

PROYECTO HARDWARE

Conecta K

Grupo Martes Mañana K



08/01/2024

GUILLERMO BAJO LABORDA, 842748@unizar.es



Universidad
Zaragoza



Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Índice

1. Resumen.....	4
2. Introducción.....	5
3. Objetivos.....	6
4. Metodología.....	7
4.1 Diseño y esquema del proyecto.....	7
5. Resultados.....	14
5.1. Presentación y análisis de los resultados.....	14
5.2. Problemas encontrados.....	16
6. Conclusiones.....	18
7. Bibliografía.....	19
8. Anexo I: Código del proyecto.....	20

1. Resumen

En el transcurso de las prácticas 2 y 3, se ha explorado el funcionamiento y la interacción con **periféricos** esenciales en la mayoría de los sistemas actuales, así como los **temporizadores** o las **entradas/salidas** de propósito general. Los temporizadores nos brindan la capacidad de generar retardos precisos, llevar a cabo tareas periódicas y medir tiempos con exactitud. Por otro lado, la **GPIO** nos permite emular dispositivos como botones o LEDs.

La gestión eficiente de estos elementos se logra mediante el uso de un **controlador de interrupciones** (VIC), que nos permite controlar el flujo del programa cuando ocurren eventos específicos. Además, exploramos la importancia de **optimizar el consumo de energía** del procesador mediante técnicas como la suspensión y reactivación del mismo, aprovechando periodos de inactividad.

En el desarrollo de esta práctica, se sigue un **enfoque modular**. Cada módulo se presenta con claridad a través de interfaces definidas en archivos de cabecera (.h), abstrayendo los detalles de bajo nivel y permitiendo una interacción sencilla desde el programa principal. La **depuración de interrupciones concurrentes**, la implementación de un **planificador** eficiente, capaz de gestionar distintos eventos del sistema y el diseño de una **cola de eventos FIFO** son aspectos clave abordados en este proyecto.

Las **líneas de interrupción externas** EINT1 y EINT2 se han configurado para poder ser utilizadas como botones con funcionalidades determinadas. En el juego, el botón 1 se emplea para cancelar jugadas mientras que el 2 para rendirse.

El enfoque específico en la jugada por **línea serie** nos proporciona una interfaz gráfica agradable, donde los comandos del usuario se reciben a través de UART0. Además, se implementa un sistema de **eliminación de condiciones de carrera** para asegurar la consistencia en la ejecución del programa. La introducción de **llamadas al sistema**, como `clock_get_us`, `read_IRQ_bit`, `enable_irq` y `disable_irq`, fortalece la robustez y seguridad del sistema al encapsular funciones críticas.

El sistema desarrollado puede operar tanto en **modo usuario** como en **supervisor**, en el cual se ejecutan las llamadas **SWI** implementadas para realizar operaciones de gestión del sistema.

Finalmente, la implementación de un **autómata** que incorpore el correcto comportamiento del juego **conecta K** asegura la adecuada ejecución y coordinación de los diversos componentes, garantizando el correcto funcionamiento del sistema solicitado en su totalidad. Este proyecto no solo amplía nuestra comprensión de los sistemas embebidos y su programación, sino que también destaca la importancia de la modularidad, eficiencia energética y gestión de eventos en el desarrollo de sistemas complejos y robustos.

2. Introducción

A lo largo de estas dos últimas prácticas de la asignatura de Proyecto Hardware, se han ido implementando diferentes **módulos** y **funcionalidades** para nuestro sistema, para finalmente, diseñar el modelo final del juego **conecta K** haciendo uso de estas funciones.

Conecta K es un juego estratégico donde dos jugadores compiten para alinear fichas en un **tablero**. El tablero tiene columnas y filas, y el objetivo es conectar un número específico de fichas en línea recta antes que el oponente. El programa trabaja con varios tipos de **periféricos**, como temporizadores, así como con interrupciones externas configuradas en los pines de entrada del sistema, las cuales serán empleadas para simular el funcionamiento de **botones** para el juego. El sistema también cuenta con un planificador de eventos que gestionará la respuesta a los eventos que se generen en el sistema.

El juego inicia mostrando las instrucciones del juego, esperando un **comando** de inicio de partida. La interacción se realiza por línea serie (UART0), que servirá como pantalla para el usuario. Después de un comando válido, se muestra el tablero y se espera la entrada de fila y columna. Errores resultan en mensajes y encendido de LEDs mediante el GPIO. Se puede cancelar un movimiento en curso pulsando el botón EINT1. El botón EINT2 en cambio sirve para rendirse e iniciar una nueva partida. Al concluir la partida por victoria o rendición, se presenta la causa de terminación de la **partida**, así como diversas estadísticas tanto de la partida como del sistema.

3. Objetivos

El desarrollo de las prácticas 2 y 3 se han orientado hacia la consecución de diversos **objetivos**, tales como los siguientes:

- Gestionar la **entrada/salida** con dispositivos básicos, asignando valores a los registros internos de la placa desde un programa en C.
- Desarrollar en C **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**.
- 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 un **protocolo de comunicación robusto para el puerto serie**.
- Identificar **condiciones de carrera**, su causa y solucionarlas.
- Realizar un **juego completo** haciendo uso de la estructura modular.

4. Metodología

4.1 Diseño y esquema del proyecto

Por simplicidad, se ha añadido al **anexo** de este documento el **código fuente**, para así poder hacer referencias al mismo de forma más cómoda y sencilla. El código fuente del anexo está comentado para aumentar su legibilidad.

Para la gestión de los diversos eventos que pueden producir el uso de los periféricos y demás componentes del sistema se ha creado una **cola de eventos FIFO**, en la que se van encolando eventos, cada uno con un dato auxiliar. Por otra parte, se ha diseñado el **planificador de eventos** para dar respuesta a estos eventos, desencolándolos y realizando las acciones necesarias para cada uno. Nuestro sistema cuenta con un total de 15 eventos, tales como *ev_TX_Serie*, *ev_Latido*, *Pulsacion_Boton*, *ev_Visualizar_Hello*, etc... Todos ellos están definidos en *evento.h* el cual forma parte del anexo.

Respecto al planificador, su funcionamiento es el siguiente: inicialmente, inicia diferentes módulos tales como el temporizador, la cola FIFO, las alarmas, los botones, etc... Inicia el autómata del juego antes de entrar en un bucle infinito en el cual repetirá este comportamiento: **esperará** a que haya **eventos** de la cola para **extraer**. Una vez haya un evento, lo extraerá y alimentará al watchdog. Para **tratar** dicho **evento**, se comprueba cual es su *IDevento* y en función de ello se realizan diversas instrucciones. Si no hay ningún evento para extraer, el procesador entra en **modo Idle** a la espera de que haya un nuevo evento disponible para su extracción.

En el diseño de este sistema, se sigue un **enfoque modular**. Cada módulo se presenta con claridad a través de interfaces definidas en archivos de cabecera (.h), abstrayendo los detalles de bajo nivel y permitiendo una interacción sencilla desde el programa principal.. De esta forma la lógica del juego y la lógica que controla los periféricos queda completamente separada, así como la interacción entre diferentes periféricos.

Este enfoque modular permite que el sistema permanezca independiente de los detalles de implementación de cada módulo. Los módulos no necesitan conocer los detalles específicos sobre el sistema, como su naturaleza basada en eventos; únicamente deben activar la función que se les ha pasado durante la inicialización cuando se dé una condición específica. En la fase de inicialización de los módulos, es posible pasar funciones como *fifo_encolar* junto con un evento correspondiente para la acción de encolar. Esta **flexibilidad** permite a los módulos adaptar su funcionalidad de acuerdo con eventos específicos, sin necesidad de que el sistema principal esté atado a detalles de implementación de cada módulo. Esta separación contribuye significativamente al **mantenimiento y la escalabilidad**

del sistema al evitar que el sistema principal esté estrechamente ligado a la lógica interna de los módulos.

A continuación, comentaremos distintos módulos más en detalle. Cabe recalcar que el código fuente de todos estos módulos forma parte del [anexo](#).

Temporizadores:

Se han implementado dos temporizadores de **propósito general** (timer 0 y 1) y un temporizador de **propósito específico** (WatchDog). Este último tiene como función resetear el procesador en caso de mal funcionamiento, mientras que los de propósito general se encargan de medir la distancia temporal entre dos acciones. Para el diseño, se han separado las funciones dependientes del hardware (**HAL** -Hardware Abstraction Layer) y las del gestor del dispositivo (**DRV** - DRiVer).

El **timer0** mide tiempos con la máxima precisión posible, pero interrumpiendo lo menos posible para evitar sobrecargar al programa principal. El **rango** del contador está determinado por el valor máximo que puede alcanzar antes de que se produzca una coincidencia. Se ha configurado de 0 a 0xFFFFFFFFE (32 bits - 1). Respecto a la **precisión**, viene determinada por la frecuencia del reloj y la resolución del contador. Se ha configurado el *Prescaler* (TOPR) a 0 para que cada ciclo de reloj el contador incremente en 1.

El **driver** es un módulo independiente del hardware nos presenta un dispositivo temporizador donde permite contar el tiempo entre dos intervalos en microsegundos. Hace uso de los servicios del módulo de abstracción del hardware. En caso de un cambio en el hardware subyacente, la única variable que probablemente necesitaría ser ajustada sería el parámetro TICKS_TO_US. Este parámetro representa la conversión de ticks a microsegundos, y su adaptación se requeriría para reflejar las diferencias en el nuevo hardware, asegurando así la correcta medición del tiempo sin afectar el resto de la lógica y funcionalidad del driver.

Respecto al **WatchDog**, se ha utilizado para salir de bucles infinitos que pudieran generarse a partir de algún tipo de malfuncionamiento. En nuestro proyecto, el WatchDog se inicia con un tiempo de 1 segundo para que si pasa más de ese tiempo sin que el planificador procese ningún mensaje, el sistema se resetee. Dado que se generan interrupciones periódicas, esto solo ocurrirá si el sistema se queda colgado. Cada vez que el planificador procesa un evento, el WatchDog se “**alimenta**”, reseteando así la cuenta atrás de ese segundo.

Cola de eventos:

Nuestro sistema funciona bajo el **paradigma orientado a eventos**. Por ello, se ha implementado una **cola FIFO** para gestionar múltiples eventos, y el planificador estará a la espera de la cola para procesar nuevos eventos.

El módulo de la cola FIFO de eventos cuenta con funciones como la inicialización de la cola, el encolado de eventos, la extracción del evento más antiguo, y la obtención de estadísticas.

Se manejan posibles **desbordamientos** de la cola, indicando un error con la activación del LED 31 usando el GPIO y deteniendo la ejecución en un bucle infinito. Por otra parte, la cola sirve de gran utilidad a la hora de depurar, almacenando información de los últimos 32 eventos generados en el sistema.

Gestión de la entrada/salida:

Los dos principales periféricos de entrada/salida diseñados en esta práctica son la **GPIO** y la **línea serie**. La GPIO es un periférico que se puede utilizar para conectar al chip elementos de entrada/salida genéricos, como por ejemplo LEDs. En el LPC2105 hay disponible un puerto de 32 bits utilizables como entrada o salida. En el juego diseñado se enciende el pin 29 cuando el jugador con el turno ha introducido un comando erróneo o el pin 32 cuando ocurre un desbordamiento en la cola de eventos.

Por otra parte, se ha configurado la línea serie (**UART0**) con el fin de ofrecer a los jugadores una **interfaz gráfica de entrada/salida agradable**. Los comandos se reciben por ahí, y así mismo el tablero y los resultados de salida son mostrados a los usuarios mediante línea serie.

Al igual que ocurre con los temporizadores de propósito general, la configuración de línea serie se rige por una clara separación de funciones entre la capa HAL (Hardware Abstraction Layer) y el gestor del dispositivo DRV (Device Driver).

La **capa HAL** inicializa la UART0 con parámetros específicos en los registros. Dada su naturaleza, limitada a la lectura y escritura de caracteres de uno en uno, una vez se recibe un carácter se transmite el contenido al DRV. Este se encarga de gestionar otras tareas asociadas con la manipulación y procesamiento de los datos de la línea serie. De la misma forma, cuando se al realizar envíos el driver comunica los caracteres de uno en uno al HAL. Es el driver el encargado de manejar los buffers de envío, finalizando la ejecución en el momento oportuno y limitando el tamaño de los envíos.

Por otro lado, la **capa DRV** ofrece operaciones de envío y recepción de arrays de caracteres al sistema, y controla y gestiona las limitaciones del hardware que restringen las operaciones a nivel de caracteres individuales. Además, para evitar bloqueos en el sistema y permitir la concurrencia en el envío de caracteres a través de la línea serie, el DRV emplea eventos, optimizando así la eficiencia y

permitiendo una gestión más dinámica de la transmisión de datos. Para la lectura, el DRV implementa una **máquina de estados** que le proporciona la capacidad de gestionar la recepción de contenido y la validación de los comandos escritos por el usuario, lo que contribuye a una interacción más segura y controlada con la interfaz UART0. Detectamos el inicio de una entrada cuando se introduce el carácter “\$” en la terminal, y hacemos uso de un buffer de tres caracteres para almacenar el contenido que el usuario escriba hasta que se introduzca el carácter “!” o se llene el buffer. Si cuando recibamos el carácter de fin el buffer tiene el tamaño esperado, se llamará a la función especificada en la inicialización del módulo con los datos recibidos. En caso contrario, el autómata pasa a estado ERR y enciende el pin correspondiente de la GPIO. Se mantiene en ERR hasta que recibe de nuevo “\$”.

