

Prácticas 2-3

Proyecto Hardware. Curso 2021-22

Juan Plo Andrés 795105 · Ignacio Ortega Lalmolda 610720

EINA - UNIZAR

15 de enero de 2022

1 ÍNDICE DE CONTENIDOS

| | | |
|----------|--|-----------|
| 1 | ÍNDICE DE CONTENIDOS | 1 |
| 2 | RESUMEN EJECUTIVO..... | 2 |
| 2.1 | Resumen | 2 |
| 3 | INTRODUCCIÓN | 3 |
| 4 | OBJETIVOS | 4 |
| 4.1 | Objetivo del proyecto | 4 |
| 5 | METODOLOGÍA | 5 |
| 5.1 | Pasos realizados | 5 |
| 5.2 | Esquema del proyecto..... | 5 |
| 5.2.1 | Ficheros del proyecto..... | 5 |
| 5.3 | Arquitectura del sistema..... | 7 |
| 5.3.1 | Esquema de los módulos que componen el sistema | 7 |
| 5.3.2 | Aplicación..... | 8 |
| 5.3.3 | Micro kernel..... | 9 |
| 5.3.4 | Manejadores | 10 |
| 5.3.5 | Dependiente del hardware..... | 11 |
| 5.4 | Descripción de cada módulo..... | 13 |
| 5.4.1 | Planificador, cola y eventos | 13 |
| 5.4.2 | Temporizadores y Gestor de alarmas | 15 |
| 5.4.3 | GPIO | 16 |
| 5.4.4 | Gestor de pulsación..... | 17 |
| 5.4.5 | Reloj de tiempo real y watchdog..... | 18 |
| 5.4.6 | Implementacion de las llamadas al sistema..... | 19 |
| 5.4.7 | Eliminación de las condiciones de carrera | 20 |
| 5.4.8 | Timer0 como fast interrupt..... | 21 |
| 5.4.9 | Integración de la I/O en el juego | 21 |
| 5.4.10 | UART | 21 |
| 6 | RESULTADOS | 24 |
| 6.1.1 | Iniciar una partida..... | 24 |
| 6.1.2 | Consultar candidatos e introducir jugada | 24 |
| 6.1.3 | No confirmar una jugada..... | 25 |
| 6.1.4 | Jugada errónea | 26 |
| 6.1.5 | Ganar partida..... | 27 |
| 6.1.6 | Nueva partida..... | 28 |
| 7 | CONCLUSIONES | 29 |

2 RESUMEN EJECUTIVO

2.1 Resumen

El trabajo desarrollado ha consistido en la realización de código para un sistema empujado para el procesador LPC2105, en concreto, se la aplicación que se ha desarrollado ha sido la del juego del Sudoku con entrada por parte del usuario de las celdas y los valores a introducir en ellas para el desarrollo del juego.

Nuestro código se divide en una serie de módulos de interacción robusta con periféricos entre los cuales se organizan de manera jerárquica. Nuestra arquitectura ofrece una capa que interaccionan con el hardware, una capa de manejadores que se encargan de la interacción con los periféricos, otra con la aplicación (el sudoku) que ofrece una interacción con el usuario bastante cómoda y mas fácil de procesar, además de un planificador que va encolando y descolando los eventos que se van generando en el sistema.

3 INTRODUCCIÓN

El trabajo a realizar consiste en el desarrollo de un sistema empotrado real para implementar una aplicación de ayuda a la resolución de sudokus, partiendo de unas librerías básicas de tratamiento de sudokus, para ello se creará una serie de módulos que interactuarán con los periféricos del sistema y con el usuario, que, en base a dicha interacción, generaran eventos los cuales serán tratados de manera eficiente por un planificador que será el núcleo del sistema.

El entorno de trabajo empleado ha sido el emulador Keil μ Vision el cual también simula el procesador con el que hemos trabajado.

Las reglas del sudoku que se emplean en la implementación son sencillas, el objetivo es el de rellenar las celdas vacías en un tablero (9x9) de tal modo que un número del 1-9 sólo pueda aparecer una vez en su fila, su columna y en su región (3x3). Nuestro programa calculará los candidatos posibles de cada celda del tablero(9x9) de acuerdo a las reglas descritas anteriormente para ayudar a los jugadores a tomar la decisión de que valor puede ir en cada celda.

4 OBJETIVOS

4.1 Objetivo del proyecto

El objetivo principal del proyecto es la construcción de un sistema empujado que ejecute una aplicación, el sudoku además de:

- Gestionar la entrada/salida con dispositivos básicos, asignando valores a los registros internos de la placa desde un programa en C (utilizando las bibliotecas de la placa)
- Desarrollar en C las rutinas de tratamiento de interrupción
- Aprender a utilizar periféricos, como los temporizadores internos de la placa y General Purpose Input/Output (GPIO)
- Depurar eventos concurrentes asíncronos
- Entender los modos de ejecución del procesador y el tratamiento de excepciones
- Ser capaces de depurar un código con varias fuentes de interrupción activas
- Desarrollar un código modular con interfaces claros y robustos para que cualquier aplicación pueda utilizar los periféricos de forma sencilla
- Diseñar un planificador que monitoriza los eventos del sistema, gestiona la respuesta a estos y reduce el consumo de energía del procesador.
- Diseñar e implementar una gestión completa y robusta de periféricos
- Componer una arquitectura de sistema que aproveche los modos del procesador
- Implementar y usar llamadas al sistema para interactuar con un dispositivo, un temporizador, y para activar y desactivar las interrupciones.
- Concebir e implementar de un protocolo de comunicación robusto para el puerto serie.
- Identificar condiciones de carrera y su causa, así como solucionarlas.

5 METODOLOGÍA

5.1 Pasos realizados

Lo primero que se ha desarrollado para el proyecto ha sido la cola de eventos. El programa principal la llama y esta se queda en un bucle infinito encolando y desencolando eventos. A continuación, se ha realizado la gestión de los temporizadores el timer1 reloj del sistema y el timer0, empleado para el encolar eventos periódicos. La siguiente parte realizada fue la configuración del GPIO para el manejo de la entrada salida con el usuario y la gestión de las interrupciones con la gestión de las pulsaciones del usuario. Se ha realizado la gestión de energía del procesador, poniéndolo en modo IDLE si el planificador no tiene nuevos eventos o en powerdown si tras 15 segundos no hay cambios en la entrada. Finalmente se ha integrado la aplicación del sudoku con la entrada salida del procesador.

Una vez terminada la parte de los módulos y el planificador empezamos a añadir nuevos módulos y modificaciones. Empezamos añadiendo una librería para permitir interactuar con el RTC y el watchdog para leer el tiempo de partida y para resetear el sistema en caso de bloqueo. Después, hicimos las modificaciones necesarias para implementar las llamadas al sistema y que los módulos usasen las mismas. Una vez con las llamadas al sistema implementadas, se pudo eliminar condiciones de carrera que podían provocar las interrupciones y cambiamos los módulos añadiendo dichas funciones para garantizar atomicidad a ciertas operaciones sensibles. Después, configuramos el timer0 para que sea tratado como una Fast Interrupt. Finalmente, mejoramos la interacción con el usuario metiendo la interacción por línea serie y definimos como debía actuar el sistema para asemejarse a un juego.

5.2 Esquema del proyecto

5.2.1 Ficheros del proyecto

En la siguiente tabla se resumen los ficheros que componen el proyecto:

| FICHEROS EN EL PROYECTO | | |
|-------------------------|--------|---|
| Fichero | Líneas | Descripción |
| startup.s | 311 | Fichero de inicio |
| SWI.s | 105 | Fichero para las software interrupt |
| softI.h | 15 | Fichero de cabeceras con los identificadores de la software interrupt |
| main.c | 6 | Fichero con el programa principal que realiza una llamada a planificador_main() |
| cola.h | 60 | Fichero de cabeceras de cola.c |
| cola.c | 85 | Funciones y tipos de datos de la cola |
| evento.h | 56 | Fichero de cabeceras de evento.c y tipos de evento |
| evento.c | 14 | Funciones para los eventos |
| planificador.h | 13 | Fichero de cabeceras de planificador.c |
| planificador.c | 156 | Función del planificador que inicializa el sistema y va gestionando los eventos |
| gestor_IO.h | 66 | Fichero de cabeceras de gestor_IO.c |
| gestor_IO.c | 92 | Funciones de los manejadores del sistema de entrada/salida por el GPIO |

