Universidad Zaragoza

Proyecto Hardware
Trabajo de la asignatura

Implementación de un juego sudoku

Guillermo Robles González - NIP: 604409

Supervisado por:
Javier Resano Ezcaray (coordinador)
María Villarroya Gaudó
Enrique Torres Moreno
Jesús Alastruey Benedé
Darío Suárez Gracia

12 de enero de 2016

Pagina intencionalmente en blanco.

Índice

1. Resumen

El presente documento es la documentación asociada al trabajo realizado en la asignatura de Proyecto Hardware ofrecida en la Universidad de Zaragoza en el año 2015. El trabajo realizado consiste en el diseño e implementación de un sistema de ayuda a la resolución de sudokus, que ayude al usuario tanto proporcionando pistas como detectando posibles errores, y de esta forma mejore la experiencia jugable.

Para la consecución del trabajo, este fue dividido en 3 partes claramente diferenciadas, esta división se hace tanto por motivos organizativos (posibilidad de delegar trabajo) como por la separación lógica existente entre las partes.

La primera parte consiste en la comprensión e implementación de un conjunto de sistemas simples, entre los que aparece un sistema de gestión de timers, un gestor de excepciones y una pila de depuración. Estos sistemas no aparecen explícitamente en el trabajo final (a excepción de los timers), sin embargo, permiten suavizar el proceso de desarrollo y acercarnos a la placa con proyectos simples y accesibles.

La segunda parte consiste en el trabajo en el problema de los rebotes (descrito con más detalle en la sección 6.2 lo cual ya conforma un proyecto de mayor complejidad, que fuerza a utilizar lo desarrollado en el apartado previo.

La tercera parte consiste en el uso de la pantalla de la placa, que ofrece una forma de comunicarse con el usuario de manera accesible a este.

Finalmente, estas tres partes (junto con el proyecto desarrollado en las primeras prácticas) se combinan para crear el trabajo deseado.

2. Introducción

Como ya se ha descrito, el presente trabajo se enmarca dentro de la asignatura de Proyecto Hardware, por lo tanto, es importante notar que durante todo el desarrollo se ha tenido presente el objetivo principalmente pedagógico del proyecto. Esto implica que, por ejemplo, se ha primado el aprendizaje variado de ciertos aspectos del hardware usados que, aunque no habrían sido muy útiles en un entorno real, sí tienen sentido desde el punto de vista de aprendizaje. Un ejemplo de esto es la decisión de ignorar el teclado matricial que, pese a que habría sido relativamente fácil de incluir en el proyecto, consiste en repetir el problema de los rebotes de nuevo, y no se considera que aporte nada en el proceso de aprendizaje.

Teniendo presente ante todo el objetivo pedagógico, también se ha tomado la decisión de usar ciertas herramientas (control de versiones, sistemas de documentado automático...) que no se había tenido oportunidad de usar en otras practicas, dada su sencillez o corto espacio en el tiempo.

Como ya se ha mencionado, el trabajo se apoya en una parte del código desarrollado en las primeras practicas, por lo que se asume el conocimiento de estas partes, por ello, no se extienden demasiado, a excepción de algunos cambios que hubo que realizar en los mismos.

3. Objetivos

Los objetivos se organizan en 2 conjuntos, los objetivos propuestos por el profesorado, y un conjunto de objetivos auto propuestos por el alumno.

Objetivos del profesorado:

- Dentro de la parte de tratamiento de rebotes tenemos:
  - Interactuar con una placa real y ser capaces de ejecutar en ella el código desarrollado en la práctica anterior.
  - Profundizar en la interacción C / Ensamblador.
  - Ser capaces de depurar el código ensamblador que genera un compilador a partir de un lenguaje en alto nivel.q
  - Ser capaces de depurar un código con varias fuentes de interrupción activas.
  - Gestionar la entrada/salida con dispositivos básicos, asignando valores a los registros internos de la placa desde un programa en C (utilizando las librerías de la placa).
  - Aprender a desarrollar en C las rutinas de tratamiento de interrupción. Aprender a utilizar los temporizadores internos de la placa y el teclado.
- Dentro de la segunda parte:
  - Finalizar el proyecto en el que se ha estado trabajando hasta conseguir un sistema empotrado autónomo de los ordenadores del laboratorio con el que se pueda jugar directamente.
  - Utilizar la pantalla LCD para visualizar el tablero.
  - Cargar el código en la memoria Flash de la placa mediante el estándar JTAG, de forma que al encenderla se pueda jugar sin necesidad de conectarse ni descargar el programa.

Objetivos del alumno:

- Uso de una adecuada disciplina de diseño, orientada a la generalización de funciones y funcionalidades.
- Organizado del código adecuada, uso de archivos de cabeceras, separación del sistema en los módulos adecuados.
- Uso de un sistema de documentado en código estándar (Doxygen)
- Uso de un sistema de control de versiones (Git)

Como ya se ha destacado, los objetivos del profesorado están principalmente orientados al aspecto pedagógico. Dentro de los objetivos del alumno destacan el uso de Git y Doxygen como sistemas de control de versiones y sistema de generación de documentación automatizado respectivamente. Esta decisión se ha tomado con el objetivo de aprovechar la extensión del proyecto para aprender y practicar ciertas herramientas y técnicas de gestión de proyectos algo complejas, poco adecuadas para usarse con las prácticas típicas de 2 semanas de duración.

4. Tecnologías y herramientas usadas

Como entorno de desarrollo se ha utilizado Eclipse en un entorno Windows, dado que era el ofrecido por el profesorado, con la ToolChain GNU, que nos permitía la compilación cruzada para sistemas particulares (en nuestro caso el procesador ARMv7 de la placa de pruebas). Para la carga y debug se ha utilizado el debugger GDB, usando el estándar JTAG para la comunicación con la placa mediante el programa OpenOCD.

Para la realización del presente documento se ha seleccionado el sistema de maquetación y control tipográfico TEX, con ayuda del conjunto de macros LATEX. Pese a ser algo más complejo que otros sistemas de maquetado del mercado (como suites ofimáticas o Scribus), ofrece un nivel de calidad bastante mayor, además de tener características especialmente orientadas al desarrollo de documentos de índole técnica o científica, como el presente (principalmente, la integración de sistemas de generación de diagramas, como tikz).

Se ha escogido Git como sistema de gestión de versiones dado que ofrece capacidades de trabajo distribuido y gran soporte al trabajo en equipo, dado que inicialmente el proyecto se aproximaba como un trabajo en grupo, pese a que se finalizó en solitario. Los otros CVS considerados (principalmente SVN, Mercurial y Fossil) o no daban tanto soporte para el trabajo en equipo, o no eran tan conocidos por el equipo de desarrollo (y se tuvo problemas para encontrar documentación satisfactoria con el tiempo disponible)

Como repositorio online elegido se ha seleccionado el servicio GitHub, dado que soporta el CVS elegido, y ofrece un conjunto de herramientas extra que simplifican el desarrollar un proyecto de cierta magnitud (Repositorio). El sistema de control de versiones ayudó a resolver varios problemas durante el desarrollo (principalmente, regresiones de software)

Durante el desarrollo se tuvo el problema de la falta del CVS elegido en los ordenadores del laboratorio, por lo que el flujo de trabajo elegido consiste en el trabajo de las 4 horas de laboratorio, seguido de un tiempo aparte en el cual se crean y realizan commits organizados, sin la creación de branches auxiliares, a excepción de en algún caso en el que se produjo discordancias entre versiones.

Como sistema de documentación en código se ha elegido Doxygen, este tipo de sistemas permiten la auto-generación de ficheros pdf o páginas html a partir de la documentación que un programador pueda escribir en el código, aunque por falta de tiempo y fuerza de trabajo no ha sido posible documentar con la profundidad deseada todos los módulos usados.

El uso de Doxygen esta relacionado con el objetivo marcado de organización de código adecuada, dado que, por ejemplo, se han realizado ciertas extensiones en el módulo `lcd` ofrecido por el profesorado, de tal forma que sería posible el dar el pdf generado por la herramienta y la librería a un alumno del curso siguiente, y este podría trabajar con la librería extendida sin necesidad de conocer su funcionamiento interno.

5. Metodología

Para la organización de tareas a alto nivel, se ha utilizado un modelo orientado a objetivos, de tal forma que cada objetivo propuesto se dividía en un conjunto de sub objetivos, repitiendo el proceso de manera recursiva hasta llegar a tareas que se consideraban asumibles por sí solas, las cuales eran asignadas una prioridad, y a continuación eran añadidas al conjunto de tareas para hacer.