Alarmas:

Las alarmas son un componente clave del sistema, el módulo gestor de alarmas simplifica significativamente la comunicación entre distintos módulos. Las alarmas permiten **la planificación de tareas una vez finalizan**. Este módulo es capaz de gestionar múltiples alarmas con distintas temporizaciones, como el timer1, configurado para interrumpir cada 10 segundos. El módulo de alarmas se ha configurado para que haya un máximo de 4 alarmas activas al mismo tiempo. Las alarmas pueden ser **periódicas** y disponen de un dato auxiliar que el planificador leerá y gestionará.

La correcta implementación del timer0 es clave para asegurar el correcto funcionamiento del módulo de alarmas, ya que al activar una nueva alarma se le asigna un **timestamp** haciendo uso de la función *temporizador_drv_leer()*, que posteriormente será empleado para comprobar si la alarma ha expirado.

Botones e interrupciones externas:

Nuestro SoC dispone dos **líneas de interrupción externa** que permiten interaccionar al procesador con dispositivos de entrada/salida que están fuera del chip. Se han utilizado **EINT1** y **EINT2** para simular la acción de botones. Una vez más, se han separado las funciones HAL (*int_externas_hal*) de las DRV (*botones*). Para usar estos botones, se han configurado los pines del sistema debido a que hay muchas más posibles conexiones que pines.

Estos botones pueden estar mucho tiempo pulsados, y las interrupciones son muy rápidas, es decir, una pulsación genera multitud de interrupciones. Por ello, para garantizar que solamente se genera una interrupción por pulsación, esta se procesa y a continuación se deshabilitan las interrupciones correspondientes a ese botón en el VIC. Acto seguido, se programa una alarma periódica de 100ms para monitorizar la pulsación y comprobar si el botón sigue pulsado. Si al expirar la alarma este sigue pulsado, se vuelve a encolar, sino, se cancela la alarma y se vuelven a habilitar las interrupciones para ese botón.

En el juego, estos botones tienen distintas funcionalidades. El botón 2 sirve como botón de rendición del jugador que tiene el turno, para a continuación empezar una nueva, mientras que el botón 1 sirve para cancelar una jugada.

Modos de consumo del sistema:

Reducir el consumo de energía y desarrollar estrategias de consumo responsable son dos puntos clave que cada vez se tienen más en cuenta en los nuevos sistemas. Por ello, en nuestro sistema se ha configurado una funcionalidad para que cuando el procesador no tenga cálculos que hacer, en lugar de ejecutar un bucle innecesario, entre en **modo Idle** y quede “dormido”. Para esto, se activa el primer bit del registro PCON (Power Control) cuando se requiera. A su vez, se ha configurado el EXTWAKE para que al recibir la pulsación de uno de los dos botones salga de este modo de ahorro de consumo.

Además, se ha configurado una alarma con un nuevo evento *USUARIO_AUSENTE*, cuya funcionalidad es pasar a modo **power-down** en caso de que pasen 12 segundos, en este caso, sin actividad del usuario.

Llamadas al sistema:

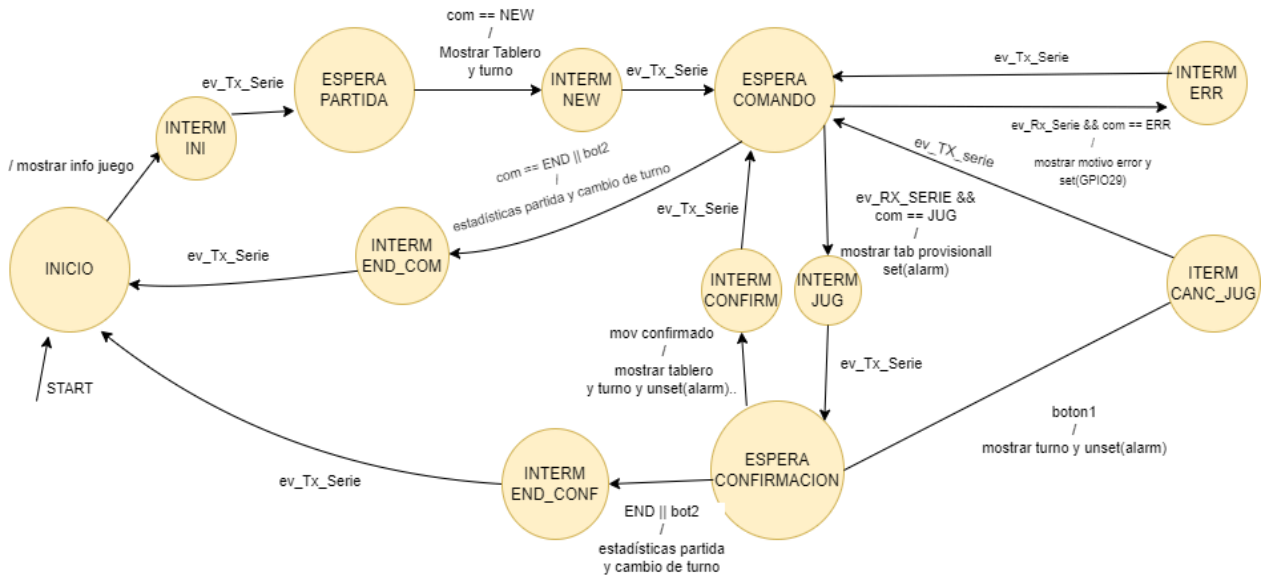
Por otra parte, se han implementado diversas **llamadas al sistema**, con el fin de poder realizar acciones como activar o desactivar interrupciones **IRQ** y **FIQ**. Para ello, se han implementado en ARM las funciones *read_IRQ_bit*, *enable_IRQ*, *disable_IRQ* y *disable_FIQ*. También se ha implementado la llamada al sistema *clock_get_us* que puede ser usado como un **reloj del sistema**.

Las llamadas al sistema para desactivar y activar interrupciones son clave a la hora de **evitar situaciones de carrera**, las cuales ocurren cuando se accede de manera concurrente a un recurso compartido, como puede ser la cola de eventos FIFO. Para prevenir estas condiciones de carrera, se han utilizado las llamadas al sistema para desactivar y reactivar interrupciones para asegurar la **actualización atómica de los índices de la cola de eventos** al leerla.

La desactivación y reactivación de excepciones también son usadas en otros contextos, como por ejemplo, antes de alimentar al WatchDog, ya que este requiere no ser interrumpido al alimentarlo.

Máquina de estados del juego:

Tras haber implementado y configurado todos estos componentes, se ha procedido a diseñar la **máquina de estados** que representa el comportamiento del juego final:



Como se aprecia el autómata, en el juego se ha hecho uso de bastantes funcionalidades de la mayoría de **módulos implementados**, tales como la pulsación de botones, el uso de línea serie para mostrar información y recibir comandos, la activación de alarmas, el uso de los LEDs del GPIO, etc...

El juego pasa por distintos estados según la situación en la que se encuentre en cada momento de la ejecución, y según los eventos que vayan sucediendo, se toman unas **transiciones** u otras. Para ello, cada vez que sucede un evento que pueda afectar al juego, el planificador llama a la función **juego_tratar_evento**, que primero actualiza las variables globales del juego en función del evento sucedido (cambia el valor booleano de botón pulsado o comando recibido, almacena en una variable el valor del comando introducido, etc...), para posteriormente **delegar el control al autómata**, que con el suceso de este nuevo evento procederá a tomar las medidas correspondientes y a avanzar de estado. De esta forma, esta máquina de estados convierte el **juego** en **autónomo**, todas las partidas siguen un camino preestablecido, solamente dependen de los dos jugadores y de las **jugadas** que realicen.

Cabe destacar que para el **tablero** y la gestión del mismo, se han utilizado diversas funciones diseñadas en la práctica I, modificando aspectos puntuales y adaptándolas a las nuevas necesidades. La versión de hay_linea utilizada ha sido la **ARM_ARM**, dado que fue la que mayor rendimiento proporcionaba de entre las diseñadas en la primera práctica.

4.2 Horas de trabajo dedicadas

	Miembros del grupo	
Tarea	Guillermo Bajo	Álvaro de Francisco
Comprensión del código fuente	2h	2h
Temporizadores	3h	4h
Entrada / salida básica GPIO	4h	4h
Cola de eventos	3h	3h
Contador - Hello World	30min	1h
Gestor alarmas	3h	4h
Interrupciones externas	4h	4h
Reducción del consumo	1h	1h
Implementación de llamadas al sistema	3h	1h
WatchDog	1h 30min	30min
Eliminación de condiciones de carrera	30min	45min
Línea serie	1h	5h
Diseño set de pruebas y demostradores	5h	5h
Diseño e implementación autómeta	2h	1h
Implementación de funciones auxiliares	1h 30min	2h
Depuración del código	7h	7h
Realización de la memoria	6h	4h
Total	48h	49h

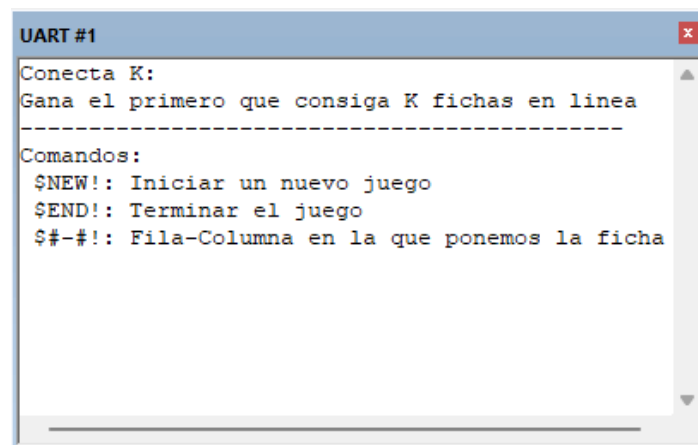
5. Resultados

5.1. Presentación y análisis de los resultados

El resultado final es un sistema que ofrece diversas interfaces de entrada/salida, como botones y una terminal para lectura y escritura, junto con la GPIO, que proporciona una interfaz de salida más versátil. Además, se presentan dos métodos adicionales para manejar acciones relacionadas con el tiempo: el temporizador del sistema y las alarmas. También se implementa el uso del watchdog para controlar bloqueos en la ejecución, el control de interrupciones IRQ para evitar condiciones de carrera, así como modos de bajo consumo que el planificador utiliza mediante la detección de alarmas y la identificación de periodos sin eventos para poner el procesador en modo idle y deep sleep.

Sobre esta base hemos añadido un juego, que podría ser sustituido por cualquier otro sin necesidad de adaptar otro código que el del propio juego. Principalmente hace uso de la línea serie para mostrar el estado del juego y recibir comandos, no obstante podemos hacer uso de la GPIO para recibir instrucciones, ver si el sistema ha detectado algún error o comprobar si el sistema ha entrado en modo *deep sleep*.

El juego comienza mostrando las posibles instrucciones que acepta el sistema y la sintaxis de las mismas. Una vez mostrado el mensaje pasa al siguiente estado, en el que espera una acción del usuario. El comando *NEW* inicia una nueva partida y mostrará un tablero vacío, dando comienzo al juego. Es en el contexto del juego donde se acepta el comando Fila-Columna, que utilizan los usuarios para colocar la ficha donde estimen oportuno. Por otro lado, disponemos del comando *END*, que nos permite finalizar la partida. El jugador que introduzca el comando se dará por perdedor



```
UART #1
Conecta K:
Gana el primero que consiga K fichas en linea
-----
Comandos:
$NEW!: Iniciar un nuevo juego
$END!: Terminar el juego
$#-#!: Fila-Columna en la que ponemos la ficha
```

Mensaje inicial del juego

Una vez se introduce el comando *NEW* en la terminal, se muestra el tablero de juego y se espera a que el jugador especifique la posición de la ficha haciendo uso del comando “\$#-#!” en el que el usuario indica la fila y columna en la que quiere poner la ficha. Inmediatamente después de introducir el

comando, se muestra donde se ha colocado la ficha utilizando un carácter especial. En este momento damos un tiempo al jugador para rectificar el movimiento pulsando el botón 1. En el caso de que lo pulse se mostrará un mensaje en el que se indica que puede volver a introducir la posición donde quiere poner la ficha repitiendo el escenario anterior. Cuando vence el temporizador, se coloca definitivamente la ficha en el tablero y el turno pasa al otro jugador. Este ciclo se repite mientras no haya un ganador y ningún jugador se rinda.

```
UART #1
-|1|2|3|4|5|6|7|
1| | | | | | |
2| | | | | | |
3| | | | | | |
4| | | | | | |
5| | | | | | |
6| | | | | | |
7| | | | | | |
Turno de jugador 2...
```

Espera a un comando

```
UART #1
-|1|2|3|4|5|6|7|
1| | | | | | |
2| | | | | | |
3| | | | | | |
4| | | |#| | | |
5| | | | | | |
6| | | | | | |
7| | | | | | |
Pulsa boton 1 para cancelar...
```

Confirmación del movimiento

```
UART #1
1| | | | | | |
2| | | | | | |
3| | | | | | |
4| | | |#| | | |
5| | | | | | |
6| | | | | | |
7| | | | | | |
Pulsa boton 1 para cancelar...
Jugada cancelada. Inserte una nueva jugada...
```

Cancelación de la jugada

```
UART #1
-|1|2|3|4|5|6|7|
1| | | | | | |
2| | | | | | |
3| | | | | | |
4| | | |N| | | |
5| | | | | | |
6| | | | | | |
7| | | | | | |
Turno de jugador 1...
```

Ficha colocada

Una vez decidido el resultado de la partida, bien sea por ganador o por rendición, se muestra información acerca de la partida. Se comunica el ganador, el tiempo total y medio que se tarda en pensar una jugada. También se muestran estadísticas propias del sistema, como el tiempo total y medio de procesamiento de la función *conecta_K_hay_linea* realizada en la práctica 1, tiempo total de procesador y el número de veces que un evento ha sido encolado junto con la suma de todos. A continuación se muestra el mensaje inicial del juego y se espera a que el usuario introduzca de nuevo el comando *NEW*.

