

PRÁCTICA 2

PROYECTO HARDWARE



19 de diciembre de 2021

álvaro fraidias y david ros

780336 y

# Resumen ejecutivo

En el siguiente documente se explica con detenimiento la implementación del juego sudoku en la placa Embest S3CEV40.

Para implementar el juego primeramente se deben configurar ciertas funcionalidades para el correcto funcionamiento de la placa. Estas funcionalidades son:

* Un temporizador para controlar los rebotes de los pulsadores.
* Un temporizador para medir los tiempos de ejecución de la placa.
* Un temporizador para controlar un led que hará de latido.
* Dos botones que se usarán para implementar las jugadas del juego sudoku.
* Un 8 led que mostrará la información del juego sudoku.
* Una cola de depuración que guarda los eventos que se producen en la placa.

Después de configurar estas funcionalidades e implementar el sudoku se miden los tiempos de las diferentes implementaciones hechas en la práctica 1 sobre los métodos para actualizar el tablero.

Como conclusiones podemos decir que la implementación del juego sudoku en la placa Embest S3CEV40 es mucho peor ya que el procesador de la placa tiene unas prestaciones menores que el procesador de un PC actual.

Contenido

[Resumen ejecutivo 1](#_Toc90819635)

[1. Introducción 4](#_Toc90819636)

[2. Objetivos 4](#_Toc90819637)

[3. Metodología 4](#_Toc90819638)

[4. Temporizadores 4](#_Toc90819639)

[4.1 Temporizador rebotes pulsador 6](#_Toc90819640)

[4.1.1 timer0\_init 8](#_Toc90819641)

[4.1.2 timer0\_trp 8](#_Toc90819642)

[4.1.3 timer0\_boton\_mantenido 9](#_Toc90819643)

[4.1.4 timer0\_trd 9](#_Toc90819644)

[4.1.5 timer0\_stop 9](#_Toc90819645)

[4.1.6 timer0\_ISR 10](#_Toc90819646)

[4.2 Biblioteca de medidas de tiempo 10](#_Toc90819647)

[4.2.1 timer1\_nit 10](#_Toc90819648)

[4.2.2 timer1\_start 11](#_Toc90819649)

[4.2.3 timer1\_count 12](#_Toc90819650)

[4.2.4 timer1\_stop 13](#_Toc90819651)

[4.2.5 timer1\_ISR 13](#_Toc90819652)

[4.3 Temporizador latido placa 14](#_Toc90819653)

[4.3.1 timer2\_init 15](#_Toc90819654)

[4.3.2 timer2\_ISR 15](#_Toc90819655)

[5. Cola de depuración 16](#_Toc90819656)

[5.1 Localización y tamaño en memoria 16](#_Toc90819657)

[5.2 Iniciar\_cola\_eventos 17](#_Toc90819658)

[5.3 Cola\_guardar\_eventos 17](#_Toc90819659)

[6. Botones 18](#_Toc90819660)

[7. Juego sudoku 20](#_Toc90819661)

[7.1 Sudoku.c 20](#_Toc90819662)

[8. Estructura sudoku 23](#_Toc90819663)

[8.1 Zona de juego 23](#_Toc90819664)

[8.2 Métodos para modificar la zona de juego 23](#_Toc90819665)

[8.2.1 sudoku\_2021.c 24](#_Toc90819666)

[8.3 Métodos para modificar las celdas del tablero 24](#_Toc90819667)

[8.3.1 celda\_es\_pista 24](#_Toc90819668)

[8.3.2 celda\_comprobar\_candidato 25](#_Toc90819669)

[9. Tiempos 25](#_Toc90819670)

[10. Problemas durante la práctica 26](#_Toc90819671)

[11. Conclusiones 28](#_Toc90819672)

[12. Bibliografía 29](#_Toc90819673)



# 1. Introducción

Para abarcar lo mejor posible la explicación de la práctica 2 de la asignatura de proyecto hardware del grado de ingeniería informática hemos definido previamente los objetivos y la metodología empleada. Acto seguido hemos ido comentando como hemos logrado la implementación de cada funcionalidad necesaria para el correcto funcionamiento de la placa.

# 2. Objetivos

* Entender el funcionamiento de la placa Embest S3CEV40.
* Interactuar con una placa real y ser capaces de ejecutar y depurar en ella el código desarrollado en la práctica anterior.
* Gestionar la entrada/salida con dispositivos básicos.
* Aprender a desarrollar en C las rutinas de tratamiento de interrupción.
* Aprender a utilizar periféricos, como los temporizadores internos de la placa y los botones.

# 3. Metodología

Para realizar correctamente la práctica hemos divido las funcionalidades en 5 pasos. Dichos pasos son los títulos de los siguientes apartados donde se explican con mayor detenimiento.