Dentro de cada día o sesión de trabajo, se seleccionaban las tareas de más prioridad

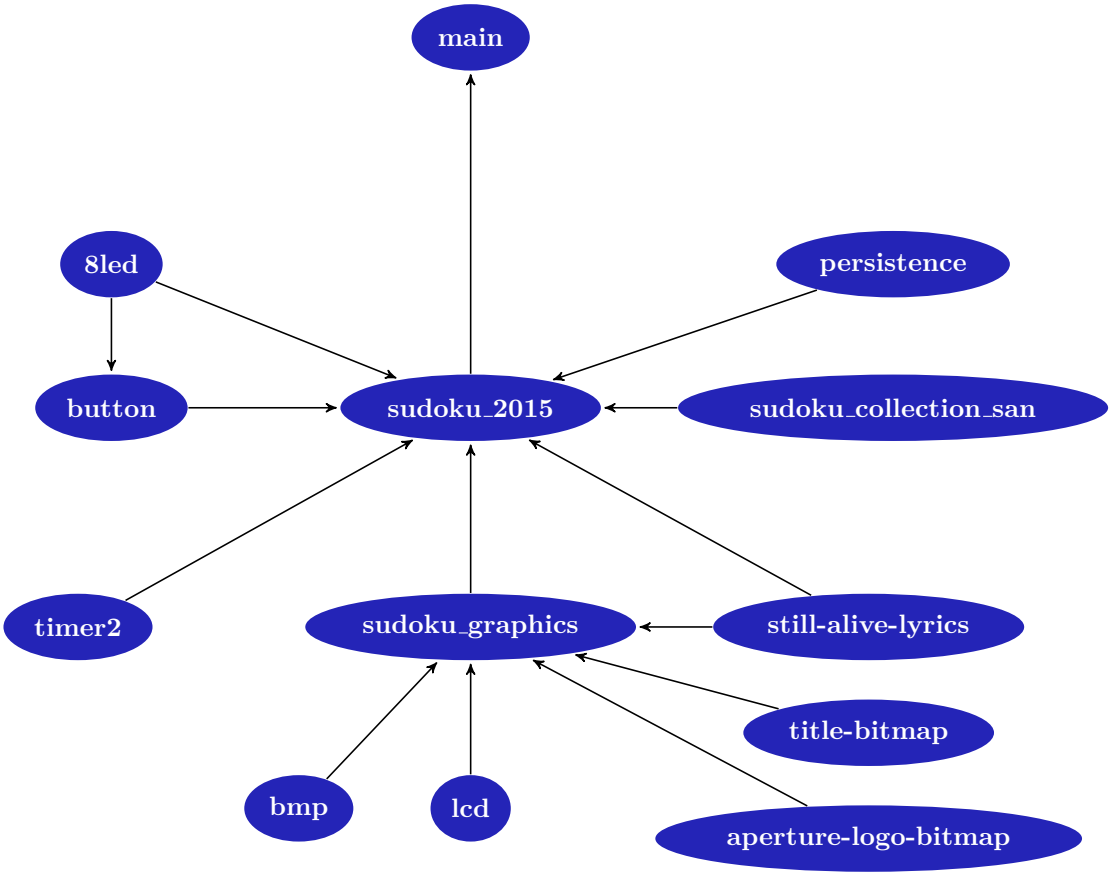

Figura 1: Diagrama de dependencias de los modulos

del conjunto de tareas, y utilizando un sistema de planificación a corto plazo basado en la conocida tabla kanban, se conseguía una visualización clara de las tareas actuales, además de permitir organizar fácilmente las tareas por prioridad, evitando que el desarrollo divague.

Respecto a la organización y priorización de funcionalidades, se a optado por jugabilidad y extensibilidad antes que optimización. En el proyecto, cada fichero de cabecera representa uno de los módulos, a excepción de algunos módulos que almacena los BitMap usados y algunas de las cadenas de texto largas, para separar los datos del código.

Cada módulo del sistema esta formado de un fichero de cabecera y un fichero fuente, a excepción del módulo `sudoku_2015`, que también incluye el fichero sudoku_candidatos.asm, el cual implementa algunas funciones de alta carga de procesado en ensamblador.

El diagrama de dependencias de los módulos aparece representado en la Figura 1 , como se puede apreciar se cumple bastante bien el objetivo de organización establecido, observándose como por ejemplo el modulo `lcd` queda completamente oculto por `sudoku_graphics`, que actúa como proxy.

En la organización en módulos existen 2 casos que se consideran poco deseados. Estos son el triángulo `button`, `8led`, `sudoku_2015` y el triángulo `sudoku_graphics`, `still-alive-lyrics`, `sudoku_2015`. Estos triángulos muestran que, por ejemplo, el módulo `button` no oculta completamente el uso del módulo `8led`. Dado que la solución de estos problemas implicaría un cambio bastante amplio en la organización de módulos, se ha decidido no solucionar esos problemas.

Para el comentario se ha elegido comentar únicamente la función `init_game()`, la cual

gestiona el juego en si, dado que se considera que el resto de funciones (como las de dibujado de ciertas pantallas) son comprensibles por si mismas. El comentario de la función aparece en el Anexo 9.1

6. Partes del proyecto

6.1. Trabajo previo

El trabajo a realizar requiere un conjunto de funcionalidades previas, que pese a no estar siempre explícitamente incluidas en el producto final, nos permiten desarrollar adecuadamente las secciones siguientes, ya sea tanto porque ofrecen funcionalidad en la que se basan(como es el caso de los timers), como porque nos permiten desarrollarlas con más comodidad (como es el caso de la pila de debug). Estas funcionalidades son:

- Conocimiento del funcionamiento de los timers, y uso de los mismos tanto para medidas de tiempo como para programar tareas.
- Implementado de una sencilla pila de debug, que nos permite almacenar estampillas temporales e información deseada, y nos simplifica por ejemplo medir cual es el tiempo que duran los rebotes, o comprobar el tiempo que se tarda en finalizar un DMA
- Manejo de excepciones, tanto para informar al programador como para eventualmente resolverlas y recuperarnos de las mismas.

6.1.1. Timers

El control de sucesos temporales es vital, tanto para ejecutar funciones periódicamente (por ejemplo, capturas de datos) como con un cierto retardo (por ejemplo, esperas).

En el proyecto, el uso de los timers en los rebotes es usado tanto para el control de los tiempos en los cuales ignoramos entradas como para medir los periodos de autorrepetición. Basado en el proyecto dado (un sencillo programa que parpadea alternativamente 2 leds) se pudo generalizar una simple librería de tiempos que cumplía las características necesitadas. El timer usado es el 2, lo cual deja también el 3 inutilizable, o, al menos, con la misma configuración (forma de onda).

La unidad de medida inicial elegida fueron los µsegundos, que posteriormente fue cambiado a milisegundos, dado que en la medida de tiempos de juego (que pueden llegar a las horas) ocurrían problemas de overflow. Sin embargo, esto podía producir problemas, dado que gran cantidad de las funciones, por ello se modifico la librería Timer2 para guardar no solo el número de µsegundos pasados, sino también los segundos por separado en su propia variable. Tras este cambio la nueva librería permite contar hasta 4294967296 segundos, o algo más de 136 años, lo que debería eliminar cualquier posible problema de overflow dentro del proyecto a tratar.

6.1.2. Pila de Debug

Una pila (lo llamamos pila pero la estructura implementada sería más cercana a una cola circular) de Debug es una zona especial de la memoria, en la cual introducimos información que podemos consultar, que en nuestra situación es usada para reconocer y ordenar sucesos

temporales, como excepciones o interrupciones, que guarda una estampilla temporal, un suceso e información extra acerca del mismo.

Cada elemento de la pila son 3 enteros (12 bytes en total), que se corresponden con la estampilla temporal del suceso, un código identificativo del suceso, y un espacio para guardar algo de información extra (por ejemplo, en el caso de las excepciones, la línea en la que ocurren)

En nuestro caso la pila se ha escogido de tamaño 20 sucesos (en total, 240 bytes, que se expanden a 256 para alinearlo correctamente en memoria), situada en la zona de las pilas, por debajo de la pila de usuario, de tal forma que podamos localizarla con facilidad.

La localización de la pila es en las direcciones siguientes a la pila de usuario, por lo que quitamos algo de espacio a la misma, sin embargo, esto no ha provocado problemas graves.

Esta pila utiliza el Timer programado previamente para la generación de las estampillas temporales, por lo que para su correcto funcionamiento el módulo Timer2 ha de ser inicializado previamente por el usuario.

6.1.3. Excepciones

Las excepciones son sucesos inesperados, generalmente asociados a situaciones de error, que nos permiten reconocer situaciones extrañas y advertir al programador, al usuario o incluso intentar recuperarnos de la misma, tanto para continuar el programa como para finalizarlo suavemente.