Como muestra del rendimiento de nuestro sistema, incluimos la traza obtenida al ganar la partida tras poner una ficha negra en la casilla (4-4).

Tiempo de uso del procesador:

Total: 7 s

Tiempo de computo de conecta_K_hay_linea:

Total: 34 us

Media: 17 us

Tiempo que al humano le cuesta pensar la jugada:

Total: 2090 ms

Media: 2090 ms

Eventos esta partida:

TOTAL EVENTOS: 9296

ALARMA_TRATAR: 7400

ev_LATIDO: 675

ev_VISUALIZAR_HELLO: 675

ev_TX_SERIE: 3

ev_RX_SERIE: 2

CONTINUA_ENVIO: 542

5.2. Problemas encontrados

En el desarrollo de las prácticas nos hemos enfrentado a distintos retos y obstáculos, tanto en la codificación de módulos como del sistema en conjunto.

Bug en alarmas:

El aumento del número de eventos causó un bug inesperado en el gestor de alarmas. No cambiamos la variable que indica el nº de tipos de eventos distintos que teníamos, lo que provocaba que se reservara menos memoria que la necesaria en el módulo *FIFO*. Esto afectaba a la estructura en la que guardábamos las alarmas, aumentando en una unidad el identificador de la primera alarma siempre que se trataba el evento de comprobación de alarmas. La dispersión entre la causa y los efectos del bug nos dificultó encontrar el problema.

Envío y construcción de arrays de caracteres:

La abstracción de línea serie que construimos trabaja con arrays de caracteres, y durante el desarrollo del módulo encontramos que, en ocasiones, no enviábamos los caracteres que debían (e.g.: envío de *NEN* en lugar de *NEW*). Para solucionar este problema, trabajamos en la forma en la que enviábamos los buffers. En lugar de utilizar buffers distintos, utilizamos un único buffer, junto con las funciones

strep y strcat para mostrar información en la terminal. Además, antes de cada uso limpiamos el buffer para evitar interferencias, lo que solucionó el problema planteado en el ejemplo.

Paso de los comandos al juego:

Cuando la línea serie recibe un comando completo, encola el evento de recepción junto con el contenido recibido. Sin embargo, no se permite encolar un buffer de tres caracteres, sino un entero sin signo. En el proceso de conversión y desconversión encontramos problemas, ya que comandos como “#-#” incluyen tanto números como caracteres especiales. Para otros comandos, como NEW o END, simplemente se requería convertirlos a valores enteros y luego en el juego, comparar estos valores enteros con el código asociado a cada comando para distinguirlos. En el comando para ingresar la jugada, el input del jugador puede variar dentro de límites específicos, lo que demanda una operación reversible. El uso del carácter “-” obstaculizaba la utilización adecuada de las funciones itoa y atoi, ya que no estaban preparadas para manejar este carácter especial. Por lo tanto, optamos por ignorar este carácter especial al enviar números.

Incorporación de las SWIs:

La integración de las SWIs representó un desafío, ya que no teníamos conocimiento sobre el funcionamiento del archivo *Startup.s*, el cual inicializa el sistema. Como resultado, nos encontramos inicialmente con dificultades para distinguir las SWIs escritas en ARM y las escritas en C, lo que requirió un tratamiento diferente en el handler.

Además, en una fase inicial, nos enfrentamos a problemas para inicializar el código .s de las interrupciones IRQ, lo que impedía su ejecución. Sin embargo, una vez que comprendimos el funcionamiento del archivo *Startup.s* y lo incluimos en el proceso, pudimos avanzar exitosamente.

Incorporación de caracter especial en el juego y buffers:

El juego Conecta K emplea la estructura TABLERO para almacenar detalles acerca de las fichas ubicadas en cada casilla y el jugador al que pertenecen. La introducción de un carácter especial '#' para permitir un cambio de jugador implicó la modificación de algunas funciones de la práctica 1.

Sin embargo, estos cambios provocaron errores en nuestro código. En particular, surgieron problemas al visualizar el contenido del tablero y al explorar la estructura de datos. La necesidad de ajustar estas funciones nos llevó a realizar numerosas pruebas para asegurar su corrección.

Finalmente, la solución a este problema surgió al modificar la forma en que manejamos los buffers para mostrar mensajes. En resumen, la complejidad añadida a estas secciones de código generó una fase intensa de verificación y resolución de problemas.

6. Conclusiones

Durante el desarrollo de las prácticas 2 y 3, hemos adquirido un conocimiento profundo sobre el funcionamiento interno y el comportamiento del procesador LPC2105, comprendiendo en detalle la lógica subyacente y sus aplicaciones prácticas.

A pesar de no comprender completamente las implicaciones del desarrollo modular durante la práctica 2, al llevar a cabo la implementación del juego en la práctica 3, hemos alcanzado una comprensión más profunda de su importancia. La experiencia nos ha demostrado la facilidad para implementar un juego que originalmente no estaba pensado para este sistema, gracias al uso de herramientas modulares que nos han convencido de su eficacia. Estas características modulares han sido fundamentales para facilitar no solo la implementación del juego, sino también para vislumbrar cómo estas prácticas contribuyen a futuras expansiones del sistema sin comprometer su estabilidad.

La flexibilidad y adaptabilidad que ofrecen estos módulos nos permitieron integrar el juego de manera más eficiente, a pesar de su diseño inicial no alineado con el sistema en cuestión. Esta situación nos hizo apreciar aún más la importancia de la modularidad en el diseño de sistemas, ya que pudimos realizar ajustes y expansiones sin perturbar el funcionamiento principal del sistema, lo cual resulta fundamental para su escalabilidad y evolución futura.

Además nos hemos enfrentado a documentación altamente técnica, compleja y redactada en otro idioma. Explorar y comprender esta documentación no solo era un requisito, sino que se convirtió en una habilidad valiosa que reconocimos como fundamental para nuestro futuro profesional, y que ya hemos empezado a aplicar en otras asignaturas.

Por otra parte, el trabajo en pareja resultó altamente satisfactorio. La buena división del trabajo y el entendimiento entre ambas partes fueron fundamentales para alcanzar nuestros objetivos. Estamos satisfechos con el código obtenido durante las prácticas 2 y 3, aunque somos conscientes de que existe margen de mejora.

Es importante mencionar que este cuatrimestre ha implicado una carga de trabajo considerable. A pesar de ello, hemos podido gestionar eficientemente las tareas asignadas, lo que ha contribuido significativamente al desarrollo exitoso del proyecto. Además, queremos expresar nuestro agradecimiento a los profesores por proponer un proyecto muy interesante que nos ha permitido profundizar en el conocimiento adquirido en asignaturas anteriores como IC y AOC y AOC2.

7. Bibliografía

Manuales del microprocesador LPC2105 - [Manuales del microprocesador LPC2105](#)

Manuales y transparencias de ARM, C - [Manuales de Referencia ARM: Ensamblador, C, Estándares](#)

8. Anexo I: Código del proyecto

Para no ocupar más espacio del necesario, hemos juntado los .h con los .c. Las funciones que tienen una cabecera estilo Doxygen son las que se ofrecen en el .h para otros módulos

[timer0_hal:](#)

```
#include <LPC210X.H>
#include "timer0_hal.h"

static volatile unsigned int timer0_count = 0;

void timer0_ISR (void) __irq{    // Generate Interrupt
    timer0_count++;
    T0IR = 1;                    // clareamos la interrupción del timer0
    VICVectAddr = 0;            // reconocemos la interrupción
}

void timer1_ISR (void) __irq{
    if (timer1_routine != NULL) timer1_routine();
    T1IR = 1;
    VICVectAddr = 0;
}

/**
 * @brief Prueba el correcto funcionamiento del temporizador.
 *
 * Esta función realiza una prueba para verificar el correcto funcionamiento del temporizador.
 * Inicia el temporizador, lo pone en marcha, lee el valor del temporizador y lo compara con
 * un valor esperado.
 * Finalmente, detiene el temporizador y verifica que el valor leído sea igual al valor
 * esperado.
 *
 * @return 0 si la prueba es exitosa, 1 en caso contrario.
 */
int temporizador_hal_test(void) {
    unsigned i;
    temporizador_hal_iniciar();
    temporizador_hal_empezar();
    i = temporizador_hal_leer();
    while ((i + 50000) > temporizador_hal_leer());
    i = temporizador_hal_parar();
    if (i != temporizador_hal_leer()) return 1; //comprobamos que pare realmente
    return 0;
}
```

```

/**
 * @brief Inicializa el temporizador 0.
 *
 * Esta función inicializa el temporizador 0, reiniciando el contador y configurando los
 * registros necesarios.
 */
void temporizador_hal_iniciar(void) {
    timer0_count = 0;
    T0PR = 0;
    T0MR0 = 0xFFFFFFE;          // valor máximo para que interrumpa lo minimo
    T0MCR = 3;                  // interrupción en la coincidencia
    VICVectCntl0 = 0x20 | 4;    // 4=timer0
    VICVectAddr0 = (unsigned long)timer0_ISR;          // funcion irq
}

/**
 * @brief Pone en marcha el temporizador 0.
 *
 * Esta función pone en marcha el temporizador 0, habilitando su contador y activando la
 * interrupción periódica.
 */
void temporizador_hal_empezar(void){
    T0TCR = 1;                  // habilitamos timer0
    timer0_count = 0;           // inicializamos cuenta
    VICIntEnable = VICIntEnable | 0x10;          // bit 4 a 1
}

/**
 * @brief Lee el valor actual del temporizador 0.
 *
 * Esta función lee el valor actual del temporizador 0 y lo devuelve en ticks.
 *
 * @return El valor actual del temporizador 0 en ticks.
 */
uint64_t temporizador_hal_leer(void){
    return timer0_count * (T0MR0 + 1) + T0TC;    // valor del reg+lo anterior
}

/**
 * @brief Detiene el temporizador 0 y devuelve el tiempo transcurrido.
 *
 * Esta función detiene el temporizador 0 y devuelve el tiempo transcurrido desde la última
 * vez que se inició el temporizador.
 *
 * @return El tiempo transcurrido en ticks desde la última vez que se inició el temporizador.
 */

```

```

*/
uint64_t temporizador_hal_parar(void){
    T0TCR = 0;    // deshabilitamos timer0
    return temporizador_hal_leer();    // devolvemos cuenta timer
}

/**
 * @brief Configura el reloj del temporizador 1 para llamar a una función de callback
 * periódicamente.
 *
 * Esta función configura el reloj del temporizador 1 para que llame a la función de callback
 * especificada cada periodo.
 * El periodo se indica en milisegundos. Si el periodo es cero, se detiene el temporizador.
 *
 * @param periodo El periodo en milisegundos.
 * @param funcion_callback La función de callback a llamar periódicamente.
 */
void temporizador_hal_reloj (uint32_t periodo, void (*funcion_callback)()) {
    if (periodo == 0)    T1TCR = 0;    // deshabilita el timer1
    else {                // repetimos el proceso de temporizador_hal_iniciar
        T1MR0 = (periodo/temporizador_hal_tick2us) * 1000 - 1;
        T1MCR = 3;
        T1TCR = 1;
        timer1_routine = funcion_callback;
        VICVectCntl5 = 0x20 | 5;
        VICVectAddr5 = (unsigned long) timer1_ISR;
        VICIntEnable = VICIntEnable | (1 << 5);
    }
}

```

timer0_driver:

```

#include "timer0_hal.h"

static void (*encolar_evento)(EVENT0_T, uint32_t);
static EVENT0_T evento;

/**
 * @brief Inicializa el temporizador.
 *
 * Esta función inicializa el hardware asociado al temporizador. No empieza a contar.
 */
void temporizador_drv_iniciar(void){
    temporizador_hal_iniciar();
}

