# Memoria técnica del proyecto del sudoku

# Proyecto Hardware. Prácticas 2-3

Gueorgui Alexandrovitch Kachan 794999

Ernesto Bielsa Gracia 798799

January 2022

# Índice

# 1. Resumen ejecutivo

En las prácticas 2 y 3 se han implementado diferentes versiones del juego del sudoku, haciendo uso de distintos periféricos.

En la práctica 2 se ha dividido la arquitectura del sistema en varios módulos que se encargan de gestionar distintos tipos de periféricos, además del módulo del sudoku, en el que se incluye la versión 'actualizar sudoku' y 'propagar sudoku' escrita en C (la más rápida en -O3). También se ha incluido un módulo 'planificador', que será el encargado de gestionar los distintos eventos que se producen durante el juego y de realizar las acciones que provocan dichos eventos.

El resultado obtenido es una aplicación que permite jugar al sudoku introduciendo y leyendo información a través de la GPIO. Los bits de lectura de la GPIO se van actualizando cada cierto tiempo. Cuando se introduce una jugada válida, los valores nuevos de los candidatos y el valor nuevo de la celda concreta seleccionada se actualizan dentro del módulo que guarda toda la información relativa al sudoku y se actualizan los bits pertinentes de la GPIO.

Permite jugar al sudoku sin problemas pero en nuestra opinión no muestra la información de la forma más clara posible y para un jugador que no esté acostumbrado a jugar así puede resultar costoso entender qué valores son los candidatos (puesto que los que NO son candidatos son los bits que están activos, lo que en un principio confunde). Además existe la dificultad añadida de que solo se puede observar una celda junto a sus candidatos en todo momento.

En la práctica 3 se mantiene la arquitectura de la práctica 2, añadiendo un nuevo periférico, la uart (pantalla). Esto permite mostrar el tablero del sudoku, además de los candidatos para cada celda, así como información textual acerca de como jugar al juego antes de empezar a jugar y un resumen al final de la partida con información relevante.
También se ha profundizado en el uso de otros periféricos ya usados en la práctica 2. En el caso de los timers, se ha incluido el uso de Watch Dog y el Real Time Clock, que se explicarán más adelante.
Por otra parte se han prescindido del uso de ciertos elementos que sí se usan el la práctica 2 pero en esta práctica no eran necesarios y que también se analizarán con más detalle en la memoria.

El resultado de esta práctica, al igual que en la práctica 2 es una aplicación que permite jugar al sudoku. Sin embargo, a diferencia de la práctica 2, al hacer uso de la uart, el juego del sudoku se puede representar de una forma mucho más clara, mostrando un tablero 9x9 tal y como se acostumbra a jugar a este juego. También se muestran más claros los candidatos, puesto que en este caso se muestran en formato decimal.
Además, como ya se ha mencionado, se han introducido mensajes para el jugador, con información que le puede ser de utilidad.

# 2. Introducción

Para poner en contexto, se está trabajando sobre el procesador ARM LPC2105, el mismo con el cual se realizó la primera práctica de la asignatura 'Proyecto Hardware' y también usada anteriormente en las asignaturas 'Arquitectura y Organización de Computadores 1 y 2'.
En la práctica 2 se aprende a trabajar con periféricos sencillos (gestionables mediante un controlador de interrupciones (VIC)) comunes a muchos sistemas de hoy en día cómo:

- Timers: permiten programar alarmas, medir tiempos y demás de forma muy precisa.
- GPIO (General-Purpose In/Out): permite emular botones, leds, o un teclado.

En la práctica 2 además se le da mucha importancia a la correcta modularidad del código, con interfaces que separan el código de alto y bajo nivel para permitir a nuestro programa principal interactuar con ellos de manera sencilla.
Ya por último, en la práctica 2 se ha desarrollado un planificador que gestionará como reaccionar ante cualquier evento que ocurra en el sistema, además de controlar el consumo de energía del procesador, lo cual es muy importante.

En este momento ya se puede jugar al sudoku, pero de una forma poco satisfactoria ya que sólo se pueden observar las celdas de uno en uno, lo cual dificulta mucho ser consciente del estado de todo el tablero a la vez.

La practica 3 tiene como finalidad completar el trabajo realizado en las prácticas 1 y 2, para acercarse lo máximo posible a un juego completo autónomo y con el procesador funcionando correctamente.
Para esto se han implementado dos contadores nuevos con funcionalidades específicas:

- Real-time clock (RTC): para medir el tiempo transcurrido durante la partida
- Watchdog (WD): este es un contador especial que permite resetear el procesador por si se cuelga el programa. Funciona de forma especial ya que hay que inicializarlo asignándole un tiempo de cuenta y entonces "alimentarlo"(reiniciarlo) de forma manual cada cierto tiempo para que nunca llegue a cero. Debido a este funcionamiento, si en algún momento no se puede "alimentar", saltará el timer y reseteará el procesador.

También se han añadido mecanismos de seguridad para asegurarse de que no ocurren condiciones de carrera (acceso a recursos compartidos de forma desincronizada, generando un estado erróneo/no deseado).
Finalmente se ha implementado el uso de la línea serie (UART0) tanto para recibir información del juego como para poder introducir las jugadas y acciones a realizar en la partida, lo cual soluciona el problema de jugabilidad que tenía la versión terminada de la práctica 2.

