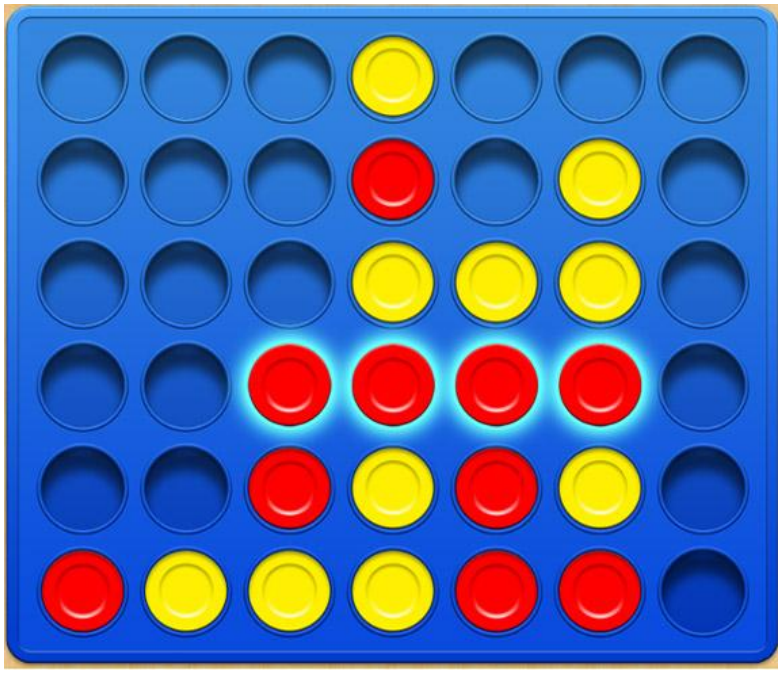
JUEGO CONECTA-K
- PRÁCTICA FINAL -

PROYECTO HARDWARE



Carlota Moncasi Gosá, 839841
Pablo Pina Gracia, 840020

Índice

1. Resumen ejecutivo

El programa Conecta K se basa en un planificador que orquesta las tareas a realizar, en función del evento que le llega. Estos eventos pueden llegar por interacción del usuario (se encola el evento en el momento) o por alarmas programadas en el mismo planificador (se encola el evento cuando salta la alarma, pasado un determinado tiempo).

El periférico de GPIO (General-Purpose I/O) va a ser utilizado como teclado, con dos de sus puertos simulando botones para que el usuario los pulse, como cuenta que muestra la función hello_world y como led encendido de indicación en caso de error. Se va a utilizar el timer 1 como periférico para construir un módulo que gestione alarmas periódicas en el sistema. La placa LPC2105 incluye dos temporizadores de propósito general (timers 0 y 1), un WatchDog para resetear el procesador en caso de bucle infinito, y un RTC (Real Time Clock) para la hora y fecha. La biblioteca a implementar se dividirá en funciones dependientes del hardware (HAL - Hardware Abstraction Layer) y del gestor de dispositivo (DRV - driver). Con el fin de gestionar ambos periféricos, se hace uso de un controlador de interrupciones (VIC) y la placa de pines que se puede consultar en las hojas técnicas proporcionadas.

La comunicación entre el programa y el usuario se realiza a través de la línea serie. El código se organiza en capas: linea_serie_drv.c y linea_serie_hal.c gestionan la transmisión y recepción de mensajes a través de línea serie, juego.c implementa la lógica del juego y utiliza funciones de bajo nivel y manejo de interrupciones. System_calls_hal.h define llamadas al sistema relacionadas con interrupciones como enable_irq(), disable_irq()..., y SWI_hal.s contiene el gestor de interrupciones de software (SWI) en ensamblador ARM y proporciona funciones para leer, habilitar y deshabilitar interrupciones IRQ y FIQ .

Además, se va a crear una Máquina de Estados Finitos (SFM) en el planificador para reducir el consumo del procesador, forzando que se duerma cuando no hay actividad. La SFM solicitará el paso del procesador a modo idle mediante la función power_hal_wait() cuando no haya eventos en la cola y a modo power-down con la función power_hal_deep_sleep(), si no hay actividad de usuario durante un tiempo determinado. Las alarmas deben ser reiniciadas una vez ya han saltado, tras la actividad del usuario y se debe reconfigurar el PLL (Phase-Locked Loop) al volver de power-down para que el programa siga funcionando como antes. También se deben deshabilitar todas las interrupciones externas al dormir.