```

```

/**
 * @brief Empieza el temporizador.
 *
 * Inicia el temporizador del sistema
 */
void temporizador_drv_empezar(void){
    temporizador_hal_empezar();
}

/**
 * @brief Lee el valor del temporizador.
 *
 * @return El valor del temporizador en microsegundos.
 */
uint64_t temporizador_drv_leer(void){
    return temporizador_hal_leer() * temporizador_hal_tick2us;
}

/**
 * @brief Para el temporizador y devuelve el valor actual.
 *
 * @return El valor del temporizador en microsegundos.
 */
uint64_t temporizador_drv_parar(void){
    return temporizador_hal_parar() * temporizador_hal_tick2us;
}

/**
 * @brief Programa un reloj periódico.
 *
 * Esta función programa el reloj para que encole un evento periódicamente en la cola del
 * planificador.
 * @param periodo El periodo del reloj en milisegundos.
 * @param funcion_encolar_evento Puntero a la función que encola el evento.
 * @param ID_evento El ID del evento a encolar.
 */
void temporizador_drv_reloj(uint32_t periodo, void (*funcion_encolar_evento)(),
EVENTO_T ID_evento) {
    void (*callback_reloj_ptr)(void) = callback_reloj;    // pasamos la funcion
    encolar_evento = funcion_encolar_evento;              // callback a hal
    evento = ID_evento;
    temporizador_hal_reloj(periodo, callback_reloj_ptr);
}

/**

```

```

* @brief Prueba del controlador del temporizador.
*
* Esta función prueba el controlador del temporizador.
*
* @return El resultado de la prueba.
*/
int timer0_driver_test(void) {
    temporizador_drv_iniciar();
    temporizador_drv_empezar();
    return temporizador_hal_test();
}
/**
* @brief Función SWI para leer el temporizador.
*
* @return El valor del temporizador en microsegundos.
*/
uint32_t __SWI_0 (void){
    return temporizador_drv_leer();
}
void callback_reloj(void) {
    encolar_evento(evento, 0);      // funcion que encola el evento en hal
}

```

FIFO:

```

#include <inttypes.h>
#include <LPC210X.H>    // LPC21XX Peripheral Registers
#include "gpio.h"
#include "system_calls.h"
#include "evento.h"

typedef struct {
    EventoCola eventos[MAX_EVENTOS];
    uint8_t inicio;
    uint8_t fin;
    GPIO_HAL_PIN_T pin_overflow;
    uint8_t overflow;
    uint32_t estadisticas[NUM_TIPO_EVENTOS];
} FIFO;

static FIFO cola;

// Inicialización de la cola. Se le pasa como parámetro el pin del
// GPIO utilizado para marcar errores.

```

```

void FIFO_iniciar(GPIO_HAL_PIN_T pin_overflow) {
    int i;
    gpio_hal_sentido(GPIO_OVERFLOW, GPIO_OVERFLOW_BITS, GPIO_HAL_PN_DIR_OUTPUT);
    cola.pin_overflow = pin_overflow;
    cola.inicio = 0;
    cola.fin = 0;
    cola.overflow = 0;
    for (i = 0; i < NUM_TIPO_EVENTOS; i++) {          // inicializamos las estad.
        cola.estadisticas[i] = 0;
    }
}

// Esta función guardará en la cola el evento. El campo ID_evento,
// que permita identificar el evento (p.e. qué interrupción ha saltado)
// y el campo auxData en caso de que el evento necesite pasar información extra.
void FIFO_encolar(EVENTO_T ID_evento, uint32_t auxData) {
    uint8_t state_irq_bit;
    state_irq_bit = read_IRQ_bit();
    if (state_irq_bit == 0)    disable_irq();          // operación atómica
    cola.eventos[cola.fin].ID_evento = ID_evento;
    cola.eventos[cola.fin].auxData = auxData;
    cola.estadisticas[ID_evento]++; // incrementamos el contador de eventos
    cola.fin++;
    if (cola.fin == MAX_EVENTOS) cola.fin = 0;        // si fin-> principio
    if (cola.fin == cola.inicio) FIFO_overflow();
    if (state_irq_bit == 0) enable_irq();             // salimos de atomicidad
}

// Si hay eventos sin procesar, devuelve un valor distinto de cero y
// el evento más antiguo sin procesar por referencia. Cero indicará que la
// cola está vacía y no se ha devuelto ningún evento.
uint8_t FIFO_extraer(EVENTO_T *ID_evento, uint32_t* auxData) {
    uint8_t state_irq_bit;
    if (cola.inicio == cola.fin) return 0;
    else {
        state_irq_bit = read_IRQ_bit();
        if (state_irq_bit == 0) disable_irq();
        *ID_evento = cola.eventos[cola.inicio].ID_evento;
        *auxData = cola.eventos[cola.inicio].auxData;
        cola.inicio++;
        if (cola.inicio == MAX_EVENTOS) cola.inicio = 0;
        if (state_irq_bit == 0)    enable_irq();
    }
}

```



```

    }
    return 1;
}

// Dado un identificador de evento nos devuelve el número total de veces
// que ese evento se ha encolado. El evento VOID nos devolverá el total de
// eventos encolados desde el inicio.
uint32_t FIFO_estadisticas(EVENTO_T ID_evento) {
    if (ID_evento == VOID) {
        int i;
        uint32_t total = 0;
        for (i = 1; i < NUM_TIPO_EVENTOS; i++) {
            total += cola.estadisticas[i];
        }
        return total;
    }
    return cola.estadisticas[ID_evento];
}

// indica overflow en la cola fifo
void FIFO_overflow(void) {
    cola.overflow = 1;
    gpio_hal_escribir(cola.pin_overflow, 0x01, 1);
    while(1);
}

/**
 * @brief Función de prueba para la funcionalidad de la cola FIFO.
 *
 * Esta función realiza pruebas en la funcionalidad de la cola FIFO. Verifica si la cola
 * detecta correctamente cuando está vacía, si encola los elementos correctamente y si las
 * estadísticas son adecuadas. También prueba la extracción de un evento de la cola.
 * @param overflow Indica si se debe probar el caso de desbordamiento de la cola.
 * @return int Retorna 0 si todas las pruebas son exitosas, de lo contrario retorna un valor
distinto de cero.
 */
int FIFO_test(int overflow) {
    uint32_t i, test_encolar;
    uint8_t test_extraer;
    EventoCola EC;
    FIFO_iniciar(31);
    test_extraer = FIFO_extraer(&EC.ID_evento, &EC.auxData);
    if (test_extraer != 0) return 1;
    for (i = 0; i < 4; i++) {

```

```

        FIFO_encolar(TIMER1, 3);
    }
    FIFO_encolar(TIMER0, 4);
    test_encolar = FIFO_estadisticas(VOID);
    if (test_encolar != 5) return 2;
    test_encolar = FIFO_estadisticas(TIMER1);
    if (test_encolar != 4) return 2;
    test_extraer = FIFO_extraer(&EC.ID_evento, &EC.auxData);
    if (EC.ID_evento != TIMER1 || EC.auxData != 3) return 3;
    if (overflow) {
        for (i = 0; i < MAX_EVENTOS-4; i++)FIFO_encolar(VOID, 5);
    }
    return 0;
}

```

GPIO:

io_reserva:

```

#define GPIO_OVERFLOW 31
#define GPIO_OVERFLOW_BITS 1
#define GPIO_HELLO_WORLD 0
#define GPIO_HELLO_WORLD_BITS 8
#define GPIO_LATIDO 16
#define GPIO_LATIDO_BITS 8
#define GPIO_SERIE_ERROR 30
#define GPIO_SERIE_ERROR_BITS 1
#define GPIO_COMANDO_ERRONEO 29
#define GPIO_COMANDO_ERRONEO_BITS 1

```

gpio_reserva:

```

#include <inttypes.h>
#include "io_reserva.h"

```

// Definición del tipo de dirección del pin del GPIO

```

typedef enum {
    GPIO_HAL_PIN_DIR_INPUT,
    GPIO_HAL_PIN_DIR_OUTPUT
} gpio_hal_pin_dir_t;

typedef uint8_t GPIO_HAL_PIN_T; // tipo de representación del pin del GPIO

```

gpio.h:

```
#include <inttypes.h>
#include <LPC210X.H>    #include "gpio_reserva.h"
// Permite emplear el GPIO y debe ser invocada
// antes de poder llamar al resto de funciones de la biblioteca.
__inline static void gpio_hal_iniciar() {
    PINSEL0 = 0;
    PINSEL1 = 0;
}
// los bits indicados se utilizarán como
// entrada o salida según la dirección.

__inline static void gpio_hal_sentido(GPIO_HAL_PIN_T gpio_inicial, uint8_t
                                     num_bits, gpio_hal_pin_dir_t direccion) {
    uint32_t mask = (1u << num_bits) - 1;
    if (direccion == GPIO_HAL_PIN_DIR_INPUT) IODIR &= ~(mask << gpio_inicial);
    else IODIR |= mask << gpio_inicial;
}
// gpio_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. Ejemplo:
//     o valor de los pines: 0xF0FAFF0
//     o bit_inicial: 12 num_bits: 4
//     o valor que retorna la función: 10 (lee los 4 bits 12-15)
__inline static uint32_t gpio_hal_leer(GPIO_HAL_PIN_T gpio_inicial, uint8_t
num_bits) {
    uint32_t mask = (1u << num_bits) - 1;
    return (IOPIN >> gpio_inicial) & mask;
}
// 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)
__inline static void
gpio_hal_escribir(GPIO_HAL_PIN_T bit_inicial, uint8_t num_bits, uint32_t valor) {
    uint32_t mask = (1u << num_bits) - 1;
    IOCLR = mask << bit_inicial;
    IOSET = (valor & mask) << bit_inicial;
}
// si todo sale bien, devuelve 0. Si falla algún punto, devuelve 0
int gpio_hal_test(void) { // implementado en un .c
```

```

    int test;
    gpio_hal_iniciar();
    gpio_hal_sentido(0, 4, GPIO_HAL_PIN_DIR_OUTPUT);
    test = gpio_hal_leer(0, 4);
    if (test != 0) return 1;
    gpio_hal_escribir(0, 2, 7);
    test = gpio_hal_leer(0, 4);
    if (test != 3) return 1;
    gpio_hal_escribir(0, 4, 0); // borra todos los valores
    test = gpio_hal_leer(0, 4);
    if (test != 0) return 1;
    gpio_hal_escribir(0, 4, 4); // escribe un 4
    test = gpio_hal_leer(0, 4);
    if (test != 4) return 1;
    return 0;
}

```

Power:

```

#include <inttypes.h>
#include <LPC210X.H> // LPC21XX Peripheral Registers

/**
 * @brief Pone el sistema en modo de bajo consumo.
 *
 * Esta función configura los pines de interrupción externa (EXTINT1 y EXTINT2) para despertar
 * el procesador del modo de bajo consumo. Luego, activa el modo de bajo consumo en el
 * registro PCON.
 */
void power_hal_down(void) {
    EXTWAKE = 6;
    PCON = (PCON | 0x1);
}

/**
 * @brief Pone el sistema en modo de sueño profundo.
 *
 * Esta función configura los pines de interrupción externa (EXTINT1 y EXTINT2) para despertar
 * el procesador del modo de sueño profundo. Luego, activa el modo de sueño profundo en el
 * registro PCON y cambia a la frecuencia del PLL para un rápido restablecimiento del sistema
 * al despertar.
 */
void power_hal_deep_sleep(void) {

```

```

        EXTWAKE = 6;
        PCON = (PCON | 0x2);
        Switch_to_PLL();
    }
/**
 * @brief Cambia la frecuencia del procesador a 60 MHz.
 *
 * Función que configura el PLL para que la frecuencia del procesador sea de 60 MHz.
 */
extern void Switch_to_PLL(void);
Implementación de la función en ARM
Switch_to_PLL
        LDR        R0, =PLL_BASE
        MOV        R1, #0xAA
        MOV        R2, #0x55

; Configure and Enable PLL
        MOV        R3, #PLLCFG_Val
        STR        R3, [R0, #PLLCFG_OFS]
        MOV        R3, #PLLCON_PLLE
        STR        R3, [R0, #PLLCON_OFS]
        STR        R1, [R0, #PLLFEED_OFS]
        STR        R2, [R0, #PLLFEED_OFS]

; Wait until PLL Locked
S_PLL_Loop    LDR        R3, [R0, #PLLSTAT_OFS]
               ANDS      R3, R3, #PLLSTAT_PLOCK
               BEQ        S_PLL_Loop

; Switch to PLL Clock
        MOV        R3, #(PLLCON_PLLE:OR:PLLCON_PLLC)
        STR        R3, [R0, #PLLCON_OFS]
        STR        R1, [R0, #PLLFEED_OFS]
        STR        R2, [R0, #PLLFEED_OFS]
        BX         LR

```

Alarmas:

```

#include <inttypes.h>
#include "evento.h"
#include "timer0_driver.h"

```

```

#define ALARMAS_MAX 4

typedef struct {
    EVENTO_T      ID_evento;
    uint64_t      timestamp;
    uint32_t      retardo;
    uint32_t      auxData;
    uint8_t       periodica;
} Alarma;

static void (*callback_encolar)(EVENTO_T ID_evento, uint32_t auxData);
static Alarma listaAlarmas[ALARMAS_MAX];
static uint8_t numAlarmas;

/**
 * @brief Inicializa el sistema de alarmas y configura la función de callback.
 *
 * @param funcion_callback Puntero a la función de callback que se ejecutará al activarse una
 * alarma.
 * @param ID_evento Identificador del evento asociado a las alarmas.
 */
void alarma_iniciar(void (*funcion_callback)(), EVENTO_T ID_evento) {
    numAlarmas = 0;
    callback_encolar = funcion_callback;
    temporizador_drv_reloj(1, callback_encolar, ID_evento);
}

void extraer_alarma(EVENTO_T ID_evento) {
    int i;
    Alarma alarma_i, ultimaAlarma;
    for (i=0; i<numAlarmas; i++) {
        alarma_i = listaAlarmas[i];
        if (alarma_i.ID_evento == ID_evento) {
            ultimaAlarma = listaAlarmas[numAlarmas-1];
            listaAlarmas[i] = ultimaAlarma;
            numAlarmas--;
        }
    }
}

/**
 * @brief Activa una alarma con el retardo especificado.
 *
 * @param ID_evento Identificador del evento asociado a la alarma.