El procesador ya incluye un sencillo sistema de gestión de excepciones, que simplemente atasca o reinicia la placa. Para manejar este sistema simplemente se han de cambiar los punteros `pISR_UNDEF`, `pISR_SWI`, `pISR_PABORT`, `pISR_DABORT` a una función, que será llamada en la situación de que salte alguna de las excepciones. Las excepciones que podemos tratar en el procesador usado son:

Puntero	Excepción que lo llama	Descripción
`pISR_RESET`	Reset	Reset por hardware de la placa
`pISR_UNDEF`	Undefined Instruction	Opcode(instrucción) no reconocido
`pISR_SWI`	Software Interrupt(SWI)	Lanzado mediante la instrucción swi
`pISR_PABORT`	Prefetch Abort	Error al realizar el fetch de instrucción
`pISR_DABORT`	Data Abort	Error al leer argumentos de una instrucción

Para reconocer la excepción en la cual nos encontramos se puede mirar el modo de procesador, dado que el lanzado de ciertas excepciones fuerza el procesador a modos particulares, por lo que mirando el modo actual se puede detectar la excepción lanzada, tanto para intentar recuperar (por ejemplo, en el caso de SWI) como para poder reconocer y guardar la excepción ocurrida.

Una situación en la cual no se puede reconocer la excepción por el modo es entre `pISR_PABORT` y `pISR_DABORT`, dado que ambas pasan al modo `Abort`; o entre `pISR_RESET` y `pISR_SWI` dado que ambas pasan al modo `Supervisor`; en estos casos la única posibilidad es utilizar varias funciones gestoras, de tal forma que por la propia función en la que estemos nos indique que excepción estamos manejando.

Dado el alcance de este proyecto, no se ha decidido complicar la librería de gestión de excepciones con estas consideraciones, pero en una situación en la cual este módulo quiera generalizarse, habría que tener en cuenta tanto la gestión correcta de algunas excepciones (por ejemplo `pISR_RESET`) como la recuperación de aquellas que lo sean (como `pISR_SWI`).

6.2. Rebotes en los botones

El problema de los rebotes ha sido uno de los primeros problemas a los que tenemos que enfrentarnos en un proyecto de estas características, y uno de los más complejos.

Este problema aparece cuando nos enfrentamos a hardware (botones) reales. Cuando hablamos de rebotes nos referimos al conjunto de señales espurias que da un botón cuando es pulsado o soltado (algunos incluso mientras son pulsados, o sin ser tocados). Estos rebotes son un problema grave, dado que mientras el usuario cree que ha pulsado el botón una única vez (asumiendo como una vez una pulsación y un soltado) el sistema ha recibido múltiples pulsaciones, y por tanto reacciona a todos ellos (en nuestro caso, el sistema incrementa su contador interno múltiples veces).

Existen múltiples soluciones a este problema, tanto por hardware como por software.

Dentro de los solucionadores por hardware aparecen 2 grandes grupos, los basados en SR latches y los basados en circuitos RC, ambos suficientemente efectivos para la mayoría de situaciones, y ampliamente usados en circuitos reales.

Desafortunadamente, el hardware escogido no posee estos sistemas, lo cual nos fuerza a utilizar solucionados por software. Al igual que para el solucionado por hardware, para el solucionado por software existen múltiples sistemas, entre los que encontramos:

- Sencillo sistema de polling, mediante el cual se compruebe el estado del botón con una cierta frecuencia (por ejemplo 250 ms) y si el botón esta pulsado, se realiza la acción asociada. Este sistema tiene el beneficio de que su implementación es trivial, pero es bastante poco efectivo (por ejemplo, si el usuario comienza a realizar pulsaciones fuera de fase respecto de las encuestas, el sistema no detectaría ninguna pulsación)

- Contar el número de "pulsaciones"(pulsaciones reales y rebotes), y solo realizar la acción cada x pulsaciones. Este sistema tiene el beneficio de que es incluso más simple de implementar. Dado que los botones usados no son consistentes en el conteo de rebotes, se abandono esta posibilidad.

- Iniciar un contador en el momento en el que llegue una "pulsación", e ignorar cualquier pulsación en los siguientes x milisegundos, en el caso en el que se desee añadir autorrepetición, gestionarla mediante un sistema de encuesta similar al primero descrito.

Para el proyecto en cuestión se ha elegido el último algoritmo, que, siendo relativamente sencillo de implementar, nos ofrece bastante protección frente a los rebotes, además de permitir integrar funciones como la autorrepetición. Dicha función consiste en la repetición de la acción asociada al botón mientras este esté pulsado, a una frecuencia deseada.

Habiendo elegido el algoritmo, es necesario decidir como se implementará. Para ello, se genera la máquina de estados detallada en la figura 2

En nuestro caso, el botón solo reacciona a 2 señales, la interrupción de pulsado (indicada en el diagrama con PULSA) y la interrupción de reloj interna (ofrecida por el Timer4, con una frecuencia de 10ms, indicada como CLK(10ms)).

Como se describe en la figura 2 .

No se ha encontrado ningún problema destacable en el implementado, a excepción quizás de la determinación de los valores TRP y TRD (ciclos de 10ms de espera en pulsado y en soltado, respectivamente). Para la determinación de los mismos se ha usado la pila de Debug ya comentada, que nos permite conocer el número de interrupciones que lanza los botones en cuestión en que intervalo de tiempo. Cabe destacar que se han elegido unos tiempos algo mas

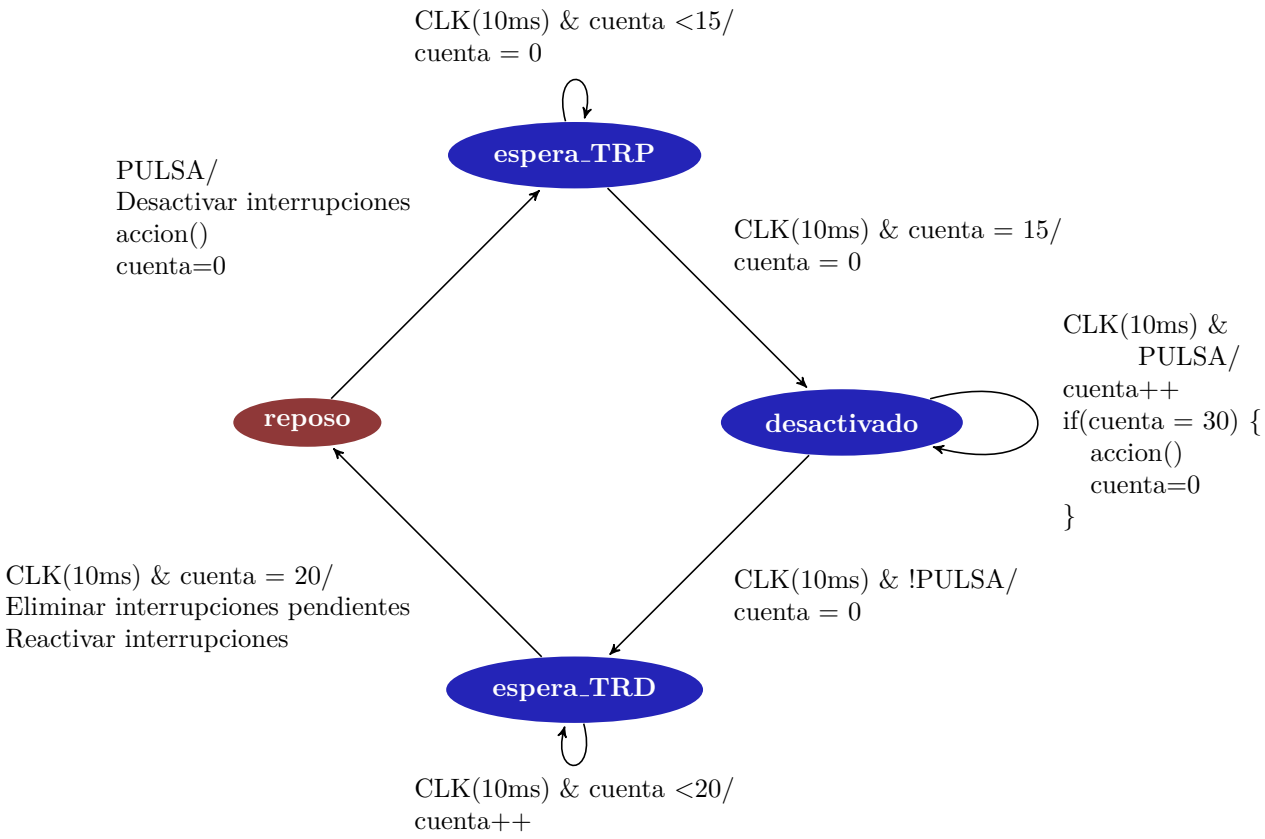

Figura 2: Diagrama de estado del sistema de botones

grandes de lo estrictamente necesario, sin embargo, se ha tomado esta decisión para permitir que el algoritmo pueda comportarse adecuadamente en las distintas placas en las cuales se ha probado, aunque se pierda algo de eficacia en situaciones en las cuales el usuario pulse los botones a gran velocidad (situaciones que, de todas formas, no tienen razón para aparecer en el sistema desarrollado).