2. Introducción

El juego Conecta-K es una versión hardware-software del tradicional juego 3 en raya o 4 en línea, pero aplicado a un tablero de m columnas por n filas, que se visualiza por la pantalla UART. A él juegan 2 jugadores, el jugador 1 que coloca fichas blancas, las cuales se muestran por pantalla como B y el segundo jugador, que coloca negras como N. En el primer turno, el jugador 1 juega 1 vez, y después será turno del jugador 2 que realizará 2 movimientos, después el jugador 1 con 2 movimientos y así sucesivamente hasta terminar la partida, bien por victoria o bien por rendición.

En este caso, el ganador es el primer jugador que consigue colocar 4 fichas en línea, ya sea horizontal, vertical o diagonal. No obstante, se puede cambiar la configuración de dimensiones del tablero, con cuántas fichas seguidas se consigue victoria,etc.
De esta manera, se asegura que sea un juego genérico y que funcione igual de bien en todas sus formas.

El usuario jugador interactúa con la máquina a través de la línea serie UART, donde escribe comandos como $NEW!, $2-3! o $END! especificando así las acciones que quiere realizar en el juego. Puede dar comienzo a una partida, ejecutar una jugada sobre el tablero, finalizar la partida e incluso cancelar una jugada antes de que pasen 3 segundos. También tiene la oportunidad de poder cancelar una jugada antes de un tiempo límite, por si el usuario se equivoca.

El proyecto Conecta K se centra en un planificador que coordina tareas basadas en eventos, ya sea por la interacción del usuario o mediante alarmas programadas en el planificador. Se empleará el periférico GPIO para simular botones y un LED indicador de cuentas, a través de dos de sus puertos, mientras que el temporizador 1 gestionará alarmas periódicas. La biblioteca a desarrollar se dividirá en funciones dependientes del hardware (HAL) y del gestor de dispositivo (DRV).

La comunicación con el usuario se realiza por línea serie, y el código está organizado en capas, desde la gestión de la línea serie hasta el manejo de interrupciones y la implementación del juego Conecta-K. La Máquina de Estados Finitos en el planificador organiza las tareas según los eventos que llegan de la cola y optimiza el consumo del procesador, activando modos de bajo consumo según la actividad del usuario y reiniciando alarmas tras la trata de eventos.

3. Objetivos

El objetivo principal es conseguir cumplir los siguientes requisitos funcionales y no funcionales del proyecto Conecta-K 2023:

- RF1: El juego deberá iniciar y terminar de manera ordenada.
- RF2: No habrá en el juego ninguna espera activa salvo las condiciones de overflow.
- RF3: Se gestionarán como llamadas al sistema la activación y desactivación de las rutinas de servicio de periféricos y la lectura del tiempo transcurrido.
- RF4: La entrada de la jugada se realizará a través de línea serie. El programa enviará al usuario/a el tablero para visualizar los movimientos por pantalla.
- RF5: Se dará opción de confirmar y cancelar mediante los botones.
- RF6: Al final de la partida, se presentarán métricas de rendimiento y calidad de servicio a través de la línea serie.
- RF7: En caso de mal funcionamiento el sistema se reseteará de forma autónoma.

- RNF1: Se puede depurar con nivel de optimización O0, pero la versión final debe funcionar correctamente en modo release con O3.

Estos objetivos se van a ir desglosando en distintas prácticas, para finalmente conformar el juego de Conecta-K en línea. Se explican a continuación las distintas tareas que se han ido definiendo para cada práctica de la asignatura:

Práctica 2a y 2b: realizar un programa en C para configurar la entrada/salida con dispositivos básicos al asignar valores a los registros internos de la placa, crear en C las rutinas de tratamiento de interrupciones, familiarizarse con el uso de periféricos como los temporizadores internos y General Purpose Input/Output (GPIO). También depurar eventos concurrentes asíncronos y las múltiples interrupciones entrantes a la vez, así como desarrollar código modular separado debidamente para facilitar la utilización de periféricos y diseñar un planificador que monitorea los eventos del sistema y gestiona su ejecución de manera eficiente.

Práctica 2c y 3: implementar y usar llamadas al sistema para interactuar con un dispositivo, un temporizador, y para activar y desactivar las interrupciones. Además, implementar un protocolo de comunicación con la pantalla UART a través del puerto serie e identificar condiciones de carrera, su causa y solucionarlas. Todo ello, asegurando el aprovechamiento de los modos del procesador.