```

```

* @param retardo Retardo en milisegundos antes de que se active la alarma.
* @param auxData Datos adicionales asociados a la alarma.
*/
void alarma_activar(EVENTO_T ID_evento, uint32_t retardo, uint32_t auxData) {
    uint8_t periodica;
    uint64_t timestamp;
    Alarma nuevaAlarma;
    extraer_alarma(ID_evento);
    if (retardo == 0) return;
    if (numAlarmas >= ALARMAS_MAX) {
        callback_encolar(ALARMA_OVERFLOW, 0);
        return;
    }
    periodica = (retardo & 0x80000000) != 0;
    retardo &= 0x7FFFFFFF;
    timestamp = temporizador_drv_leer();
    nuevaAlarma.ID_evento = ID_evento;
    nuevaAlarma.timestamp = timestamp + retardo*1000;
    nuevaAlarma.retardo = retardo;
    nuevaAlarma.auxData = auxData;
    nuevaAlarma.periodica = periodica;
    listaAlarmas[numAlarmas] = nuevaAlarma;
    numAlarmas++;
}
/**
* @brief Función que se ejecuta periódicamente para tratar las alarmas.
*/
void alarma_tratar_evento(void) {
    int i;
    uint64_t now = temporizador_drv_leer();
    Alarma alarma_i;
    for (i=0; i<numAlarmas; i++) {
        alarma_i = listaAlarmas[i];
        if (alarma_i.timestamp <= now) {
            callback_encolar(alarma_i.ID_evento, alarma_i.auxData);
            if(alarma_i.ID_evento==ev_LATIDO)
                alarma_i.ID_evento = ev_LATIDO;
            if (alarma_i.periodica != 0) {
                alarma_i.retardo |= 0x80000000;
                alarma_activar(alarma_i.ID_evento, alarma_i.retardo,

```

```

        alarma_i.auxData);
    }
    else    extraer_alarma(alarma_i.ID_evento);
}
}

/**
 * @brief Realiza pruebas de las funciones relacionadas con las alarmas.
 *
 * @param FIFO_encolar Puntero a la función de encolado de eventos en la FIFO.
 * @param FIFO_iniciar Puntero a la función de inicialización de la FIFO.
 * @return 0 si las pruebas son exitosas, 1 en caso contrario.
 */
int alarmas_test(void (*FIFO_encolar)(EVENTO_T ID_evento, uint32_t auxData),
                void (*FIFO_iniciar)(GPIO_HAL_PIN_T pin_overflow)){
    uint8_t  fifoExt;
    uint32_t retardoPeriodico, auxData;
    EVENTO_T ID_evento;
    uint8_t sigue = 1;
    temporizador_drv_iniciar();
    temporizador_drv_empezar();
    FIFO_iniciar(GPIO_OVERFLOW);
    alarma_iniciar(FIFO_encolar, ALARMA_TRATAR);
    retardoPeriodico = 30 | 0x80000000;
    alarma_activar(VOID, 20, 3); // Alarma con ID 1, retardo de 2000
    alarma_activar(TIMER0, retardoPeriodico, 0); // Alarma con ID 2, retardo 3000
    while (temporizador_drv_leer() < 22 && sigue) {
        alarma_tratar_evento();
        fifoExt = FIFO_extraer(&ID_evento, &auxData);
        if (fifoExt != 0) {
            if (ID_evento != 0)    return 1;
            if (auxData != 3)      return 1;
        } else    sigue = 0;
    }
    sigue = 1;
    while (temporizador_drv_leer() < 32 && sigue) {
        alarma_tratar_evento();
        fifoExt = FIFO_extraer(&ID_evento, &auxData);
        if (fifoExt != 0) {          // no ha descolado nada
            if (ID_evento != 1)    return 1;

```



```

        if (auxData != 0) return 1;
    } else        sigue = 0;
}
sigue = 1;
while (temporizador_drv_leer() < 64) {
    alarma_tratar_evento();
    fifoExt = FIFO_extraer(&ID_evento, &auxData);
    if (fifoExt != 0) {        // no ha desencolado nada
        if (ID_evento != 1)    return 1;
        if (auxData != 0)      return 1;
    } else        sigue = 0;
    alarma_activar(TIMER0, 0, 0); // Alarma con ID 2, retardo de 3000
}
while (temporizador_drv_leer() < 94) {
    alarma_tratar_evento();
    fifoExt = FIFO_extraer(&ID_evento, &auxData);
    if (fifoExt != 0) return 1;    // no ha desencolado nada
}
alarma_activar(VOID, 20, 3);
alarma_activar(TIMER0, 20, 3);
alarma_activar(TIMER1, 20, 3);
alarma_activar(ALARMA_OVERFLOW, 20, 3);
alarma_activar(ALARMA_TRATAR, 20, 3);
fifoExt = FIFO_extraer(&ID_evento, &auxData);
if (fifoExt != 1) return 1;    // no ha desencolado nada
if (ID_evento != ALARMA_OVERFLOW) return 1;
return 0;
}

```

Int externas hal:

```

#include <LPC210X.H>
#include <stddef.h>
#include <inttypes.h>

```

```

static void (*interrupcion_eint1)(uint8_t);
static void (*interrupcion_eint2)(uint8_t);

```

```

void eint1_ISR(void) __irq {
    VICIntEnClr = 0x00008000;
    if (interrupcion_eint1 != NULL) interrupcion_eint1(1);
}

```

```

        EXTINT = EXTINT | 2;
        VICVectAddr = 0;
    }
/**
 * @brief Inicializa la interrupción EINT1.
 *
 * Esta función configura los registros necesarios para habilitar la interrupción EINT1.
 *
 * @param interrupcion_eint_par Puntero a la función de interrupción EINT1.
 */
void eint1_iniciar(void (*interrupcion_eint_par)(uint8_t)) {
    EXTINT = EXTINT | 2;
    VICVectAddr2 = (unsigned long)eint1_ISR;
    PINSEL0      = PINSEL0 & 0xCFFFFFFF;
    PINSEL0      = PINSEL0 | 0x20000000;
    VICVectCntl2 = 0x20 | 15;
    VICIntEnable = VICIntEnable | 0x00008000;
    interrupcion_eint1 = interrupcion_eint_par;
}
/**
 * @brief Lee el estado del botón asociado a EINT1.
 *
 * Esta función lee el estado del botón asociado a EINT1 y devuelve su valor.
 *
 * @return Estado del botón asociado a EINT1 (0 o 1).
 */
unsigned int eint1_read_button(void){
    EXTINT = EXTINT | 2;
    return (EXTINT & (1 << 1)) >> 1;
};
/**
 * @brief Habilita la interrupción EINT1.
 *
 * Esta función habilita la interrupción EINT1.
 */
void eint1_enable(void){
    EXTINT = EXTINT | 2;
    VICIntEnable = VICIntEnable | 0x00008000;
}

void eint2_ISR(void) __irq {

```

```

        VICIntEnClr = 0x00010000;
        if (interrupcion_eint2 != NULL) interrupcion_eint2(2);
        VICVectAddr = 0;
        EXTINT = EXTINT | 4;
    }

/**
 * @brief Inicializa la interrupción EINT2.
 *
 * Esta función configura los registros necesarios para habilitar la interrupción EINT2.
 *
 * @param interrupcion_eint_par Puntero a la función de interrupción EINT2.
 */
void eint2_iniciar(void (*interrupcion_eint_par)(uint8_t)) {
    EXTINT = EXTINT | 4;
    VICVectAddr3 = (unsigned long)eint2_ISR;
    PINSEL0      = PINSEL0 & 0x3FFFFFFF;
    PINSEL0      = PINSEL0 | 0x80000000;
    VICVectCntl3 = 0x20 | 16;
    VICIntEnable = VICIntEnable | 0x00010000;
    interrupcion_eint2 = interrupcion_eint_par;
}

/**
 * @brief Lee el estado del botón asociado a EINT2.
 *
 * Esta función lee el estado del botón asociado a EINT2 y devuelve su valor.
 *
 * @return Estado del botón asociado a EINT2 (0 o 1).
 */
unsigned int eint2_read_button(void){
    EXTINT = EXTINT | 4;
    return (EXTINT & (1 << 2)) >> 2;
};

/**
 * @brief Habilita la interrupción EINT2.
 *
 * Esta función habilita la interrupción EINT2.
 */
void eint2_enable(void){
    EXTINT = EXTINT | 4;
    VICIntEnable = VICIntEnable | 0x00010000;
}

```

```

/**
 * @brief Función de prueba para el módulo de interrupciones externas.
 *
 * Esta función realiza una prueba de las funciones de manejo de interrupciones externas EINT1
 * y EINT2.
 * @return 0 si la prueba se completó correctamente.
 */
int int_externas_hal_test(void) {
    eint1_iniciar(NULL);
    eint2_iniciar(NULL);
    while (eint1_read_button() != 1) {}
    while (eint2_read_button() != 1) {}
    return 0;
}

```

Botones:

```

#include "botones.h"

static enum ESTADO estado_boton_1, estado_boton_2;
static void (*callback_encolar)(EVENTO_T ID_evento, uint32_t auxData);
static EVENTO_T eventoEncolar;
static void (*callback_temp)(EVENTO_T ID_evento, uint32_t retardo, uint32_t auxData);
static EVENTO_T eventoTemp;

/**
 * @brief Maneja la interrupción generada por los botones.
 *
 * Esta función se ejecuta cuando se produce una interrupción por pulsación de un botón.
 * Si el botón no estaba previamente pulsado, se encola un evento de pulsación y se actualiza
 * el estado del botón.
 *
 * @param num_boton Número del botón que generó la interrupción.
 */
void interrupcion_eint(uint8_t num_boton) {
    // si el boton no esta pulsado y se interrumpe, encolamos evento
    // pulsacion y establecemos estado del boton como pulsado
    if (num_boton == 1) {
        if (estado_boton_1 == NO_PULSADO)
            callback_encolar(eventoEncolar, num_boton);
    } else if (num_boton == 2) {
        if (estado_boton_2 == NO_PULSADO)
            callback_encolar(eventoEncolar, num_boton);
    }
}

```

```

    }
}

/**
 * @brief Inicializa el módulo de botones.
 *
 * Esta función se encarga de inicializar el módulo de botones, estableciendo el estado
 inicial de los botones
 * y configurando las rutinas de interrupción correspondientes.
 *
 * @param callbackCola Puntero a la función que se ejecutará al encolar un evento.
 * @param ev_pulsacion Evento que se encolará al detectar una pulsación de botón.
 * @param callback_temporizador Puntero a la función que se ejecutará al activar un
 temporizador.
 * @param ev_temporizador Evento que se encolará al activar un temporizador.
 */
void botones_iniciar(void (*callbackCola)(), EVENTO_T ev_pulsacion,
    void (*callback_temporizador)(), EVENTO_T ev_temporizador) {
    estado_boton_1 = NO_PULSADO; // establecemos estado de los botones
    estado_boton_2 = NO_PULSADO;
    eint1_iniciar(&interrupcion_eint); // inicializamos los botones
    eint2_iniciar(&interrupcion_eint);
    eventoEncolar = ev_pulsacion;
    callback_encolar = callbackCola;
    eventoTemp = ev_temporizador;
    callback_temp = callback_temporizador;
}

/**
 * @brief Comprueba si algún botón está pulsado.
 *
 * Esta función comprueba si alguno de los botones está pulsado.
 * Si ninguno de los botones está pulsado, desactiva la comprobación y encola un evento de
 finalización.
 *
 * @return 1 si algún botón está pulsado, 0 en caso contrario.
 */
int boton_esta_pulsado(void) {
    // actualizamos el estado de los botones
    if (eint1_read_button() == 0) {
        estado_boton_1 = NO_PULSADO;
        eint1_enable();
    }
}

```

```

        if (eint2_read_button() == 0) {
            estado_boton_2 = NO_PULSADO;
            eint2_enable();
        }
        if (estado_boton_1 == NO_PULSADO && estado_boton_2 == NO_PULSADO) {
            // si no está pulsado ningun boton desactivamos la comprobación
            callback_temp(eventoTemp, 0, 0);
            return 0;
        }
        return 1;
    }
}

/**
 * @brief Maneja la pulsación de un botón.
 *
 * Esta función debe ejecutar cuando se detecta una pulsación de botón.
 * Realiza las acciones correspondientes a la pulsación, como encolar un evento y actualizar
 * el estado del botón.
 *
 * @param num_boton Número del botón que fue pulsado.
 */
void handle_pulsacion_boton(uint8_t num_boton) {
    callback_temp(eventoTemp, 100 | 0x80000000, 0);
    if (num_boton == 1) estado_boton_1 = PULSADO;
    else if (num_boton == 2) estado_boton_2 = PULSADO;
}

/**
 * @brief Prueba el módulo de botones.
 *
 * Esta función realiza una prueba del módulo de botones, ejecutando una serie de pruebas
 * internas.
 * @return 0 si todas las pruebas pasan correctamente, un valor distinto de 0 en caso
 * contrario.
 */
int botones_test(void) {
    return int_externas_hal_test();
}