# 4. Temporizadores

Los temporizadores son los elementos de un sistema que son prácticamente necesarios. En esta práctica hemos utilizado 3 temporizadores (a partir de ahora llamados “timers”) para realizar las siguientes funcionalidades:

* **Timer 0** = Controlar los rebotes al pulsar y soltar un botón.
* **Timer 1** = Medir tiempos de ejecución de la placa (necesarios para la cola de depuración)
* **Timer 2** = Controlar un latido led para comprobar que el programa se ejecuta correctamente a lo largo del tiempo.

La placa S3C44B0X tiene 6 temporizadores de 16 bits. Cada uno puede ser configurado para operar por interrupciones o por DMA.

Son contadores descendentes que pueden ser inicializados con un determinado valor. En cada ciclo de reloj interno se decrementan. Cuando llegan al valor de comparación generan una interrupción que debemos tratar.

A la hora de inicializar los timers debemos tener en cuenta las siguientes tablas donde se muestran los registros que debemos configurar y sus direcciones en memoria:

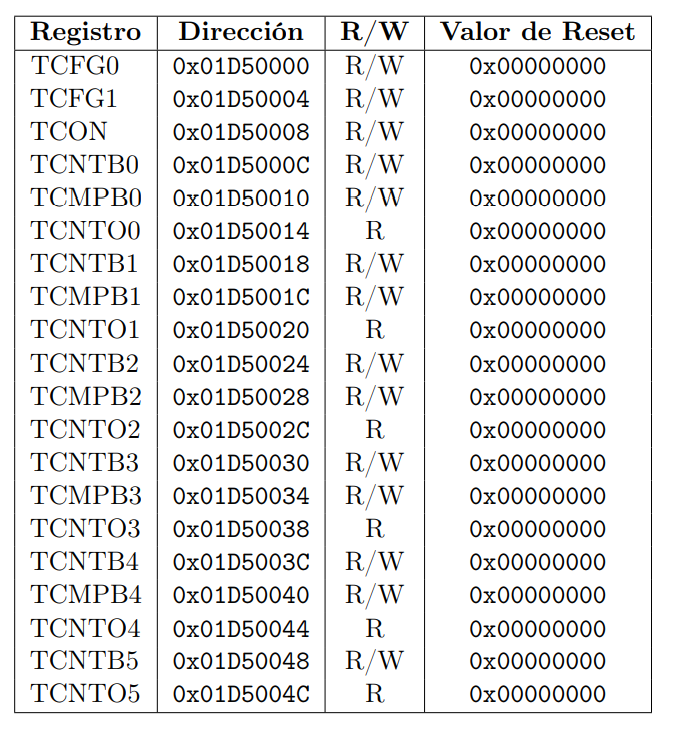


Ilustración 1: Registros para la utilización de los temporizadores

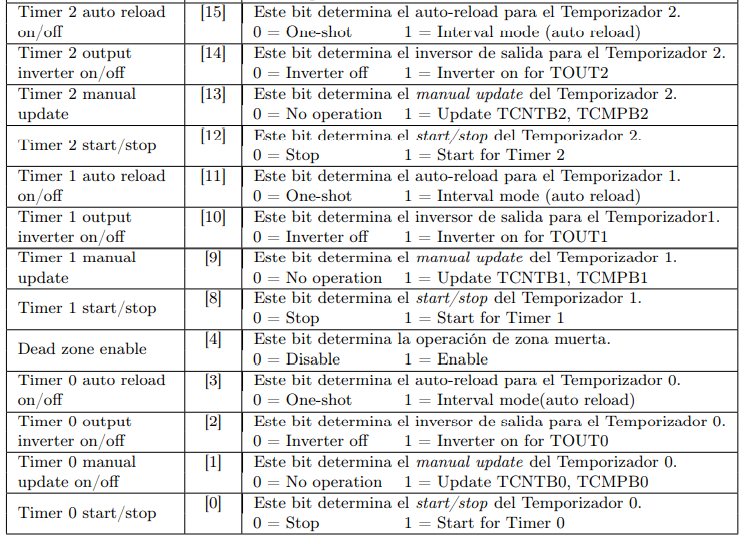


Ilustración : Registro TCON

* **TCFG0**: En este registro se ajusta el pre-escalado.

**TCFG1**: En este registro se almacena el valor del divisor (puede ser 2,4,8,16 o 32).

En nuestro caso hemos usado el 0000 (entrada del multiplexor 0 para obtener un divisor de ½ como se muestra en la siguiente ilustración):

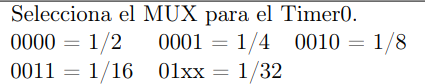


Ilustración 3: Descripción del registro TCFG1

* **TCNTBn**: Se almacena el valor de recarga (si el timer está en modo “auto-reload”). “Auto-reload” significa que cuando la cuenta del timer llega al valor del comparador, se vuelve a cargar el valor inicial para que vuelva a ejecutarse. Este valor se carga en TCNTn cuando el timer llega al valor de comparación.

Para que el valor que escribimos en TCNTBn pase directamente a TCNTn debemos activar el “manual-update” bit.