4. Metodología

El proyecto se organiza en dos módulos distintos, HAL (Hardware Abstraction Layer) y Driver, con el objetivo de separar procedimientos de bajo nivel y abstraer el hardware del programa. Esto se hace para abstraer el hardware del programa, de este modo, se podría usar el mismo código con distintos procesadores cambiando tan sólo el módulo dependiente del hardware. El módulo HAL se dedica a funciones a nivel de hardware, mientras que el Driver o controlador, realiza funciones abstraídas del hardware, aunque en muchas ocasiones se limita a llamar a funciones del HAL. Ambos módulos contienen funciones simples para la inicialización, inicio, lectura y detención de un contador, junto con una constante para convertir los ticks a microsegundos y cada uno de los dos bloques, tendrá un fichero de código y su fichero de cabeceras correspondiente.

Práctica 2 A

- temporizador_hal y temporizador_drv

En el módulo hal se programan dos interrupciones de temporizador (timer0_ISR y timer1_ISR). Además, se añade una función en ambos módulos, HAL y DRV, que programa el reloj para permitir que se llame a la función callback periódicamente.

Por otro lado, en el driver se programa una "software interrupt", que se explicará más adelante. Y como se indica en el paso 4 de esta práctica, se añaden a estos módulos dos funciones: una dependiente del hardware en el módulo hal, que programa el reloj para que llame a la función callback, pasada por parámetro, con una frecuencia determinada y otra en el driver, que programa el reloj para que encole un evento periódicamente en el planificador.

- gpio_hal.h y io_reserva.h

En este módulo se desarrollará una pequeña biblioteca que nos abstraiga del hardware e interactúe con los GPIO. Se realiza en un fichero de cabeceras para que el compilador interprete el "inline", al igual que en celda.h. Para ello, se definirá el conjunto de direcciones E/S del GPIO y el tipo GPIO_HAL_PIN_T que es un entero de 32 bits. Por otro lado, se definen funciones para iniciar el GPIO, poner sus pines en modo input o output, leer el valor de un pin o escribir un valor en los pines.
Por otro lado, en io_reserva.h, se les asignará un nombre más humano a los pines que vayamos a utilizar, como por ejemplo, "#define GPIO_OVERFLOW 31", para asignarle al pin 31 un nombre representativo de desbordamiento.

- fifo.h y fifo.c

En estos módulos se implementará una estructura de una cola de eventos FIFO (First In First Out), para encolar los distintos eventos. Para su pleno funcionamiento, se implementa en primer lugar el tipo de datos EVENTO_T, y diversas funciones para poder inicializar la cola, encolar un evento, extraer un evento y mostrar las estadísticas de eventos encolados (número de eventos de cada tipo). Al ser una cola circular, tiene un tamaño limitado y puede que desaparezca un evento sin ser tratado por el planificador; cuando ocurra esta situación, se encenderá el pin correspondiente al overflow, definido en io_reserva.h.

- hello_world

Se trata de básicamente un programa de prueba, en el que se muestra una cuenta encendiendo 8 “leds” en el GPIO. En este módulo nos encontramos con una serie de funciones para inicializarlo, y otra para actualizar el contador (incrementarlo en 1) y cambiar el estado de los leds, según el valor de la cuenta. A esta última función se le llama desde el planificador cada 10 milisegundos.

- power

Se trata de un módulo que hace que el procesador no desperdicie energía. Cuando el procesador no tenga cálculos que hacer, en lugar de ejecutar un bucle innecesario, lo vamos a dormir para que reduzca su consumo (modo idle). Para ello, se implementa la función power_hal_wait(), en la que se activa el modo idle poniendo a 1 el bit de menos peso del registro PCON (Power Control Register).
Más adelante, en este mismo módulo, se añadirá otra función para desconectar más partes del procesador.

Práctica 2 B

- alarmas

En este módulo se implementa un gestor de alarmas, el cual nos permitirá programar alarmas para que se “disparen” o salten, pasado un tiempo estipulado. Para ello, se ha definido un struct Alarma que contiene los campos necesarios para conocer si la alarma es periódica o no, si está activa o no lo está, cada cuánto tiempo se debe disparar, etc. Existe una serie de funciones que nos permiten inicializar el array de alarmas, activar y desactivar una alarma, así como tratar un evento asociado a una alarma. Si no se pueden activar más alarmas porque el array está lleno, se encola un evento de overflow, que se indicará en el GPIO y se parará la ejecución.