6.3. Interacción y Ejecución del sistema

La interacción se realiza principalmente a través de los 2 botones de la placa. Por comodidad, el juego se ha separado en un conjunto de menús o pantallas, cada uno con una función asociada en `sudoku_graphics`, estos menús son `title_screen`, `final_screen`, `aperture`, `instructions`, `sudoku`. Cada menú tiene asociado un estado, a excepción del menú `sudoku`, que contiene 3 estados (`esperando_fila`, `esperando_columna`, `esperando_valor`). El menú `aperture` contiene a su vez 119 subestados, que consisten en las distintas líneas de los créditos, pero no se consideraba que mereciera la pena el añadir otros 119 estados al espacio de estados actual. Este subestado es mantenido por la variable `iterador_aperture`.

La organización de este espacio de estados se muestra en la figura 3

6.4. Gráficos

Para la gestión de los gráficos se ha intentado desplazar todo el sistema al módulo `sudoku_graphics`, que se encarga de gestionar toda la comunicación con la pantalla, ofreciendo una interfaz de alto nivel (con funciones como `sudoku_graphics_fill_from_data(cuadricula)`, que rellena la pantalla con los números de la cuadrícula pasada). Este modelo de abstracción nos permitiría reutilizar el código de la función Main en caso de desear llevar un sistema similar.

En cualquiera de los casos en los que ha sido posible, se ha utilizado el sistema de macros del preprocesador de C para crear código adaptable, de tal forma que si se deseara cambiar la resolución de funcionamiento, o cambiar la fuente de escritura, se pueda hacer sin más que cambiar las pocas macros que describen tamaños relativos y posiciones. Esto tiene el efecto negativo de que la cuadricula en la pantalla no es tan grande como pudiera, dado que para poder adaptarse correctamente, las medidas han de cumplir ciertas propiedades (este problema se describe con más detalle en el propio fichero fuente `sudoku_graphics.h`). También ocurre que algunas funciones se complican más de lo estrictamente necesario, pero ambos aspectos son aceptados.

La actualización de los gráficos es uno de los temas que mas complicaciones ha traído, mas exactamente la decisión del momento de actualización de la pantalla.

En el proyecto inicialmente mandado, solo existían 2 posibles fuentes de actualización de la pantalla (el momento en el cual el usuario introducía un número y el momento en el que avanzaba el reloj), por lo que se podía montar un sistema de actualización a demanda, en el cual simplemente se actualizan ciertas zonas de la pantalla en ciertos momentos (por ejemplo, la zona en la que mostramos el reloj en el momento en el que este cambie).

Sin embargo, algunas de las extensiones que se han decidido realizar (principalmente, la adición de un cursor, que permite eliminar completamente el 8 segmentos del proyecto; y también la adición de algunas animaciones simples) provocan que aparezcan más fuentes de posible actualización, por lo que se tomó la decisión de cambiar a un sistema de actualización constante, que refresca la pantalla siempre que sea posible, eliminando todo el contenido y

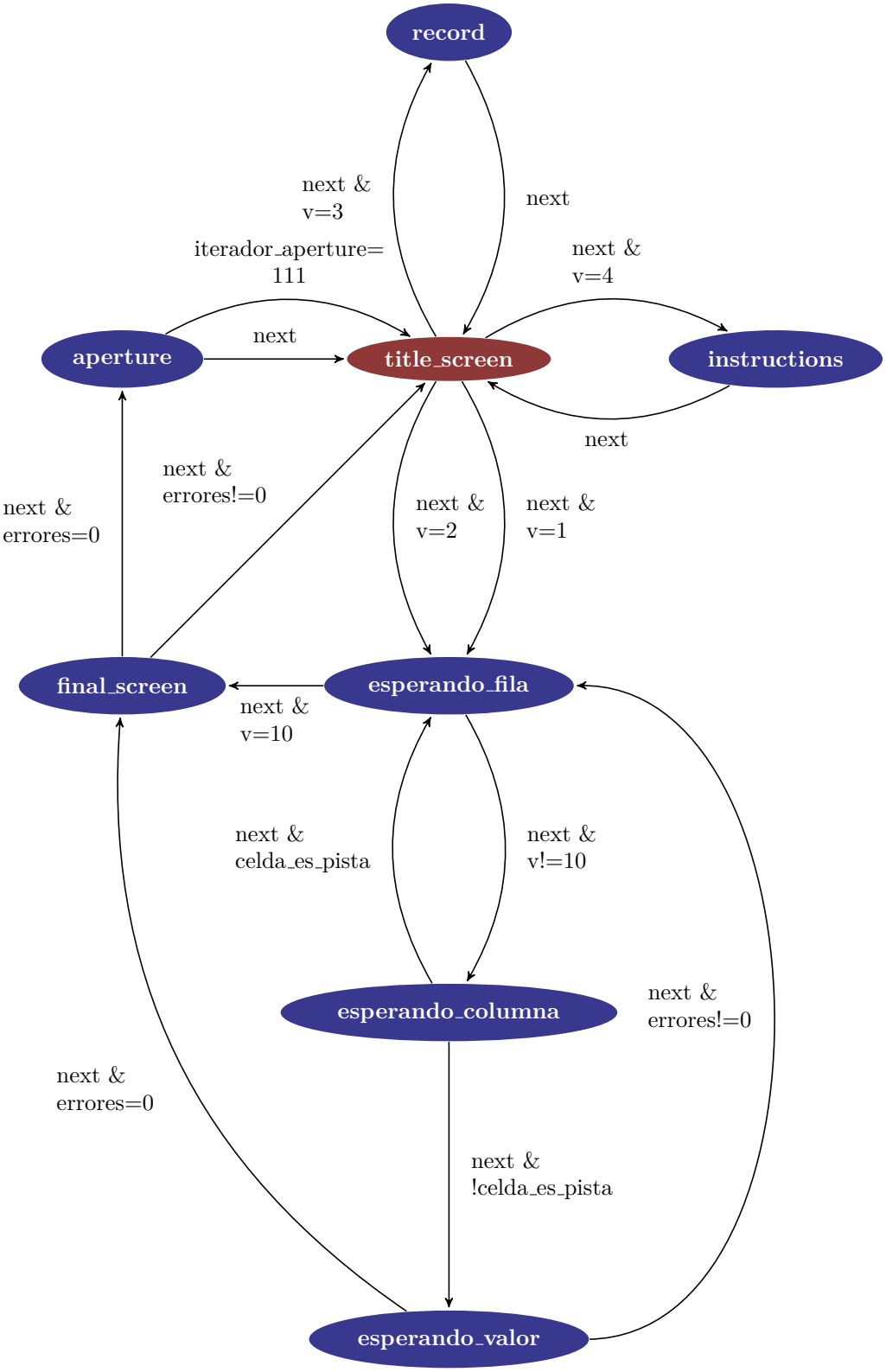

Figura 3: Diagrama de estado del juego de sudoku
Dada su complejidad, se ha decidido no expandir las acciones realizadas en cada estado

recalculándolo de nuevo. Esto tiene algunos efectos negativos, entre ello, si cambia el segundo durante una actualización de la pantalla, un segundo en la pantalla puede durar ligeramente más que un segundo real, pero como se mantiene perfectamente la cuenta interna, se considera un problema asumible. Otro de los problemas que aparece en alguno de los sistemas de prueba es una ligera sensación de parpadeo, pero dado que solo apareció en una de las placas de prueba durante el desarrollo, no se considera un problema alarmante.

Para la organización del sistema se a utilizado un modelo-vista-controlador, en la cual el controlador es proporcionado por la el segundo bloque de la función `init_game()`, que se encarga de actualizar la vista, consistente en la pantalla, siguiendo la información del modelo, que se reparte entre el modulo botón y el segundo bloque de la función `init_game()` (esto es contrario a la filosofía de encapsulamiento, y provoca que, por ejemplo, el modulo button no sea todo lo general que pudiera, esto se comenta con más profundidad en la sección 8.1). Los beneficios de usar un modelo como este son inmediatos, dado que nos permite manejar con comodidad múltiples fuentes de información (botones, sudoku, tiempo...) y múltiples presentaciones de la información (pantalla, cuadricula...), y mantenerlos sincronizados sin problemas.

7. Resultados

Dado que se escogió un desarrollo basado en objetivos, los resultados serán analizados de la misma forma.