* **TCMPBn**: Guarda un valor para comparar y comprobar si se ha llegado al final.
* **TCON**: En este registro se realiza el control de los temporizadores (puesta en marcha, parada, actualización, modo auto reload, etc).

Para saber exactamente cómo funcionan debemos entender que cada par de temporizadores (0-1, 2-3, 4-5) comparten un módulo de pre-escalado y un divisor de frecuencia.

## 4.1 Temporizador rebotes pulsador

Al trabajar por primera vez con una placa real física vimos que los componentes no funcionan perfectamente y tienen ciertos problemas que debemos tratar. Uno de estos problemas son los rebotes que tienen los botones al ser pulsados y depulsados.

Un ejemplo de esto se puede ver en la siguiente ilustración:

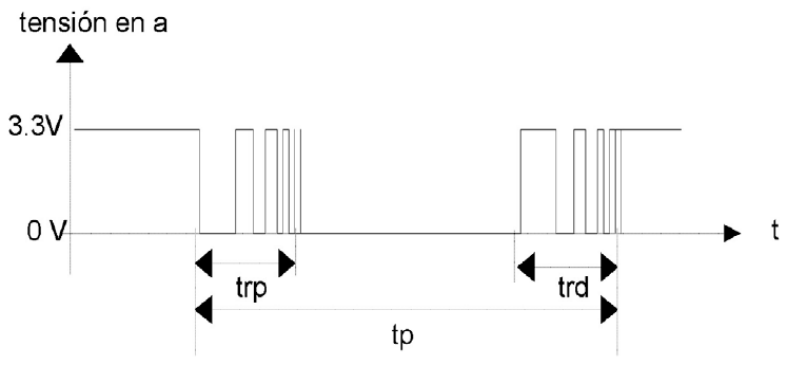


Ilustración 4: Rebotes al pulsar un botón.

Para que funcionen correctamente debemos implementar un temporizador para que establezca dos valores necesarios para evitar los rebotes:

* TRP: Tiempo necesario para ignorar los rebotes al pulsar un botón.
* TRD: Tiempo necesario para ignorar los rebotes al soltar un botón.

Para calcular estos tiempos hemos realizado pruebas donde pulsábamos un botón y veíamos en la cola de depuración los rebotes para calcular aproximadamente el tiempo necesario para que no surjan. Un ejemplo de estas pruebas se ve en la siguiente ilustración:

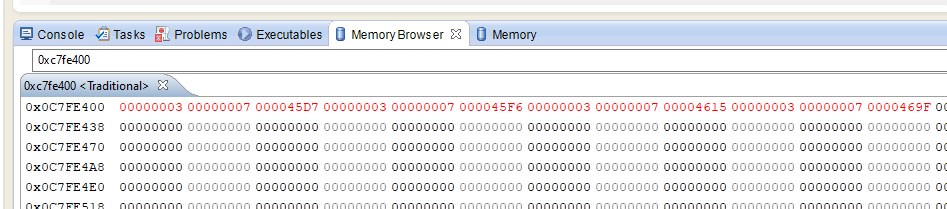


Ilustración 5: Visualización de los rebotes en la memoria

Con estas pruebas y el siguiente texto en el documento “Sistema de memoria y de entrada/salida en la placa S3CEV40” (enlace 1º de la bibliografía), en la página 13 comprobamos que el tiempo necesario para trp y trd debería ser de 100ms:

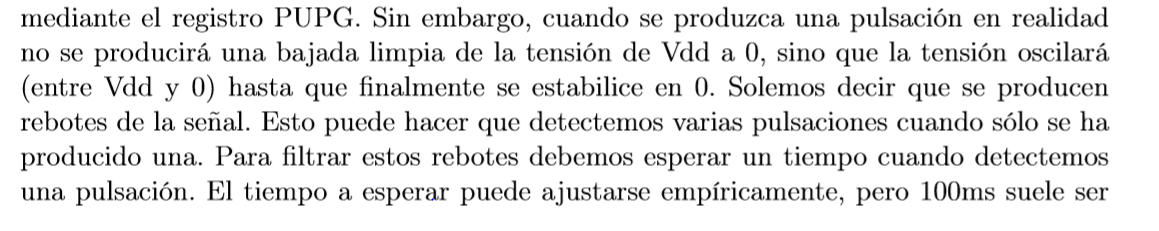


Ilustración 6: Página 13 Sistema de memoria y de entrada/salida en la placa S3CEV40

Por consiguiente, el valor inicial del timer debería ser el resultado de la siguiente operación:

Frecuencia del reloj\* tiempo trp = 64.000 \* 0,1 = 6400

Para calcular el tiempo de “timer0\_boton\_mantenido()” hacemos la misma operación, pero con 50ms:

Frecuencia del reloj\* tiempo trp = 64.000 \* 0,05 = 3200

### 4.1.1 timer0\_init

A diferencia del timer 2 (se explicará posteriormente), en este método no se le da valores al registro “TCNTB0” ya que este irá tomando valores según se ejecuten los métodos timer0\_trp, timer0\_mantenido o timer0\_trd.