- int_externas_hal y botones

El módulo dependiente de hardware, int_externas_hal, administra las interrupciones externas, inicializa las interrupciones de los botones, y gestiona su tratamiento. Implementa un mecanismo para detectar nuevas pulsaciones y limpiar pulsaciones anteriores.

En el módulo independiente botones.c, se encuentran cuatro funciones esenciales que inicializan y gestionan los botones, evitando que una pulsación genere múltiples interrupciones, mediante un mecanismo de alarma que verifica la continuidad de la pulsación. Se incorporan maquina_estados_EINT1 y maquina_estados_EINT2, que comprueban el estado del botón y si está pulsado, actualizan su estado cuando se libera. Ambas funciones son idénticas, salvo que una trata el botón 1 y la otra el botón 2.

Adicionalmente, botones.c utiliza una función callback, gestionar_pulsacion, para manejar eventos asociados a las pulsaciones, tras la interrupción del botón correspondiente. Esta función es llamada al detectar una pulsación, encargándose de encolar el evento correspondiente y activar alarmas según el botón pulsado.

- power

En esta sección, se incorpora a la funcionalidad del módulo "power" de la Práctica 2A una nueva función diseñada para reducir aún más el consumo del procesador, llevándolo al estado de "power-down". Para habilitar este modo, se introduce la constante "USUARIO_AUSENTE" en el planificador, que define el límite de tiempo durante el cual el procesador puede estar inactivo. Se configura una alarma que se activa una vez transcurrido ese período.
Entonces, cada vez que se detecta actividad por parte del usuario y la alarma aún no ha sido disparada (es decir, antes de que transcurra el periodo establecido por "USUARIO_AUSENTE"), se cancela y reprograma la alarma, reiniciando así el tiempo de procesador. Es relevante señalar que, en este modo, el procesador se despierta con cada interrupción que llega, gracias a la configuración del registro "EXTWAKE" del LPC2105.

- Demostrador

Se crea el módulo juego.c con dos visualizaciones: un contador, el cual empieza en cero y aumenta con cada pulsación del botón EINT1, se decrementa con cada EINT2, y una segunda variable llamada intervalo, que almacena el tiempo transcurrido entre las dos últimas pulsaciones. Con cada nueva pulsación, juego creará el evento ev_VISUALIZAR_CUENTA colocando la cuenta en el campo auxiliar. El módulo visualizar, que habrá sido inicializado por el planificador, mostrará tanto el contador como el intervalo, a través del GPIO desde el pin 23 hasta el 16, mediante las funciones del gpio_hal. Acerca del hello_world, en su inicialización se programa una alarma para generar el evento ev_LATIDO, cada 10 ms. El planificador, al gestionar este evento,

llamará a la nueva función hello_world_tratar_evento y la visualización se llevará a cabo mediante un evento adicional, ev_VISUALIZAR_HELLO.

Práctica 2 C

- Startup.s

Tras las últimas líneas del fichero Startup, donde se establece el SP (r13) de cada modo de ejecución, se ha añadido el siguiente código:

```
;  Enter User Mode and set its Stack Pointer
        MSR     CPSR_c, #Mode_USR
        MOV     SP, R0
        SUB     SL, SP, #USR_Stack_Size
```

Así se consigue que, antes de ejecutar el programa principal, llamada a main, la ejecución pase a modo usuario, restringiendo ciertas operaciones para que sean realizadas únicamente por el núcleo del SO y hacer el sistema más seguro.

- SWI_hal.s y System_calls_hal.h

El código ensamblador en SWI_hal.s proporciona la implementación real de las funciones que se definen en System_calls_hal.h. El código C llama a las funciones definidas en System_calls_hal.h para interactuar con las interrupciones del sistema, y estas llamadas se convertirán en llamadas al SWI que están implementadas en el fichero ensamblador SWI_hal.s.

- linea_serie_hal.c

Proporciona funciones de bajo nivel para la inicialización y manipulación del hardware de la línea serie. Incluye una interrupción que maneja la recepción y transmisión de datos.
Configura registros específicos del hardware para la comunicación serie.
Define una interrupción para manejar eventos relacionados con la línea serie.