Objetivos del profesorado:

- Dentro de la parte de tratamiento de rebotes tenemos:

  - **Interactuar con una placa real:**
    No se tuvo ningún problema destacable, a excepción de los cambios necesarios a realizar en el fichero `44b_init`, para eliminar el problema de depositar las variables en zonas de solo lectura. Existe la pequeña incomodidad de tener que mantener dos `44b_init`, uno para las pruebas con GDB y otro para el momento de cargar en la placa. Se intentó usar el sistema de macros del compilador para arreglar este problema, pero no se consiguió terminar.
  - **Profundizar en la interacción C / Ensamblador:**
    Esto apareció principalmente en las llamadas mutuas entre funciones C y ensamblador. Al principio fue un poco costoso gestionar adecuadamente los marcos de pila, y comprobar que todas las funciones de tratamiento de rutina estaban marcadas como `__attribute__((interrupt(``IRQ'')))`. Como decisión de diseño basada en la legibilidad, se ha mantenido siempre separado el código en C y ensamblador. Un ejemplo de esta situación es en el tratamiento de excepciones, en el cual se había de mirar el modo en el que nos encontrábamos, y en vez de usar sentencias `asm` de gcc para incrustar el ensamblador necesario en el archivo fuente en C, este ensamblador se mueve a su propio fichero fuente. Esta decisión también fue tomada con el objetivo de obtener fuentes en C lo más portables posible.
  - **Ser capaces de depurar el código ensamblador generado por un compilador:**

En general no se tuvo que profundizar demasiado en el ensamblador, dado que no se tuvo ningún problema extraño que así lo requiriese. Por curiosidad, si que se comprobó algo del código generado, para compararlo con el escrito por el alumno durante las primeras practicas. Por ejemplo, el código generado por el compilador (en -W0, es decir, el menor nivel de optimizado) guarda los contadores de bucles en memoria, y no utiliza post/pre indexado. Al igual que con los contadores, el compilador tiene tendencia a evitar guardar variables en registros. En las comprobaciones también se comprobó el código generado al poner la opción -W3, que fuerza el máximo optimizado posible. En este caso se puede observar como el compilador deja de almacenar los contadores de bucle en memoria, además de realizar otras optimizaciones, como desenrollar bucles.

- **Ser capaces de depurar un código con varias fuentes de interrupción activas:**
  Gracias al desarrollo de la pila de Debug (6.1.2sección) no se tuvieron problemas al gestionar las distintas fuentes de interrupción, dado que la pila nos permitía reconstruir los sucesos ordenadamente en el tiempo, e identificar rápidamente las fuentes que provocan ciertas reacciones. Sin embargo, si que se tuvo problemas con las máscaras de interrupción, dado que algunas funciones de inicialización (entre las que se encuentran tanto las ofrecidas por la placa como algo del código ofrecido por el profesorado) tenían el problema de que limpiaban la máscara para luego activar su propio bit, provocando que a partir de ese momento fueran ignoradas el resto de interrupciones.
- **Gestionar la entrada/salida con dispositivos básicos, asignando valores a los registros internos de la placa desde C:**
  Gracias a las librerías de desarrollo de la placa, fue sencillo el uso de los periféricos.
- **Aprender a desarrollar en C las rutinas de tratamiento de interrupción:**
  Aparte del adecuado uso de alguno de los atributos del compilador usado (`__attribute__((interrupt(``IRQ'')))`) y el evitar las rutinas de interrupción demasiado grandes, no se tuvo ningún problema durante la consecución de este objetivo.

- Dentro de la segunda parte:

  - **Utilizar la pantalla LCD para visualizar el tablero:**
    Ya se han mencionado los problemas que se tuvieron, como la frecuencia de actualizado, o los problemas de parpadeo en una de las placas, pero se resolvieron satisfactoriamente. Principalmente se usó la librería lcd ofrecida con la placa, aunque se extendió con algunas funciones para permitir ciertas funcionalidades no disponibles en la librería básica.
  - **Cargar el código en la memoria Flash de la placa mediante el estándar JTAG:**
    Como ya se ha mencionado, fueron necesarios algunos cambios a los archivos de inicialización de la placa (`44b_init`), que copiaban el código de la zona de solo lectura de la memoria a una zona de la memoria en la cual se tenía permiso de lectura y escritura, para evitar problemas al acceder y editar variables.

Objetivos del alumno:

- **Uso de una adecuada disciplina de diseño, orientada a la generalización de funciones y funcionalidades:**
  En general se cumplió la disciplina deseada. Para ello se usaron algunas herramientas externas (como programas que comprobaban el estilo, para asegurar la consistencia).

- **Organizado del código adecuada, uso de archivos de cabeceras, separación del sistema en los módulos adecuados:** La organización final se considera suficientemente adecuada para el nivel al que se está. Todo módulo tiene asociado su correspondiente fichero de cabecera, que además contiene la documentación, por lo que ese aspecto se considera adecuado. La separación del sistema es la adecuada, a excepción de los problemas ya mencionados en la 5sección.

- **Uso de un sistema de documentado en código estándar (Doxygen):**
  Se han comentado mediante este sistema todos los ficheros de cabecera del proyecto, por lo que se considera que se ha cumplido

- **Uso de un sistema de control de versiones (Git):**
  En general, se considera cumplido. El problema de no tener Git en los ordenadores del laboratorio forzó un flujo de trabajo no estándar, pero en general se obtuvo la experiencia con VCS deseada.

En conjunto, se considera que se han obtenido unos resultados satisfactorios, tanto desde el punto de vista puramente pedagógico como desde el de trabajo. Todas las mejoras respecto de la base se considera que mejora la experiencia del jugador.

8. Conclusiones

Las conclusiones se centra principalmente en el aspecto pedagógico del trabajo.
En general, se han cumplido los objetivos propuestos, y dada la longitud temporal y la complejidad del mismo, ha ofrecido la posibilidad de enfrentarse a problemas y utilizar herramientas inadecuadas para una práctica común de 2 semanas. Se considera que se ha obtenido un producto adecuado.

8.1. Margen de mejora

- El módulo Button no es suficientemente general, dado que esta integrado tanto el control del 7 segmentos como algunas variables de control (next, update...). Esto se podría arreglar haciendo que en el constructor el botón reciba 2 punteros a función, que serán las que llamará cuando sea pulsado alguno de los botones; además de un conjunto de parámetros (si el botón tiene auto repetición, tiempo de repetición...). No se ha realizado dado que se considera que complicaría el trabajo innecesariamente.

- Muchas de las funciones creadas para tratar textos, tanto fijos como en movimiento, podrían ser refactorizadas a su propio módulo TextUtils con dependencia de Lcd, persiguiendo el ya mencionado objetivo de una correcta organización del código. No se ha realizado por falta de fuerza de trabajo.

- No se han añadido efectos de sonido al proyecto por falta de fuerza de trabajo. Sin embargo, todo el desarrollo que se ha realizado se describe en la sección 9.2

- Las animaciones de los créditos son bastante crudas, y tienen posibilidad de mejorar. Una opción sería añadir suavizado, provocando que los créditos se muevan más lentamente. No se ha realizado por falta de tiempo.

- Actualmente, el sistema de récords únicamente guarda el menor tiempo de juego con éxito. Se considera que mejoraría bastante introduciendo un sistema de puntuación (algo tan sencillo como una media ponderada del numero de errores durante la partida con el numero de segundos que haya durado sería suficiente), lo cual permitiría la creación de una tabla de los jugadores con más puntos.

- Respecto a los elementos de posible mejora ofrecidos por el profesorado:

  - La pantalla táctil fue descartada porque se consideró que no sería aprovechada a menos que se pensara de nuevo todo el modelo de interacción, y dadas las fuertes restricciones de tiempo sufridas, dicho refactorizado no parecía accesible. En cualquier caso, algunos de los experimentos realizados demostraron que el modo de interacción no era demasiado adecuado, dado que con el tamaño de cuadrícula usado, una tarea tan trivial como seleccionar una casilla se podía convertir en una tarea frustrante, por la falta de precisión de la pantalla (que junto con la no linealidad[1] de las coordenadas que daba dificultaban la tarea de interpretar los toques.)
  - El puerto serie fue descartado porque no parecía un modo adecuado de interactuar con un sistema de estas características (sin embargo, se puede notar como un sistema así permitiría, por ejemplo, un teclado externo).
  - El teclado matricial fue el único elemento de interacción que se vio que ofrecía posibilidades que sobrepasaban su coste de implementación, sin embargo la falta de fuerza de trabajo y el momento tardío en el que se consideró su introducción obligó a abandonar esa línea de trabajo.