En este caso el preescalado (registro TCFG0) será de 255 para que no nos salga una frecuencia muy elevada. El divisor será 00 y el comparador 0.

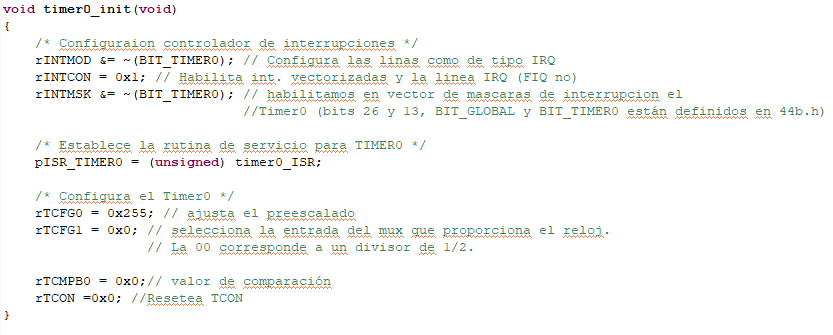


Ilustración 7: Método timer0\_init()

### 4.1.2 timer0\_trp

Este método como hemos mencionado anteriormente será el encargado de eliminar los rebotes al pulsar un botón. La funcionalidad es muy simple: le da el valor inicial de 100ms al registro TCNTB0, activa el bit con la puerta lógica OR “manual-update” del registro TCON y después desactiva dicho bit a la vez que activa el bit “start-stop” con la puerta lógica XOR.

En este caso no activamos el bit “auto-reload” ya que no queremos que se reinicie cuando se acabe.

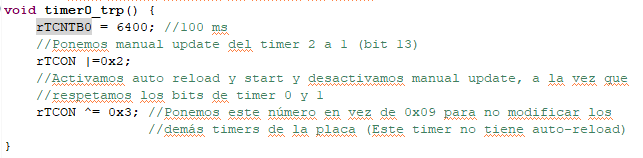


Ilustración 8: Método timer0\_trp()

### 4.1.3 timer0\_boton\_mantenido

Este método hace prácticamente lo mismo que el método anterior salvo por dos diferencias:

* Carga en el registro TCNTB0 el valor equivalente a 50ms.
* Activa el bit “auto-reload” ya que en este caso si que queremos que el timer se reinicie cuando acabe.

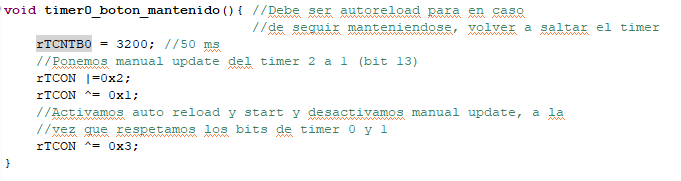


Ilustración 9: Método timer0\_boton\_mantenido

### 4.1.4 timer0\_trd

Este método hace exactamente lo mismo que en el método timer0\_trp.

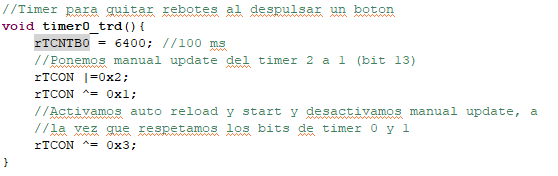


Ilustración 10: Método timer0\_trd

### 4.1.5 timer0\_stop

Este método se encarga de deshabilitar el timer.

En la práctica no haría falta ya que el último método que ejecuta el timer es el método “timer0\_trd” donde se estipula que el bit “auto-reload” esté desactivado por lo que se finalizará cuando llegue al final. Aun así lo hemos hecho para desactivar el bit “star-stop”.

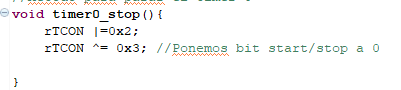


Ilustración 11: Método timer0\_stop()

### 4.1.6 timer0\_ISR

Este método se ejecuta cada vez que el timer genera una interrupción en el procesador. Este método simplemente llama al método “maquina\_estados\_rebotes()” para que trate los rebotes:

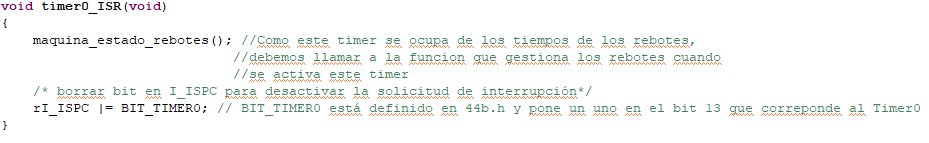


Ilustración 12: Método timer0\_ISR()

## 4.2 Biblioteca de medidas de tiempo

Hemos necesitado implementar una biblioteca de medidas de tiempo para poder calcular el tiempo de ciertas funcionalidades de la práctica (medir tiempos de ejecución de la placa o para saber cuándo se ha producido una interrupción que se guardará en la cola de depuración…). Para implementar dichas funcionalidades hemos usado el timer 1. Este timer debe ser lo más preciso posible e interrumpir lo menos posible al procesador. Para implementar estas medidas de tiempo hemos realizado los siguientes métodos:

### 4.2.1 timer1\_nit

Para hacer este timer lo más preciso posible hemos descubierto que la frecuencia máxima a la que puede trabajar un temporizador es 32MHz ya que la frecuencia máxima de la placa es 64MHz.