# 3. Objetivos

Los objetivos principales, tanto de la práctica 2 como de la práctica 3 son crear una aplicación que permita jugar al sudoku, de una forma distinta en cada práctica, siendo en el caso de la práctica 2 interactuando a través de la GPIO y en el caso de la práctica 3 a través de la uart.

Otro objetivo que se pone es que en ambos casos el código ha de ser modular y que el código que controle cada periférico esté separado del resto, así como del código que controla la lógica del juego. Este último, el planificador, deberá encargarse únicamente de recibir eventos y ejecutar las acciones pertinentes, intentando en la medida de lo posible evitar cualquier conocimiento acerca aquello que está gestionando (es decir no saber si está jugando al sudoku o al tetris, por ejemplo).

El mayor objetivo como estudiante a nivel formativo es el de aprender a solucionar todos nuestros problemas ya sea sabiendo usar un manual tan completo y extenso como es el del procesador LPC2105, como buscando por internet soluciones a problemas comunes que ya han tenido anteriormente otras personas, para poder resolver todos nuestros problemas de forma individual.

Se busca aprender a controlar e interactuar con los periféricos a través de C, utilizando el manual de referencia para informarse de la forma de usar los periféricos.

En la práctica 2 se establece como objetivo familiarizarse con el uso de los timers, la GPIO y la gestión de energía del procesador, además de la gestión de interrupciones externas (eint1 y eint2, botones).
En la práctica 3 se busca extender el conocimiento del uso de los timers, con la inclusión del Watch Dog y el Real Clock Timer. También se exige aprender a usar otro periférico, la uart, que servirá para mostrar información del juego y para introducir jugadas.
Además, en la práctica 3, se implementarán llamadas al sistemas para poder desactivar las interrupciones fiq e irq, lo que permitirá eliminar condiciones de carrera.

Por último se busca crear un código seguro, que minimice al máximo los errores que pueda cometer el usuario y que puedan desencadenar problemas en la ejecución de la aplicación, asi como eliminar posibles condiciones de carrera que puedan provocar las interrupciones fiq y irq (como ya se ha mencionado anteriormente).En relación con la seguridad, en la práctica 3 también se buscará que los cambios entre distintos modos de ejecución se realicen de forma fiable.

# 4. Metodología

Para realizar estas 2 prácticas se ha trabajado durante las sesiones de prácticas usando el método de trabajo 'Pair Programming' que consiste en trabajar por parejas en un mismo ordenador para tener dos cerebros pensando en vez de uno. Antes de finalizar las sesiones se hacía una lista de puntos a hacer en casa y se repartian entre los 2 para traer hecho a la siguiente sesión.

A continuación en este apartado se va a analizar el esquema del proyecto explicando los pasos realizados para crear el código con el cual se ha obtenido el resultado final.

Para la práctica 2 no se tuvo en cuenta una estructura demasiado modular, además no se siguió un estilo definido (no se utilizó la misma nomenclatura en el nombre de las funciones). Los periféricos que se han utilizado para esta práctica han sido la GPIO, los botones y los timers, además de la creación de un manejador de estados de energía.

En la práctica 3, se ha intentado, en la medida de lo posible, seguir una estructura modular, por lo que se ha modificado parte del código de la práctica 2 para que cumpla esta condición. Además, se ha decidido utilizar una nomenclatura común para todos los módulos de proyecto. La nomenclatura utilizada es **snake case**, puesto que nos ha parecido la más cómoda para los nombres de funciones y variables que queríamos poner. Otra cosa importante a destacar de esta práctica es la inclusión de un nuevo periférico, la uart, que permite mostrar por pantalla un tablero de sudoku e introducir jugadas por teclado, y que no estaba en la práctica 2.
A continuación se incluyen una serie de apartados en los que se habla de los distintos módulos implementados y se explican brevemente los aspectos de diseño y desarrollo que se han considerado más importantes en cada uno de ellos.

## 4.1. Eventos

El primer paso de la realización de esta práctica fue la creación de este módulo, formado por un fichero .h. Se muestran a continuación todos los eventos que pueden ocurrir durante el juego en la práctica 2:

```
#define alarmaSet 0
#define pulsacion1 1
#define pulsacion2 2
#define tempPeriodico 3
#define compruebaSuelta 4 // Este sobra
#define PDown 5 //--> Se programa alarma de 15 segundos que gemera el evento de power
    down
#define visualizar 6
#define apagarValidacion 7
```

Listing 4.1: Eventos usados para la práctica 2

En un principio, se crearon los eventos desde el 0 hasta el 5, que eran los eventos que se nos vinieron a la cabeza con más claridad y nos parecían más lógicos de añadir. Más tarde nos vimos obligados añadir el 6 y el 7, y quitamos el 4 (aunque no se borró por un despiste). Esto es lo que significa cada uno:

- **alarmaSet**: Este evento indica que hay que guardar una nueva alarma, con los datos auxiliares que tenga guardado el evento (el tiempo que tiene que esperar la alarma y si es periódica o no).
- **pulsacion1**: Evento que ocurre en el caso de que se detecte una pulsación de eint1.
- **pulsacion2**: Evento que ocurre en el caso de que se detecte una pulsación de eint2.
- **tempPeriodico**: Evento que indica que ya ha transcurrido el tiempo del temporizador periódico (explicado en el apartado 4.4, de los timers).
- **PDown**: Evento que se genera para poner el procesador en modo Power Down.
- **visualizar**: Evento que indica que es necesario actualizar los valores de la GPIO para que el jugador pueda ver si han cambiado.
- **apagarValidacion**: Evento generado cuando sea necesario apagar el bit de validación que se enciende tras introducir un valor correcto y que tiene que durar 3 segundos encendido.