```

Hello world:

```

#include <inttypes.h>
#include "timer0_driver.h"
#include "evento.h"

```

```

static int temp = 0;
static EVENTO_T eventoEncolar;
/**
 * @brief Inicializa el módulo "Hello World".
 *
 * Esta función se encarga de inicializar el módulo "Hello World". Configura los pines de
 * salida necesarios, activa una alarma y guarda el evento de encolar y la función de callback
 * proporcionados.
 * @param funcion_callback Puntero a la función de callback que se llamará al encolar un
 * evento.
 * @param ID_evento_alarma Identificador del evento de alarma.
 * @param ID_evento_encolar Identificador del evento de encolar.
 */
void hello_world_iniciar(void (*funcion_callback)(), EVENTO_T ID_evento_alarma,
EVENTO_T ID_evento_encolar) {
    gpio_hal_sentido(GPIO_HELLO_WORLD, GPIO_HELLO_WORLD_BITS, GPIO_HAL_P_OUTPUT);
    temp = 0;
    alarma_activar(ID_evento_alarma, 10 | 0x80000000, 0);
    eventoEncolar = ID_evento_encolar;
    callback_encolar = funcion_callback;
}
/**
 * @brief Realiza una operación de "tick-tack" en el módulo "Hello World".
 *
 * Representa en binario el valor del contador temporal en la salida.
 */
void hello_world_tick_tack(void) {
    temp++;
    gpio_hal_escribir(GPIO_HELLO_WORLD, GPIO_HELLO_WORLD_BITS, temp%(1 << 8));
}
/**
 * @brief Trata un evento en el módulo "Hello World".
 *
 * Incrementa el contador temporal y llama a la función de callback proporcionada
 * pasando el evento de encolar y el contador temporal como argumentos.
 */
void hello_world_tratar_evento(void) {
    temp++;
    callback_encolar(eventoEncolar, temp);
}

```

Visualizar:

```
#include "gpio.h"

/**
 * @brief Inicializa la salida.
 *
 * Configura los pines correspondientes como salidas.
 */
void visualizar_iniciar(void) {
    // Configurar los pines del 23 al 16 como salidas
    gpio_hal_sentido(GPIO_LATIDO, GPIO_LATIDO_BITS, GPIO_HAL_PIN_DIR_OUTPUT);
}

/**
 * @brief Muestra una cuenta en la salida.
 *
 * @param cuenta El valor de la cuenta a mostrar.
 */
void visualizar_cuenta(int cuenta){
    gpio_hal_escribir(GPIO_LATIDO, GPIO_LATIDO_BITS, cuenta);
}

/**
 * @brief Muestra el valor de la variable en la salida.
 *
 * @param temp El valor de la variable a mostrar.
 */
void visualizar_hello_world(int temp) {
    gpio_hal_escribir(GPIO_HELLO_WORLD, GPIO_HELLO_WORLD_BITS, temp%(1 << 8));
}
```

Watchdog:

```
#include <LPC210X.H>    // LPC21XX Peripheral Registers
#include "system_calls.h"

/**
 * @brief Inicializa el watchdog timer con el tiempo de cuenta especificado en segundos.
 *
 * @param sec Tiempo de cuenta en segundos.
 */
void WD_hal_iniciar(int sec) {
    WDTC = sec * 15000000 / 4;    // Configurar el tiempo de cuenta en segundos
    WDMOD = 3;                    // Configurar el watchdog para generar un reset y habilita
}
```



```

/**
 * @brief "Alimenta" el watchdog timer para evitar el reset.
 *
 * Esta función debe ser llamada periódicamente para evitar que el watchdog timer
 * genere un reset. Deshabilita las interrupciones antes de alimentar el watchdog
 * y las habilita nuevamente después.
 */
void WD_hal_feed() {
    uint8_t state_irq_bit    = read_IRQ_bit();
    disable_fiq();
    if (state_irq_bit == 0) disable_irq();
    WDFEED = 0xAA;
    WDFEED = 0x55;
    if (state_irq_bit == 0)    enable_irq();
}

/**
 * @brief Función de prueba con un bucle infinito para comprobar el watchdog.
 *
 * Esta función inicializa el watchdog timer con un tiempo de cuenta de 5 segundos
 * y luego llama a la función WD_hal_feed() para "alimentar" el watchdog. A continuación,
 * entra en un bucle infinito para simular una ejecución continua del programa.
 */
void WD_hal_test() {
    WD_hal_iniciar(5);
    WD_hal_feed();
    while (1) {}
}

```

System_calls.c:

```

#include "system_calls.h"
#include "timer0_driver.h"

// testea las llamadas al sistema
uint32_t system_calls_test(void){
    uint32_t tiempo1;
    uint32_t tiempo2;
    uint32_t bitIRQ;
    temporizador_drv_iniciar();
    temporizador_drv_empezar();
    enable_irq();
    tiempo1 = temporizador_drv_leer();

```

```

    tiempo2 = clock_getus();
    tiempo1 = tiempo1 + 3;
    if(tiempo1 != tiempo2) return 1;
    bitIRQ = read_IRQ_bit()
    if(bitIRQ != 0) return 1;
    disable_irq();
    bitIRQ = read_IRQ_bit();
    if(bitIRQ != 1) return 1;
    return 0;
}

```

System_calls.h:

```

#include <LPC210X.H>    // LPC21XX Peripheral Registers
#include <stdint.h>

uint32_t __swi(1) read_IRQ_bit(void);
void __swi(0xFF) enable_irq(void);
void __swi(0xFE) disable_irq(void);
void __swi(0xFD) disable_fiq(void);
uint32_t system_calls_test(void);

```

Linea_serie_hal:

```

#include <LPC210X.H>    // LPC21XX Peripheral Registers
#include <stddef.h>
#include "system_calls.h"

static void (*interrupcion_serie0)(uint8_t);

void serie0_ISR(void) __irq {
    if (interrupcion_serie0 != NULL && serial0_hal_char_recibido())
        interrupcion_serie0(U0RBR);
    VICVectAddr = 0;    // Reconoce interrupcion escribiendo un 0
}

/**
 * @brief Inicializa la interfaz serie 0.
 * @param interrupcion_serie0_par Puntero a la función de interrupción para la serie 0.
 */
void serial0_hal_iniciar(void (*interrupcion_serie0_par)(uint8_t)) {
    PINSEL0 |= 0x5;          /* Enable RxD1 and TxD1          */
    U0LCR = 0x83;            /* 8 bits, no Parity, 1 Stop bit */
    U0DLL = 97;              /* 9600 Baud Rate @ 15MHz VPB Clock */
}

```

```

        U0LCR = 0x03;                                /* DLAB = 0 */
        U0IER = 3;
        interrupcion_serie0 = interrupcion_serie0_par;
        VICVectCntl4 = 0x20 | 6;
        VICVectAddr4 = (unsigned long)serie0_ISR;
        VICIntEnable = VICIntEnable | 0x00000040;
    }
    /**
     * @brief Verifica si se ha recibido un carácter en la interfaz serie 0.
     * @return 1 si se ha recibido un carácter, 0 en caso contrario.
     */
    uint32_t serial0_hal_char_recibido(void) {
        return (U0IIR & 0xE) == 4;
    }
    /**
     * @brief Escribe un carácter en la interfaz serie 0.
     * @param caracter El carácter a escribir.
     */
    void serial0_hal_escribir(char caracter) {
        while (!(U0LSR & 0x20));
        U0THR = caracter;
    }

```

Linea serie drv:

```

#include "system_calls.h"
#include "linea_serie_hal.h"
#include "gpio.h"
#include "evento.h"
#include "utils.h"

static void (*callback_encolar)(EVENTO_T ID_evento, uint32_t auxData);

enum ESTADO_RECIBIR {
    ALM,
    WAIT,
    LEN_ERROR,
};

enum ESTADO_ENVIAR {
    FIN,
    ENVIAR,
};

static enum ESTADO_RECIBIR estado_recibir;
static EVENTO_T evFinRecibir, evFinMostrar, evContinuaMostrar;

```

```

static char buffer_recibir[3];
static char buffer_enviar[2000];
static uint32_t index_recibir = 0;
static uint32_t index_enviar = 0;
/**
 * @brief Función de interrupción de la línea serie.
 *
 * Esta función se llama cuando se produce una interrupción en la línea serie.
 * Analiza el estado actual de recepción y realiza las acciones correspondientes.
 *
 * @param caracter El caracter recibido.
 */
void interrupcion_serie(uint8_t caracter) {
    if (estado_recibir == WAIT) {
        if (caracter == '$') {
            estado_recibir = ALM;
            index_recibir = 0;
        }
    } else if (estado_recibir == ALM) {
        if (caracter == '!') {
            estado_recibir = WAIT;
            if (index_recibir == 3) {
                uint32_t auxData = atoi(buffer_recibir);
                callback_encolar(evFinRecibir, auxData);
            }
        } else if (index_recibir >= 3) {
            estado_recibir = LEN_ERROR;
            gpio_hal_escribir(GPIO_SERIE_ERROR, GPIO_SERIE_ERROR_BITS, 1);
        } else {
            buffer_recibir[index_recibir] = caracter;
            index_recibir++;
        }
    } else if (estado_recibir == LEN_ERROR) {
        if (caracter == '$') {
            index_recibir = 0;
            estado_recibir = ALM;
            gpio_hal_escribir(GPIO_SERIE_ERROR, GPIO_SERIE_ERROR_BITS, 0);
        }
    } else while(1);
}

```

```

/**
 * @brief Inicia el controlador de la línea serie.
 *
 * Esta función inicializa el controlador de la línea serie.
 * Configura los eventos de finalización de recepción, finalización de mostrar y continuación
 * de mostrar. También configura la función de callback para encolar eventos.
 *
 * @param funcion_callback La función de callback para encolar eventos.
 * @param finRecibir El evento de finalización de recepción.
 * @param finMostrar El evento de finalización de mostrar.
 * @param continuaMostrar El evento de continuación de mostrar.
 */
void linea_serie_drv_iniciar(void (*funcion_callback)(), EVENTO_T finRecibir,
    EVENTO_T finMostrar, EVENTO_T continuaMostrar) {
    estado_recibir = WAIT;
    serial0_hal_iniciar(&interrupcion_serie);
    index_enviar = 0;
    index_recibir = 0;
    evFinRecibir = finRecibir;
    evFinMostrar = finMostrar;
    evContinuaMostrar = continuaMostrar;
    callback_encolar = funcion_callback;
}

/**
 * @brief Envía un array a través de la línea serie.
 *
 * Esta función envía un array de caracteres a través de la línea serie.
 * El envío se realiza de forma asíncrona, es decir, la función no bloquea.
 *
 * @param salida El array de caracteres a enviar.
 */
void linea_serie_drv_enviar_array(char salida[]) {
    int i;
    if (index_enviar != 0) return;
    for (i=0; salida[i] != '\0' && i<2000; i++) {
        buffer_enviar[i] = salida[i];
    }
    buffer_enviar[i] = salida[i];
    serial0_hal_escribir(buffer_enviar[index_enviar++]);
    callback_encolar(evContinuaMostrar, 0);
}

```

```

/**
 * @brief Continúa el envío de datos a través de la línea serie.
 *
 * Esta función continúa el envío de datos a través de la línea serie.
 * Después de enviar todos los datos, se envía un carácter de nueva línea y se llama al evento
 * de finalización de mostrar.
 */
void linea_serie_drv_continuar_envio(void) {
    if (buffer_enviar[index_enviar] != '\0') {
        serial0_hal_escribir(buffer_enviar[index_enviar++]);
        callback_encolar(evContinuaMostrar, 0);
    } else {
        serial0_hal_escribir('\n');
        callback_encolar(evFinMostrar, 0);
        index_enviar = 0;
    }
}

```

Utils:

```

#include <inttypes.h>
#include <stddef.h>

/**
 * @brief Convierte una cadena de caracteres en un número entero sin signo.
 * @param buffer_carga Cadena de caracteres a convertir.
 * @return El número entero sin signo resultante.
 */
uint32_t atoi(const char* buffer_carga){
    uint8_t i;
    uint32_t num = 0;
    for (i = 0; buffer_carga[i] != '\0'; ++i) {
        num = num * 10;
        if (buffer_carga[i] != '-') num += (buffer_carga[i] - '0');
    }
    return num;
}

/**
 * @brief Convierte un número entero en una cadena de caracteres.
 * @param num Número entero a convertir.
 * @param str Cadena de caracteres donde se almacenará el resultado.
 */
void itoa(int num, char str[]) {
    int i = 0;