Para ajustar el nivel de frecuencia debemos prestar atención a la ecuación que define este nivel:

Frecuencia de cada tick (cada escalón) =

Por consiguiente, debemos establecer un nivel de pre-escalado (0) y un divisor (1/2) mínimo.

Para que este temporizador interrumpa lo menos posible al procesador debemos poner un valor inicial elevado. Para ello hemos usado el valor por defecto que tenía el timer proporcionado por el profesor: “65535”.

En este método, como se puede ver en la siguiente ilustración, simplemente configuramos los registros necesarios para que el timer acepte interrupciones IRQ a la vez que le damos los valores iniciales:

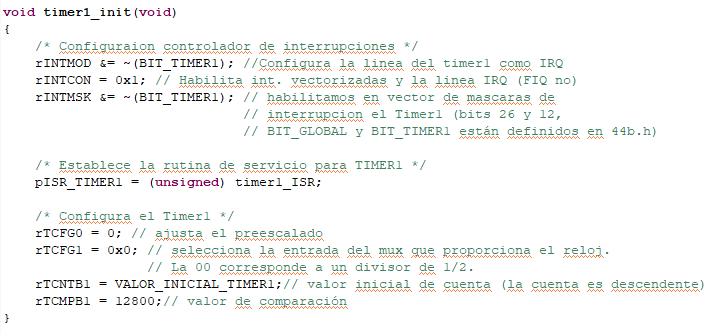


Ilustración 13: Método timer1\_init

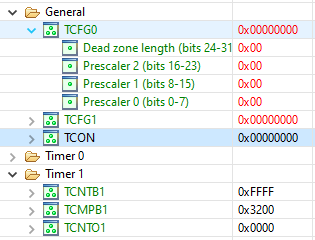


Ilustración 14: Registros timer1 después de ejecutar timer1\_init

### 4.2.2 timer1\_start

En este método debemos activar el timer. Para ello le damos el valor inicial al registro “TCNT01” y editamos el registro TCON dos veces:

* La primera vez debemos activar el bit “manual-update”. Para ello ejecutamos sobre este registro una operación lógica OR ya que así no afectará a ningún otro bit.
* La segunda vez activamos los bits “auto-reload” y “start-stop” a la vez que eliminamos el bit “manual-update”. En este caso usamos una operación lógica XOR porque, como se muestra a continuación, la XOR solo afectaría en caso de poner un 1:

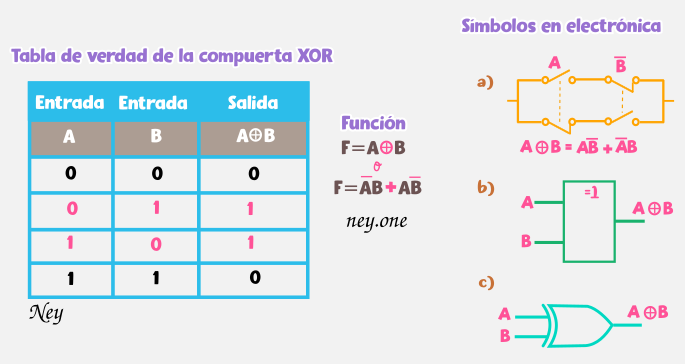


Ilustración 15: Tabla de verdad XOR

El método explicado anteriormente es el siguiente:

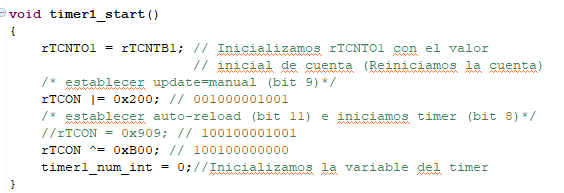


Ilustración 16: Método timer1\_start()

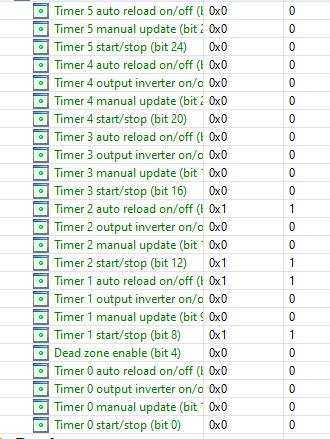


Ilustración 17: Registro TCON después de usar timer1\_start()

### 4.2.3 timer1\_count

Este método se encarga de contar el tiempo desde que se activó (desde que se ejecutó timer1\_start()).