Probablemente se debería de haber usado un **enum** en vez de definir constantes para los distintos eventos, puesto que es más seguro, pero esto no se nos ocurrió y se nos indicó en la corrección de la práctica 2.

En la práctica 3 se ha corregido esto y se ha usado enum. Además, se han quitado las variables que no se usaban y se han añadido nuevas. Estas son las nuevas variables:

- **nuevo_caracter**: Este evento indica que se ha recibido un nuevo caracter procedente de la uart, escrito por el usuario, por lo que hay que almacenarlo.
- **cout_caracter**: Evento que ocurre en el caso de que se haya terminado de escribir un caracter y la uart esté lista para escribir otro, en el caso de que lo haya.
- **jugada**: Evento que ocurre en el caso de que se detecte una jugada introducida por el usuario, lo que inicia un periodo de 3 segundos en el que el usuario tiene que aceptar la jugada.
- **fin_jugada**: Evento que indica que ya ha transcurrido el tiempo que tiene el usuario para aceptar o rechazar la jugada, por lo que se rechaza automáticamente
- **fin_partida**: Evento que se genera cuando el jugador acaba el sudoku o decide finalizar la partida, una vez se escribe por la uart el mensaje final.

En la siguiente imagen aparece un fragemnto de código con todos los eventos utilizados para la práctica 3:

```
enum {alarmaSet =0,
      pulsacion1 = 1,
      pulsacion2 =2,
      tempPeriodico =3,
      PDown =4,
      nuevo_caracter = 7,
      cout_caracter = 8,
      jugada = 9,
      fin_jugada = 11,
      fin_partida = 12};
```

Listing 4.2: Eventos usados para la práctica 3

## 4.2. Planificador

Este es el fichero que contiene el código necesario para gestionar el juego. Inicia el juego, espera a los eventos y realiza las acciones necesarias en función del evento que llega. El diseño elegido para poder jugar tantas partidas como queramos ha sido utilizar 2 bucles infinitos anidados. Del primero no se sale nunca y sirve para empezar/reiniciar el juego. Del segundo se sale cuando el usuario indica que quiere acabar la partida (escribe valor 0 en fila 0 y columna 0) y se entra en modo Power Down.
El procesador se queda entonces dormido hasta que el jugador pulse cualquiera de los 2 botones. En ese momento comienza una nueva iteración del bucle exterior que reinicia el juego.

Al inicio de cada partida se ha decidido invocar a una función para limpiar los eventos que pudiesen haber quedado guardados en la cola. También se vuelve a dejar el sudoku con sus candidatos y pistas iniciales, borrando los valores introducidos en la partida anterior.
La última cosa a reseñar del diseño del inicio de la partida es el hecho de que aquí es donde se prepara y encola la programación de la alarma de 15 segundos que en el caso de que se disparase pone el procesador en modo Power Down. A posteriori se entendió que muchas de estas acciones no correspondían al planficador, cosa que se ha corregido en la práctica 3
Pasando al interior del segundo bucle cabe reseñar las acciones que se realizan en el caso de producirse un evento de pulsación.
Según nuestra implementación, se comprueba si el botón se ha dejado de pulsar y si la pulsación del botón no ha sido para salir del modo Power Down. En caso de que se cumplan las 2 condiciones anteriores se comprueba si el valor a introducir es correcto y se procede a añadirlo.

En la práctica 3 lo más importante a destacar es que a diferencia de la práctica 2, en este planificador no se inicializa ni se interacciona directamente con ningún periférico sino que se

utilizan funciones de diversos módulos para gestionar los periféricos, corrigiendo de esta forma lo mencionado en el anterior párrafo.

## 4.3. Cola

Aquí se define un cola circular en la que se guardarán los eventos, además de las funciones necesarias para gestionar la cola (guardar elementos, leer,etc). El diseño de la cola se ha planteado como un array de 32 valores de tipo **elemento**, que es un struct que contiene el tipo de evento que es (con un uint8_t basta pues el número de eventos no es muy grande y no puede ser negativo), los datos auxiliares si el evento requiere alguno y la estampilla temporal del momento en el que se ha guardado el evento en la cola.
En caso de producirse overflow en la cola esta se queda iterando en un bucle infinito.
Por último cabe mencionar que se ha creado una función distinta para leer cada uno de los 3 valores que guarda el elemento que esté en primer lugar en la cola, aunque al final se han acabado usando únicamente las funciones que leen el tipo de evento y sus datos auxiliares, mientras que la función que lee la estampilla temporal del elemento no se ha usado.

