1,8,5,2,3,2,1,3,1,7,3,2,6,4,14,3,30,4,5,7}
};
struct circuito Masters = {      // Definición del circuito de Masters
    {0,1,0,-1,0,-1,1,0,1,0,1,-1,0,-1,0,1,0,-1,1,0,-1,1,0,-1,1,-1,1,0,-1,0,1,0,0},
    {3,4,2,4,3,2,7,3,3,4,2,2,3,6,3,3,4,2,3,4,1,8,11,4,3,6,1,8,5,2,2,6,3,7,2,3,4,5,4,1,2,8,3,0}
};

Definición de cabeceras de funciones
-----
void iniciaRally(struct circuito Circuito, int velocidad);

void juegoRally(struct circuito Circuito, int velocidad, char *pSuelo, int *pOrigen, int *pTramo, int *pNumRep, int *pPosCoche, int *pMeta, int *pFinRepeticion);

void actualizaPosicion(struct circuito Circuito, char *pSuelo, int *pOrigen, int *pTramo, int *pPosCoche, char direccion);

void inicializaSuelo(char *pSuelo);
```

```
void situaTramo(struct tramo Tramo, char *pSuelo, int *pOrigen);

void sigueCircuito(struct Circuito Circuito, char *pSuelo, int *pOrigen, int *pTramo, int *pNumRep, int *pMeta, int miroFuturo, int *pFinRepeticion);

void decrementaRepeticion (struct Circuito Circuito, int *pTramo, int *pNumRep, int miroFuturo, int *pFinRepeticion);

BOOL compruebaMeta(struct Circuito Circuito, char *pSuelo, int *pOrigen, int *pTramo, int *pNumRep, int *pPosCoche);

void muestraTramo(char *pSuelo);

BOOL detectaColision(char *pSuelo, int *pPosCoche, char direccion);

void setCircuito (int velocidad);

int setVelocidad();

void setConfig();

void limpiaCoche(char *pSuelo, int *pPosCoche);

void disparaSalida(void);

void situaVehiculo(char *pSuelo, int *pPosCoche);

void borraVehiculo(char *pSuelo, int *pPosCoche);

void muestraTramoFuturo(struct Circuito Circuito, char *pSuelo, int *pOrigen, int *pTramo, int *pNumRep, int *pMeta, int *pFinRepeticion);

void mensajeFinal(int meta);
```



rally.c

Rally.c

```
#include "rally.h"
#include "interfaz.h"
```

```

void iniciaRally(struct Circuito Circuito, int velocidad)
Función que crea las variables y punteros necesarias para el funcionamiento del juego.
-----
void iniciaRally(struct Circuito Circuito, int velocidad) {

    char suelo[TAM_SUELO];
    char *pSuelo = &suelo[0];
                                                // Creación de la variable suelo. Contendrá la carretera y el terreno
                                                // Creación del puntero que apunta al vector suelo

    int origen = (TAM_SUELO/2) - LONG_TRAMO;
    int *pOrigen = &origen;
                                                // Variable que indica la posición comienzo del tramo dentro del suelo
                                                // Creación del puntero que apunta al origen (principio) del tramo

    int tramo = 0;
                                                // Variable que controla el tipo de tramo, si es recto, izquierda o
derecha
    int *pTramo = &tramo;
                                                // Creación del puntero que apunta al tipo de tramo

    int numRep = Circuito.repeticiones[tramo] - 1;
    int *pNumRep = &numRep;
                                                // Variable que contiene el número de repeticiones del tipo de tramo
                                                // Creación del puntero que apunta al número de repeticiones de tramo

    int posCoche = origen + (LONG_TRAMO/2);
                                                // Variable que contiene la posición del coche en el terreno.

Inicialmente
    int *pPosCoche = &posCoche;
                                                // se encuentra en el centro de la calzada.
                                                // Creación del puntero que apunta a la posición del coche

    int meta = 0;
    int *pMeta = &meta;
                                                // Creación de la variable que indica si se alcanza la meta
                                                // Creación del puntero que apunta al flag de meta

    int finRepeticion = 0;
tipo de tramo
    int *pFinRepeticion = &finRepeticion;
                                                // Creación del flag que indica si han acabado las repeticiones de un
                                                // Creación del puntero que apunta al flag de fin de repeticiones

    seteaRefresco(0);
    inicializaSuelo(pSuelo);
                                                // Reset del tiempo de refresco
                                                // Se inicializa el terreno (sin calzada aún)

    situaVehiculo(pSuelo,pPosCoche);
                                                // Si sitúa el coche en la calzada

    disparoSalida();
                                                // Se hace la cuenta atrás "preparados, listos, ya"

    // Se llama a la función principal de juego Rally
    juegoRally(Circuito, velocidad, pSuelo, pOrigen, pTramo, pNumRep, pPosCoche, pMeta, pFinRepeticion);
}

```

```

void juegoRally(struct Circuito Circuito, int velocidad, char *pSuelo, int *pOrigen, int *pTramo, int *pNumRep, int *pPosCoche, int
*pMeta, int *pFinRepeticion)

Función principal del juego. Gestiona la actualización del tramo visible por el jugador, y el movimiento del jugador.
-----
void juegoRally(struct Circuito Circuito, int velocidad, char *pSuelo, int *pOrigen, int *pTramo, int *pNumRep, int *pPosCoche, int
*pMeta, int *pFinRepeticion) {

    int j = 0;                                // Índice que indicará el número de tramo representado.