| | | |
|--------------------|-----|---|
| gestor_pulsacion.h | 29 | Fichero de cabeceras de gestor_pulsacion.c |
| gestor_pulsacion.c | 125 | Funciones de los manejadores del sistema de las pulsaciones en EINT1 y EINT2 |
| gestor_energia.h | 23 | Fichero de cabeceras de gestor_energia.c |
| gestor_energia.c | 28 | Funciones de los manejadores del sistema para gestionar los modos IDLE y power down |
| gestor_alarmas.h | 64 | Fichero de cabeceras de gestor_alarmas.c |
| gestor_alarmas.c | 160 | Funciones de los manejadores del sistema para gestionar las alarmas |
| sudoku_2021.h | 30 | Fichero de cabeceras de sudoku_2021.c |
| sudoku_2021.c | 94 | Funciones que interactúan directamente con el sudoku |
| celda.h | 123 | Contiene las operaciones que se aplican a una celda |
| tableros.h | 94 | Contiene una serie de tableros en hexadecimal con la situación inicial de cada uno para cada versión. También incluye un tablero solución para verificar la corrección de cada versión ejecutada. |
| gpio.h | 38 | Fichero de cabeceras de gpio.c |
| gpio.c | 97 | Funciones dependientes del hardware que interactúan con el GPIO |
| buttons.h | 30 | Fichero de cabeceras de buttons.c |
| buttons.c | 70 | Funciones dependientes del hardware que interactúan con los botones EINT1 y EINT2 |
| idle_pwdwn.h | 21 | Fichero de cabeceras de idle_pwdwn.c |
| idle_pwdwn.c | 22 | Funciones dependientes del hardware que ponen al procesador en modo idle o powerdown |
| timers.h | 37 | Fichero de cabeceras de timers.c |
| timers.c | 95 | Funciones dependientes del hardware que configuran el timer 0 y el timer 1 |
| relojes.h | 39 | Fichero de cabeceras de relojes.c |
| relojes.c | 73 | Funciones dependientes del hardware que configuran el RTC y el Watch Dog |
| serial_port.h | 86 | Fichero de cabeceras de serial_port.c |
| serial_port.c | 482 | Funciones dependientes del hardware que configuran la UART |

5.3 Arquitectura del sistema

La arquitectura del sistema que se ha desarrollado sigue el siguiente esquema, en donde cada módulo se comunica o bien con otro módulo, o bien con el planificador a través del encolado y desencolado de eventos.

5.3.1 Esquema de los módulos que componen el sistema

En la imagen se muestra como se ha organizado la comunicación entre módulos de la aplicación:

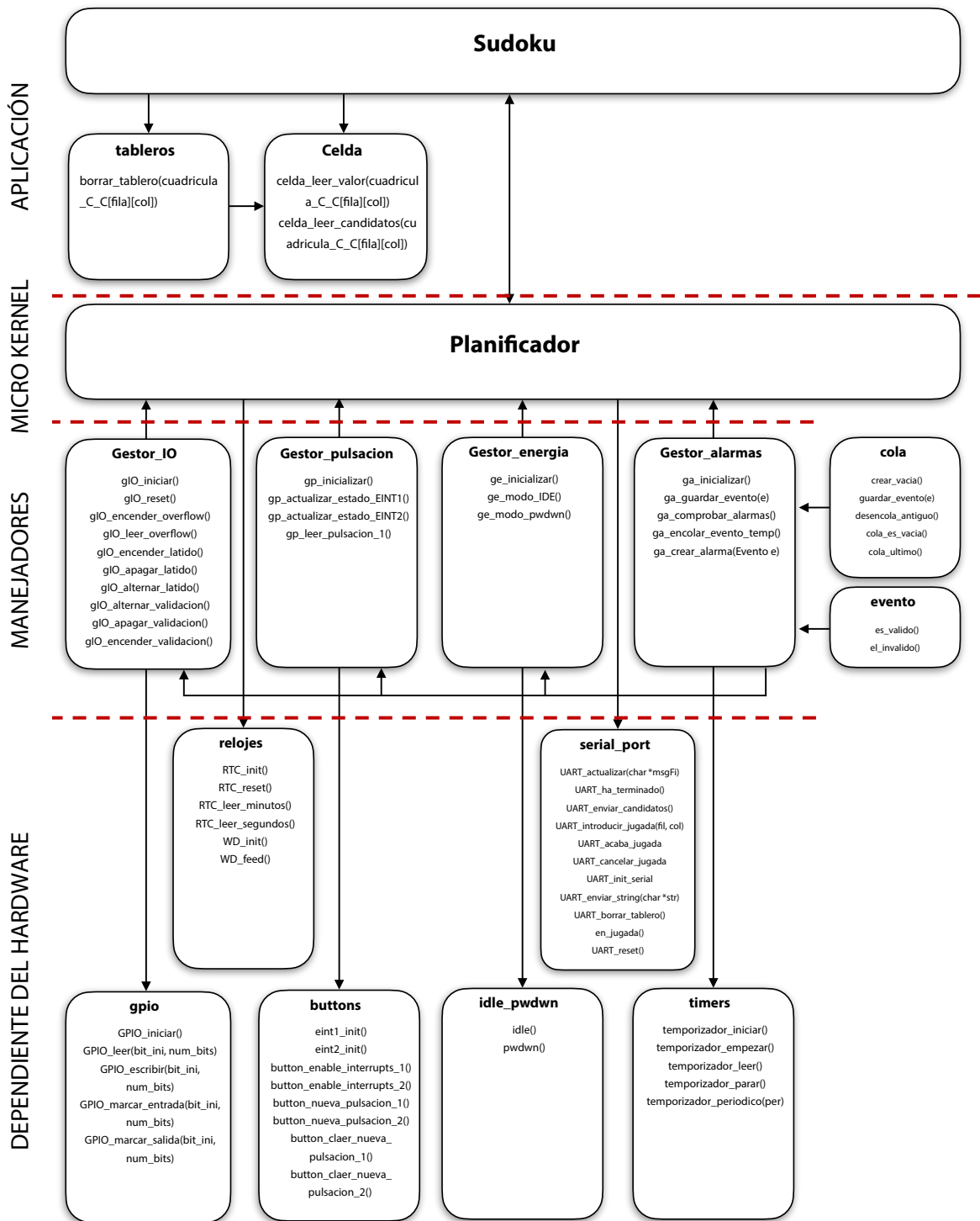


Figura 1: Esquema de los módulos

En las siguientes tablas se incluyen las funciones que componen cada fichero, así como una pequeña descripción de su comportamiento y parámetros para cada capa del sistema.

5.3.2 Aplicación

| SUDOKU | | |
|--|--------------------------|--|
| Fichero | Funciones | Descripción |
| sudoku.c sudoku.h | candidatos_actualizar_c | Versión del código en C. Recibe como parámetro una celda. Calcula todas las listas de candidatos (tras borrar o cambiar un valor) y llama a "candidatos_propagar_c". Devuelve el número de celdas vacías. |
| | candidatos_propagar_c | Versión en C. Recibe como parámetro una celda, su fila y su columna. Propaga el valor de una celda para actualizar la lista de candidatos de su fila, columna y región. Recibe la celda a propagar como parámetro. |
| | tiempo_actualizar | Devuelve tiempo contado que tarda la funcion candidatos actualizar |
| | reset_tiempo_actualizar | Pone a 0 el tiempo contado que tarda la funcion candidatos actualizar |
| celda.h | celda_eliminar_candidato | Recibe como parámetro el puntero a una celda y un valor. Elimina el candidato que coincida con valor. |
| | celda_poner_valor | Recibe como parámetro el puntero a una celda y un valor. Pone como valor el valor recibido. |
| | celda_leer_valor | Recibe como parámetro una celda. Devuelve el valor almacenado en ella. |
| | esVacía | Recibe como parámetro una celda. Devuelve 0 si su valor es igual a 0. |
| | eliminaCandidatos | Recibe como parámetro el puntero a una celda. Pone a 0 todos sus candidatos. |
| | esPista | Devuelve 1 si la celda seleccionada es pista. 0 en caso contrario |
| | esError | Devuelve 1 si la celda seleccionada es error. 0 en caso contrario |
| | celda_borrarError | Pone a 0 el bit de error de la celda *celdaptr |
| | celda_ponerError | Pone a 1 el bit de error de la celda *celdaptr |
| | celda_noEsCandidato | Devuelve 1 si no es candidato valor en la celda. 0 en caso contrario |
| tableros.h | borrar_tablero | Borra el tablero para dejarlo solo con las pistas |
| | check_resuelto | Comprueba si el tablero ha sido resuelto mirando que ninguna celda sea error y que todas tengan valor |

5.3.3 Micro kernel

| MICRO KERNEL | | |
|----------------|----------------------------|---|
| Fichero | Funciones | Descripción |
| SWI.s | SWI_Handler | |
| | __disable_isr | Desabilita las interrupciones isr |
| | __enable_isr | Habilir las interrupciones isr |
| | __disable_isr_fiq | Desabilita las interrupciones isr y las fiq |
| | __enable_isr_fiq | Habilita las interrupciones isr y las fiq |
| cola.c | cola_crear_vacia | Crea la cola circular vacía, es decir, sin eventos. |
| | cola_guardar_evento | Si numEventos < maxEventos devuelve la cola circular resultante de añadir a c el Evento e. Si no, devuelve la cola resultante de añadir a c el Evento e, eliminando el Evento más antiguo de la cola, activando el flag V (Overflow). |
| | cola_desencola_mas_antiguo | Si numEventos > 0, devuelve el Evento más antiguo de la cola (el primero) y devuelve la cola circular resultante de eliminar de la cola dicho Evento. En caso contrario, devuelve un Evento inválido. |
| | cola_es_vacia | Devuelve 1 (verdad) sí y sólo si la cola c no tiene ningún Evento. Devuelve 0 (falso) en caso contrario. |
| | cola_ultimo | Si numEventos > 0, devuelve el Evento más nuevo encolado en la cola (el último). En caso contrario, devuelve un evento inválido. |
| | vaciar_cola | Vacía la cola y crea una nueva vacía |
| | | |
| evento.c | es_valido | Devuelve 1 si el evento tiene id 255 InvalidID. Devuelve 0 en caso contrario. |
| | el_invalido | Devuelve un evento inválido |
| planificador.c | planificador_main | Planificador infinito que va desencolando eventos e inicializa todo el sistema. Si la cola se llena WatchDog reiniciará la partida. |
| main.c | main | Programa principal que llama a planificador_main |