Acerca de las interrupciones UART (Universal Asynchronous Receiver/Transmitter), se manejan interrupciones relacionadas con la recepción (RDA, Receive Data Available) y la transmisión (THRE, Transmitter Holding Register Empty) de datos a través del UART0 en la interrupción de línea serie, como funciones callback: linea_serie_ISR() que llama a RDA o a THRE, según el valor del registro U0IIR (Interrupt Identification Register).

La siguiente función de linea_serie_drv.c:

void linea_serie_drv_inicializar(void(*funcion_encolar_evento)(EVENTO_T, uint32_t)){
FIFO_encolar = funcion_encolar_evento;
linea_serie_hal_inicializar(&linea_serie_drv_maquina_estados,&linea_serie_drv_continuar_envio);

es la que llama a la función de linea_serie_hal.c:
void
linea_serie_hal_inicializar(void(*funcion_callback1)(),void(*funcion_callback2)()){

- linea_serie_drv.c

Implementa un controlador de línea serie para la comunicación. Utiliza una máquina de estados para gestionar la recepción de datos y crear comandos y proporciona funciones para inicializar el controlador, enviar arrays de caracteres y continuar con la transmisión.

Utiliza un buffer para almacenar datos a enviar y almacena un puntero a la función que encola eventos para su posterior llamada, ya que se evita incluir el fichero fifo.c en el fichero.

La máquina de estados detecta caracteres especiales ('$' y '!') para gestionar la recepción y procesamiento de datos en dos estados posibles: VACIO y LLENO. En el estado VACIO, se almacenan datos en el vector “carga” hasta alcanzar un máximo de 3 caracteres. Si se recibe un '!', se verifica la longitud total de “carga” y se encola un evento ev_comprobar, para que el planificador compruebe que la cadena recibida es correcta. En el estado LLENO, al recibir '$', se reinicia para recibir una nueva carga.

linea_serie_drv_continuar_envio utiliza linea_serie_hal_escribir(buffer.dato[buffer.indiceEnvio]) para enviar datos al UART. Este proceso se repite hasta que se hayan enviado todos los datos en el búfer.

Los registros U0THR (UART0 Transmitter Holding Register) y U0RBR (Universal Asynchronous Receiver Buffer), para la comunicación a través del UART (Universal Asynchronous Receiver/Transmitter), son parte del hardware de comunicación serie del microcontrolador LPC210X:

- **U0THR**: Este registro es utilizado por linea_serie_drv.c para escribir datos que se enviarán a través del UART0. En el código, se observa que se utiliza en la función linea_serie_hal_escribir(char c) para enviar caracteres.

- **U0RBR**: Este registro es utilizado por linea_serie_hal.c para leer datos que se reciben a través del UART0. En el código, se observa que se utiliza en la función linea_serie_hal_leer(void) para recibir caracteres.

- juego.c

Implementa la lógica principal del juego. Utiliza un tablero y funciones relacionadas para cargar y visualizar el estado del juego y maneja comandos (diferenciados según el auxData del evento) como $NEW!, $END!, $#-#! y $TAB!, provenientes de la línea serie y realiza acciones correspondientes a cada uno.

Con la función clock_get_us de System_calls_hal.h calcula y visualiza el tiempo transcurrido durante la ejecución del juego, es decir, muestra cuánto tarda el tablero en aparecer por pantalla.

Una condición de carrera es un error a evitar que se produce al acceder de manera concurrente a la cola de eventos compartida. El problema radica en que al leer un dato de la cola, mientras desde el planificador se llama a FIFO_encolar() o a otras rutinas de servicio o interrupción que tengan que ver con la cola, se impide realizar la acción correctamente ya que se solapan los procesos y no se llega al resultado esperado.
Por ello, se han desactivado las interrupciones en la función FIFO_extraer() mediante llamadas al sistema y al final de la función, se han restaurado los valores de bit de interrupción que tenía al inicio.

Además, antes de alimentar el watchdog, siguiendo el manual de la placa, se han deshabilitado todas las interrupciones y se han habilitado nuevamente después de la alimentación.

Práctica 3

En la máquina de estados del juego, para ahorrarnos el estado de *esperar el comando RX_SERIE* (que indica el fin de la recepción de un comando), hemos implementado una cola circular que actúa como buffer de mensajes. De esta manera, no se necesita esperar a que acabe un comando para mandar el siguiente, sino que se pueden enviar varios mensajes al mismo tiempo y se encolan uno seguido del otro en el buffer, siempre y cuando haya espacio suficiente. Este “buffer” se vacía cada vez que se escribe algo, para evitar que se llene provocando entonces un overflow.