9. Anexos

9.1. `init_game()`

Versión comentada de la función `init_game()`, consistente en el bucle principal del juego.

```
void init_game(void) {
  // Tiempo completo de juego
  int tiempo_juego = 0;
  // Tiempo invertido en recalcular la tabla
  int tiempo_calculos = 0;
  // Auxiliar: timestamp de inicio de calculos
  int tiempo_ini_calculo = 0;
  // Auxiliar: timestamp de final de calculos
  int tiempo_fin_calculo = 0;
  // Auxiliar: timestamp de final de juego
  int tiempo_juego_final = 0;
  // Indica si se ha leido el tiempo de fin de juego
  int tiempo_final_no_leido = 1;
  // Timestamp en el cual se comenzaron a reproducir los creditos
```

[1] Resultados ofrecidos por las investigaciones de los compañeros David Nicuesa Aranda y Alberto Álvarez Aldea

```
int tiempo_base_aperture = 0;
// Auxiliar: indica si ya se ha tomado el timestamp de los creditos
int tiempo_base_aperture_no_leido = 1;
// Indica que linea de los creditos se esta reproduciendo ahora
int iterador_aperture = -1;
// Indica si la tabla actual contiene errores
int errores = 1;
// Auxiliar: ultimo record leido
int last_record;
// Mantiene el ultimo momento en el que se avanzo en los creditos
int last_time_aperture = -1;
// Inicializacion del sistema de timers
Timer2_Inicializar();
Button_init(1, 4);
Timer2_Reiniciar();
Timer2_Empezar();
// Inicializacion de variables
uint8_t columna = 1;
uint8_t fila = 1;
int valor = 0;
// Estado de juego actual, comenzamos en la pantalla de titulo
Game_state estadoJuego = title_screen;
// El juego es un bucle infinito
while (1) {
  // Leemos el timestamp actual
  tiempo_juego = Timer2_Leer() / 1000000;
  // Bloque 1: gestion del renderizado de la pantalla
  // Limpimos el buffer virtual
  sudoku_graphics_clear_screen_buffer();
  if (estadoJuego == title_screen) {
    // Estamos en la pantalla de titulo, la dibujamos, se le pasa el valor actual
    // del boton para que sepa que opcion remarcar en el menu
    sudoku_graphics_print_title_screen(Button_valor_actual());
  } else if (estadoJuego == final_screen) {
    // Gestion de las animaciones, el reloj de juego debe detenerse al acabar la
    // partida, sin embargo el contador debe continuar para poder gestionar el
    // temporizado de los creditos
    if (tiempo_final_no_leido) {
      // Bajamos la bandera, indicando que ya se ha leido el tiempo de juego
      tiempo_final_no_leido = 0;
      // Leemos el tiempo de juego
      tiempo_juego_final = tiempo_juego;
      // Comprobacion de record. Leemos el record previo, si el actual es menor
      // lo actualizamos
      last_record = Persistence_read_int();
      if ((tiempo_juego_final < last_record) && (!errores)) {
        Persistence_save_int(tiempo_juego_final);
      }
    }
    // Imprimos la pantalla final, pasandole toda la informacion que ha de mostrar,
    // y la existencia de errores para saber si ha de mostrar la pantalla de exito
    // o fracaso. Tambien se le pasa el record previo para saber si se ha de mostrar
    // el mensaje de felicitacion o no.
    sudoku_graphics_print_final_screen(tiempo_juego_final,
        tiempo_calculos, last_record, errores);
  } else if (estadoJuego == aperture) {
    // Estamos mostrando la pantalla de creditos, por ello se ha de gestionar el
    // contador iterador_aperture para saber por que linea vamos.
    if (tiempo_base_aperture_no_leido) {
      tiempo_base_aperture_no_leido = 0;
      tiempo_base_aperture = tiempo_juego;
    }
    // Gestiona si el iterador ha de avanzar, se tiene en cuenta el valor del
    // boton para permitir variar la velocidad
    if (((tiempo_juego - tiempo_base_aperture) %(4 - Button_valor_actual()) == 0)
        && tiempo_juego != last_time_aperture) {
      iterador_aperture += 1;
      last_time_aperture = tiempo_juego;
    }
    // Finalmente, dibujamos la pantalla, pasandole el numero de linea que queremos
    // dibujar
```

```
    sudoku_graphics_print_still_alive(iterador_aperture);
  } else if (estadoJuego == instructions) {
    // Si estamos en el estado de instrucciones, las imprimimos
    sudoku_graphics_print_instructions();
  } else if (estadoJuego == record) {
    // Si estamos en el estado de record, lo imprimimos
    // Obviamente, hemos de pasarle el record
    sudoku_graphics_print_record(Persistence_read_int());
  } else {
    // Actualizamos la pantalla de juego
    // Dibujamos la ''base'', consistente en la cuadricula, junto con los numeros en los
    // margenes, ademas de el mensaje de ''Pulse A para terminar''
    sudoku_graphics_draw_base();
    // Rellenamos la cuadricula con la informacion del sudoku, por lo que hemos de
    // pasarle el puntero a la cuadricula. Esta funcion se encarga de remarcar pistas y
    // errores como sea necesario, ademas de dibujar los candidatos de cada cuadro de la
    // manera adecuada
    sudoku_graphics_fill_from_data(cuadricula);
    // Dibujamos los tiempos en el margen de la derecha, de la manera deseada
    sudoku_graphics_draw_time(tiempo_juego, tiempo_calculos);
    // Por comodidad del usuario, le recordamos en que estado se encuentra mediante
    // un mensaje en pantalla
    switch (estadoJuego) {
      // Tras seleccionar el estado correcto, se le pasa a la funcion el valor deseado
    case esperando_fila:
      sudoku_graphics_draw_state(0, fila);
      break;
    case esperando_columna:
      sudoku_graphics_draw_state(1, columna);
      break;
    case esperando_valor:
      sudoku_graphics_draw_state(2, valor);
      break;
    default:
      // En cualquier otra situacion, no dibujamos nada
      break;
    }
    // Dibujamos el cursor que indica en que cuadro estamos
    // No se ha de dibujar nada si se esta fuera de la cuadricula (como al seleccionar
    // fila A)
    if (fila < 10) {
      // Remarcamos el cuadrado
      sudoku_graphics_remark_square(fila - 1, columna - 1);
      // Si la celda NO es pista, dibujamos puntos en el interior del cuadro, de esa
      // forma el usuario es capaz de ver rapidamente si esta insertando un posible
      // candidato
      if (!celda_es_pista(cuadricula[fila - 1][columna - 1])) {
        // Marcamos el punto
        sudoku_graphics_mark_error_in_square(fila - 1, columna - 1,
            valor);
      }
    } else {
      // Si estamos fuera de la cuadricula no dibujamos nada
    }
  }
  // Finalmente, pasamos lo dibujado del buffer virtual al buffer de la pantalla real
  sudoku_graphics_update_lcd();

  // Bloque 2: gestion de cambios de estado u operaciones sobre datos
  // Maquina de estados
  switch (estadoJuego) {
    // En caso de encontrarnos en la pantalla de titulo
  case title_screen:
    // Reaccionamos a la pulsacion de boton derecho
    if (Button_next()) {
      // Bajamos la pulsacion
      Button_low_next();
      // Seleccionamos modo de juego (cuadricula/record/instrucciones)
      int valor_actual = Button_valor_actual();
      // Depende de la opcion de menu escogida
      if (valor_actual == 1) {
```

```
        // Cuadricula ''aleatoria''
        // Si realmente fuera necesaria aleatoriedad, existen cientos de
        // implementaciones de docenas de generadore de numeros aleatiorios,
        // el escogido seria un Mersenne twister.
        // Descomprimimos la cuadricula escogida aleatoriamente en el espacio
        // reservado para la cuadricula de juego actual
        sudoku_collection_descomprime(
            cuadriculas[Timer2_Leer() %NUM_CUADRICULAS],
            cuadricula);
      } else if (valor_actual == 2) {
        // Cuadricula especial (casi finalizada)
        // Descomprimimos la cuadricula especial
        sudoku_collection_descomprime(cuadriculaCasiResuelta,
            cuadricula);
      } else if (valor_actual == 3) {
        // Mostrar records
        // Desactivamos el boton
        Button_reconfigure_range(1, 1);
        // Pasamos al estado destino
        estadoJuego = record;
        break;
      } else if (valor_actual == 4) {
        // Seleccion de instrucciones
        // Desactivamos boton
        Button_reconfigure_range(1, 1);
        // Pasamos al estado destino
        estadoJuego = instructions;
        break;
      }
      // Comenzamos a registrar los tiempos de calculo
      tiempo_calculos = 0;
      tiempo_ini_calculo = Timer2_Leer() / 1000;
      // Primer recalculo de la cuadricula recien cargada, necesario para marcar
      // correctamente los candidatos. Recordemos que la version compresa no los almacena
      errores = sudoku_recalcular(cuadricula);
      tiempo_fin_calculo = Timer2_Leer() / 1000;
      tiempo_calculos += (tiempo_fin_calculo - tiempo_ini_calculo);
      // Reseteamos todas las variables
      columna = 1;
      fila = 1;
      valor = 0;
      tiempo_juego = 0;
      tiempo_juego_final = 0;
      tiempo_final_no_leido = 1;
      tiempo_base_aperture = 0;
      tiempo_base_aperture_no_leido = 1;
      iterador_aperture = -1;
      last_time_aperture = -1;
      // Reconfiguramos el boton
      Button_reconfigure_range(1, 10);
      // Dibujamos la nueva tabla
      // Esta situado aqui para evitar retardos y parpadeos
      sudoku_graphics_clear_screen_buffer();
      sudoku_graphics_draw_base();
      sudoku_graphics_fill_from_data(cuadricula);
      sudoku_graphics_draw_time(0, 0);
      sudoku_graphics_draw_state(0, 0);
      sudoku_graphics_update_lcd();
      // Pasamos al estado deseado
      estadoJuego = esperando_fila;
      // Reiniciamos el timer a 0
      Timer2_Reiniciar();
    }
    break;
  case esperando_fila:
    // Estamos esperando la fila
    // Leemos y guardamos el boton
    fila = Button_valor_actual();
    // Si toca cambiar de estado...
    if (Button_next()) {
      // Bajamos la bandera
```

```
      Button_low_next();
      // Comprobamos si se ha introducido el valor especial que indica salir
      if (Button_valor_actual() != 10) {
        // Releemos de nuevo (puede que el usuario pulse los botones de manera inconsistente)
        fila = Button_valor_actual();
        // Ponemos el boton en el rango deseado
        Button_reconfigure_range(1, 9);
        // Pasamos al estado deseado
        estadoJuego = esperando_columna;
      } else {
        // El usuario ha introducido el valor especial de finalizacion
        // Reseteamos variables
        fila = 1;
        columna = 1;
        valor = 0;
        // Desactivamos boton
        Button_reconfigure_range(1, 1);
        // Pasamos a la pantalla final
        estadoJuego = final_screen;
      }
    }
    break;
  case esperando_columna:
    // Estamos esperando a que el usuario introduzca la columna
    // Leemos el valor de columna actual
    columna = Button_valor_actual();
    // Si se ha de pasar al siguiente estado
    if (Button_next()) {
      // Bajamos bandera
      Button_low_next();
      // Es imposible escribir en las celdas marcadas como pista
      if (!celda_es_pista(cuadricula[fila - 1][columna - 1])) {
        // La celda NO es pista, por lo que el usuario puede escribir
        columna = Button_valor_actual();
        Button_reconfigure_range(0, 9);
        estadoJuego = esperando_valor;
      } else {
        // La celda es pista, reiniciamos valores y volvemos al estado previo
        columna = 1;
        fila = 1;
        valor = 0;
        Button_reconfigure_range(1, 10);
        estadoJuego = esperando_fila;
      }
    }
    break;
  case esperando_valor:
    // El usario esta introduciendo valor
    valor = Button_valor_actual();
    // Si toca cambiar de estado
    if (Button_next()) {
      // Bajamos bandera
      Button_low_next();
      // Leemos el valor del boton
      valor = Button_valor_actual();
      // Actualizamos el sudoku con el nuevo valor
      celda_poner_valor(&(cuadricula[fila - 1][columna - 1]), valor);
      // Recalculamos, midiendo el tiempo
      tiempo_ini_calculo = Timer2_Leer() / 1000;
      errores = sudoku_recalcular(cuadricula);
      tiempo_fin_calculo = Timer2_Leer() / 1000;
      tiempo_calculos += (tiempo_fin_calculo - tiempo_ini_calculo);
      // Reseteamos varaibles
      fila = 1;
      columna = 1;
      valor = 0;
      // En caso de que haya errores...
      if (errores) {
        // Si hay errores (o casillas vacias), continuamos jugando
        Button_reconfigure_range(1, 10);
        estadoJuego = esperando_fila;
```

```
        } else {
          // En caso de que no haya errores ni casillas vacias, hemos finalizado
          Button_reconfigure_range(1, 1);
          estadoJuego = final_screen;
        }
      }
      break;
    case final_screen:
      // Pantalla final, damos la oportunidad al usuario de mirar sus resultados
      // Cuando el usuario desee, pasamos al siguiente estado
      if (Button_next()) {
        Button_low_next();
        // En caso de que haya errores
        if (!errores) {
          // Si no hay errores, se va a la pantalla de creditos
          // Configuramos el boton para soportar los distintos niveles de velocidad
          Button_reconfigure_range(1, 3);
          // Pasamos al estado deseado
          estadoJuego = aperture;
        } else {
          // En caso de que haya errores, se vuelve a la pantalla inicial directamente
          // Recofiguramos boton
          Button_reconfigure_range(1, 4);
          // Pasamos al estado deseado
          estadoJuego = title_screen;
        }
      }
      break;
    case aperture:
      // Nos encontramos en la pantalla de creditos
      // De este estado se sale con 2 condiciones, que el usuario desee interrumpirlo o
      // que se acaben de mostrar los creditos
      if (Button_next() || iterador_aperture == (STILL_ALIVE_SIZE - 6)) {
        // Bajamos bandera, reconfiguramos boton, cambiamos de estado.
        Button_low_next();
        Button_reconfigure_range(1, 4);
        estadoJuego = title_screen;
      }
      break;
    case instructions:
      // Estado de instrucciones, esta pantalla no tiene interactividad
      if (Button_next()) {
        // Bajamos bandera, reconfiguramos boton, cambiamos de estado.
        Button_low_next();
        Button_reconfigure_range(1, 4);
        estadoJuego = title_screen;
      }
      break;
    case record:
      // Estado de record, esta pantalla no tiene interactividad
      if (Button_next()) {
        // Bajamos bandera, reconfiguramos boton, cambiamos de estado.
        Button_low_next();
        Button_reconfigure_range(1, 4);
        estadoJuego = title_screen;
      }
      break;
    default:
      break;
    }
  }
}
```