5.3.4 Manejadores

| GESTORES | | |
|--|----------------------------|--|
| Fichero | Funciones | Descripción |
| gestor_IO.c gestor_IO.h | gIO_inicializar | Inicializa el gestor_IO iniciando la GPIO y pone una alarma periódica que Check_Entrada para mostrar en el GPIO el valor de la celda seleccionada (valor y candidatos) |
| | gIO_reset | Vuelve a poner una alarma periodica para el latido tras resetear una partida |
| | gIO_encender_overflow | Escribe el bit de overflow del GPIO para saber si ha desbordado la cola de eventos |
| | gIO_leer_overflow | Lee el estado del bit de overflow del GPIO para saber si ha desbordado la cola de eventos |
| | gIO_apagar_validacion | Apaga el led de validación (bit 13 GPIO) |
| | gIO_encender_validacion | Enciende el led de validación (bit 13 GPIO) |
| | gIO_encender_latido | Enciende el bit 31 del GPIO |
| | gIO_apagar_latido | Apaga el bit 31 del GPIO |
| | gIO_alternar_latido | Alterna el latido del GPIO encendiendo el bit 31 si estaba apagado y apagandolo si estaba encendido |
| | gIO_alternar_validacion | Alterna la validacion del GPIO encendiendo el bit 13 si estaba apagado y apagandolo si estaba encendido |
| gestor_pulsacion.c gestor_pulsacion.h | gp_inicializar | Inicializa las interrupciones en EINT1 |
| | gp_leer_pulsacion_1 | Devuelve 1 si el estado es PULSADO. Devuelve 0 en caso contrario |
| | leer_estado_1 | Devuelve el estado de EINT1 |
| | leer_entrada_1 | Devuelve 1 si el botón sigue pulsado, 0 en caso contrario |
| | gp_actualizar_estado_EINT1 | Comprueba si se ha levantado la pulsacion y actualiza el estado. Si se ha levantado la pulsación vuelve a habilitar las interrupciones en el VIC |
| gestor_energia.c gestor_energia.h | ge_inicializar | Inicia la alarma para poner al procesador a PWDDOWN |
| | ge_modos_IDE | Pone al procesador en modo IDE |
| | ge_modos_pwdwn | Pone al procesador en modo powerdown |

5.3.5 Dependiente del hardware

| DEPENDIENTE DEL HARWARE | | |
|--|---|---|
| Fichero | Funciones | Descripción |
| gpio.c gpio.h | GPIO_iniciar | Inicializa el GPIO |
| | GPIO_leer (uint8_t bit_inicial, uint8_t num_bits) | bit_inicial indica el primer bit a leer. num_bits indica cuántos bits queremos leer. La función devuelve un entero con el valor de los bits indicados. |
| | GPIO_escribir (uint8_t bit_inicial, uint8_t num_bits, uint32_t valor) | similar al anterior, pero en lugar de leer escribe en los bits indicados el valor (si valor no puede representarse en los bits indicados se escribirá los num_bits menos significativos a partir del inicial) |
| | GPIO_marcar_entrada (uint8_t bit_inicial, uint8_t num_bits) | Los bits indicados se utilizarán como pines de entrada desde bit_inicial hasta bit_inicial + num_bits. |
| | GPIO_marcar_salida (uint8_t bit_inicial, uint8_t num_bits) | Los bits indicados se utilizarán como pines de salida desde bit_inicial hasta bit_inicial + num_bits. |
| buttons.c buttons.h | eint1_ISR | ISR para las EINT1 |
| | button_nueva_pulsacion_1 | Devuelve 1 si ha habido una nueva pulsación |
| | button_clear_nueva_pulsacion_1 | Resetea a 0 las nuevas pulsaciones |
| | button_enable_interrupts_1 | Habilita interrupciones en VIC |
| | eint1_init | Habilita las interrupciones EINT1 en el pin P0.14 |
| idle_pwdwn.c idle_pwdwn.h | idle | Pone al procesador en modo IDLE para ahorrar energía |
| | pwdwn | Pone al procesador en modo powerDown y sólo se despertará por las interrupciones externas marcadas EINT1 |
| timers.c timers.h | temporizador_iniciar | Inicializa el timer1 para que pueda ser usado |
| | temporizador_empezar | Inicia la cuenta en el timer1 |
| | reset_timer1 | Resetea el timer 1 |
| | temporizador_leer | Devuelve el tiempo que lleva contado timer1 |
| | temporizador_parar | Para el temporizador y devuelve el tiempo transcurrido |
| | temporizador_peridico | Programa el timer0 para que encole un evento periódicamente en ms |
| relojes.c relojes.h | RTC_init | Inicializa el RTC |
| | RTC_reset | Resetea el RTC |
| | RTC_leer_minutos | Devuelve los minutos que lleva contados el RTC |
| | RTC_leer_segundos | Devuelve los segundos que lleva contados el RTC |
| | WD_init | Inicializa el WatchDog y se prepara para alimentarlo con una alarma periodica cada 5 seg |
| | Alimenta el WatchDog | WD_feed |

| | | |
|--------------------------------|------------------------|--|
| serial_port.c serial_port.h | UART_actualizar | Actualiza el sudoku mostrado en la UART |
| | UART_ha_terminado | Devuelve 1 si se ha terminado de enviar el string a la UART. 0 en caso contrario |
| | UART_enviar_candidatos | <p>Escribe por la UART los candidatos de la celda correspondiente a la fila fil y de la columna col.</p> <p>Si la celda seleccionada era una pista devuelve el mensaje "La celda seleccionada es una pista"</p> |
| | UART_introducir_jugada | <p>Procesa la jugada que el usuario quiere introducir en la celda correspondiente a la fila fil y la columna col. Si la celda seleccionada era una pista devuelve el mensaje "La celda en la que quieres introducir valor es una pista"</p> <p>Si la jugada es valida, envia el mensaje "¿Confirmar jugada?" la realiza, pone una alarma a 3 seg para que el usuario valide la jugada y enciende con otra alarma el led de latido.</p> |
| | UART_acaba_jugada | Apaga el led de latido de validación con una alarma con su tiempo a 0 |
| | UART_cancelar_jugada | Si el usuario no valida la jugada, se cancela y muestra el mensaje "Jugada cancelada" |
| | UART_init_serial | Inicializa las interrupciones de la UART0 |
| | UART_enviar_string | Manda el string a la UART. Copia el string en un buffer send_buffer y manda el primer caracter del buffer a la UART |
| | UART_borrar_tablero | Borra el tablero y recalcula los candidatos |
| | en_jugada | Devuelve 0 si estamos en medio de una jugada, 1 en caso contrario |
| | UART_reset | Prepara la UART para un reinicio de partida |

5.4 Descripción de cada módulo

5.4.1 Planificador, cola y eventos

Para comenzar a bordar el diseño del planificador se desarrolla el elemento más importante, la cola de eventos, definida en cola.h y cola.c. esta será la responsable de ir encolando y desencolando los eventos que nos vayan llegando, sin embargo, antes del diseño de la cola pensamos en como representar los eventos que guardará.

A. Eventos

Para comenzar a bordar el diseño del planificador se desarrolla el elemento más importante, la cola de eventos, definida en cola.h y cola.c. esta será la responsable de ir encolando y desencolando los eventos que nos vayan llegando, sin embargo, antes del diseño de la cola pensamos en como representar los eventos que guardará.

Definimos en evento.h un struct que se compone de lo siguiente:

```
/*
 * Cada Evento es caracterizado por su ID (único), además cada
 * evento tiene una serie de datos asociados y una estampilla temporal que
 * indica cuando el evento fue generado.
 */
typedef struct Evento Evento;

struct Evento {
    uint8_t      ID_evento; //entero de 8 bits para el identificador del mismo
    uint32_t      auxData;   //32 bits para información necesaria para tratar con el evento
    uint32_t      timestamp; //32 bits que guarda el tiempo en ms que fue creado el evento
};
```

Para cada evento, tenemos una serie de identificadores los cuales definimos mediante una enumeración, poco a poco a medida que los vamos necesitando mientras desarrollamos el proyecto:

```
#define INVALID_ID 255 //ID de Los eventos inválidos

/*
 * Tipo de evento invalido
 */
static Evento eventoInvalido = {INVALID_ID, 0, 0};

/*
 * Tipo para definir el id de cada tipo de evento
 */
enum {
    Temp_perio = 0,
    Set_Alarma = 1,
    Check_Pulsacion_EINT1 = 2,
    Pulsacion_EINT1 = 3,
    Power_Down = 4,
    Apagar_Validacion = 5,
    Terminar = 6,
    Latido = 7,
```

```

    No_Confir_Jugada = 8,
    Latido_Validacion = 9,
    Candidatos = 10,
    Jugada = 11,
    Feed = 12,
    Start = 13,
    Check_Terminado_PWDOWN = 14
};

```

Además, añadimos una función `es_valido(Evento* e)` para saber que eventos son validos, debido a que hay funciones que han de devolver un dato de tipo evento de manera forzosa. Se define un evento inválido con identificador 255 el cual es devuelto en dichas ocasiones.