Como otra consideración importante acerca de la nueva cola, ocurre una condición de carrera que se debe abordar. Se deben deshabilitar las interrupciones antes de encolar cualquier mensaje, ya que, en el caso de estar activas las interrupciones externas, la

concurrencia de procesos podría hacer que se estuviese encolando un mensaje y extrayendo otro, al mismo tiempo. Además, antes de salir de la función *buffer_encolar*(), se debe restaurar el bit de interrupción, al que había previamente, antes de iniciar la función. Este mismo problema se solucionó ya en la práctica 2, pero con la cola FIFO.

También hay que tener en cuenta la relación entre la línea serie con la nueva cola buffer, que ya no es un registro. En la función *linea_serie_drv_enviar_array*() del fichero *linea_serie_drv.c*, se presenta una mini máquina de estados para gestionar el envío de mensajes a la línea serie: si ya se está enviando un string, basta con encolar el nuevo envío ya que se enviará después del envío actual. Si no se está enviando nada (buffer vacío), se encola el nuevo mensaje y se crea un nuevo envío.

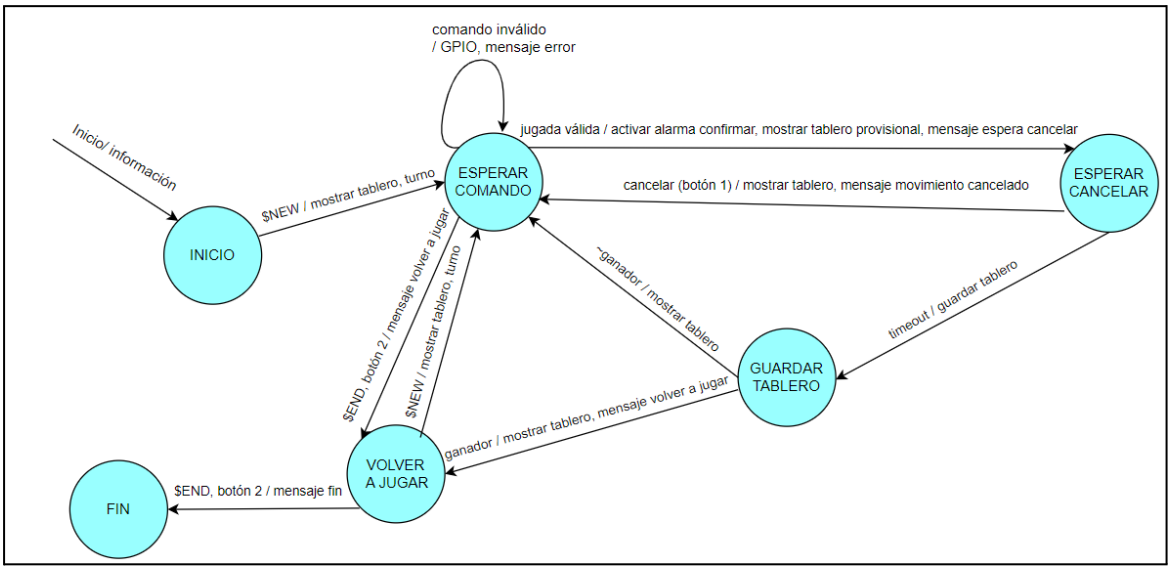
En la práctica 2, se utilizaban los botones 1 y 2 para gestionar la actividad del usuario. En esta práctica, cada botón tiene su funcionalidad propia: si el jugador pulsa el botón 1 antes de que pasen 3 segundos desde la última jugada, se cancela la jugada realizada y se muestra el tablero anterior. El botón 2 sirve para indicar que el jugador se rinde y se termina la partida actual.

Para hacer que vuelva a funcionar el Hello_world, se ha cambiado en el fichero *io_reserva.h* el pin GPIO_HELLO_WORLD que tenía anteriormente asignado, al pin GPIO016, puesto que el 14 y el 15 han sido ocupados por los botones 1 y 2.

Si el planificador pasa más de un segundo sin procesar ningún mensaje o evento, se reseteará el sistema. Este requisito lo conseguimos cumplir con el Watchdog, que entra en un while(1) si sucede.