Para la práctica 3 se ha añadido una comprobación a la hora de guardar un nuevo evento en la cola para desactivar las interrupciones irq y fiq en el caso de ser necesario, para así evitar condiciones de carrera. Se han decidido usar los registros VICIRQStatus y VICFIQStatus, para averiguar si se está intentando guardar un evento desde alguna interrupción de estos tipos. Otra posibilidad contemplada fue la de utilizar el registro CPSR, en el que hay 2 bits que indican si hay alguna IRQ o FIQ activa, pero como por nuestra parte no se encontró ninguna ventaja significante para usar esta segunda opción se dejó tal y como estaba.

A continuación se muestra la función utilizada para guardar un evento nuevo, tal y como está en la práctica 3, con las deshabilitaciones de las IRQ y FIQ:

```
void cola_guardar_eventos(uint8_t ID_evento, uint32_t auxData) {
    struct elemento aux;
  while(lleno()){ }
    if(VICIRQStatus != 0){ // Desactivar fiq
        disable_fiq();
        if (front == -1) front = 0;
        rear = (rear + 1) % SIZE; // & (SIZE - 1)
        aux.ID_evento = ID_evento;
        aux.auxData = auxData;
        aux.tiempo = clock_gettime(); //poner marca de tiempo
        items[rear] = aux;
        enable_fiq();
    }else if(VICFIQStatus != 0){
        //Leer tiempo con temporizador_leer en vez de clock_gettime()
        if (front == -1) front = 0;
        rear = (rear + 1) % SIZE;
```

```
17          aux.ID_evento = ID_evento;
18          aux.auxData = auxData;
19          aux.tiempo = temporizador_leer(); //poner marca de tiempo
20          items[rear] = aux;
21      }else{ // Desactivar ambas interrupciones
22          disable_isr_fiq();
23          if (front == -1) front = 0;
24          rear = (rear + 1) % SIZE;
25          aux.ID_evento = ID_evento;
26          aux.auxData = auxData;
27          aux.tiempo = clock_gettime(); //poner marca de tiempo
28          items[rear] = aux;
29          enable_isr_fiq();
30      }
31  }
```

Listing 4.3: Función para encolar eventos

Aquí es probable que para reutilizar código hubiera sido mejor no repetir las mismas acciones dentro de cada condición, y simplemente volver a comprobar los registros VICIRQStatus y VICFIQStatus para habilitar las interrupciones pertinentes.

## 4.4. Timers

En este módulo lo más importante a destacar son las funciones para la inicialización de cada uno de los timers, timer0 y timer1 del LPC. A la hora de diseñar la solución se ha decidido asignarle a las interrupciones de los timers las prioridades más altas en el VIC, puesto que se ha considerado que de todos los periféricos estos son los más importantes, sobre todo si queremos obtener la máxima precisión posible a la hora de obtener datos temporales.
Para el caso del timer1, que tiene que ser periódico, se ha usado un Match Register (MR0), donde se guarda el tiempo que tiene que pasar entre interrupciones.
La RSI de este timer lo único que hace es guardar un evento de tempPeriodico, visto en el apartado de los eventos y registrar que la interrupción ha sido atendida.

Para la práctica 3 en este módulo se añade la implementación de la llamada al sistema **clock_gettime()**, para acceder al timer1 vía interrupción. Además se establece la interrupción del timer0 como una Fast Interrupt. El código de la función de inicialización del timer0 se puede ver a continuación. Para hacer que sea una interrupción FIQ se ha indicado con una escritura en el registro VICIntSelect, en la línea 10:

```
1  void temporizador_periodico (int periodo) {
2      //  int periodo = 1->1ms, 2->2ms...
3      timer0_int_count = 0;
4      T0MR0 = (60000*periodo)-1;  // 1ms = 60.000-1 counts
5      T0MCR = 3;              // Genera interrupci n cuando el valor de MR0 es alcanzado
```

```
6       T0TCR = 1;                  // Timer0 Enable
7       VICVectAddr1 = (unsigned long)timer0_ISR;       // set interrupt vector in 0
8       VICVectCntl1 = 0x20 | 4;
9       VICIntEnable = VICIntEnable | 0x00000010;       // Enable Timer0 Interrupt
10      VICIntSelect = VICIntSelect | 0x00000010;       // Set as IFQ Interrupt
11  }
```

Listing 4.4: Función para inicializar el timer0

## 4.5. Gestor_Alarmas

Como su nombre indica, crea y gestiona las alarmas que se programan para generar eventos. Las alarmas se han decidido guardar en un struct, que contiene 3 arrays, con tamaño para 8 valores todos. Un array contiene el evento que se quiere provocar. Otro contiene el tiempo que le queda a la alarma para disparase y el último indica si el i-esimo hueco del resto de los arrays está libre o no. Si el i-esimo valor es 1, quiere decir que el hueco está libre.
Dentro de este módulo hay 2 funciones, una para guardar alarmas y otra para comprobar si ha saltado alguna alarma. Para guardar una nueva alarma se comprueba principalmente si esta alarma ya existe (en cuyo caso se procede a su cancelación o su actualización) y en caso contrario se comprueba también si existe un hueco donde guardar esta nueva alarma.