    while(TRUE) {                             // Bucle principal de mostrar el tramo de circuito

        char direccion;                      // Creación de la variable para guardar la dirección tomada por el usuario
        BOOL hayColision = FALSE;             // Flag que detecta si ha habido una colisión

        finRefresco = FALSE;                  // Reseteo del flag de final de refresco
        seteaRefresco(REFRESCO*velocidad);    // Se setea el tiempo hasta que aparezca el siguiente tramo

        direccion = teclado();               // Esperamos que el usuario mueva el coche

        if (j > 0) borraVehiculo(pSuelo, pPosCoche);      // En caso de no ser el primer tramo, se borra el vehículo del
                                                               // tramo anterior y se pinta el rastro

        inicializaSuelo(pSuelo);              // Vaciado del escenario (suelo) de trabajo
        // Se tramita el siguiente tramo
        sigueCircuito(Circuito, pSuelo, pOrigen, pTramo, pNumRep, pMeta, 0, pFinRepeticion);

        hayColision = detectaColision(pSuelo, pPosCoche, direccion); // Comprobación de existencia de colisión del coche

        // Se actualiza la posición del coche según haya indicado el usuario
        actualizaPosicion(Circuito, pSuelo, pOrigen, pTramo, pPosCoche, direccion);

        // En caso de que se llegue a la meta, se sale del bucle principal
        if (compruebaMeta(Circuito, pSuelo, pOrigen, pTramo, pNumRep, pPosCoche)) break;

        muestraTramo(pSuelo);                // Se imprime por pantalla el tramo actual
        // Se imprime el próximo tramo
        muestraTramoFuturo(Circuito, pSuelo, pOrigen, pTramo, pNumRep, pMeta, pFinRepeticion);

        if (hayColision) {                  // En caso de colisión, se imprime un mensaje final
            mensajeFinal(0);
            return;
        }
        j = 1;
    }
}

```

```
    }
    mensajeFinal(1);           // En caso de llegar a meta, se imprime el mensaje correspondiente.
    return;
}
```

```
void setConfiguracionRally(void)  
Función principal de configuración del juego. Permitirá seleccionar una velocidad y un circuito.  
-----  
void setConfiguracionRally(void) {  
  
    int velocidad;                      // Variable que almacena la velocidad indicada por el usuario  
  
    output("\nBienvenido a LSED Rally\n");   // Mensaje de bienvenida del juego  
  
    velocidad = setVelocidad();           // Configuración de la velocidad  
    setCircuito(velocidad);               // Configuración del circuito de juego  
    return;  
}
```

```
int setVelocidad(void  
Esta función obtiene el dato de velocidad elegido por el usuario.  
-----  
int setVelocidad(void) {  
  
    BYTE velocidad;                                // Variable que almacenará la velocidad introducida por el usuairo  
    do {  
        output("\nEscoge la velocidad que deseas: \[1 Supersónico | 9 Caracol]: ");      // Mensaje de elección  
        velocidad = teclado() - '0';                // Captura de la velocidad indicada por el usuario  
        outNum(10, velocidad, SIN_SIGNO);            // Se muestra la velocidad elegida por pantalla  
    } while (velocidad < 1 || velocidad > 9);      // Comprobación de respuesta válida  
  
    return velocidad;  
}
```

```
void setCircuito (int velocidad)

Función que configura el circuito elegido por el usuario.
-----
void setCircuito (int velocidad) {

    BYTE circuito;           // Variable que almacena el circuito elegido por el usuario

    do {
        output("\nEscoge la copa que quieras jugar:");
        output("\n\n[1 Copa Novatos LSED ]");
        output("\n\n[2 Copa Monitores LSED]");
        output("\n\n[3 Copa Profes LSED ]");
        output("\n\n[4 Copa Masters LSED ]");
        circuito = teclado() - '0';
        outNum(10, circuito, SIN_SIGNO);
        output("\n");

    } while (circuito < 1 || circuito > 4);           // Comprobación de respuesta válida

    switch(circuito){

        case 1:
            iniciaRally(Novatos, velocidad);           // Inicialización de la copa Novatos LSED
            break;
        case 2:
            iniciaRally(Monitores, velocidad);          // Inicialización de la copa Monitores LSED
            break;
        case 3:
            iniciaRally(Profes, velocidad);             // Inicialización del de la copa Profes LSED
            break;
        case 4:
            iniciaRally(Masters, velocidad);            // Inicialización del de la copa Masters LSED
            break;
        default:
            break;
    }
}
```

```
void actualizaPosicion(struct circuito Circuito, char *pSuelo, int *pOrigen, int *pTramo, int *pPosCoche, char dirección)
-----
void actualizaPosicion(struct circuito Circuito, char *pSuelo, int *pOrigen, int *pTramo, int *pPosCoche, char dirección) {
    switch (dirección) {                                // Se comprueba la dirección introducida por el jugador
        case IZQUIERDA:                               // Se mueve el coche una posición a la izquierda
            *pPosCoche -= 1;
            situaVehiculo(pSuelo, pPosCoche);
            break;
        case DERECHA:                                // Se mueve el coche una posición a la derecha
            *pPosCoche += 1;
            situaVehiculo(pSuelo, pPosCoche);
            break;
        default:                                     // Se mantiene al coche en su posición original
            situaVehiculo(pSuelo, pPosCoche);
            break;
    }
    return;
}
```

```
void inicializaSuelo(char *pSuelo)
Rellena el suelo con el caracter del terreno (reseteo del suelo)
-----
void inicializaSuelo(char *pSuelo) {
    BYTE indice;                                // Variable para iterar el suelo
    for (indice = 0; indice < TAM_SUELO; indice++) { // Bucle de iteración del vector de suelo
        *(pSuelo+indice) = ESPACIO;              // Reseteo de todos los valores al carácter del terreno
    }
    return;
}
```