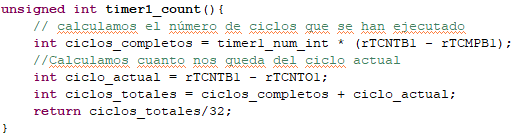


Ilustración 18: Método timer1\_count()

El funcionamiento es simple: como se mostrará en el punto 4.2.5 cada vez que salta la interrupción del timer se incrementa la variable “timer1\_num\_int” por lo que, al ejecutar timer1\_count() debemos tener en cuenta todas las veces que se ha ejecutado este timer y guardarlo en la variable “ciclos\_completos”. También debemos tener en cuenta cuanto nos queda para acabar el ciclo actual. Para ello guardamos en la variable “ciclo\_actual” la diferencia entre el valor inicial del timer y el valor de comparación. Una vez tenemos esos dos valores los sumamos y los dividimos entre 32 (frecuencia) para tener una mejor precisión.

### 4.2.4 timer1\_stop

Este método se encarga de realizar dos funciones:

* Parar el timer
* Devolver el tiempo desde que se llamó a timer1\_start()

Para parar el tiempo hacemos lo contrario que en el caso timer1\_start(): Primero activamos con una puerta lógica OR el bit “manual-update” y después desactivamos con una puerta lógica XOR los bits “manual-update”, “start-stop” y “auto-reload”.

Para devolver el tiempo creamos una variable auxiliar y llamamos a timer1\_count(). Después devolvemos esta variable. El método es el siguiente:

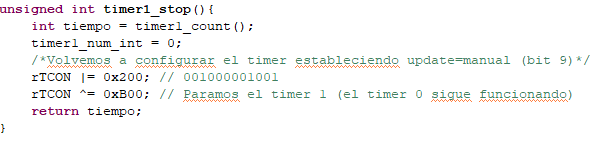


Ilustración 19: Método timer1\_stop()

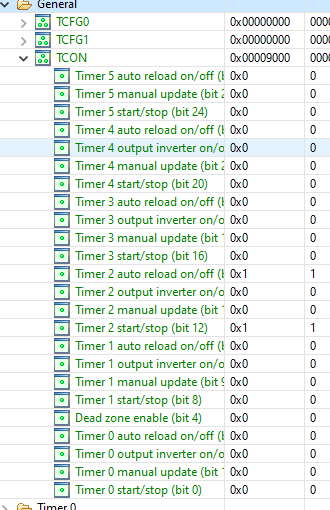
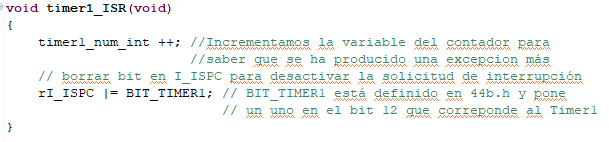


Ilustración 20: Registro TCON después de ejecutar timer1\_stop

### 4.2.5 timer1\_ISR

Este método se ejecuta cuando el timer 1 llega al valor de comparación y produce una interrupción en el procesador. Este método simplemente aumenta en 1 la variable “timer1\_num\_int”.

El código es el siguiente:



Para comprobar que la implementación de este temporizador funciona correctamente hemos realizado unas pruebas en el main donde guardamos en la cola de depuración una lectura antes y después de ejecutar un delay de 1ms, 10ms y 1s:

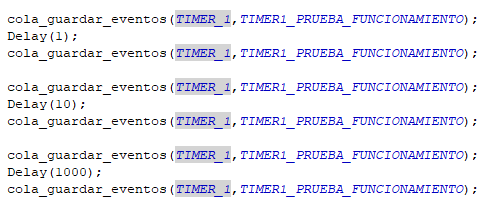


Ilustración 21: Pruebas timer 1

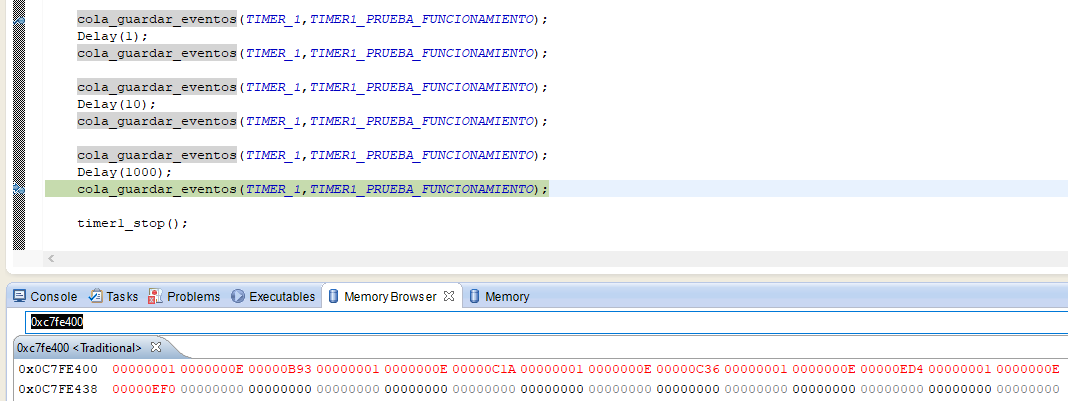


Ilustración 22: Cola de depuración después de ejecutar pruebas timer 1

## 4.3 Temporizador latido placa

Para implementar el latido de la placa debemos implementar un 3º timer (timer 2) que genere 60 eventos por minuto, haciendo que el led de la derecha parpadee a 4Hz.