Respecto a prácticas anteriores, se ha tenido que modificar toda clase que utilizaba la cola FIFO y los GPIO de io_reserva.h, de forma que ahora, en vez de incluir los ficheros fifo.h y io_reserva.h, la cola se pasa como parámetro actuando como una función callback y para utilizar un GPIO en concreto, se define directamente en el fichero como #define GPIO_OVERFLOW por ejemplo.

5. Máquina de estados



- **Inicio**: El juego comienza mostrando por línea serie al jugador las instrucciones a seguir y la información de bienvenida. Si llega el comando $NEW!, se muestra el tablero inicial (tablero_test) y se asigna el turno inicial al jugador 1. Se pasa al estado Esperar comando.

- **Esperar comando:** Se espera que el jugador escriba un comando por línea serie. Si el comando es inválido, se queda en el mismo estado, activando el GPIO029 y mostrando un mensaje de error (específico para cada tipo de error: *columna inválida* si al introducir una jugada #-#! se escriben dígitos menores que 1 o mayores que 7, o si esa celda ya está ocupada, y *comando incorrecto* si se escribe cualquier cosa que no sea un comando válido en el juego). Si, por el contrario, es una jugada válida (#-#! entre 1 y 7), se activa la alarma de confirmar la jugada durante 3 segundos y se muestra un mensaje informativo. Además, se muestra el tablero provisional (fichas b y n se muestran en minúsculas) con la nueva jugada y se pasa al estado Esperar cancelar.

  De otra forma, si el jugador escribe el comando $END! o pulsa el botón 2, el jugador se rinde y se pasa al estado Volver a jugar, mostrando un mensaje de pregunta como: ¿quieres volver a jugar?

- **Esperar cancelar**: Se espera 3 segundos a que el usuario pulse el botón 1 para cancelar la jugada. Si se acaba el tiempo de la alarma y el jugador no lo ha hecho, se pasa al estado de Guardar tablero y se confirma la jugada realizada guardando el tablero actual como definitivo. Si, por el contrario, ha pulsado el botón 1 antes de que salte la alarma, se pasa al estado de Esperar comando y se muestra el tablero anterior y el mensaje de: Movimiento cancelado.

- **Guardar tablero:** Se almacena el tablero actual definitivo (fichas b y n se muestran en mayúsculas) como confirmado. Si hay un ganador, se acaba la partida y se muestra el mensaje de pregunta al jugador para ver si quiere jugar de nuevo, se pasa al estado Volver a jugar. Si, en cambio, no hay ganador, se muestra el tablero recién guardado y se pasa al estado Esperar comando para seguir la partida.

- **Volver a jugar:** Se ha terminado la partida. El jugador elige entre crear una nueva partida, o rendirse y dejar de jugar. Si se pulsa el botón 2 o se escribe el comando $END!, se termina el juego, se muestra un mensaje de fin y se pasa al estado Fin. Si, por el contrario, se escribe el comando $NEW!, se empieza una nueva partida alternando el jugador inicial, teniendo en cuenta el de la partida anterior y se pasa al estado Esperar comando, mostrando el tablero vacío y alternando el turno del jugador que empezó la partida ahora terminada.

- **Fin:** Se ha acabado el juego.

Para la sección de estadísticas sobre el juego, se muestra por línea serie:
- ➔ Causa por la que se termina (botón de reiniciar, victoria de uno de los jugadores, ...)
- ➔ Tiempo total de uso de procesador en esta partida (sin power-down).
- ➔ Total y media de tiempo de computo de conecta_K_hay_linea.
- ➔ Total y media de tiempo que al humano le cuesta pensar jugada.
- ➔ Total de eventos encolados en esta partida e histograma por tipo de evento.

En resumen, se ha decidido que, si en el estado Esperar comando, se recibe un comando erróneo, se trata encendiendo un led GPIO y mostrando un mensaje de error, pero si sucede en el resto de estados, se ignora el comando erróneo y no se hace nada al respecto. El botón 1 sirve para cancelar la jugada realizada, si se pulsa en menos de 3 segundos después de haberla realizado y el botón 2, tiene la misma función que el comando $END!, que es terminar la partida. Una vez llega a su fin, se muestra un mensaje por pantalla que permite al usuario volver a jugar otra partida, o bien, abandonar el juego definitivamente, pulsando el botón 2 o el comando $END. Siempre aparece la causa de fin por línea serie y se muestran las estadísticas del tiempo que se ha tardado para realizar cada tarea durante la partida.