En la práctica 3, en este módulo, se añaden funciones para poner diversas alarmas, liberando al planificador de la necesidad de saber la estructura interna de las alarmas. También se añade una función de inicialización del módulo, que se encarga de inicializar los timers y poner las alarmas iniciales necesarias.
A continuación se muestra parte del código que puede resultar de más interés, el de la función de inicialización del módulo:

```
1   void GestorAlarmaInit(){
2       int i;
3       for(i=0; i<numHuecos; i++){
4           misEventos.estaLibre[i] = 1;
5       }
6       temporizador_periodico(tiempo_ms);
7       temporizador_iniciar();
8       temporizador_empezar();
9       gestor_alarma_poner_alarma_power_down();
10  }
```

Listing 4.5: Función inicialización para el gestor de alarmas

Lo primero que se hace es asegurar que todos los huecos para las alarmas estén libres. Tras esto se puede ver que en las líneas 6,7 y 8 se inician ambos timers (puesto que este módulo es el que se encarga de gestionarlos, liberando al planificador de dicha tarea).

Por último se pone una alarma de 15 segundos que al saltar pone el procesador en modo Power Down.

## 4.6. Gestor_Energia

Este módulo incluye únicamente 2 funciones, una para poner el procesador en modo Power Down y otro para ponerlo en modo Idle.

## 4.7. Gestor_Pulsacion

En este módulo en la práctica 2, se incluye una función que activa las interrupciones para los botones eint1 y eint2, asignándoles las posiciones 14 y 15 del VIC, ya que se ha considerado que este tipo de interrupciones son menos importantes que otras como las de los timers, por ejemplo. También se incluyen las rutinas de servicio de interrupción para los 2 botones, cuya función es la de indicar al procesador que la interrupción ha sido atendida y guardar en la cola lo necesario para que haya un evento de tipo pulsación.
Por último, aquí es donde están también las funciones que realizan la comprobación de si un botón se ha dejado de pulsar.

Como ya se ha mencionado, para la práctica 3 se ha intentado realizar un diseño más modular y por tanto este módulo se ha decidido transformar en una interfaz para los botones, siendo este módulo el único que interactúa con el nivel que controla el hardware de los botones.
Todo lo relativo al manejo de los botones se ha pasado a otro módulo que se ha decidido llamar **buttons**.

## 4.8. Buttons

Este módulo se añadió en la práctica 3 e incluye exactamente los mismo que el módulo Gestor_Pulsación en la práctica 2, una función para inicializar los botones, las rutinas de excepción de sus interrupciones y unas funciones para comprobar si siguen pulsados.
Se decidió añadir en la práctica 3 para incrementar la modularidad del código

## 4.9. gestorGPIO

Módulo que controla el periférico GPIO, con funciones para inicializarlo y para leer y escribir en él. Aquí se incluyen 2 funciones para determinar qué bits de la GPIO son de entrada y cuales de salida, lo cual se consigue principalmente manipulando el registro **IODIR**, escribiendo un 1 en los bits que se quieren marcar como de salida y un 0 en los que se quieren establecer como entrada.
También se incluyen 2 funciones, una para leer de la GPIO y otra para escribir en la misma, lo que

se ha conseguido leyendo o escribiendo en el registro **IOPIN**.

Para la práctica 3 este fichero no se ha modificado, es más, ni siquiera se ha usado puesto que ahora la entrada/salida de información se ha efectuado a través de la uart.

## 4.10. Gestor_IO

A la hora de hacer el diseño se ha planteado este módulo como una interfaz, que permitiese al planificador manipular la GPIO sin necesidad de conocer su funcionamiento interno, simplemento invocando funciones que realicen lo que se quiere hacer en todo momento.
Para esto se dispone de una función de inicialización del módulo, que se encarga también de inicializar el módulo gestorGPIO, descrito anteriormente.
Además, se incluyen funciones para leer y escribir los valores de Fila, Columna, el valor de la Celda, el valor nuevo que se quiere introducir, los Candidatos, el bit de error, el bit de Overflow y el bit de Idle.
Posteriormente se llegó a la conclusión de que algunas funciones eran innecesarios, puesto que no tiene sentido hacer una función de escritura en bits de la GPIO declarados como de lectura, por ejemplo. Esto se corrigió en la práctica 3.

En la práctica 3, se ha decidido utilizar este módulo para gestionar la información que escriba el usuario en la uart, ya sea una jugada o un mensaje para empezar o acabar la partida. Esta decisión ha sido tomada debido a que este módulo gestionaba la entrada/salida de información de la GPIO en la práctica 2 y parece lógico que haga lo mismo en la práctica 3, esta vez con la uart. Para ello se ha incluido una función que se invoca cada vez que el usuario escribe en la uart. Esta función procesa el caracter y lo guarda en un buffer. En el caso de detectar un mensaje válido genera el evento correspondiente en función del mensaje recibido

## 4.11. sudoku

En este módulo se ha decidido dejar 2 funciones, quitando el resto que había en la práctica 1. Estas funciones son las de 'actualizar_candidatos' y 'propagar_candidatos escritas' en C, puesto que se vio que con una optimización -03 eran las más rápidas.