```

```

    int sign = 1;
    if (num < 0) {
        sign = -1;
        num = -num;
    }
    do {
        str[i++] = num % 10 + '0';
        num = num / 10;
    } while (num != 0);
    if (sign == -1) str[i++] = '-';
    str[i] = '\0';
    reverse(str, i);
}

/**
 * @brief Invierte una cadena de caracteres.
 * @param str Cadena de caracteres a invertir.
 * @param length Longitud de la cadena de caracteres.
 */
void reverse(char str[], int length) {
    int start = 0;
    int end = length - 1;
    while (start < end) {
        char temp = str[start];
        str[start] = str[end];
        str[end] = temp;
        start++;
        end--;
    }
}

/**
 * @brief Concatena dos cadenas de caracteres.
 * @param dest Cadena de caracteres de destino.
 * @param src Cadena de caracteres fuente.
 * @return Puntero a la cadena de caracteres de destino.
 */
char *strcat (char *dest, const char *src){
    const char *p;
    char *q;
    for (q = dest; *q != '\0'; q++)
        ;

```

```

        for(p = src; *p != '\0'; p++, q++) *q = *p;
        *q = '\0';
        return dest;
    }

/**
 * @brief Copia una cadena de caracteres en otra.
 * @param dest Cadena de caracteres de destino.
 * @param source Cadena de caracteres fuente.
 */
void strcpy(char dest[], const char source[]) {
    int i = 0;
    while ((dest[i] = source[i]) != '\0') i++;
}

/**
 * @brief Compara dos cadenas de caracteres.
 * @param s1 Primera cadena de caracteres a comparar.
 * @param s2 Segunda cadena de caracteres a comparar.
 * @return Un valor negativo si s1 es menor que s2, un valor positivo si s1 es mayor que s2, o
 * 0 si son iguales.
 */
int strcmp(const char* s1, const char* s2) {
    while(*s1 && (*s1 == *s2)) {
        s1++;
        s2++;
    }
    return *(const unsigned char*)s1 - *(const unsigned char*)s2;
}

/**
 * @brief Llena un bloque de memoria con un valor específico.
 * @param dst Puntero al bloque de memoria.
 * @param len Longitud del bloque de memoria.
 * @param val Valor a asignar a cada byte del bloque de memoria.
 */
void ms(char *dst, size_t len, char val) {
    char *p;
    for (p = dst; p < dst + len; ++p) {
        *p = val;
    }
}

```


Planificador:

```
#include <stdint.h>
#include "fifo.h"
#include "hello_world.h"
#include "power.h"
#include "alarmas.h"
#include "botones.h"
#include "juego.h"
#include "visualizar.h"
#include "linea_serie_drv.h"
#include "watchdog.h"
#include "evento.h"

#define WATCHDOG_TIMEOUT 1
static uint8_t powerDown;

/**
 * @brief Gestiona los eventos del sistema.
 *
 * Esta función se encarga de gestionar los eventos del sistema, ejecutando las funciones
 * asociadas a cada evento.
 */
void planificador() {
    temporizador_drv_iniciar();          // tiempo de las alarmas
    temporizador_drv_empezar();
    FIFO_iniciar(GPIO_OVERFLOW);        // cola de eventos
    alarma_iniciar(FIFO_encolar, ALARMA_TRATAR);
    botones_iniciar(FIFO_encolar, PULSACION_BOTON, alarma_activar, COMPROBAR_BOTON)
    linea_serie_drv_iniciar(FIFO_encolar, ev_RX_SERIE, ev_TX_SERIE,
                           CONTINUA_ENVIO);
    hello_world_iniciar(FIFO_encolar, ev_LATIDO, ev_VISUALIZAR_HELLO);
    visualizar_iniciar();
    WD_hal_iniciar(WATCHDOG_TIMEOUT);
    powerDown = 0;
    alarma_activar(USUARIO_AUSENTE, TIEMPO_USR_AUSENTE, 0);
    juego_iniciar();    // alarma confirmacion jugada
    while (1) {
        EventoCola EC;
        uint32_t hayDato;
```

```

hayDato = FIFO_extraer(&EC.ID_evento, &EC.auxData);
if (hayDato) {
    WD_hal_feed();
    switch (EC.ID_evento) {
        case ALARMA_TRATAR:
            alarma_tratar_evento();
            break;

        case ALARMA_OVERFLOW:
            FIFO_overflow();
            break;

        case PULSACION_BOTON:
            alarma_activar(USUARIO_AUSENTE, TIEMPO_USR_AUSENTE, 0);
            handle_pulsacion_boton(EC.auxData);
            if (powerDown) powerDown = 0;
            else juego_tratar_evento(EC.ID_evento, EC.auxData);
            break;

        case COMPROBAR_BOTON:
            boton_esta_pulsado();
            break;

        case ev_VISUALIZAR_CUENTA:
            visualizar_cuenta(EC.auxData);
            break;

        case ev_LATIDO:
            hello_world_tratar_evento();
            break;

        case ev_VISUALIZAR_HELLO:
            visualizar_hello_world(EC.auxData);
            break;

        case ev_RX_SERIE:
            alarma_activar(USUARIO_AUSENTE,
                           TIEMPO_USR_AUSENTE, 0);
            juego_tratar_evento(EC.ID_evento, EC.auxData);
    }
}

```

```

        break;

    case ev_TX_SERIE:
        juego_tratar_evento(EC.ID_evento, EC.auxData);
        break;

    case CONTINUA_ENVIO:
        linea_serie_drv_continuar_envio();
        break;

    case JUGADA_CONFIRM_TEMP:
        juego_tratar_evento(EC.ID_evento, EC.auxData);
        break;

    case USUARIO_AUSENTE:
        powerDown = 1;
        power_hal_deep_sleep();
        break;

    default:
        break;
    }
} else {
    power_hal_down();
}
}
}

```

Test:

```

#include <inttypes.h>
#include "timer0_driver.h"
#include "fifo.h"
#include "gpio.h"
#include "alarmas.h"
#include "botones.h"
#include "watchdog.h"

/**
 * @brief Realiza una jugada en el juego.
 */

```

```

* Esta función realiza una jugada en el juego, insertando una ficha en la posición
* especificada.
* Si la jugada es válida, se inserta la ficha y se comprueba si hay una línea de cuatro
* fichas del mismo color. Si hay ganador, "ganador"=1. Si no hay ganador, "ganador"=0.
*
* @param row La fila en la que se desea realizar la jugada.
* @param column La columna en la que se desea realizar la jugada.
* @param colour El color de la ficha a insertar.
*/
int test_all(void) {
    uint8_t test;
    test = botones_test();
    if (test)    return 1;
    test = timer0_driver_test();
    if (test)    return 1;
    test = FIFO_test(0);
    if (test)    return 1;
    test = gpio_hal_test();
    if (test)    return 1;
    test = alarmas_test(FIFO_encolar, FIFO_iniciar);
    if (test)    return 1;
    return 0;
}

/**
* @brief Invoca la función test del watchdog
*
* Invoca la función test del watchdog. No usar en entornos de producción, produce bucle
* infinito.
*/
void test_watchdog(void) {
    WD_hal_test();
}

```

Definición de “evento”:

```

#include <inttypes.h>
#define MAX_EVENTOS 32
#define NUM_TIPO_EVENTOS 14
/**
* @brief Enumeración que representa los diferentes tipos de eventos.
*

```

* Los valores de esta enumeración representan los distintos tipos de eventos que pueden
 * ocurrir en el sistema. Cada valor tiene un significado específico y se utiliza para
 * identificar el tipo de evento que se está manejando.
 */

```
typedef enum {
    VOID = 0,
    TIMER0 = 1,
    TIMER1 = 2,
    ALARMA_OVERFLOW = 3,
    ALARMA_TRATAR = 4,
    PULSACION_BOTON = 5,
    COMPROBAR_BOTON = 6,
    USUARIO_AUSENTE = 7,
    ev_VISUALIZAR_CUENTA = 8,
    ev_LATIDO = 9,
    ev_VISUALIZAR_HELLO = 10,
    ev_TX_SERIE = 11,
    ev_RX_SERIE = 12,
    CONTINUA_ENVIO = 13,
    JUGADA_CONFIRM_TEMP = 14,
} EVENTO_T;
```

```
typedef struct {
    EVENTO_T ID_evento;
    uint32_t auxData;
} EventoCola;
```

Juego:

```
#include <inttypes.h>
#include "linea_serie_drv.h"
#include "timer0_driver.h"
#include "tablero.h"
#include "celda.h"
#include "config_conecta_K.h"
#include "alarmas.h"
#include "utils.h"

enum COMANDO {
    NOC,
    END,
```

```

        NEW,
        TAB,
        JUG,
        ERR,
};

enum ESTADO_CONECTAK {
    INICIO = 0,
    INTERM_INI = 1,
    ESPERA_PARTIDA = 2,
    INTERM_NEW = 3,
    ESPERA_COMANDO = 4,
    INTERM_ERR = 5,
    INTERM_JUG = 6,
    ESPERA_CONFIRMACION = 7,
    INTERM_CONFIRM = 8,
    INTERM_CANC_JUG = 9,
    INTERM_END_COM = 10,
    INTERM_END_CONF = 11,
};

/**
 * @brief Comprueba si hay una línea de cuatro fichas del mismo color.
 *
 * Esta función comprueba si hay una línea de cuatro fichas del mismo color. Corresponde a la
 * cabecera de la función implementada en ensamblador.
 *
 * @param t El tablero de juego.
 * @param row La fila en la que se ha realizado la jugada.
 * @param column La columna en la que se ha realizado la jugada.
 * @param colour El color de la ficha que se ha insertado.
 */
uint8_t conecta_K_hay_linea_arm_arm(TABLER0 *t,uint8_t fila,uint8_t columna,uint8_t
color);

static uint8_t primera_partida = 1;
static TABLER0 cuadrricula;
static uint8_t turno, ganador, num_jugadas, posicion_ocupada;
//Variables automata
char info_juego[] = "Conecta K:\nGana el primero que consiga K fichas en
linea\n-----\nComandos:\n $NEW!: Iniciar un

```

nuevo juego\n \$END!: Terminar el juego\n \$#-#!: Fila-Columna en la que ponemos la ficha\n";

```
static enum ESTADO_CONECTAK estado;
```

```
static uint32_t estadisticas_fifo[NUM_TIPO_EVENTOS];
```

```
//Input
```

```
static uint32_t boton_data;
```

```
static uint8_t boton_pulsado;
```

```
static enum COMANDO comando_data;
```

```
static uint8_t comando_recibido;
```

```
static uint8_t hay_tx;
```

```
//Control automata
```

```
static uint8_t columna_jugada = 0;
```

```
static uint8_t fila_jugada = 0;
```

```
static uint8_t jugada_confirmada = 0;
```

```
//Variables para las estadisticas
```

```
static uint64_t tiempo_total_procesador, tiempo_parc_procesador;
```

```
static uint64_t tiempo_total_hay_linea, tiempo_parc_hay_linea;
```

```
static uint32_t llamadas_hay_linea;
```

```
static uint64_t tiempo_total_humano, tiempo_parc_humano;
```

```
char buffer[1200], buff_itoa[6], tablero[200];
```

```
/**
```

```
 * @brief Inicializa el juego.
```

```
 *
```

```
 * Esta función se encarga de inicializar todas las variables y estructuras necesarias para
```

```
 * comenzar el juego por primera vez.
```

```
 */
```

```
void juego_iniciar(void) {
```

```
    int i;
```

```
    estado = INICIO;
```

```
    conecta_K_automata();
```

```
    for (i=0; i<NUM_TIPO_EVENTOS; i++) {
```

```
        estadisticas_fifo[i] = 0;
```

```
    }
```

```
}
```

```
void juego_reiniciar_partida(void){
```

```
    turno = 2;
```

```
    ganador = 0;
```

```
    num_jugadas = 0;
```

```
    posicion_ocupada = 0;
```

```

    boton_data = 0;
    boton_pulsado = 0;
    comando_data = NOC;
    comando_recibido = 0;
    hay_tx = 0;
    columna_jugada = 0;
    fila_jugada = 0;
    jugada_confirmada = 0;
    tablero_inicializar(&cuadricula);
    if (primera_partida == 1) {
        conecta_K_test_cargar_tablero(&cuadricula);
        primera_partida = 0;
    }
    tiempo_parc_procesador = temporizador_drv_leer();
    tiempo_total_hay_linea = 0;
    tiempo_total_humano = 0;
}

/**
 * @brief Trata un evento del juego.
 *
 * Esta función se encarga de tratar un evento del juego, como una pulsación de botón o un
 * comando recibido.
 * @param ID_evento El identificador del evento.
 * @param auxData Datos adicionales asociados al evento.
 */
void juego_tratar_evento(EVENTO_T ID_evento, uint32_t auxData){
    if (ID_evento == PULSACION_BOTON) {
        boton_data = auxData;
        boton_pulsado = 1;
    }
    else if (ID_evento == ev_RX_SERIE) {
        sel_comando(auxData);
        comando_recibido = 1;
    }
    else if (ID_evento == ev_TX_SERIE){
        hay_tx = 1;
    }
    else if (ID_evento == JUGADA_CONFIRM_TEMP){
        jugada_confirmada = 1;
    }
}