Con la opción de cancelar la jugada, se muestra por línea serie el tablero provisional, que presenta la ficha de la jugada provisional en minúscula, ya que la jugada en cuestión todavía no está confirmada. Una vez termina la alarma de 3 segundos sin pulsar el botón, se confirma la jugada y se muestra el tablero ya definitivo, con la nueva jugada y todas las fichas en mayúsculas.

6. Tiempo de dedicación

TIEMPO	Carlota Moncasi	Pablo Pina
Comprensión del enunciado	3 horas	3 horas
Máquina de estados	2 horas	1 hora
Nuevas funciones	1 hora	1 hora
Cambios en código de prácticas anteriores	5 horas	3 horas
Depuración	20 horas	13 horas
Análisis de rendimiento	4 horas	4 horas
Memoria	10 horas	8 horas
Total	45 horas	33 horas

La práctica 1 nos resultó más costosa, ya que era la primera vez que programábamos con periféricos en el entorno de Keil y todavía nos estábamos iniciando en la asignatura intentando entender, poco a poco, cómo funcionaba la placa LPC2105. Para realizar la práctica 2, tardamos mucho tiempo ya que se dividía en 3 partes (A,B y C) y comprendía toda la lógica y los aspectos técnicos importantes de todo el proyecto. Sin embargo, finalmente la práctica 3 fue la más rápida, aunque también la más frustrante ya que tuvimos que arreglar algún error de prácticas anteriores y comprobar que todo funcionase correctamente. Pero también disfrutamos de la práctica final, al ver los resultados esperados, mientras nos divertíamos jugando al Conecta K, durante la última fase de pruebas.

7. Conclusiones

Entre las dificultades encontradas, al hacer que se ejecutase con normalidad la práctica 3, encontramos que la funcion_callback_RDA y la funcion_callback_THRE guardaban valores ASCII, en lugar de una dirección de memoria, y por tanto, el programa resultaba en un error **Prefetch Abort** al llegar a la interrupción de linea_serie_ISR. Esto se debía a que usábamos la función strcat() para concatenar char*, pero esta requería añadir '\0' al final de cada cadena de caracteres, y no funcionaba bien a la hora de mostrar los mensajes seguidos. Nos dimos cuenta de que los mensajes se sobreescribían en memoria, unos sobre otros llegando a repetirse.

Así que, para evitar este problema, lo que hicimos fue llamar directamente a linea_serie_drv_enviar_array() y conseguimos que todos los mensajes se escribieran correctamente, sin que apareciese el error de Prefetch Abort durante su ejecución.

Por otra parte, en la penúltima corrección de laboratorio, se comprobó que el hello_world funcionaba a la perfección; sin embargo, en la última corrección, el movimiento dejó de visualizarse, al añadir toda la funcionalidad del programa. En principio, el código relevante al módulo es correcto y no debería surgir ningún problema.

Algo a resaltar también es que en entregas anteriores, no llegamos a comprobar que la función FIFO_estadisticas() funcionara correctamente, ya que en ese momento todavía no se utilizaba. Entonces, para esta práctica tuvimos que incluir el vector numVeces[NUM_EVENTOS] en el struct de la cola queue, con el fin de almacenar el número de eventos encolados por cada tipo.

Como extra, hemos añadido la funcionalidad de mostrar por la pantalla UART, lo que se introduce en ella mientras el usuario lo va escribiendo. De esta manera, se evitan confusiones y erratas al probar el programa. Lo que no tuvimos en cuenta, ya que no nos dio tiempo, fue evitar permitir la interrupción por teclado en el caso de estar mostrando el tablero o mensajes por la UART, puesto que se solaparían y se sobreescribirían ambos.

Por último, añadir que nos hemos esforzado mucho para sacar adelante este proyecto y estamos muy orgullosos del trabajo realizado. Hemos aprendido mucho sobre hardware pero también de software, al implementar bastante código en C. Por otra parte, la asignatura también nos ha hecho organizarnos para ir al día, con las entregas semanales. En resumen, hemos disfrutado mucho de la práctica, sobre todo porque se puede ver físicamente cómo funciona el juego e interactuar con él y ver que el resultado conseguido es el esperado es toda una recompensa.