Para la práctica 3 se han introducido unas cuantas funciones más. Una decisión importante de diseño tomada es que no se pueden introducir o borrar valores que vayan contra las normas del juego (por ejemplo, borrar un valor que es una pista), por lo que algunas de las funciones tienen como objetivo evitar precisamente eso.
Otra decisión tomada es que para sacar el sudoku por pantalla se hace una copia del sudoku entero, por lo que se ha hecho una función que obtiene la información del sudoku de la memoria y

la guarda en un array, transformando los valores a formato decimal, para que sean comprendidos fácilmente por el usuario. Otra opción contemplada fue la de ir mandado línea por línea, pero nos pareció más práctica la primera opción.

## 4.12. Gestor_Visualización

Este es un módulo que se ha decidido crear para gestionar toda la información que llega o se escribe en la uart. Para resumir el contenido,hay una función para inicializar el periférico, 2 funciones que se encargan de generar el mensaje de inicio y fin de partida, una función que recibe los mensajes y comienza la escritura por pantalla y otra función que se encarga de seguir escribiendo los mensajes hasta que se acaben.
En la inicialización del periférico se ha decidido darle una prioridad de 13 a las interrupciones de la uart, justo delante de las de los botones, pero con prioridad bastante menor que la de los timers, que se siguen considerando los más importantes.
También hay una rutina de servicio de interrupción en la que se utiliza el registro **U0IIR** para averiguar si la interrupción se producido por una escritura en la uart o una lectura de la misma. El código de esta rutina se puede observar a continuación:

```
void uart0_ISR (void) __irq{
    if(((U0IIR >> 1) & 0x111) == 1 ){
        cola_guardar_eventos(cout_caracter,0);
    }
    else{
        uint32_t hola = U0RBR;
        cola_guardar_eventos(nuevo_caracter,hola);
    }
    VICVectAddr = 0;
}
```

Listing 4.6: RSI de la uart

## 4.13. RTC

Este módulo se añade en la práctica 3 con el objetivo calcular el tiempo de cómputo durante la partida. En este módulo únicamente hay una función para inicializar el RTC y otras 2 para leer los minutos y segundos.

## 4.14. WT

También añadido para la práctica 3 junto con el RTC. Tiene una función de inicialización en la que se programa el reseteo del procesador tras 20 segundos de inactividad (sin alimentar al watchdog).

Además se ha creado una función para "alimentar" al watchdog. En esta función cabe destacar que se ha asegurado la desactivación de todas la interrupciones (irq y fiq), entre las escrituras consecutivas realizadas para evitar que se pueda generar una excepción.

## 4.15. Startup

Para la práctica 2 en este fichero se han añadido una serie de líneas en código ensamblador, desde la líneas 331, que sirven para volver a configurar el PLL, para seguir trabando con una frecuencia de procesador de 60 MHz, una vez el procesador se despierta del modo Power Down.

Para la práctica 3 se han realizado varias modificaciones. La primera en la línea 58, en la que se le asigna cierto espacio a la pila de FIQ. Tras esto, en las líneas 154 y 167 se hacen import de los códigos de la rutina de excepción del timer0 y el fichero de SWI. El código del timer se establece como una interrupción FIQ en la línea 165, tal y como se puede ver a continuación, en la línea 11 del siguiente fragmento de código (en la primera línea del fragmento también se puede observar el import de la RSI del timer0, mencionado anteriormente):

```
1                IMPORT timer0_ISR
2
3  Vectors       LDR     PC, Reset_Addr
4                LDR     PC, Undef_Addr
5                LDR     PC, SWI_Addr
6                LDR     PC, PAbt_Addr
7                LDR     PC, DAbt_Addr
8                NOP                          ; Reserved Vector
9  ;             LDR     PC, IRQ_Addr
10               LDR     PC, [PC, #-0x0FF0]   ; Vector from VicVectAddr
11               LDR     PC, =timer0_ISR
```

Listing 4.7: Asignación de la RSI del timer0 como interrupción FIQ

## 4.16. SWI

En este fichero se han escrito, en código ensamblador, las funciones necesarias para desactivar las interrupciones fiq, las irq y ambas a la vez. Para ello se ha cargado el SPSR que guarda el Status Register del modo usuario del que venimos (ahora estamos en modo superusuario) y se modifican los bits IRQ o FIQ (o ambos).
Este fichero sirve también para saltar al código de la interrupción 0 del SWI, definida como **clock_gettime()**, que se encuentra en el módulo de los timers.
Un ejemplo de como se inhabilitan las interrupciones se muestra a continuación:

```
1  __disable_isr_fiq
2
3      LDMFD   SP!, {R8, R12}      ; Load R8, SPSR
```

```
4   ORR     R12, R12, #0x40     ; Bit de fiq a 1 desactiva este tipo de interrupciones
5   ORR     R12, R12, #0x80     ; Bit de irq a 1 desactiva este tipo de interrupciones
6   MSR     SPSR_cxsf, R12      ; Set SPSR
7   LDMFD   SP!, {R12, PC}^     ; Restore R12 and Return
```

Listing 4.8: Inhabilitación de las interrupciones FIQ y IRQ

En este caso se inhabilitan las interrupciones IRQ y FIQ. Para vover a habilitarlas habría que volver a poner los respectivos bits a 0 (con una instrucción **BIC** bastaría).