B. Cola de eventos

Para el diseño de la cola creamos un struct que se compone de lo siguiente:

- Array de "Eventos" de tamaño 32
- Un entero que guarda el índice del evento mas antiguo
- Un entero que guarda el índice del siguiente evento encolado
- Un entero que cuente el numero de eventos guardados en el array

```

#define MAX_EVENTOS 32

typedef struct Cola Cola;

struct Cola {
    Evento eventos[MAX_EVENTOS];
    uint8_t indPrimEv;      // Índice del evento más antiguo de la cola
    uint8_t indProxEv;      // Índice del siguiente evento a encolar
    uint8_t numEventos;
};

```

Creamos las operaciones básicas para gestionar una cola:

- `cola_crear_vacia()` para crear una cola vacía
- `cola_guardar_evento(Evento e)` para añadir un evento a la cola y en caso de pasarnos activar el led de overflow del GPIO
- `cola_desencola_mas_antigua()` para sacar de la cola el evento mas antiguo y devolverlo
- `cola_es_vacia()` para comprobar si la cola es vacía, devuelve 1 en caso afirmativo y 0 en caso contrario
- `cola_ultimo()` devuelve el evento mas nuevo encolado

C. Planificador

Se trata del programa que va a estar ejecutándose de manera infinita. Su comportamiento es el siguiente:

1. Inicializa todos los componentes del sistema creando la cola vacía, marca las entradas y salidas del GPIO para la interacción con el Sudoku, inicializa el gestor de alarmas, pulsación, energía y pasa a un bucle infinito.
2. Una vez comenzado el bucle infinito, primero se lee el bit de overflow del GPIO para saber si la cola de eventos ha dado error (se ha desbordado) en cuyo caso para la ejecución.
3. A continuación, va mirando si hay eventos en la cola y desencola el más antiguo (cola FIFO) y para cada tipo de evento si es válido realiza las operaciones que tenga que hacer.
4. Si la cola hubiese sido vacía, se pone al procesador en modo IDLE.

5.4.2 Temporizadores y Gestor de alarmas

A. Temporizadores

Para el desarrollo del sistema se utilizará el timer0 para la gestión de las alarmas y el timer1 para medir el tiempo del sistema.

Creamos las siguientes operaciones para interactuar con los temporizadores en donde llevaremos la cuenta de las interrupciones del timer1 con un entero volátil:

1. **Temporizador iniciar():** configura el timer1 para que interrumpa el numero mínimo de veces osea, cada $2^{32} - 1$ tics de reloj, además de configurar el reloj de los relojes para que trabajen a la máxima frecuencia posible. También habilitamos sus interrupciones y le asignamos su RSI, timer_ISR() que aumenta la cuenta de interrupciones del timer1
2. **Temporizador_empezar():** pone en marcha el timer1
3. **Temporizador_leer():** devuelve en us el tiempo del sistema
4. **Temporizador_parar():** para el timer1 y devuelve en us el tiempo del sistema
5. **Temporizador_periodico(int periodo):** configura el timer0 para generar una interrupcion cada <periodo> ms y se habilita sus interrupciones además de asignarle su RSI
6. **timer0_ISR():** encola un evento de comprobar alarmas en la cola de eventos

B. Gestor de alarmas

El gestor de alarmas es el encargado de iniciar los temporizadores y gestionar las alarmas que se configuran, para ello tiene un vector de ocho alarmas en donde cada uno de sus índices guardan una Alarma definida en gestor_alarmas.h del siguiente modo:

```
/*
 * Tipo de datos de una alarma
 */
typedef struct Alarma Alarma;
struct Alarma {
    uint8_t     esValida;           //indica 1 si es valida 0 no es valida
```



```

uint8_t  IDevento;
uint8_t  esPeriodica;
int      periodo;           //ms a disparar la alarma
int      timestamp;         //timestamp actual en ms
int      timeToLeave;        //tiempo en el que se ha de sacar la alarma en ms
};

```

Las funciones incluidas en gestor_alarmas.c son las siguientes:

1. **ga_inicializar(void):** marca todas las alarmas a inválidas e inicia los temporizadores. El temporizador periódico se marca a 50ms, debido a que todas las alarmas que se encolan son múltiplo de 50.
2. **ga_comprobar_alarmas(void):** cada vez que llega un evento de temporizador periódico que desencola el planificador se comprueban todas las alarmas con el instante de tiempo actual para ver si se han de encolar en el planificador. Si ese es el caso, si una alarma no es periódica se desactiva. Si es periódica se actualiza su timeToLeave.
3. **actualizar_alarma(Alarma al):** operación interna. Comprueba si una nueva alarma que nos llega está ya en la lista de alarmas. Si está y el tiempo indicado en la nueva alarma es cero, la alarma se desactiva, si no, actualiza el tiempo timeToLeave de la alarma. Si existe la alarma devuelve 1, si no devuelve 0.
4. **crear_alarma(Evento e):** operación interna. Crea una alarma a partir de un evento Set_alarma.
5. **ga_guardar_evento(Evento e):** Crea una alarma a partir de un evento. Si no se ha de actualizar la alarma, la introduce en el vector de alarmas

5.4.3 GPIO

Para la interacción con el GPIO creamos gpio.h con distintas operaciones:

1. **GPIO_iniciar():** que configura los pines del GPIO para que este de acuerdo al sistema:
bit 31: output | bit 30: output | bit 27-24: input | bit 23-20: input | bit 19-16: input | bit 15-14: input | bit 13: output | bit 12-4 : output | bit 3-0: output
2. **GPIO_leer(uint8_t bit_inicial, uint8_t num_bits):** que devuelve un entero de 32 bits con los bits leídos desde <bit_inicial> hasta <bit_inicial> + <num_bits>
3. **GPIO_escribir(uint8_t bit_inicial, uint8_t num_bits, uint32_t valor):** que escribe <valor> desde <bit_inicial> hasta <bit_inicial> + <num_bits> y en el caso de que <valor> tuviese mas bits que <num_bits> escribiera los de menos peso.
4. **GPIO_marcar_entrada(uint8_t bit_inicial, uint8_t num_bits):** configura el GPIO para que los bits desde <bit_inicial> hasta <bit_inicial> + <num_bits> sean leds de entrada.
5. **GPIO_marcar_salida(uint8_t bit_inicial, uint8_t num_bits):** configura el GPIO para que los bits desde <bit_inicial> hasta <bit_inicial> + <num_bits> sean leds de salida.

5.4.4 Gestor de pulsación

El gestor de pulsación definido en `gestor_pulsacion.c` es el encargado de indicarnos cuando se ha levantado la pulsación del botón por parte del usuario. En la RSI del botón, la primera vez que nos interrumpen se deshabilitan las interrupciones de botón en el VIC para que sólo nos interrumpan una vez. Se crea una pequeña máquina de estados que según la entrada pone a pulsado o no pulsado el estado del botón. Si ha sido despulsado, vuelve a habilitar las interrupciones en el VIC del siguiente modo.

```
/*
 * Comprueba si se ha levantado la pulsacion y actualiza el estado.
 * si se ha levantado la pulsación vuelve a habilitar las interrupciones
 * en el VIC
 */
void gp_actualizar_estado_EINT1(void) {
    unsigned int estado, entrada;
    int retardo;
    Evento eAlarma;

    if (button_nueva_pulsacion_1() == 1) { //ha habido una nueva pulsacion se ajusta el estado
        estado_pulsacion_EINT1 = PULSADO;
    }
    estado = Leer_estado_1();
    entrada = Leer_entrada_1();
    switch(estado) {
        case PULSADO:
            if (entrada == 0) { // se ha despulsado el boton
                estado_pulsacion_EINT1 = NO_PULSADO;
                button_enable_interrupts_1();
            }
            break;
        case NO_PULSADO:
            if (entrada == 1) { // se ha pulsado
                estado_pulsacion_EINT1 = PULSADO;
            }
            break;
    }
}
```

En donde `button_enable_interrupts_1` simplemente habilita en el VIC Y `leer_entrada` lee la entrada actual del botón del siguiente modo:

```
unsigned int Leer_entrada_1(void) {
    EXTINT = EXTINT | 2; // clear interrupt flag de EINT0, EINT1 y EINT2
    if ((EXTINT & 0x2) == 2) {
        return 1; //EL botón sigue pulsado
    }
    return 0; //EL botón no está pulsado
}
```

Podemos también dibujar el siguiente autómata (figura 2) que define el comportamiento de `gp_actualizar_entrada()`:

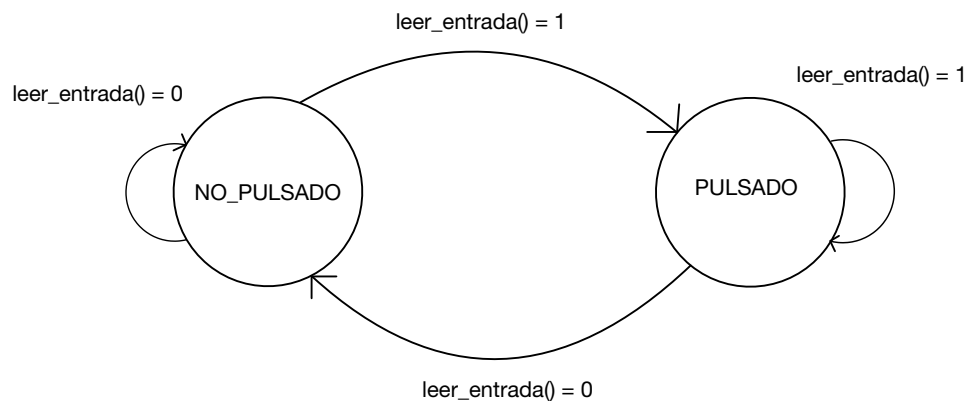


Figura 2: autómata para la pulsación de los botones

5.4.5 Reloj de tiempo real y watchdog

Para tratar con el RTC se crea una función `RTC_init` que resetea el mismo y lo pone en marcha para contar a tiempo real.

```

void RTC_init(void){
    //Tratar con el CCR o registro de control del RTC(5 bits)
    CCR = 2;    //Resetear La cuenta(Poner a 1 el bit 1 del CCR)
    CCR = 1;    //Enablear el reloj(Poner el bit 0 a 1) y desactivar el reset(poner bit 1 a 0)

    //Modificar PREINT para La frecuencia
    PREINT = 1829; //(CCLK(MHz)/32768 - 1) CCLK = 60MHz
    PREFRAC = 34560; // Lo que se trunca al hacer La division
    PCOMP = PCOMP | 0x100;
}
  
```

Del RTC queremos sacar los minutos y los segundos de partida por lo que creamos las funciones `RTC_leer_minutos` y `RTC_leer_segundos` que devuelven un entero de 32 bits de los minutos y los segundos respectivamente. Además, se crea una función `RTC_reset` que hace que las funciones anteriores devuelvan los minutos y los segundos respectivos a la nueva partida creada.

```

uint32_t RTC_leer_minutos(void) {
    uint32_t res;
    res = MIN - old_min;
    return res;
}

uint32_t RTC_leer_segundos(void) {
    uint32_t res;
    res = SEC - old_sec;
    return res;
}

void RTC_reset(void){
    old_min = MIN;
    old_sec = SEC;
    CCR = 0;
}
  
```

```
}

```

En lo relativo al watchdog, se crea una función WD_init que recibe como parámetro los segundos para que el reloj reinicie al sistema en caso de no alimentación y lo configura para ello, también creamos la función WD_feed que garantiza la alimentación del watchdog en exclusión mutua.

```
void WD_init(int sec){
    //Poner el WDEN(bit 0 del WDMOD) y el WDRESET(bit 1 del WDMOD) a 1
    //Asignar <sec> como tiempo para resetear para el WD
    WDTC = sec * 256 * 61476;
    WDMOD = 3;
    disable_isr_fiq();
    set_Alarma(Feed, 5000, 1);
    enable_isr_fiq();
}

void WD_feed(void){
    disable_isr_fiq();
    WDFEED = 0xAA;
    WDFEED = 0x55;
    enable_isr_fiq();
}
```

5.4.6 Implementacion de las llamadas al sistema

La implementación de las llamadas al sistema se hace mediante las Software Interrupt para así pasar de modo usuario a modo supervisor, lo cual nos permite realizar ciertas operaciones restringidas para el modo usuario.

Creamos las funciones enable_isr, enable_isr_fiq, disable_isr, disable_isr_fiq, para deshabilitar las interrupciones y así evitar condiciones de carrera. La implementación es sencilla, simplemente cambiamos la palabra de estado del modo usuario para que el bit I y el bit F queden a 1 para deshabilitarlas o a 0 para habilitarlas.

```
void __swi(0xFF) enable_isr (void);
void __swi(0xFE) disable_isr (void);
void __swi(0xFD) enable_isr_fiq (void);
void __swi(0xFC) disable_isr_fiq (void);

__enable_isr
    LDMFD SP!, {R8, R12}      ; Load R8, SPSR
    PUSH {r0}
    mvn r0, #0x80
    and r12, r12, r0
    POP {r0}
    MSR SPSR_cxsf, R12      ; Set SPSR
    LDMFD SP!, {R12, PC}^    ; Restore R12 and Return

__disable_isr
    LDMFD SP!, {R8, R12}      ; Load R8, SPSR
    PUSH {r0}
    mov r0, #0x80
    orr r12, r12, r0
    POP {r0}
    MSR SPSR_cxsf, R12      ; Set SPSR
    LDMFD SP!, {R12, PC}^    ; Restore R12 and Return

__enable_isr_fiq
```

| | |
|-------------------|--|
| | <pre> LDMFD SP!, {R8, R12} ; Load R8, SPSR PUSH {r0} mvn r0,#0xC0 and r12,r12,r0 POP {r0} MSR SPSR_cxsf, R12 ; Set SPSR LDMFD SP!, {R12, PC}^ ; Restore R12 and Return </pre> |
| __disable_isr_fiq | <pre> LDMFD SP!, {R8, R12} ; Load R8, SPSR PUSH {r0} mov r0,#0xC0 orr r12,r12,r0 POP {r0} MSR SPSR_cxsf, R12 ; Set SPSR LDMFD SP!, {R12, PC}^ ; Restore R12 and Return </pre> |

La función que lee el temporizador para hacer timestamps de eventos y gestionar alarmas es cambiada por una llamada al sistema clock_gettime la cual llama directamente a temporizador_leer y devuelve el resultado de la misma.

| |
|---|
| uint32_t __swi(0) cLock_gettime(void); |
| <pre> uint32_t __SWI_0 (void){ return temporizador_leer(); } </pre> |

5.4.7 Eliminación de las condiciones de carrera

Gracias a las llamadas al sistema que deshabilitan las interrupciones podemos asegurar la atomicidad de ciertas secuencias de instrucciones, tales como la alimentación del watchdog la cual requiere de dos escrituras simultaneas, para impedir la adición de un evento a la cola cuando se esta encolando o desencolando un evento y para evitar la corrupción de la aplicación cuando una ISR interrumpe a otra o una FIQ.

| |
|--|
| <pre> void WD_feed(void){ disable_isr_fiq(); WDFEED = 0xAA; WDFEED = 0x55; enable_isr_fiq(); } </pre> |
| <pre> Evento cola_desencola_mas_antiguo(void) { Evento e; disable_isr_fiq(); e = el_invalido(); if(!cola_es_vacia()) { e = c.eventos[c.indPrimEv]; c.indPrimEv = (c.indPrimEv + 1) % MAX_EVENTOS; c.numEventos--; } enable_isr_fiq(); return e; } </pre> |
| <pre> void timer1_ISR(void) __irq { disable_isr_fiq(); timer1_int_count++; } </pre> |

```

T1IR = 1;
VICVectAddr = 0;
enable_isr_fiq();
}

```

5.4.8 Timer0 como fast interrupt

Para ello al inicializar el timer0 ya no le asignamos una dirección en el VICVectAddr y por ello tenemos que modificar el Startup para decirle que hacer cuando salte su interrupción y asignarle un espacio de pila.

| | | |
|----------------|-----|------------|
| IRQ_Stack_Size | EQU | 0x00000080 |
| FIQ_Addr | DCD | timer0_ISR |

5.4.9 Integración de la I/O en el juego

Como ya se ha mostrado antes se configuran los leds del GPIO para este cometido:

1. GPIO13: led validar jugada
2. GPIO14: EINT1 (escribir dato en celda seleccionada)
3. GPIO30: overflow
4. GPIO31: latido modo idle

5.4.10 UART

A. Inicialización de la UART

La implementación de la interacción con el usuario se realiza mediante la UART1.

Para ello primero se inicializa la UART1 de la siguiente manera para que nos pueda interrumpir:

```

void UART_init_serial(void) {
    PINSEL0 = PINSEL0 | 0x5;           // Initialize Serial Interface
    U0LCR = 0x83;                      // Enable Rx/D1 and Tx/D1
    U0DLL = 97;                        // 8 bits, no Parity, 1 Stop bit
    U0LCR = 0x03;                      // Div Latch LSB 9600 Baud Rate @ 15MHz VPB Clock
    //habilitar interrupciones de uart1
    VICIntEnable = VICIntEnable | 0x00000040;
    VICVectAddr5 = (unsigned long)serial_ISR;
    VICVectCntl5 = 0x20 | 6;
    U0IER = U0IER | 0x3;               // Interrupt Enable Register, habilitar las RBR
                                     // (Receiver send_buffer Register) y las THR
                                     // (Transmit Holding Register)
}

```

B. Enviar un string a la UART

Una vez inicializada la UART nos interesa tener una función para poder enviar de golpe un string a la UART y no tener que enviar carácter por carácter:

```

void UART_enviar_string(char *string) {
    memset(&send_buffer[0], 0, sizeof(send_buffer)); //Wipe previous buffer
    size_send_buffer = 0;
    index_send_buffer = 0;
}

```

```

terminado = 0;

//Copiar el string a mi 'send_buffer' y controlar que no nos salgamos del limite
while (string[size_send_buffer] != '\0' && size_send_buffer < MAX_SEND_BUFFER) {
    send_buffer[size_send_buffer] = string[size_send_buffer];
    size_send_buffer++;
}
U0THR = send_buffer[index_send_buffer];           //manda el primer caracter a la UART0
index_send_buffer++;
}

```

En donde se le pasa a la función un char* que se copia a un buffer (send_buffer). Una vez copiado dicho buffer se le manda a la UART el primer carácter, entonces, una vez que nos empiece a interrumpir se le mandará el siguiente carácter de ese buffer dentro de la RSI hasta que termine.