```
void situaTramo(struct tramo Tramo, char *pSuelo, int *pOrigen)

Escribe en el vector de suelo un tramo de calzada con sus límites izquierdo y derecho, según sea la dirección del mismo.
-----
void situaTramo(struct tramo Tramo, char *pSuelo, int *pOrigen) {

    BYTE indice = 0;
    int longitud = Tramo.anchura+1;                                // Variable que controla la longitud del tramo

    while (longitud > 0) {                                         // Comprobamos que no se haya pintado todo el tramo

        if (indice == *pOrigen+Tramo.anchura) {                     // Pintar segundo límite de tramo
            *(pSuelo+indice) = Tramo.limite;
            *(pSuelo+indice+1) = Tramo.limite;
            *(pSuelo+indice+2) = NEUMATICO;
            longitud--;
        }
        if (indice > *pOrigen && indice < *pOrigen+Tramo.anchura) { // Pintar relleno del tramo
            *(pSuelo+indice) = Tramo.relleno;
            longitud--;
        }

        if (indice == *pOrigen) {                                     // Pintar el primer límite de tramo
            *(pSuelo+indice-2) = NEUMATICO;
            *(pSuelo+indice-1) = Tramo.limite;
            *(pSuelo+indice) = Tramo.limite;
            longitud--;
        }
        indice++;
    }
    return;
}
```

```
void sigueCircuito(struct Circuito Circuito, char *pSuelo, int *pOrigen, int *pTramo, int *pNumRep, int *pMeta, int miroFuturo, int *pFinRepeticion)

Esta función se encarga de gestionar el siguiente tramo a mostrar por pantalla, decrementando el número de repeticiones del mismo.
-----
void sigueCircuito(struct Circuito Circuito, char *pSuelo, int *pOrigen, int *pTramo, int *pNumRep, int *pMeta, int miroFuturo, int *pFinRepeticion) {

    // Se obtiene el offset que genera el siguiente tramo, que puede ser izquierda, derecha o recto
    int siguienteTramo = Circuito.tramos[*pTramo];
    *pOrigen += siguienteTramo;                                // Actualizamos el comienzo de la calzada en el vector suelo

    switch (siguienteTramo) {                                    // Analizamos según el tipo de tramo siguiente

        case -1:
            situaTramo(Izquierda, pSuelo, pOrigen);           // Caso en que el tramo sea hacia la izquierda
            if (miroFuturo == 1) {                             // Comprobación de que estábamos observando un tramo futuro
                *pOrigen = *pOrigen + 1;                      // Restablecimiento del puntero origen
            }
            break;
        case 1:
            situaTramo(Derecha, pSuelo, pOrigen);           // Caso en que el tramo sea hacia la derecha
            if (miroFuturo == 1) {                             // Comprobación de que estábamos observando un tramo futuro
                *pOrigen = *pOrigen - 1;                      // Restablecimiento del puntero origen
            }
            break;
        default:
            situaTramo(Recto, pSuelo, pOrigen);             // Caso en que el tramo sea recto
            if (miroFuturo == 1) {                             // Comprobación de que estábamos observando un tramo futuro
                *pOrigen = *pOrigen - 1;                      // Restablecimiento del puntero origen
            }
            break;
    }
    decrementaRepeticion(Circuito, pTramo, pNumRep, miroFuturo, pFinRepeticion); // Se decrementa el número de repeticiones
    return;
}
```

```
void decrementaRepeticion( struct circuito Circuito, int *pTramo, int *pNumRep, int miroFuturo, int *pFinRepeticion)

Se decrementa el número de repeticiones de tramo durante el desarrollo del circuito
-----
void decrementaRepeticion (struct circuito Circuito, int *pTramo, int *pNumRep, int miroFuturo, int *pFinRepeticion) {

    if (miroFuturo == 0) {                                // Comprobación de que no estamos visualizando un tramo futuro
        if (*pNumRep != 0) {                            // En caso de que aún queden repeticiones de tramo, se decrementa en uno
            *pNumRep -= 1;
        } else {
            *pFinRepeticion = 1;           // Se activa el flag de fin de repeticiones
            *pTramo += 1;      // Se incrementa la variable tramo para localizar la dirección del siguiente tramo
            *pNumRep = Circuito.repeticiones[*pTramo] - 1;    // Actualización del número de repeticiones de tramo
        }
    }
}
```

```
BOOL compruebaMeta(struct Circuito Circuito, char *pSuelo, int *pOrigen, int *pTramo, int *pNumRep, int *pPosCoche)

Función que comprueba si se ha llegado al final del circuito, y en caso afirmativo, muestra por pantalla la línea de meta.
-----
BOOL compruebaMeta(struct Circuito Circuito, char *pSuelo, int *pOrigen, int *pTramo, int *pNumRep, int *pPosCoche) {

    // Comprobación de que se ha llegado al final del circuito (tramo de tipo 0 con número de repeticiones 0)
    if ( *pNumRep == 0 && Circuito.repeticiones[(*pTramo)+1] == 0 ) {

        situaTramo(Meta, pSuelo, pOrigen); // Se sitúa el tramo meta al final del circuito
        situaVehiculo(pSuelo, pPosCoche); // Se sitúa al vehículo en la meta manteniendo su posición anterior
        muestraTramo(pSuelo); // Se imprime la línea de meta por pantalla
        retardo(200000); // Se produce una pequeña espera para mostrar más adelante el mensaje de meta
        return TRUE; // Fin del circuito
    }
    return FALSE;
}
```