Para calcular el valor inicial del timer esto hemos utilizado las siguientes funciones:

Frecuencia de cada tick (cada escalón) = = = 125000

Tiempo de cada tick (cada escalón) = = = 0,000008

Interrupciones por segundo = Tiempo de cada tick \* Valor inicial timer

Valor inicial timer = = = 125000

Una vez calculado el valor inicial del contador nos disponemos a implementar los métodos para que funcione correctamente.

Este timer solo tiene dos métodos:

* Timer2\_init()
* Timer2\_ISR()

### 4.3.1 timer2\_init

Se encarga de dar los valores iniciales al timer a la vez que activarlo:

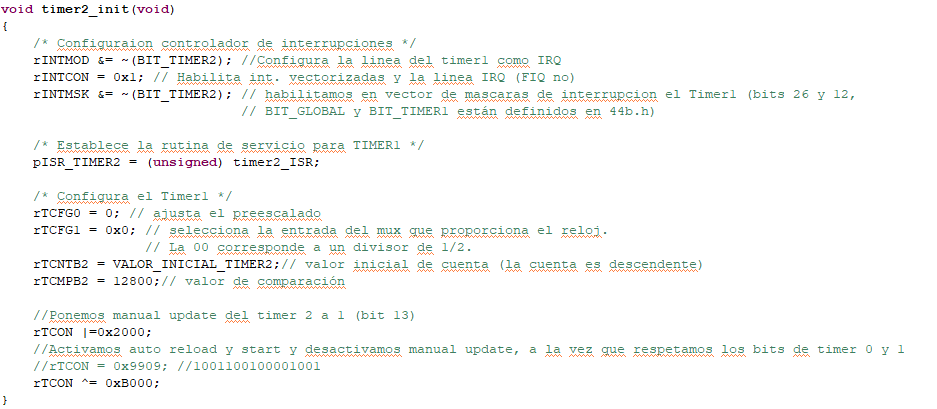


Ilustración : Método timer2\_init()

### 4.3.2 timer2\_ISR

Cuando el timer 2 interrumpe al procesador ejecuta este método. En este método simplemente modifica la variable “variable\_latido” para que el programa principal (main) pueda comprobar una condición y acceder a la parte del código que ejecuta el latido:

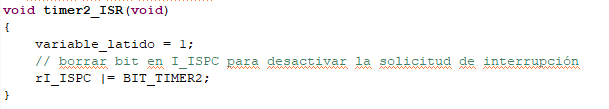


Ilustración : Método timer2\_ISR()

El código que se ejecuta en el programa principal es el siguiente:

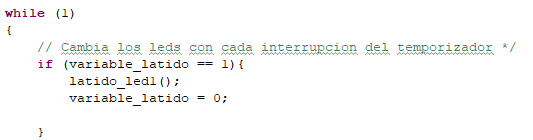


Ilustración : Latido programa principal (main.c)

En este código, el programa comprueba infinitamente si se ha cambiado “variable\_latido”. En caso de que se haya cambiado, cambia el estado del led derecho (más adelante se verá el código que implementa esta funcionalidad).

# 5. Cola de depuración

Como vamos a tener varias funciones concurrentes en nuestro programa debemos crear una cola de depuración donde almacenar cada interrupción para simplificar la labor de depurar el código en caso de que sea necesario. Esta pila recogerá información relevante acerca de cada interrupción.

De esta forma, en cualquier momento se puede parar la ejecución y, observando la cola, poder determinar el orden de activación de las interrupciones.

Para saber exactamente que interrupción se ha ejecutado hemos creado una clase llamada “eventos.h” donde declarar los diferentes tipos de interrupciones:

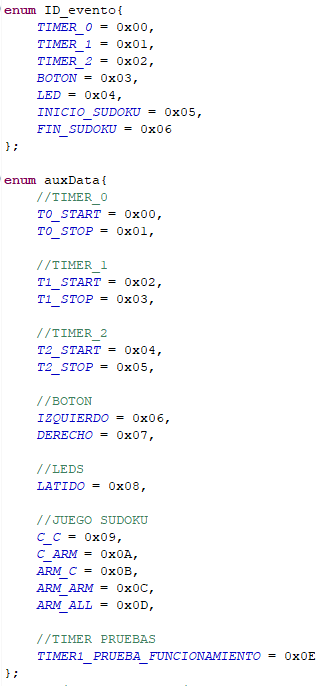


Ilustración 26: Clase eventos.h

## 5.1 Localización y tamaño en memoria

Navegando entre los ficheros proporcionados por el profesor nos dimos cuenta de que en el archivo “44binit.asm” aparecían las zonas de memoria reservadas para las pilas:

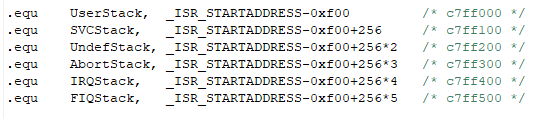


Ilustración : Espacio reservado para pilas en el fichero 44binit.asm.