C. RSI de la UART

Dentro de la RSI lo primero que se mira es si la RSI ha sido invocada por un mensaje para mostrar por pantalla o bien si ha sido invocada por la entrada de un usuario aplicando una máscara al U0IIR para quedarnos con los bits 3:1 que nos dicen que tipo de interrupción ha sido.

Si se ha producido una interrupción por un input de un usuario en la UART

- Se actualiza la alarma de powerdown (segumos jugando) y se resetea la cuenta.
- Actualizamos una máquina de estados (que explicaremos más adelante) que comprueba si el carácter introducido es correcto.
- Leemos el carácter recibido en la UART

Sino si ha sido porque queríamos escribir algo en la UART

- Llamamos a continuar_msj que envía el siguiente carácter del buffer a la UART

En la siguiente tabla podemos observar la ISR de la UART:

```

void serial_ISR (void) __irq {
    int mask;
    int retardo;
    disable_isr_fiq();
    VICVectAddr = 0;
    mask = 0xE;           // Nos quedamos con los bits 3:1 nos dicen que tipo de interr ha sido
    mask &= U0IIR;

    // Recibo datos de UART0 (3:1 = 010) (man pg 88)
    if (mask == 0x4) {
        if (UART_ha_terminado() && jugada == 0){
            //Actualizamos la alarma para resetear el countdown para irnos a powerdown
            retardo = TIME_PWDN & 0x07FFFFFF;           // Asegurarnos que retardo es de 23bits
            set_Alarma(Power_Down, retardo, 1);
            if (index_rec_buffer < MAX_REC_BUFFER) {
                rec_buffer[index_rec_buffer] = U0RBR; //nos da el char introducido en UART
                actualizar_estado(rec_buffer[index_rec_buffer]);
            }
            //Comprobación de cada tipo de comando para generar el evento correspondiente
            if (rec_buffer[index_rec_buffer] == '!') { // Fin del mensaje
                if (strcmp(rec_buffer, "#RST!") == 0) {

```

```

        // Terminar partida
        Evento terminar;
        empezado = 0;
        terminar.ID_evento = Terminar;
        cola_guardar_evento(terminar);
    }
    (...)
}
// Escribo datos en UART0 (3:1 = 010) (pg 88)
else if (mask == 0x2) {
    continuar_msj();
}
enable_isr_fiq();
}

```

D. Máquina de estados

Como hemos mencionado anteriormente cada vez que el usuario escribe un carácter en la UART comprobamos que el carácter es válido, es decir, que pertenezca a un comando correcto el cual debe ser:

- #NEW! Para comenzar una partida
- #RST! Para terminarla
- #fcC! Para mostrar los candidatos de una fila 'f' y de una columna 'c'
- #fcvs! Para introducir una jugada en la fila 'f' y columna 'c' con valor 'v' y checksum 's'

Si por algún casual el usuario introduce un comando erróneo, la máquina de estados lo detectará y enviará a la UART el string con el mensaje "Instrucción no reconocida" y volverá al estado inicial "ESPERA_CMD".

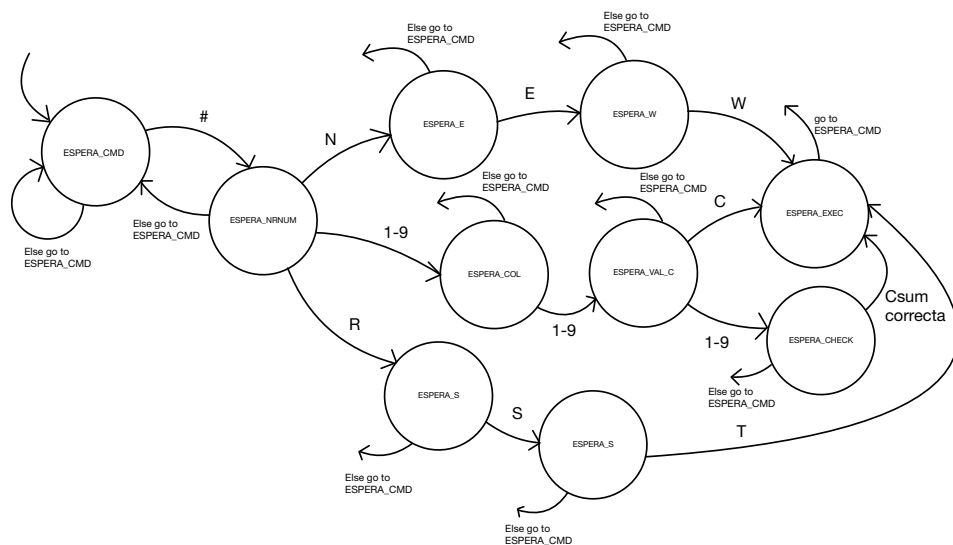


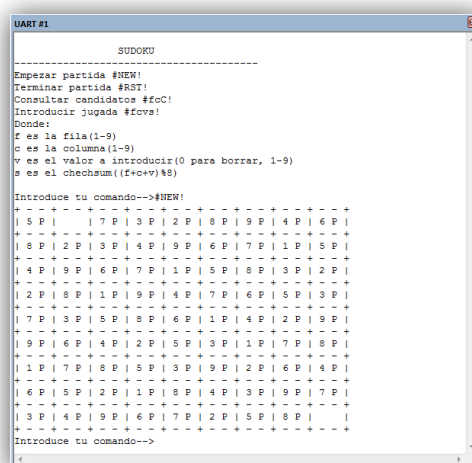
Figura 3: esquema de la máquina de estados

6 RESULTADOS

6.1.1 Iniciar una partida

Antes de ejecutarlo hay que asegurarse de tener **View > Periodic Window Update** habilitado en la sesión de debug y tener abiertas las ventanas **View > Serial Windows > UART#1** y **Peripherals > GPIO**.

Aquí se muestra como al iniciar el programa te muestra los comandos posibles y su formato, se introduce NEW para empezar la partida. Se muestra el tablero por pantalla y pide otro comando.



```
UART #1
SUDOKU
-----
Empezar partida #NEW!
Terminar partida #RST!
Consultar candidatos #fC!
Introducir jugada #fCv!
Donde:
f es la fila(1-9)
c es la columna(1-9)
v es el valor a introducir(0 para borrar, 1-9)
s es el checksum((f+c+v)%8)

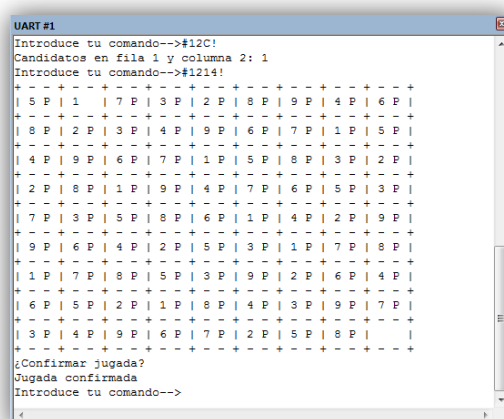
Introduce tu comando-->#NEW!
+---+---+---+---+---+---+---+---+---+
| 5 P |   | 7 P | 3 P | 2 P | 8 P | 9 P | 4 P | 6 P |
+---+---+---+---+---+---+---+---+---+
| 8 P | 2 P | 3 P | 4 P | 9 P | 6 P | 7 P | 1 P | 5 P |
+---+---+---+---+---+---+---+---+---+
| 4 P | 9 P | 6 P | 7 P | 1 P | 5 P | 8 P | 3 P | 2 P |
+---+---+---+---+---+---+---+---+---+
| 2 P | 8 P | 1 P | 9 P | 4 P | 7 P | 6 P | 5 P | 3 P |
+---+---+---+---+---+---+---+---+---+
| 7 P | 3 P | 5 P | 8 P | 6 P | 1 P | 4 P | 2 P | 9 P |
+---+---+---+---+---+---+---+---+---+
| 9 P | 6 P | 4 P | 2 P | 5 P | 3 P | 1 P | 7 P | 8 P |
+---+---+---+---+---+---+---+---+---+
| 1 P | 7 P | 8 P | 5 P | 3 P | 9 P | 2 P | 6 P | 4 P |
+---+---+---+---+---+---+---+---+---+
| 6 P | 5 P | 2 P | 1 P | 8 P | 4 P | 3 P | 9 P | 7 P |
+---+---+---+---+---+---+---+---+---+
| 3 P | 4 P | 9 P | 6 P | 7 P | 2 P | 5 P | 8 P | 1 P |
+---+---+---+---+---+---+---+---+---+
Introduce tu comando-->
```

Figura 4: iniciar una partida

6.1.2 Consultar candidatos e introducir jugada

Una vez empezada la partida ponemos el comando de consulta de candidatos, en este caso los candidatos de la celda en la fila 1 y la columna 2 y muestra que el único candidato posible es el 1.