9.2. Musica

La placa incluye un chip de sonido PHILIPS UDA1341TS, que permite tanto la grabación como la reproducción de sonidos. El chip usa una cola FIFO a nivel de byte, en la cual se

pueden introducir bytes mediante un DMA, de tal forma que los reproduzca. El formato de reproducción válido es WAV estéreo, a un ratio de muestreo de 22050 Hz, eliminando la cabecera. El archivo puede ser cargado en memoria RAM para pruebas mediante el comando restore de GDB, que permite el mover archivos binarios del sistema de archivos del ordenador a la memoria RAM de la placa. La zona recomendada es a partir de la dirección 0x0C310000, justo detrás del buffer real de la pantalla. El comando exacto sería

`restore` *`fichero`* `binary 0x0C310000` *`start`*

El valor *`start`* indica el offset a partir del cual se comenzará a leer el fichero, y es usado para evitar leer la cabecera y pasarla también a la placa, para obtener el valor específico se ha de abrir el fichero de sonido con un editor hexadecimal y obtener la dirección inmediatamente siguiente a la cadena "data", la cual indica el comienzo de la zona de datos (en el archivo incluido con los proyectos ofrecidos por el profesorado, este valor es 0x28)

Para cargar el archivo de musica a la placa para soportar reseteos se ha de incrustar manualmente en el archivo .bin usando un editor hexadecimal antes de cargarlo en la placa. Existen espacios en blanco al final del fichero que lo permiten. No se ha probado a realizar el DMA directamente desde la ROM, pero no hay razón para pensar que podría dar problemas. En caso de que surjan problemas, es necesario modificar el archivo `44b_init` para que mueva el fichero de la ROM a una zona de la RAM válida (por ejemplo, la ya descrita 0x0C310000).

Uno de los problemas que surgen al utilizar este sistema es la altísima velocidad de DMA, que es tan rápida que distorsiona el archivo usado. Para esto se plantean varias soluciones:

- Retardar el DMA
- Disminuir la velocidad de consumo de la cola
- Alargar el archivo de sonido

La primera solución es inviable, dado que el sistema DMA de la placa usada no permite retardo, una posibilidad sería el crear un gestor de DMA a nivel de software, que se encargara de temporizar adecuadamente las transmisiones, pero se comprobó que era demasiado difícil para las restricciones de tiempo.

La segunda opción se probó, pero incluso a la mínima velocidad disponible la velocidad era demasiado elevada.

Por ello, se decidió desarrollar la tercera opción, que pese a ser la menos elegante era la única que era accesible. Para ello se usó el software Audacity para alargar manualmente la longitud de el archivo, comprobando tras algunas pruebas que el factor que mejor funcionaba era 12.2, es decir, que un efecto de sonido de 1 segundo se alargaría a unos 12 segundos, los cuales ocuparían aproximadamente 1092949 bytes, o algo más de 1 MB (sin contar cabeceras de ficheros, que no han de ser pasados a la placa).

Para la reproducción de los ficheros se ha usado el modulo `iis_wav` incluido en los proyectos de ejemplos, del cual tuvo que adaptarse la función `void Playwave(int times)` para que recibiera la dirección del fichero en la memoria de la placa y la longitud del mismo.

El otro gran problema que aparece es que el contador del DMA es de 20 bits, lo cual equivale a un contador máximo de 1048576 bytes, lo cual es ligeramente menor que el tamaño en memoria de un segundo de sonido (consistente en 1092949 bytes, o 1.05 MB), por ello, en caso de desear reproducir más de un segundo se deberá adaptar el módulo para que automáticamente programe otro DMA en el momento en el que termine el previo, de tal forma que se puedan reproducir con suavidad.