```
void muestraTramo(char *pSuelo)
    Imprime por pantalla la situación del vector suelo, que contiene el terreno, la calzada y el vehículo.
-----
void muestraTramo(char *pSuelo) {
    BYTE indice;                                // Variable para iterar el vector suelo
    for (indice = 0; indice < TAM_SUELO; indice++) { // Bucle de recorrido del terreno
        outch(*(pSuelo+indice));                // Muestra por pantalla cada elemento del suelo
    }
    output("\n");
}
```

```
BOOL detectaColision(char *pSuelo, int *pPosCoche, char direccion)          *
Esta función se encarga de detectar la posible colisión del coche con los límites del circuito. Para ello detecta cuando el coche sale
de la calzada y toca la segunda linea de arcén.
-----
BOOL detectaColision(char *pSuelo, int *pPosCoche, char direccion) {
    BYTE dir_num;                      // Se crea la variable que traduce la dirección del usuario a offset en la calzada
    switch (direccion) {
        case IZQUIERDA:                // Caso en que el vehículo giró a la izquierda
            dir_num = -1;
            break;
        case DERECHA:                 // Caso en que el vehículo giró a la derecha
            dir_num = 1;
            break;
        default:                      // Caso en que no haya giro
            dir_num = 0;
            break;
    }
    // Comprobación de que el coche permanece en la calzada
    return ( *(pSuelo+(*pPosCoche) + dir_num)) != GRAVILLA );
}
```

```
void disparoSalida(void)

Se escribe una cuenta atrás inicial para preparar al jugador al ritmo de "3, 2, 1, GO!"

-----
void disparoSalida(void) {

    BYTE i;
    output("\n\n");
    for (i = 3; i > 0; i--) {
        output("\b");                                // Bucle de iteración de número
        outNum(10, i, SIN_SIGNO);                   // Se borra el anterior carácter
        retardo(500000);                            // Se muestra el número de cuenta atrás
                                                // Producimos un pequeño retardo
    }
    output("\bGO!\n");                            // Mostramos el mensaje de disparo
}
```

```
void situaVehiculo(char *pSuelo, int *pPosCoche
```

Esta función escribe el vehículo en una determinada posición. El vehículo consta de una carrocería que se representa por un rectángulo, y los cuatro neumáticos laterales.

```
void situaVehiculo(char *pSuelo, int *pPosCoche) {
```

```
    *(pSuelo + *pPosCoche-1) = RUEDAS;  
    *(pSuelo + *pPosCoche) = COCHE;  
    *(pSuelo + *pPosCoche+1) = RUEDAS;
```

```
    // Se escribe el carácter de los neumáticos izquierdos  
    // Se escribe el carácter de la carrocería del vehículo  
    // Se escribe el carácter de los neumáticos derechos
```

```
}
```

```
void borraVehiculo(char *pSuelo, int *pPosCoche)
```

Esta función escribe un rastro del coche en los tramos del circuito que van quedando atrás. Para ello se borra el tramo actual y el tramo futuro, para luego escribir el tramo con el rastro del coche.

```
-----  
void borraVehiculo(char *pSuelo, int *pPosCoche) {  
  
    BYTE i;                                // Variable para recorrer el doble borrado  
  
    for (i = 0; i < 2*(TAM_SUELO+1); i++) {    // Bucle de doble borrado de suelo (tramo actual y tramo futuro)  
        output("\b");  
    }  
    *(pSuelo+*pPosCoche) = RASTRO;           // Se escribe un rastro del coche tras avanzar  
    muestraTramo(pSuelo);                    // Se imprime por pantalla el tramo anterior con rastro  
    *(pSuelo+*pPosCoche) = COCHE;            // Se restablece la posición del coche para imprimir correctamente el próximo tramo  
}
```

```
void muestraTramoFuturo( struct Circuito Circuito, char *pSuelo, int *pOrigen, int *pTramo, int *pNumRep, int *pMeta, int
*pFinRepeticion)

Esta función imprime por pantalla el siguiente tramo del circuito, para ayudar al usuario a predecir la dirección de la calzada.
-----
void muestraTramoFuturo(struct Circuito Circuito, char *pSuelo, int *pOrigen, int *pTramo, int *pNumRep, int *pMeta, int
*pFinRepeticion) {

    inicializaSuelo(pSuelo);                                     // Se inicializa el terreno
    // Análisis del próximo tramo a dibujar
    sigueCircuito(Circuito, pSuelo, pOrigen, pTramo, pNumRep, pMeta, 1, pFinRepeticion);
    muestraTramo(pSuelo);                                       // Se imprime el tramo por pantalla
}
```

```
void mensajeFinal(int meta)
Se imprime un mensaje final por pantalla en función de cómo haya acabado la partida.
-----
void mensajeFinal(int meta) {

    switch(meta) {                                // Comprobación de si se ha llegado a meta

        case 0:
            output("\n;Ha habido un choque!\n");
            break;                                  // Mensaje para el caso en que se detecte colisión

        case 1:
            output("\n;Has llegado a la meta!\n"); // mensaje para el caso en que se pise la línea de final de circuito
            break;

        default:
            break;
    }
    retardo(200000);                            // Introducción de un pequeño retardo
    finRefresco = FALSE;                        // Reseteo del flag de fin de refresco de pantalla
    seteaRefresco(0);                           // Reseteo del tiempo de refresco de pantalla
}
```



tftp_main.c