Después, se introduce el comando de jugada en la fila 1, columna 2 con valor 1, entonces se muestra el tablero con la jugada introducida y el programa pide una confirmación de jugada, la cual se confirmará mediante el led 14 del GPIO. Una vez confirmada se comunica por pantalla y pide un nuevo comando.

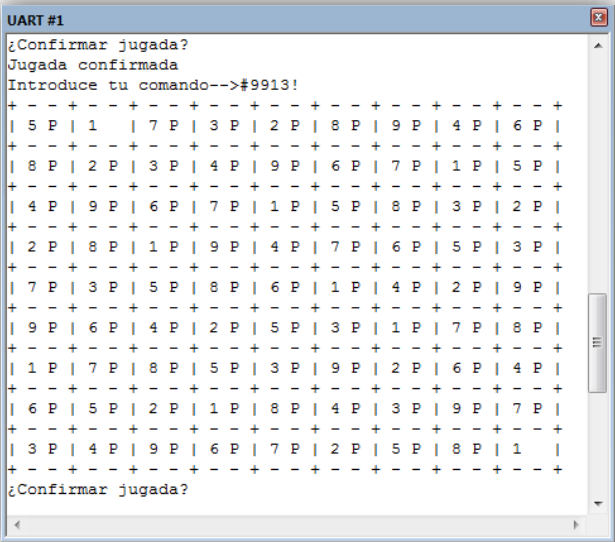


```
UART #1
Introduce tu comando-->#fC!
Candidatos en fila 1 y columna 2: 1
Introduce tu comando-->#fCv!
+---+---+---+---+---+---+---+---+---+
| 5 P | 1 | 7 P | 3 P | 2 P | 8 P | 9 P | 4 P | 6 P |
+---+---+---+---+---+---+---+---+---+
| 8 P | 2 P | 3 P | 4 P | 9 P | 6 P | 7 P | 1 P | 5 P |
+---+---+---+---+---+---+---+---+---+
| 4 P | 9 P | 6 P | 7 P | 1 P | 5 P | 8 P | 3 P | 2 P |
+---+---+---+---+---+---+---+---+---+
| 2 P | 8 P | 1 P | 9 P | 4 P | 7 P | 6 P | 5 P | 3 P |
+---+---+---+---+---+---+---+---+---+
| 7 P | 3 P | 5 P | 8 P | 6 P | 1 P | 4 P | 2 P | 9 P |
+---+---+---+---+---+---+---+---+---+
| 9 P | 6 P | 4 P | 2 P | 5 P | 3 P | 1 P | 7 P | 8 P |
+---+---+---+---+---+---+---+---+---+
| 1 P | 7 P | 8 P | 5 P | 3 P | 9 P | 2 P | 6 P | 4 P |
+---+---+---+---+---+---+---+---+---+
| 6 P | 5 P | 2 P | 1 P | 8 P | 4 P | 3 P | 9 P | 7 P |
+---+---+---+---+---+---+---+---+---+
| 3 P | 4 P | 9 P | 6 P | 7 P | 2 P | 5 P | 8 P | 1 P |
+---+---+---+---+---+---+---+---+---+
¿Confirmar jugada?
Jugada confirmada
Introduce tu comando-->
```

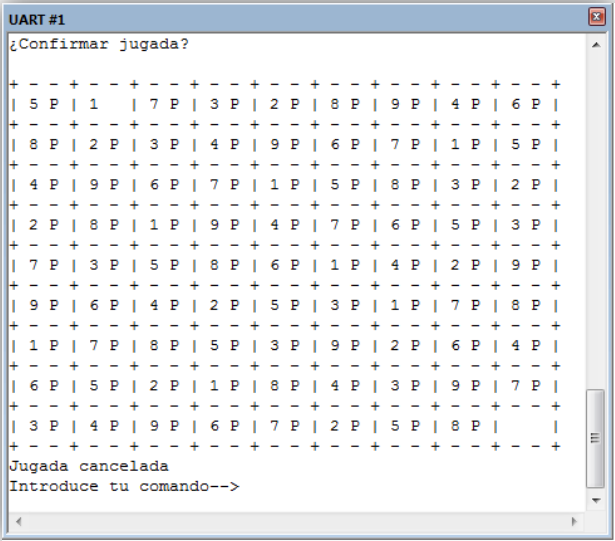
Figura 5: mostrar candidatos e introducir jugada

6.1.3 No confirmar una jugada

Ahora se consultan los candidatos de la celda en la fila 9 y columna 9 y muestra que el único posible es 1, entonces se introduce la jugada, pero pasan los 3 segundos de confirmación y la jugada se cancela, mostrando el tablero sin la jugada introducida.



```
UART #1
¿Confirmar jugada?
Jugada confirmada
Introduce tu comando-->#9913!
+ - + - + - + - + - + - + - + - + - + - +
| 5 P | 1 | 7 P | 3 P | 2 P | 8 P | 9 P | 4 P | 6 P |
+ - + - + - + - + - + - + - + - + - + - +
| 8 P | 2 P | 3 P | 4 P | 9 P | 6 P | 7 P | 1 P | 5 P |
+ - + - + - + - + - + - + - + - + - + - +
| 4 P | 9 P | 6 P | 7 P | 1 P | 5 P | 8 P | 3 P | 2 P |
+ - + - + - + - + - + - + - + - + - + - +
| 2 P | 8 P | 1 P | 9 P | 4 P | 7 P | 6 P | 5 P | 3 P |
+ - + - + - + - + - + - + - + - + - + - +
| 7 P | 3 P | 5 P | 8 P | 6 P | 1 P | 4 P | 2 P | 9 P |
+ - + - + - + - + - + - + - + - + - + - +
| 9 P | 6 P | 4 P | 2 P | 5 P | 3 P | 1 P | 7 P | 8 P |
+ - + - + - + - + - + - + - + - + - + - +
| 1 P | 7 P | 8 P | 5 P | 3 P | 9 P | 2 P | 6 P | 4 P |
+ - + - + - + - + - + - + - + - + - + - +
| 6 P | 5 P | 2 P | 1 P | 8 P | 4 P | 3 P | 9 P | 7 P |
+ - + - + - + - + - + - + - + - + - + - +
| 3 P | 4 P | 9 P | 6 P | 7 P | 2 P | 5 P | 8 P | 1 |
+ - + - + - + - + - + - + - + - + - + - +
¿Confirmar jugada?
```

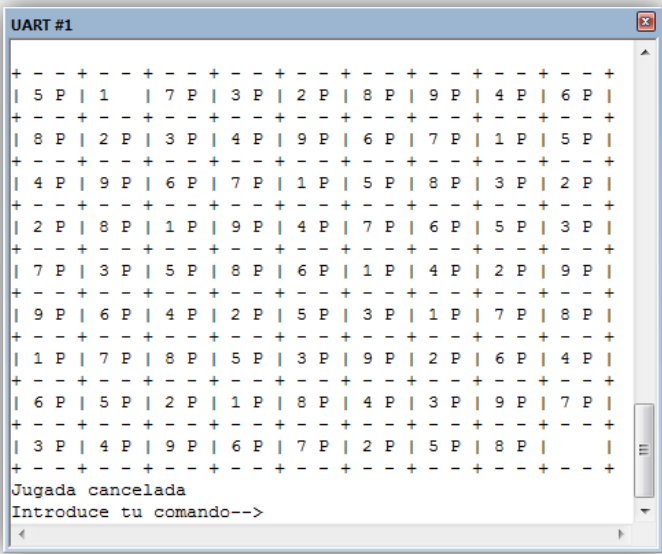
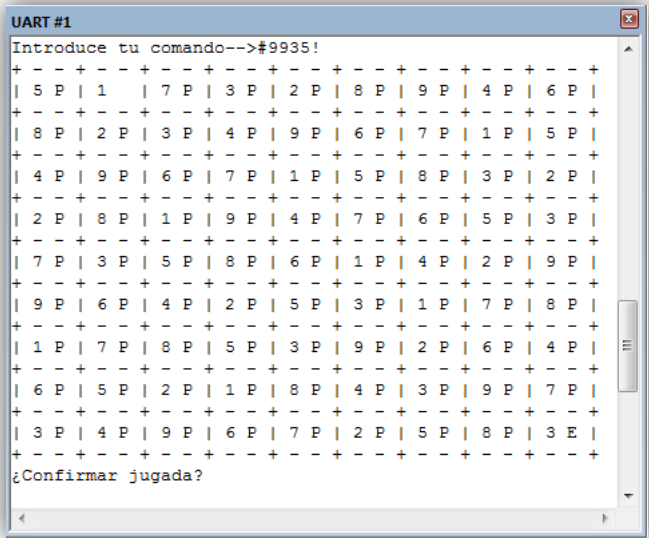


```
UART #1
¿Confirmar jugada?
Jugada cancelada
Introduce tu comando-->
```