## 4.17. Horas de dedicación

| **Tabla con la horas dedicadas por alumno y práctica** | | |
|---|---|---|
| **Alumno** | **Práctica 2** | **Práctica 3** |
| Gueorgui | 30h | 27h |
| Ernesto | 25h | 20h |

# 5. Resultados

El resultado final del juego se puede ver en las siguientes imagenes.

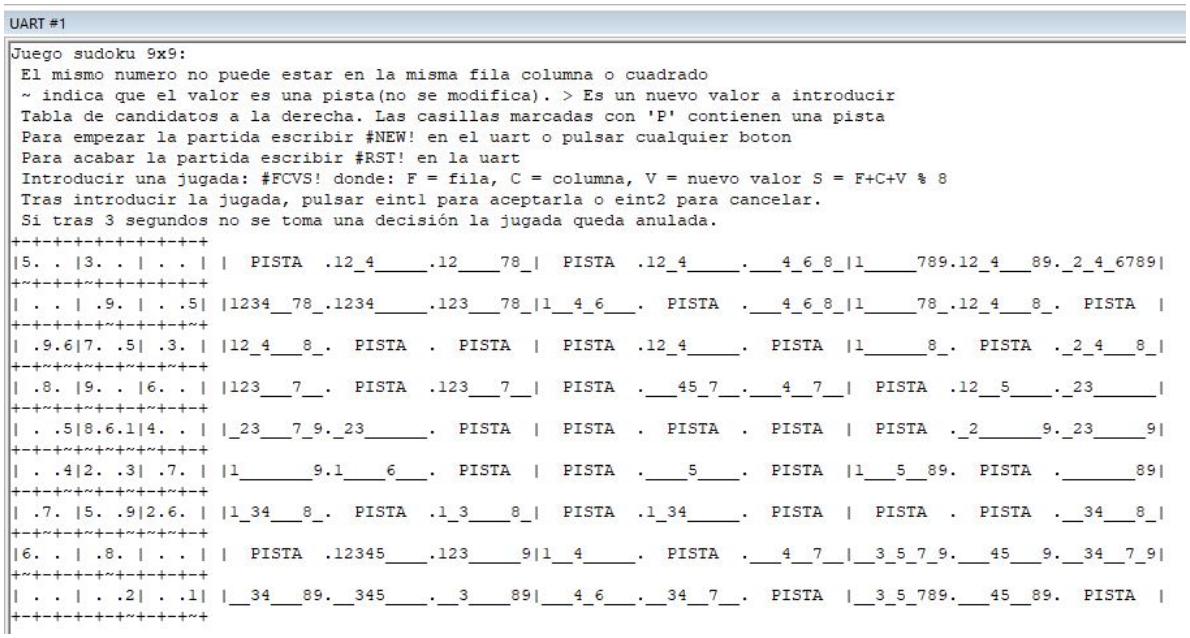


Figura 1: Texto mostrado al inicio de la partida y tablero de Sudoku y de candidatos. Como se puede ver, es muy facil y satisfactorio jugar con este tablero de candidatos ya que muestra de forma muy visible los candidatos posibles de cada celda y si una celda es una pista o no.

```
Partida finalizada por haber tecleado #RST!
Fin de la partida despues de 0 minutos y 5 segundos.
El total del tiempo de computo es: 173ms
Para volver a jugar pulsar cualquiera de los 2 botones
```

Figura 2: Texto mostrado al terminar una partida con el comando 'RST!'.

Se ha obtenido un resultado muy completo ya que cumple todos los requisitos y funcionalidades pedidas en los guiones de las prácticas.

De forma adicional se podría haber perfeccionado más el trabajo si se hubiese implementado una toolbox que emulase los botones de la GPIO para que el jugador no necesite saber el funcionamiento de la GPIO y pueda jugar de forma mas cómoda y facil.

También se podría haber hecho una revisión final del tiempo de ejecución de cada función para intentar optimizar las funciones más costosas o para intentar reducir líneas de código para reducir el tamaño del programa.

Debido a la correcta modularidad del código, este se podría reutilizar en futuros proyectos, lo cual siempre es algo positivo a la hora de reducir tiempo y esfuerzos en proximos trabajos.

# 6. Conclusión

A lo largo de estas 2 prácticas, se han construido 2 versiones distintas, perfectamente funcionales ambas, del juego del sudoku, apoyándose en parte del trabajo realizado en la práctica 1. Para ello ha sido necesario estudiar a fondo el procesador LPC2105, para aprender a usar los periféricos adecuados para realizar lo que se nos pide. Esto ha sido posible en gran medida gracias al manual del procesador que se nos proporciona como material de la práctica.