```
Tftp_main.c

#include "stdlib.h"
#include "m5272c3.h"
#include "nif.h"
#include "fec.h"
#include "arp.h"
#include "ip.h"
#include "udp.h"
#include "tftp.h"
#include "timer.h"
#include "tftp_main.h"
#include "m5272lib.h"
#include "mcf5xxx.h"

#include "m5272lib.c"
#include "stdlib.c"
#include "timer.c"

#include "fec.c"
#include "arp.c"
#include "ip.c"
#include "tftp.c"

/* Define an interface to the FEC */
NIF          fec_nif;
IP_INFO      ip_info;
ARP_INFO arp_info;

//RSSH: #define USERSPACE (SDRAM_ADDRESS + 0x40000)
#define USERSPACE (SDRAM_ADDRESS + 0x0000)

static char input[MAX_LINE];
static const char PROMPT[] = "TFTP> ";
static const char HELPMMSG[] = "\nEnter 'help' for help.\n\n";
static const char SYNTAX[] = "Error: Invalid syntax for: %s\n";
static const char INVCMD[] = "Error: No such command: %s\n";
static const char INVALUE[] = "Error: Invalid value: %s\n";
static const char HELPFORMAT[] = "%8s %-30s %s %s\n";
static const char INVOPT[] = "Error: Invalid set/show option: %s\n";
static const char OPTFMT[] = "%12s: ";
static const char MACFMT[] = "%02X:%02X:%02X:%02X:%02X:%02X\n";
static const char SETERR[] = "Error: Invalid MAC address: %s\n";
```

```
static const char IPFMT[] = "%d.%d.%d.%d\n";
static const char SETERR1[] = "Error: Invalid IP address: %s\n";
static IP_ADDR client = {192,168,0,240};
static IP_ADDR gateway = {192,168,0,1};
static IP_ADDR netmask = {255,255,255,0};
static IP_ADDR server = {192,168,0,37};
static unsigned char mac[6] = {0x00, 0x0B, 0xCB, 0xFF, 0xF0, 0x00};
static char filename[MAX_FN] = "sudoku.txt";
uint32 md_last_address;

CMD CMDTAB[] =
{
    {"help", 0, 0, help, "Display this help message", ""},
    {"read", 0, 1, read, "Read from TFTP host", "<filename>" },
    {"md", 0, 1, md, "Memory display", "<address>" },
    {"set", 0, 2, set, "Set parameter", "option value" },
    {"show", 0, 0, show, "Show parameters", "" },
    {"write", 1, 2, write, "Write to TFTP host", "bytes <filename>" },
    {"quit", 0, 0, quit, "Exit TFTP", " " }
};
const int NUM_CMD = sizeof(CMDTAB)/sizeof(CMD);

SETCMD SETCMTAB[] =
{
    {"server", set_server, "<server IP address>" },
    {"client", set_client, "<board IP address>" },
    {"gateway", set_gateway, "<gateway IP address>" },
    {"netmask", set_netmask, "<netmask IP address>" },
    {"mac", set_mac, "<ethernet address>" },
    {"filename", set_filename, "<file to transfer>" }
};
const int NUM_SETCMD = sizeof(SETCMTAB)/sizeof(SETCMD);
```

```
void main_tftp (void)
{
    md_last_address = USERSPACE;

    output("\n\n");
    output("Running TFTP Network Stack\n");
    //output("Built on %s %s\n", __DATE__, __TIME__);
    output("Built on DATE TIME\n");

    #if (DEBUG)
        output("DEBUG prints ON\n");
    #endif

    output("\nEscriba 'help' para desplegar el menu de ayuda.\n\n");

    while (1)
    {
        output("TFTP> ");
        run_cmd();
    }
}
```

```
static void
get_user_input (char *userline)
{
    char line[MAX_LINE];
    int pos, ch;

    pos = 0;

    ch = (int)inch();
    while ((ch != 0x0D /* CR */) &&
           (ch != 0x0A /* LF/NL */) &&
           (pos < MAX_LINE))
    {
        switch (ch)
        {
            case 0x08:          /* Backspace */
            case 0x7F:          /* Delete */
                if (pos > 0)
                {
                    pos -= 1;
                    outch(0x08);      /* backspace */
                    outch(' ');
                    outch(0x08);      /* backspace */
                }
                break;
            default:
                if ((pos+1) < MAX_LINE)
                {
                    /* only printable characters */
                    if ((ch > 0x1f) && (ch < 0x80))
                    {
                        line[pos++] = (char)ch;
                        outch((char)ch);
                    }
                }
                break;
        }

        ch = (int)inch();
    }

    outch(0x0D);      /* CR */
    outch(0x0A);      /* LF */
}
```

```
if (!pos && (strncasecmp(userline,"md",2) == 0))
{
    /* Allow 'md' command to be repeated */
}
else
{
    line[pos] = '\0';
    strcpy(userline,line);
}
```

```
static int
make_argv (char * cmdline, char * argv[])
{
    int argc, i, in_text;

    /* break cmdline into strings and argv */
    /* it is permissible for argv to be NULL, in which case */
    /* the purpose of this routine becomes to count args */
    argc = 0;
    i = 0;
    in_text = FALSE;
    while (cmdline[i] != '\0') /* getline() must place 0x00 on end */
    {
        if (((cmdline[i] == ' ') || (cmdline[i] == '\t')) )
        {
            if (in_text)
            {
                /* end of command line argument */
                cmdline[i] = '\0';
                in_text = FALSE;
            }
            else
            {
                /* still looking for next argument */
            }
        }
        else
        {
            /* got non-whitespace character */
            if (in_text)
            {
            }
            else
            {
                /* start of an argument */
                in_text = TRUE;
                if (argc < MAX_ARGS)
                {
                    if (argv != NULL)
                        argv[argc] = &cmdline[i];
                    argc++;
                }
            }
        }
    }
}
```