Figuras 6 y 7: no confirmar jugada

6.1.4 Jugada errónea

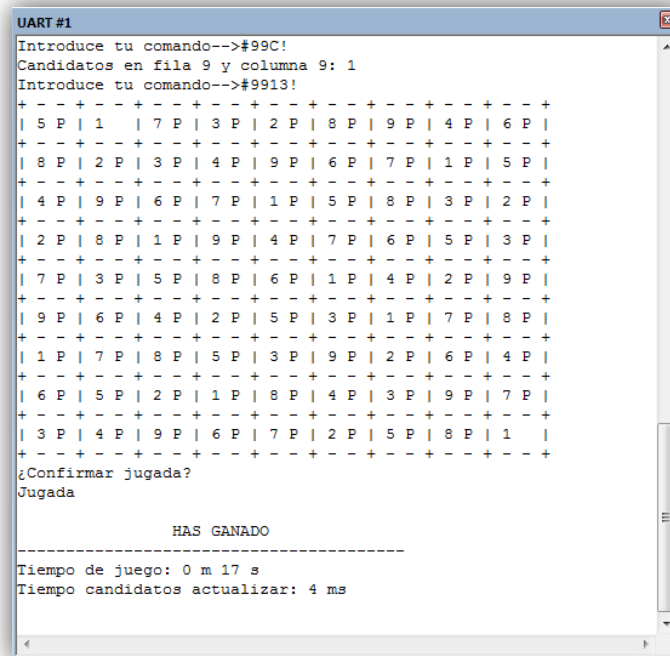
En caso de introducir una jugada errónea, un valor que no sea candidato, mostrara el tablero con el valor 3 introducido, pero con una E indicando que esa celda genera un error, por lo que no confirmamos la jugada.



Figuras 8 y 9: jugada errónea sin confirmar

6.1.5 Ganar partida

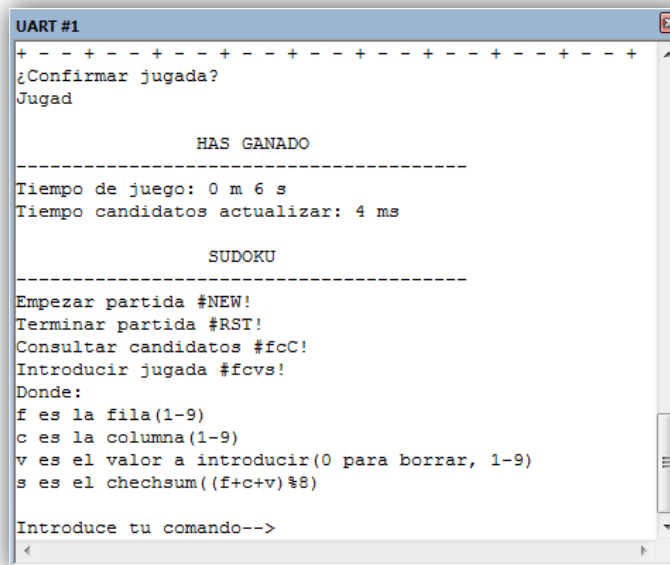
Ahora se introduce la jugada ganadora y una vez confirmada la jugada muestra el mensaje de HAS GANADO y termina la partida mostrando el tiempo de juego y el tiempo usado en calcular candidatos y muestra de nuevo el mensaje para empezar una nueva partida.



```
UART #1
Introduce tu comando-->#99C!
Candidatos en fila 9 y columna 9: 1
Introduce tu comando-->#9913!
+ - + - + - + - + - + - + - + - + - + - +
| 5 P | 1 | 7 P | 3 P | 2 P | 8 P | 9 P | 4 P | 6 P |
+ - + - + - + - + - + - + - + - + - + - +
| 8 P | 2 P | 3 P | 4 P | 9 P | 6 P | 7 P | 1 P | 5 P |
+ - + - + - + - + - + - + - + - + - + - +
| 4 P | 9 P | 6 P | 7 P | 1 P | 5 P | 8 P | 3 P | 2 P |
+ - + - + - + - + - + - + - + - + - + - +
| 2 P | 8 P | 1 P | 9 P | 4 P | 7 P | 6 P | 5 P | 3 P |
+ - + - + - + - + - + - + - + - + - + - +
| 7 P | 3 P | 5 P | 8 P | 6 P | 1 P | 4 P | 2 P | 9 P |
+ - + - + - + - + - + - + - + - + - + - +
| 9 P | 6 P | 4 P | 2 P | 5 P | 3 P | 1 P | 7 P | 8 P |
+ - + - + - + - + - + - + - + - + - + - +
| 1 P | 7 P | 8 P | 5 P | 3 P | 9 P | 2 P | 6 P | 4 P |
+ - + - + - + - + - + - + - + - + - + - +
| 6 P | 5 P | 2 P | 1 P | 8 P | 4 P | 3 P | 9 P | 7 P |
+ - + - + - + - + - + - + - + - + - + - +
| 3 P | 4 P | 9 P | 6 P | 7 P | 2 P | 5 P | 8 P | 1 P |
+ - + - + - + - + - + - + - + - + - + - +
¿Confirmar jugada?
Jugada

HAS GANADO

-----
Tiempo de juego: 0 m 17 s
Tiempo candidatos actualizar: 4 ms
```



```
UART #1
+ - + - + - + - + - + - + - + - + - + - +
¿Confirmar jugada?
Jugad

HAS GANADO

-----
Tiempo de juego: 0 m 6 s
Tiempo candidatos actualizar: 4 ms

SUDOKU

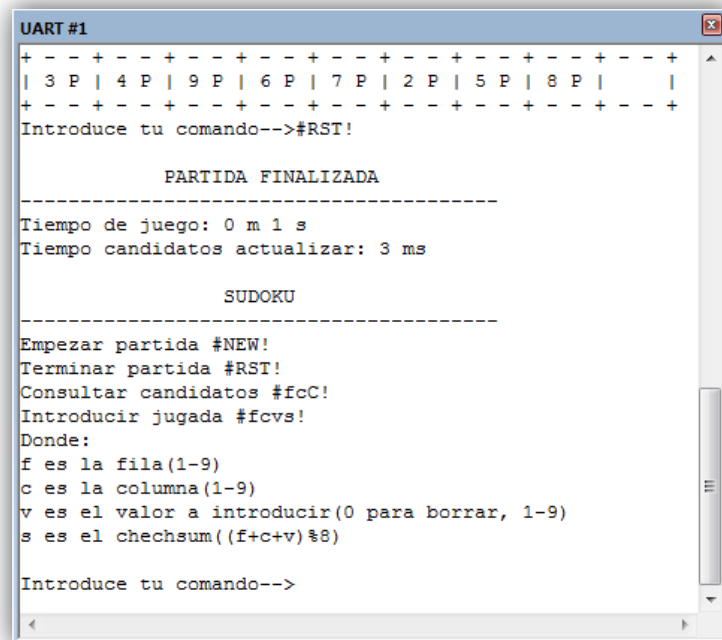
-----
Empezar partida #NEW!
Terminar partida #RST!
Consultar candidatos #fcC!
Introducir jugada #fcvs!
Donde:
f es la fila(1-9)
c es la columna(1-9)
v es el valor a introducir(0 para borrar, 1-9)
s es el checksum((f+c+v)%8)

Introduce tu comando-->
```

Figuras 10 y 11: jugada ganadora y muestra de menú inicial

6.1.6 Nueva partida

Entonces se introduce de nuevo el comando NEW para empezar una nueva partida para terminarla con el comando RST y muestra el tiempo de la nueva partida. Una vez finalizada la partida se invoca al modo powerdown del que podremos salir con el botón EINT1 y mostrará el menú principal de nuevo.



```
UART #1
+ - + - + - + - + - + - + - + - + - + - +
| 3 P | 4 P | 9 P | 6 P | 7 P | 2 P | 5 P | 8 P |
+ - + - + - + - + - + - + - + - + - + - +
Introduce tu comando-->#RST!

PARTIDA FINALIZADA
-----
Tiempo de juego: 0 m 1 s
Tiempo candidatos actualizar: 3 ms

SUDOKU
-----
Empezar partida #NEW!
Terminar partida #RST!
Consultar candidatos #fcC!
Introducir jugada #fcvs!
Donde:
f es la fila(1-9)
c es la columna(1-9)
v es el valor a introducir(0 para borrar, 1-9)
s es el checksum((f+c+v)%8)

Introduce tu comando-->
```

Figura 12: nueva partida con reset

7 CONCLUSIONES

Para terminar, el hecho de haber programado nosotros en bajo nivel nos permite optimizar la ejecución y controlar los aspectos de interacción con el jugador, sin embargo, lleva más tiempo y es más complicado crear uno mismo el sistema que ejecuta la aplicación.

La mayor ventaja que nos aporta nuestra implementación es un mayor control, de tal modo que podemos ajustar el comportamiento del dispositivo a nuestro gusto haciendo cosas como, por ejemplo, seleccionar las entradas/salidas customizar las invocaciones a modos como powerdown, elegir las políticas de planificación y gestión de eventos, etc.