El último problema encontrado es que la función `Playwave` es bloqueante, lo cual significa que no se puede realizar ninguna acción mientras se reproduce sonido, para arreglar eso hemos de separar el siguiente fragmento de código de `Playwave`:

```
while (iDMADone == 0)
        ;       // DMA end ?
rIISCON = 0x0;
```

el cual se encarga de esperar al finalizado del DMA, y a continuación apaga el sistema de sonido, y mover la línea `rIISCON = 0x0;` a la ISR del DMA, de tal forma que el sistema de sonido no permanezca encendido constantemente.

Dado que la placa usada tiene varios módulos DMA, se puede actualizar la pantalla y reproducir música simultáneamente, y en los experimentos realizados el hacer ambas acciones simultáneamente no afecta de manera notable a la calidad del sonido, aunque si hace la frecuencia de refresco de pantalla algo más lenta, por lo que podría dar problemas en ciertas aplicaciones.

Tanto la complejidad del manejo de sonido como los amplios requerimientos de espacio que usa lo hacen inadecuado para la adición de una banda sonora completa al juego, sin embargo, si puede usarse para añadir pequeños efectos de sonido (como una fanfarria de victoria) que mejoren la experiencia del jugador.

9.3. Persistencia

La persistencia consiste en la capacidad de un sistema de información de almacenar datos e información frente a situaciones de corte eléctrico. Particularizándolo a nuestra situación, consiste en la capacidad del juego de almacenar información entre ciclos de apagado-encendido. La información que se podría guardar consisten en partidas a medias, puntuaciones, configuraciones...

El sistema usado contiene un disco solido interno, en el que se podría guardar información no volátil (descrito en la sección 3.4.3 del manual de usuario), sin embargo, requiere manejo de muy bajo nivel, y no existían ejemplos de su uso en los proyectos ofrecidos por el profesorado. Por ello, se decidió investigar otros métodos. Muchos periféricos contienen registros reales, que no forman parte de la RAM, y por tanto no son volátiles, sin embargo la mayoría son reseteados a valores especiales al reiniciar la placa, generalmente a valores 0. Esta sobre escritura es por hardware, y por tanto imposible de evitar. Sin embargo, existen algunos registros que no están afectados por este reseteo. Un ejemplo de los mismos es los registros del RTC (Real Time Clock) interno, que permanecen alimentados por la pila integrada en la placa, y no se borran al apagar.

En un principio se consideró la posibilidad de almacenar información directamente sobre el RTC interno, dado que el proyecto no lo requiere para nada, y por tanto no se usa. Sin embargo, no se pudo averiguar como desactivarlo, y sin desactivarlo el hardware lo auto incrementaba constantemente.

Sin embargo, existen otros registros del RTC que no son auto-incrementados, y estos son los registros de alarma. Estos registros son usados para programar alarmas, de tal forma que el sistema llame a una función en un momento programado. Estos registros son `rALMYEAR`, `rALMMON`, `rALMDAY`, `rALMHOUR`, `rALMMIN`, `rALMSEC`.

Esto nos proporciona un total de 48 bits con los que trabajar, en los cuales podemos almacenar la información deseada, es importante notar como en el manual se indica que de cada

registro de 8 bits, el más significativo aparecía como no utilizado[1]; por lo que realmente sólo se tiene 42 bits para usar. Como ejemplo de su uso, en el proyecto desarrollado se utilizan para almacenar un entero que consiste en el menor tiempo obtenido a lo largo de todas las partidas.

En el sistema actual se utiliza un entero de 4 bytes para almacenar el menor tiempo, es decir, permitimos partidas de hasta 4294967296 segundos, lo cual equivalen a 1193046 horas o 136 años, una duración inviable. Una posible mejora sería el usar un short (de media palabra, 16 bits) para almacenar el mejor tiempo, esto nos daría unas partidas de un máximo de 18 horas, lo cual es bastante más normal. Además, esto tendría el efecto secundario de permitirnos guardar hasta 3 récords, y no solo uno.

El módulo `Persistence` actual permite acceso indexado a nivel de char a los 6 bytes del sistema de alarma del RTC, además de permitir guardar un entero que se superpone en memoria a los bytes 2-5, de forma similar al funcionamiento de las estructuras union de C. Este sistema es suficiente para el desarrollo realizado. Sin embargo, aparte de su reducido espacio, tiene los problemas ya mencionados.

9.4. Múltiples cuadrículas

Una de las mayores mejoras realizadas al sistema es la inclusión de la capacidad de gestionar múltiples tablas, lo cual permitía proporcionar al usuario una cuadrícula nueva en cada juego.

En un principio, se llegó a plantear la posibilidad de que la placa generara los sudokus proceduralmente, sin embargo se comprobó que aunque los algoritmos para la generación de sudokus no eran especialmente complejos, la placa tenía problemas para su ejecución, dado el elevado coste computacional. Dentro de la generación de sudokus, también se plantea la posible solución de la generación de sudokus mediante transformaciones sobre una cuadrícula ya validada. Un ejemplo de dichas transformaciones serían rotaciones de la cuadrícula, intercambios de columnas dentro de la misma subcuadrícula (igualmente aplicables a filas), o intercambios de los bloques de 3 columnas pertenecientes a una subcuadrícula (también aplicables a filas). Esto nos permitiría tener un conjunto base de cuadrículas y aplicarles transformaciones para obtener nuevas subcuadrículas para ofrecer al usuario. Esta posibilidad no ha sido implementada por falta de tiempo.

El camino medio que se eligió fue el guardar cierto número de cuadrículas en memoria, y escoger en tiempo de ejecución una. Rápidamente se observó el elevado consumo de memoria que esta opción causaba, por ello, se consideró la posibilidad de desarrollar un sencillo algoritmo de compresión que permitiera guardar una cantidad elevada de tablas en la ROM sin necesidad de un gran consumo de espacio ni variar el código desarrollado en las primeras prácticas.

El sistema de compresión usado se basa en 3 factores:

- No es necesario almacenar el padding que utilizábamos en las primeras practicas para poder usar cómodamente los visores de memoria del debugger.

[1]Para comprobar si dicho bit existía realmente, se realizaron ciclos de 30 lecturas y escrituras, a una velocidad de una por segundo, alternando 1 y 0 en esa posición, e introduciendo apagados, y los resultados fueron que el sistema no tocaba ese bit (es decir, podía ser tratado como uno más), sin embargo, no se puede afirmar con seguridad que el sistema no utilice esos bits para alguna operación interna, y cambien de manera inesperada. El modulo `Persistence` actual no toma en cuenta esta posibilidad.

4 bits	1 bit	1 bit	1 bit	9 bits
Valor	Es pista	Es error	Sin usar	Candidatos

Valor: Indica el número contenido en la casilla, está en el rango [0,9], con el valor 0 en caso de que la casilla esté vacía.
Es pista: Indica si la casilla es pista inicial. Será pista si este bit vale 1.
Es error: Indica si el valor en la casilla no cumple las normas del sudoku. La casilla es errónea si este bit vale 1.
Candidatos: Indica que números encajan en la casilla. El bit 0 indica si el 1 es candidato, el bit 1 indica si el 2 es candidato...
Las casillas se guardan en memoria como una lista de medias palabras del formato arriba indicado, con 7 medias palabras de padding, de tal forma que cada línea del sudoku ocupa 256 bytes en memoria, y el sudoku entero ocupa 9 líneas, o 2304 bytes.

Figura 4: Formato de las casillas del sudoku en memoria

4 bits	4 bits
Valor casilla 1	Valor casilla 2

Valor casilla 1: Primeros 4 bits de la primera casilla. Está en el rango [0-9], siendo 0 si la casilla es vacía.
Valor casilla 2: Primeros 4 bits de la primera casilla. Está en el rango [0-9], siendo 0 si la casilla es vacía.

Figura 5: Formato de las casillas compresas

- No es necesario almacenar que casillas son candidatos, podemos indicar las casillas vacías con un 0.
- No es necesario guardar los candidatos de una casilla, dado que podemos recalcularlos con relativa facilidad y bajo coste temporal.

Gracias al mismo del formato usado originalmente para almacenar una casilla en 16 bits y un sudoku en 2304 bits (expresado en la figura 4), se pasa a un formato que ocupa únicamente 4 bits por casilla, no requiere padding y sólo ocupa 328 bits, una reducción por un factor de 7, descrito en la figura 5)

10. Bibliografía

- Manuales de consulta ofrecidos por el profesorado
- Proyectos de la placa ofrecidos por el profesorado
- http://www.eng.utah.edu/~cs5780/debouncing.pdf
- https://en.wikipedia.org/wiki/Mersenne_Twister
- https://en.wikipedia.org/wiki/Sudoku
- http://infocenter.arm.com/help/index.jsp (especialmente el manual de referencia de ARM7)
- http://www.sudoku-solutions.com/
- http://sudoku.taskfour.de/
- dryicons.com/blog/2009/08/14/a-simple-algorithm-for-generating-sudoku-puzzles/
- http://stackoverflow.com/questions/6924216/how-to-generate-sudoku-boards-with-unique-solutions
- http://zhangroup.aporc.org/images/files/Paper_3485.pdf