```
        else
            /*return argc;*/
            break;
    }

}
i++;      /* proceed to next character */
}
if (argv != NULL)
    argv[argc] = NULL;
return argc;
}
```

```
static uint32
get_value (char *s, int *success, int base)
{
    uint32 value;
    char *p;

    value = strtoul(s,&p,base);
    if ((value == 0) && (p == s))
    {
        *success = FALSE;
        return 0;
    }
    else
    {
        *success = TRUE;
        return value;
    }
}
```

```
void
run_cmd (void)
{
    int argc;
    char *argv[MAX_ARGS + 1]; /* One extra for NULL terminator */

    get_user_input(input);

    argc = make_argv(input, argv);

    if (argc)
    {
        int i;
        for (i = 0; i < NUM_CMD; i++)
        {
            if (strcasecmp(CMDTAB[i].cmd, argv[0]) == 0)
            {
                if (((argc-1) >= CMDTAB[i].min_args) &&
                    ((argc-1) <= CMDTAB[i].max_args))
                {
                    CMDTAB[i].func(argc, argv);
                    return;
                }
                else
                {
                    output("Error de sintaxis");
                    output(argv[0]);
                    output("\n");
                    return;
                }
            }
        }
        output("Error: no existe el comando:");
        output(argv[0]);
        output("\n");
        output("\nEscriba 'help' para desplegar el menu de ayuda.\n\n");
    }
}
```

```
void
help (int argc, char **argv)
{
    int index;

    (void)argc;
    (void)argv;

    output("\n");
    for (index = 0; index < NUM_CMD; index++)
    {
        output("\n");
        output("Nombre del comando: ");
        output(CMDTAB[index].cmd);
        output("\n");
        output("Descripcion del comando: ");
        output(CMDTAB[index].description);
        output("\n");
        output("Parametros: ");
        output(CMDTAB[index].syntax);
        output("\n");
    }
    output("\n");
}
```

```
void
read (int argc, char **argv)
{
    uint32 addr = USERSPACE;
    char *fn;
    int ch;

    if (argc == 2)
        fn = filename;
        //fn = (char *)&argv[1][0];
    else
        fn = filename;

    /* Initialize the timer for network use */
    timer_init(TIMER_NETWORK, TIMER_NETWORK_PERIOD, SYSTEM_CLOCK, TIMER_NETWORK_LEVEL);

    /* Enable FEC interrupts to ColdFire core */
    mbar_writeLong(MCFSIM_ICR3,MCF5272_SIM_ICR_ERX_IL(FEC_LEVEL));

    /* Initialize network device */
    fec_init(&fec_nif);

    /* Write ethernet address in the NIF structure */
    fec_nif.hwa[0] = mac[0];
    fec_nif.hwa[1] = mac[1];
    fec_nif.hwa[2] = mac[2];
    fec_nif.hwa[3] = mac[3];
    fec_nif.hwa[4] = mac[4];
    fec_nif.hwa[5] = mac[5];

    /* Initialize Network Buffers */
    nbuf_init();

    /* Initialize ARP */
    arp_init(&arp_info);
    nif_bind_protocol(&fec_nif,FRAME_ARP,(void *)arp_handler,(void *)&arp_info);

    /* Initialize IP */
    ip_init(&ip_info,client,gateway,netmask);
    nif_bind_protocol(&fec_nif,FRAME_IP,(void *)ip_handler,(void *)&ip_info);

    /* Initialize UDP */
    udp_init();
```

```
output("Reading ");
output(fn);
output(" desde: ");
outNum(10,server[0],SIN_SIGNO);
output(".");
outNum(10,server[1],SIN_SIGNO);
output(".");
outNum(10,server[2],SIN_SIGNO);
output(".");
outNum(10,server[3],SIN_SIGNO);
output(" al espacio de usuario: ");
outNum(16,addr,SIN_SIGNO);
output("\n");

/* Open the TFTP connection */
if (tftp_read(&fec_nif, fn, server))
{
    while (1)
    {
        // ch = tftp_inch();
        ch = tftp_in_char();

        if (ch == -1)
        {
            /* We're done */
            break;
        }
        else
        {
            *(unsigned char *)addr++ = (unsigned char)ch;
        }
    }

    /* Close the TFTP connection */
    tftp_end(TRUE);
}
}
```

```
void
write (int argc, char **argv)
{
    uint32 bytes;
    int success;
    char *fn;

    bytes = get_value(argv[1],&success,10);
    if (!success)
    {
        output("Error: valor no valido: ");
        output(argv[1]);
        output("\n");
        return;
    }

    if (argc == 3)
        fn = (char *)&argv[2][0];
    else
        fn = filename;

    /* Initialize the timer for network use */
    timer_init(TIMER_NETWORK, TIMER_NETWORK_PERIOD,
               SYSTEM_CLOCK, TIMER_NETWORK_LEVEL);

    /* Enable FEC interrupts to ColdFire core */
    mbar_writeLong(MCFSIM_ICR3,MCF5272_SIM_ICR_ERX_IL(FEC_LEVEL));

    /* Initialize network device */
    fec_init(&fec_nif);

    /* Write ethernet address in the NIF structure */
    fec_nif.hwa[0] = mac[0];
    fec_nif.hwa[1] = mac[1];
    fec_nif.hwa[2] = mac[2];
    fec_nif.hwa[3] = mac[3];
    fec_nif.hwa[4] = mac[4];
    fec_nif.hwa[5] = mac[5];