Como ya se ha mencionado unas cuantas veces, en la práctica 2 se ha construido una primera versión del sudoku, utilizando principalmente la GPIO, aunque haciendo uso también de los botones, los timers y alternando entre distintos modos del procesador según convenía.
El sudoku obtenido en esta práctica, en nuestra opinión, tiene un gran punto débil y es que es complicado de entender su funcionamiento, sobre todo para alguien poco experimentado en el uso de la GPIO. También resulta tedioso el extraer cierta información que ofrece la GPIO. Por ejemplo, para saber el valor de una determinada celda hay que hacer la conversión de hexadecimal a decimal lo que siempre cuesta un poco incluso si se está acostumbrado a trabajar con esta notación. Saber cuales son los candidatos también resulta complicado a veces.
Otro punto débil de esta implementación es que no se dispone de una información completa de todo el sudoku y los candidatos, únicamente se ve una celda y sus candidatos. Un punto fuerte con respecto a la implementación de la práctica 3 puede ser que, en el caso de que todo lo anterior no suponga una molestia al jugador, con esta versión se puede jugar más rápido, pues no hay que esperar a que salga todo el sudoku y las pistas por la uart.

La segunda versión del sudoku hace uso de la uart para mostrar la información del sudoku. Aquí, como ya se menciona, se solucionan varios de los puntos débiles que tenía la anterior implementacion, puesto que se muestra una información más global y clara del juego. Un punto débil que puede tener esta implementación, es la forma de introducir las jugadas, puesto que a pesar de que de esta forma se consigue cierta seguridad para evitar errores, puede resultar tedioso calcular el checksum para el usuario.
De nuevo cabe mencionar que el punto más fuerte a destacar de nuestra implementación es el hecho de que sea imposible de introducir o borrar valores que vayan contra las normas del juego, mostrando al usuario un mensaje a través de la uart con la explicación del error que ha cometido (el número introducido no estaba entre los candidatos o se está intentando borrar una celda que es una pista).

En lo personal, estas 2 prácticas nos han servido para aprender a como intentar sacar el máximo partido a un procesador. Nos ha permitido entender mejor el uso de los distintos periféricos, así como los distintos modos de ejecución que tiene, cómo distinguirlos y alternar entre ellos de forma segura, aprendiendo a usar el manual del procesador para extraer la información necesaria.

# 7. Correcciones solicitadas en las entregas previas

## 7.1. Práctica2

Como se ha resaltado en numerosas ocasiones a lo largo de la memoria, el código de la práctica 2 carecía prácticamente de modularidad, lo cual se señaló en la presentación del código de la práctica.
Esto, como también se ha indicado ya, se ha intentado corregir en la medida de lo posible en la práctica 3.
Otro problema señalado durante la presentación fue la falta de homogeneidad en cuanto a la nomenclatura utilizada para los nombres de funciones.
Con el fin de evitar el mismo problema, antes de comenzar con la realización de la práctica 3 se tomó la decisión de usar la misma nomenclatura de programación para el nombre de todas las funciones, siendo la nomenclatura elegida **Snake case**, pues se cree que es con la que más claridad se pueden representar los nombres elegidos para nuestras funciones, como ya se había mencionado al comienzo del apartado de Metodología.

## 7.2. Práctica 3

Durante la presentación del 22 de Diciembre se señalaron varios aspectos a mejorar en el código del proyecto, para ser presentado de nuevo el 10 de Enero.
El fallo más importante que se resaltó fue que no se comprobaba desde donde se estaba intentando guardar un evento en la cola, puesto que esto podía ocurrir estando en una interrupción fiq, una interrupción irq o en modo usuario. Para cada caso había que realizar acciones distintas, cosa que no se hizo. Esto se solucionó usando los registros VicIRQStatus y VicFIQStatus.
Otro error señalado fue que a la hora de guardar el contenido del mensaje a escribir en la uart, esto se hacía copiando la dirección de memoria donde estaba el mensaje, en lugar de copiar el mensaje entero, lo que podía dar lugar a problemas si la zona de memoria donde se encuentra el mensaje se modifica. Esto se solucionó con un sencillo strcpy().
También nos fue señalado un fallo en la rutina de servicio de interrupción de la uart, en la que por un despiste, solo se escribía en el VicVectAddr para dar por atendida la petición en el else, donde se entra solo cuando se recibe un caracter desde la uart. Esto se solucionó haciendo la escritura del VicVectAddr fuera del else.
Por último, otros fallos que se señalaron, fueron el uso de variables de tipo **int** para variables que no necesitaban signo y que deberían de ser **uint**, así como el uso de **static** para las variables de ámbito global, para garantizar que solo exista en la unidad de compilación en la que se encuentre declarado.

# 8. Bibliografía

En cuanto a la bibliografía, los 2 recursos más utilizados han sido los siguientes:

- **Manual de usuario lpc2104-lpc2105-lpc2106.**
- **https://community.arm.com**

El primer recurso es el manual proporcionado al comienzo del curso, del que se ha extraído la mayoría de la información que se ha necesitado.
El segundo recurso es una página web en la que hay una sección de preguntas entre las cuales se han encontrado preguntas similares a las que teníamos nosotros y a las que no encontrabamos solución en el manual. En esta página se han encontrado soluciones a algunos problemas que tuvimos en el desarrollo de la práctica.