```



```

        conecta_K_automata();
    }
    /**
     * @brief Ejecuta el autómata del juego.
     *
     * Esta función ejecuta el autómata del juego, controlando el flujo del juego y realizando las
     * acciones correspondientes a cada estado.
     */
    void conecta_K_automata(void){
        ms(buffer, 1200, '\0');
        if(estado == INICIO){
            juego_reiniciar_partida();
            linea_serie_drv_enviar_array(info_juego);
            estado = INTERM_INI;
        }
        else if(estado == INTERM_INI && hay_tx){
            hay_tx = 0;
            estado = ESPERA_PARTIDA;
        }
        else if(estado==ESPERA_PARTIDA &&(comando_recibido && comando_data == NEW)){
            strcpy(buffer, conecta_K_visualizar_tablero());
            itoa(turno, buff_itoa);
            strcat(buffer, "Turno de jugador ");
            strcat(buffer, buff_itoa);
            strcat(buffer, "...\\n");
            linea_serie_drv_enviar_array(buffer);
            comando_recibido = 0;
            comando_data = NOC;
            estado = INTERM_NEW;
        }

        else if(estado == INTERM_NEW && hay_tx){
            hay_tx = 0;
            estado = ESPERA_COMANDO;
            tiempo_parc_humano = temporizador_drv_leer();
        }
        else if(estado == ESPERA_COMANDO){
            // CASO PULSACION BOTON 2 PARA TERMINAR PARTIDA
            if (boton_pulsado && boton_data == 2) {
                crear_msjFin(buffer, "boton2");
            }
        }
    }

```

```

        linea_serie_drv_enviar_array(buffer);
        boton_data = 0;
        boton_pulsado = 0;
        estado = INTERM_END_COM;
        turno = (turno == 1) ? 2 : 1;
    }
    // CASO COMANDO RECIBIDO
    else if(comando_recibido){
        if(comando_data == ERR){
            if(columna_jugada == 0){
                strcpy(buffer, "Error: Columna invalida\n");
            }
            else if(fila_jugada == 0){
                strcpy(buffer, "Error: Fila invalida\n");
            }
            else {
                strcpy(buffer, "Error. Comando desconocido: ");
                strcat(buffer, comandoToString(comando_data));
                strcat(buffer, "\n");
            }
            linea_serie_drv_enviar_array(buffer);
            gpio_hal_escribir(GPIO_COMANDO_ERRONEO,
                              GPIO_COMANDO_ERRONEO_BITS, 1);
            estado = INTERM_ERR;
        }
        else if(comando_data == END){
            gpio_hal_escribir(GPIO_COMANDO_ERRONEO,
                              GPIO_COMANDO_ERRONEO_BITS, 0);
            crear_msjFin(buffer, "end");
            linea_serie_drv_enviar_array(buffer);
            estado = INTERM_END_CONF;
        }
        else if(comando_data == JUG){
            tiempo_total_humano += temporizador_drv_leer() -
                                tiempo_parc_humano;
            gpio_hal_escribir(GPIO_COMANDO_ERRONEO,
                              GPIO_COMANDO_ERRONEO_BITS, 0);
            // 3 es el color de la ficha provisional

```

```

        conecta_K_jugada(fila_jugada-1, columna_jugada-1, 3);
    if (posicion_ocupada) {
        strcpy(buffer, "Error: Posicion ocupada. Pon la ficha en
            una casilla libre\n");
        estado = INTERM_ERR;
    } else {
        strcpy(buffer, conecta_K_visualizar_tablero());
        strcat(buffer, "Pulsa boton 1 para cancelar...\n");
        tablero_borrar_celda(&cuadricula, fila_jugada-1,
            columna_jugada-1);
        alarma_activar(JUGADA_CONFIRM_TEMP, 3000, 0);
        estado = INTERM_JUG;
    }
    linea_serie_drv_enviar_array(buffer);
}
comando_recibido = 0;
comando_data = NOC;
}

}

else if(estado == ESPERA_CONFIRMACION){
if((boton_pulsado&&boton_data==2)|| (comando_recibido&&comando_data==END)){
    estado = INTERM_END_CONF;
    if(boton_pulsado){
        crear_msjFin(buffer, "boton2");
        boton_pulsado = 0;
        boton_data = 0;
    }
    if(comando_recibido){
        crear_msjFin(buffer, "end");
        comando_recibido = 0;
        comando_data =NOC;
    }
    linea_serie_drv_enviar_array(buffer);
}

else if(boton_pulsado && boton_data == 1){
    strcpy(buffer, "Jugada cancelada. Inserte una nueva
        jugada...\n");
    linea_serie_drv_enviar_array(buffer);
}

```

```

        estado = INTERM_CANC_JUG;
        boton_pulsado = 0;
        boton_data = 0;
    }
    else if (jugada_confirmada){
        // borrar celda
        conecta_K_jugada(fila_jugada-1 , columna_jugada-1, turno);
        strcpy(buffer, conecta_K_visualizar_tablero());
        fila_jugada = 0;
        columna_jugada = 0;
        num_jugadas++;
        if (ganador){
            crear_msjFin(buffer, "ganador");
            estado = INTERM_END_CONF;
        } else {
            turno = (turno == 1) ? 2 : 1;
            itoa(turno, buff_itoa);
            strcat(buffer, "Turno de jugador ");
            strcat(buffer, buff_itoa);
            strcat(buffer, "...\\n");
            estado = INTERM_CONFIRM;
        }
        linea_serie_drv_enviar_array(buffer);
        jugada_confirmada = 0;
    }
}
else if(estado == INTERM_ERR && hay_tx){
    hay_tx = 0;
    estado = ESPERA_COMANDO;
}
else if(estado == INTERM_END_COM && hay_tx){
    hay_tx = 0;
    estado = INICIO;
    conecta_K_automata();
}
else if(estado == INTERM_JUG && hay_tx){
    hay_tx = 0;
    estado = ESPERA_CONFIRMACION;
}
}

```

```

else if(estado == INTERM_CONFIRM && hay_tx){
    hay_tx = 0;
    estado = ESPERA_COMANDO;
}
else if(estado == INTERM_CANC_JUG && hay_tx){
    hay_tx = 0;
    estado = ESPERA_COMANDO;
}
else if(estado == INTERM_END_CONF && hay_tx){
    hay_tx = 0;
    estado = INICIO;
    conecta_K_automata();
}
}

/**
 * @brief Visualiza el tablero de juego.
 *
 * Esta función genera una representación visual del tablero de juego y la devuelve como una
 * cadena de caracteres.
 * @return La representación visual del tablero de juego.
 */
char* conecta_K_visualizar_tablero(void) {
    uint8_t f, c;
    char caracter;
    int fila_ancho = (NUM_COLUMNAS+1)*2+1;
    for (f = 0; f < NUM_FILAS + 1; ++f) {
        for (c = 0; c < NUM_COLUMNAS + 1; ++c) {
            if (f == 0) {
                if (c == 0) caracter = '-';
                else caracter = c + 48;
            } else if (c == 0) caracter = f + 48;
            else {
                uint8_t contenido_celda=tablero_leer_celda(&cuadricula,
                                                            f-1, c-1);

                if (contenido_celda == 0x04) caracter = '#';
                else if (contenido_celda == 0x05) caracter = 'B';
                else if (contenido_celda == 0x06) caracter = 'N'; // 4e
                else caracter = ' '; // 20
            }
        }
    }
}

```

```

        tablero[f*fila_ancho + c*2] = caracter;
        tablero[f*fila_ancho + c*2 +1] = '|';
    }
    tablero[f*fila_ancho + c*2] = '\n';
}
return tablero;
}

/**
 * @brief Realiza una jugada en el juego.
 *
 * Esta función realiza una jugada en el juego, insertando una ficha en la posición
 * especificada. Si la jugada es válida, se inserta la ficha y se comprueba si hay una línea de
 * cuatro fichas del mismo color. Si hay ganador, "ganador"=1. Si no hay ganador, "ganador"=0.
 *
 * @param row La fila en la que se desea realizar la jugada.
 * @param column La columna en la que se desea realizar la jugada.
 * @param colour El color de la ficha a insertar.
 */
void conecta_K_jugada(uint8_t row, uint8_t column, uint8_t colour){
    if(tablero_fila_valida(row)&&tablero_columna_valida(column)
        && tablero_color_valido(colour)){
        //podríamos no validarla ya que tablero_insertar_valor vuelve a
        validar
        if (celda_vacia(tablero_leer_celda(&cuadricula, row, column))){
            //tablero_insertar tambien chequea si esta libre esa
            posicion_ocupada = 0;
            if(tablero_insertar_color(&cuadricula, row, column,
                                    colour) == EXITO) {
                tiempo_parc_hay_linea = temporizador_drv_leer();
                llamadas_hay_linea++;
                ganador = conecta_K_hay_linea_arm_arm(&cuadricula, row,
                                                        column, colour);
                tiempo_total_hay_linea += temporizador_drv_leer() -
                                        tiempo_parc_hay_linea;
            } else {
                while(1);    // no ha insertado bien
            }
        } else {
            posicion_ocupada = 1;    //celda no vacia

```

```

        }
    } else {
        // fuera de rango fila, columna o color
        while(1);
    }
}

void crear_msjFin(char* dest, const char* parametro) {
    int i, numEventos;
    tiempo_total_procesador = temporizador_drv_leer() - tiempo_parc_procesador;
    strcat(dest, "FIN DE LA PARTIDA: El jugador ");
    itoa(turno, buff_itoa);
    strcat(dest, buff_itoa);
    if (strcmp(parametro, "boton2") == 0) {
        strcat(dest, " ha pulsado el boton 2.\n");
    } else if (strcmp(parametro, "end") == 0) {
        strcat(dest, " ha introducido el comando END.\n");
    } else if (strcmp(parametro, "ganador") == 0){
        strcat(dest, " ha ganado.\n");
    } else {
        strcat(dest, "-----ERROR-----");
    }

    strcat(dest, "Tiempo de uso del procesador:\nTotal: ");
    itoa(tiempo_total_procesador/1000000, buff_itoa);    // pasamos a segundos
    strcat(dest, buff_itoa);
    strcat(dest, " s\n\nTiempo de computo de conecta_K_hay_linea:\nTotal: ");
    itoa(tiempo_total_hay_linea, buff_itoa);
    strcat(dest, buff_itoa);
    strcat(dest, " us\nMedia: ");
    itoa(tiempo_total_hay_linea/llamadas_hay_linea, buff_itoa);
    strcat(dest, buff_itoa);
    strcat(dest, " us\n\nTiempo que al humano le cuesta pensar la
        jugada:\nTotal: ");
    itoa(tiempo_total_humano/1000, buff_itoa);
    strcat(dest, buff_itoa);
    strcat(dest, " ms\nMedia: ");
    itoa((tiempo_total_humano/num_jugadas)/1000, buff_itoa);
    strcat(dest, buff_itoa);
    strcat(dest, " ms\n\nEventos esta partida: \n");
    for (i = 0; i<NUM_TIPO_EVENTOS; i++) {

```

```

        strcat(dest, eventoToString((EVENTO_T)i));
        strcat(dest, ": ");
        numEventos = FIFO_estadisticas((EVENTO_T)i);
        itoa(numEventos - estadisticas_fifo[i], buff_itoa);
        estadisticas_fifo[i] = numEventos;
        strcat(dest, buff_itoa);
        strcat(dest, "\n");
    }
}

void conecta_K_test_cargar_tablero(TABLER0 *t) {
    size_t f, c;
    #include "tablero_test.h"
    for(f = 0; f < NUM_FILAS; ++f){
        for(c = 0; c < NUM_COLUMNAS; ++c) {
            tablero_insertar_color (t, f, c, tablero_test[f][c]);
        }
    }
}

void sel_comando(uint32_t comando) {
    fila_jugada = 0;
    columna_jugada = 0;
    if (atoi("NEW") == comando)
        comando_data = NEW;
    else if (atoi("END") == comando)
        comando_data = END;
    else {
        itoa(comando, buff_itoa);
        buff_itoa[1] = 0; // torpeza. cambiar?
        fila_jugada = atoi(&buff_itoa[0]);
        columna_jugada = atoi(&buff_itoa[2]);
        if (fila_jugada > 0 && fila_jugada <= 7 &&
            columna_jugada > 0 && columna_jugada <= 7) {
            comando_data = JUG;
        }
        else {
            comando_data = ERR;
        }
    }
}
}

```



```

char* eventoToString(EVENTO_T evento) {
    switch (evento) {
        case VOID: return "TOTAL EVENTOS";
        case TIMER0: return "TIMER0";
        case TIMER1: return "TIMER1";
        case ALARMA_OVERFLOW: return "ALARMA_OVERFLOW";
        case ALARMA_TRATAR: return "ALARMA_TRATAR";
        case PULSACION_BOTON: return "PULSACION_BOTON";
        case COMPROBAR_BOTON: return "COMPROBAR_BOTON";
        case USUARIO_AUSENTE: return "USUARIO_AUSENTE";
        case ev_VISUALIZAR_CUENTA: return "ev_VISUALIZAR_CUENTA";
        case ev_LATIDO: return "ev_LATIDO";
        case ev_VISUALIZAR_HELLO: return "ev_VISUALIZAR_HELLO";
        case ev_TX_SERIE: return "ev_TX_SERIE";
        case ev_RX_SERIE: return "ev_RX_SERIE";
        case CONTINUA_ENVIO: return "CONTINUA_ENVIO";
        default: return "ERROR";
    }
}

char* comandoToString(enum COMANDO com) {
    switch (com) {
        case NOC: return "NOC";
        case END: return "END";
        case NEW: return "NEW";
        case ERR: return "ERR";
        case JUG: return "JUG";
        default: return "ERR";
    }
}

```