    /* Initialize Network Buffers */
    nbuf_init();

    /* Initialize ARP */
    arp_init(&arp_info);
```

```
nif_bind_protocol(&fec_nif, FRAME_ARP, (void *)arp_handler, (void *)&arp_info);

/* Initialize IP */
ip_init(&ip_info, client, gateway, netmask);
nif_bind_protocol(&fec_nif, FRAME_IP, (void *)ip_handler, (void *)&ip_info);

/* Initialize UDP */
udp_init();

/*output("Sending data from 0x%08X to 0x%08X to %d.%d.%d.%d\n",
       USERSPACE, USERSPACE + bytes,
       server[0], server[1], server[2], server[3]);
*/
output("\nEnviando los datos almacenados entre el comienzo ");
output("\nde la direccion del espacio de usuario \n");
output(" al espacio de usuario mas los bytes a enviar");
output("\n");
output(" a la direccion: ");
outNum(10,server[0],SIN_SIGNO);
output(".");
outNum(10,server[1],SIN_SIGNO);
output(".");
outNum(10,server[2],SIN_SIGNO);
output(".");
outNum(10,server[3],SIN_SIGNO);
output("\n");

output("Nombre del fichero remoto: ");
output(fn);
output("\n");

}
tftp_write(&fec_nif, fn, server, USERSPACE, USERSPACE + bytes);
```

```
void
quit (int argc, char **argv)
{
    (void)argc;
    (void)argv;

    output ("\nExiting TFTP\n");
    while (1)
        ;
}
```

```
static void
dump_mem (uint32 begin, uint32 end)
{
    uint32 data;
    uint32 curr;
    char line[16];
    char *lcur;
    int i, ch, j;
    BOOL escribeSolucion = FALSE;

    curr = begin;
    j = 0;

    do
    {
        output("\n");
        output("Posicion actual de memoria: ");
        outNum(16,curr,SIN_SIGNO);
        output("\n");
        output("\n");

        lcur = line;
        i = 0;

        while (i < 16)
        {
            data = *((uint32 *)curr);
            output("El dato almacenado es: ");
            outNum(16,data,SIN_SIGNO);
            output("\n");

            *(uint32 *)lcur = data;
            curr += 4;
            lcur += 4;
            i += 4;
        }

        output("\n");
        output("Que se traduce por:");
        for (i = 0; i < 16; i++)
        {
            ch = line[i];
```

```
    if ((ch >= ' ') && (ch <= '~'))  
        outch(ch);  
    else  
        output(".");  
        //output("\n");  
  
    if ( escribeSolucion ) {  
        if ( (i + j*16) == 0) {  
            //sudokuRed.inicial[80] = ch;  
        } else {  
            //sudokuRed.solucion[(i + j*16 - 1)] = ch;  
        }  
    } else {  
        //sudokuRed.inicial[(i + j*16)] = ch;  
        if ( (i+j*16) == 79 ) {  
            escribeSolucion = TRUE;  
            j = -1;  
        }  
    }  
}  
output("\n");  
j++;  
}  
while (curr < end);  
}
```

```
void
md (int argc, char **argv)
{
    int success;
    uint32 begin, end;

    if (argc > 1)
    {
        begin = get_value(argv[1],&success,16);
        if (!success)
        {
            output("Error:valor no válido: ");
            output(argv[1]);
            output("\n");
            return;
        }
        begin = begin & 0xFFFFFFFFFC;
        end = begin + (8 * 16);    //
    }
    else
    {
        begin = md_last_address;
        end = begin + (11 * 16);    //
    }

    dump_mem(begin,end);
    md_last_address = end;
}
```

```
static int
ip_convert_address (char *ipstring, uint8 ipaddr[])
{
    int i,j,k,l;

    l = strlen(ipstring);

    for (i = 0; i < l; i++)
    {
        if ((ipstring[i] != '.') && ((ipstring[i] < '0') ||
                                       (ipstring[i] > '9')))
        {
            return 0;
        }
    }
    /* the characters in the string are at least valid */
    j = 0;
    k = 0;
    for (i = 0; i < l; i++)
    {
        if (ipstring[i] != '.')
        {
            if (++j > 3)      /* number of digits in portion */
                return 0;
        }
        else
        {
            if (++k > 3)      /* number of periods */
                return 0;

            if (j == 0)          /* number of digits in portion */
                return 0;
            else
                j = 0;
        }
    }

    /* at this point, all the pieces are good, form ip_address */
    k = 0; j = 0;
    for (i = 0; i < l; i++)
    {
        if (ipstring[i] != '.')
        {
            k = (k * 10) + ipstring[i] - '0';
        }
    }
}
```

```
        }
        else
        {
            ipaddr[j++] = (unsigned char)k;
            k = 0;
        }
    }
    ipaddr[j] = (uint8)k;
    return 1;
}
```

```
static int
mac_convert_address (char *macstring, unsigned char macaddr[])
{
    uint8 i, digits, colons;
    int success, length;
    char *str;
    length = strlen(macstring);
    str = macstring;
    digits = colons = 0;

    for (i = 0; i < length; i++)
    {
        if (macstring[i] != ':')
        {
            if (++digits > 3)
                return 0;
        }
        else
        {
            if (++colons > 5)
                return 0;
            else if (digits == 0)
                return 0;
            else
            {
                digits = 0;
                macstring[i] = (uint8)NULL;
                macaddr[colons-1] = (uint8)get_value(str,&success,16);
                if (!success)
                    return 0;
                str = &macstring[i+1];
            }
        }
    }
    if (colons != 5)
        return 0;

    macaddr[colons] = (uint8)get_value(str,&success,16);
    if (!success)
        return 0;

    return 1;
}
```

```
void
set_server (int argc, char **argv)
{
    (void)argc;

    if (argv[2] == NULL)
    {
        //output(IPFMT, server[0], server[1], server[2], server[3]);
        output("Servidor: ");
        outNum(10,server[0],SIN_SIGNO);
        output(".");
        outNum(10,server[1],SIN_SIGNO);
        output(".");
        outNum(10,server[2],SIN_SIGNO);
        output(".");
        outNum(10,server[3],SIN_SIGNO);
        output("\n");
        return;
    }
    if (!ip_convert_address(argv[2],server))
    {
        output("Error:direccion IP no valida: ");
        output(argv[2]);
        output("\n");
        return;
    }
}
```

```
void
set_client (int argc, char **argv)
{
    (void)argc;

    if (argv[2] == NULL)
    {
        //output(IPFMT, client[0], client[1], client[2], client[3]);
        output("Cliente: ");
        outNum(10,client[0],SIN_SIGNO);
        output(".");
        outNum(10,client[1],SIN_SIGNO);
        output(".");
        outNum(10,client[2],SIN_SIGNO);
        output(".");
        outNum(10,client[3],SIN_SIGNO);
        output("\n");
        return;
    }
    if (!ip_convert_address(argv[2],client))
    {
        output("Error:direccion IP no valida: ");
        output(argv[2]);
        output("\n");
        return;
    }
}
```

```
void
set_gateway (int argc, char **argv)
{
    (void)argc;

    if (argv[2] == NULL)
    {
        //output(IPFMT, gateway[0], gateway[1], gateway[2], gateway[3]);
        output("Gateway: ");
        outNum(10,gateway [0],SIN_SIGNO);
        output(".");
        outNum(10,gateway [1],SIN_SIGNO);
        output(".");
        outNum(10,gateway [2],SIN_SIGNO);
        output(".");
        outNum(10,gateway [3],SIN_SIGNO);
        output("\n");

        return;
    }
    if (!ip_convert_address(argv[2],gateway))
    {
        output("Error:direccion IP no valida: ");
        output(argv[2]);
        output("\n");
        return;
    }
}
```

```
void
set_netmask (int argc, char **argv)
{
    (void)argc;

    if (argv[2] == NULL)
    {
        //output(IPFMT, netmask[0], netmask[1], netmask[2], netmask[3]);
        output("Netmask: ");
        outNum(10,netmask [0],SIN_SIGNO);
        output(".");
        outNum(10,netmask [1],SIN_SIGNO);
        output(".");
        outNum(10,netmask [2],SIN_SIGNO);
        output(".");
        outNum(10,netmask [3],SIN_SIGNO);
        output("\n");
        return;
    }
    if (!ip_convert_address(argv[2],netmask))
    {
        output("Error:direccion IP no valida: ");
        output(argv[2]);
        output("\n");
        return;
    }
}
```

```
void
set_filename (int argc, char **argv)
{
    (void)argc;

    if (argv[2] == NULL)
    {
        output("Nombre del archivo: ");
        output(filename);
        output("\n");
        return;
    }
    strncpy(filename, argv[2], MAX_FN);
}
```

```
void
set_mac (int argc, char **argv)
{
    (void)argc;

    if (argv[2] == NULL)
    {
        //output(MACFMT, mac[0], mac[1], mac[2], mac[3], mac[4], mac[5]);
        output("Direccion MAC: ");
        outNum(10,mac [0],SIN_SIGNO);
        output(".");
        outNum(10,mac [1],SIN_SIGNO);
        output(".");
        outNum(10,mac [2],SIN_SIGNO);
        output(".");
        outNum(10,mac [3],SIN_SIGNO);
        output("\n");
        return;
    }
    if (!mac_convert_address(argv[2],mac))
    {
        output("Error:direccion IP no valida: ");
        output(argv[2]);
        output("\n");
        return;
    }
}
```

```
void
set (int argc, char **argv)
{
    int index;

    if (argc == 1)
    {
        output("\n");
        output("Posibles opciones de 'set': \n");
        for (index = 0; index < NUM_SETCMD; ++index)
        {
            output("\nOpcion : ");
            output(SETCMDTAB[index].option);
            output("\n");

            output("Su sintaxis es: ");
            output(SETCMDTAB[index].syntax);
            output("\n");
            output("\n");
        }
        output("\n");
        return;
    }

    if (argc != 3)
    {
        output("Error: Argumento de la lista no valido\n");
        return;
    }

    for (index = 0; index < NUM_SETCMD; index++)
    {
        if (strcasecmp(SETCMDTAB[index].option, argv[1]) == 0)
        {
            SETCMDTAB[index].func(argc, argv);
            return;
        }
    }
    output("Error:opcion set/show no valida: ");
    output(argv[1]);
    output("\n");
}
```

```
void
show (int argc, char **argv)
{
    int index;

    /* Show all Option settings */
    argc = 2;
    argv[2] = NULL;
    output("\n");
    for (index = 0; index < NUM_SETCMD; index++)
    {
        //output(OPTFMT, SETCMTTAB[index].option);
        output("La opcion de la tabla de comandos es: ");
        output(SETCMTTAB[index].option);
        output("\n");

        SETCMTTAB[index].func(argc, argv);
    }
    output("\n");
}
```