Es por esto que decidimos crear nuestra propia cola en dichas zonas de memoria. En cuanto al tamaño elegimos un tamaño de 768 bytes (256 \* 3) ya que debemos guardar eventos de 3 en 3 (más adelante hablaremos con detenimiento sobre esto). El inicio de la cola y el tamaño se definen en la siguiente ilustración:



Ilustración : Dirección y tamaño de la cola de depuración

## 5.2 Iniciar\_cola\_eventos

Para iniciar la cola de eventos lo que hemos hecho es, mediante un bucle, recorrer toda la cola e iniciar los valores a 0:

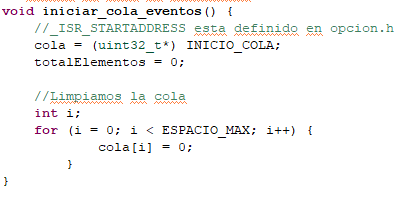


Ilustración 29: Método iniciar\_cola\_eventos()

## 5.3 Cola\_guardar\_eventos

Lo primero que debemos hacer a la hora de guardar un evento en la cola de depuración es calcular el tiempo en el que se ha producido. Para eso llamamos a timer1\_count() y guardamos el valor en la variable “tiempo\_evento” para introducirlo más adelante. También debemos hacer una especie de casteo con el ID\_evento ya que viene dado con un entero sin signo de 8 bits y debemos guardarlo como un entero sin signo de 32 bits por lo que debemos desplazarlo 24 bits con la operación “<<”.

Por último, como queremos implementar una cola circular debemos inicializar la cola al valor “INICIO\_COLA” y desde esa zona incrementar hasta “ESPACIO\_MAX”, donde se reiniciará. Por ello, siempre que deseemos introducir un evento debemos ver si estamos al final de la cola para, en caso de estar al final, reiniciar dicha cola.

El método es el siguiente:

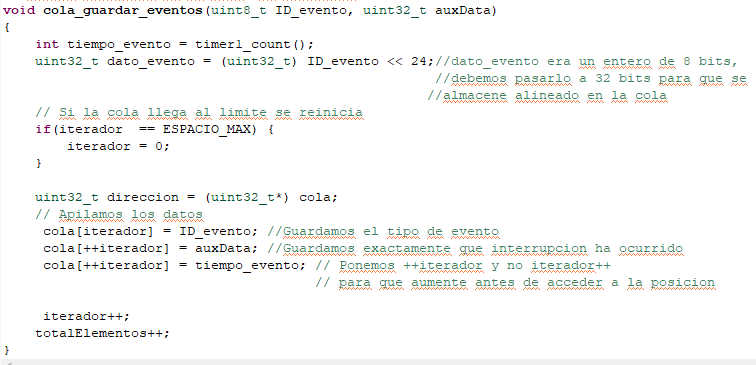


Ilustración 30: Método cola\_guardar\_eventos()

A la hora de guardar la información en la cola vimos que podíamos hacerlo programando en arm con la función “asm” desde C, pero vimos que con punteros era más fácil visualmente.

# 6. Botones

Al ejecutarse la interrupción de los botones, se bloquean las demás interrupciones, se activa timer0\_trp() y cuando este timer interrumpe al procesador, llama a la función “**maquina\_estado\_rebotes()**” que gestiona los rebotes. En esta máquina de estados se encuentran los siguientes estados:

* ESTADO\_INICIAL.
* ESTADO\_PULSADO.
* ESTADO\_FINAL.

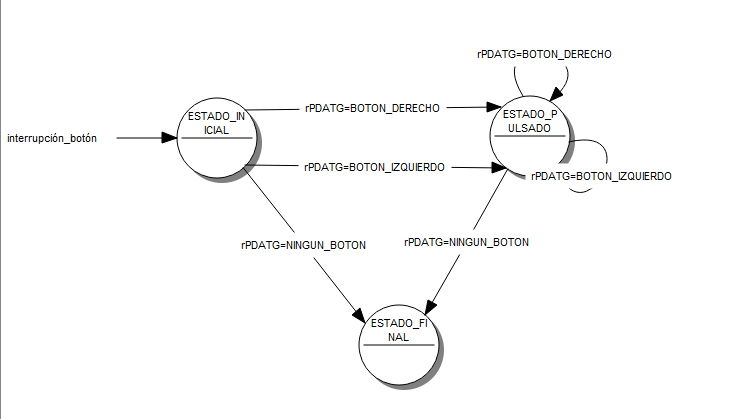
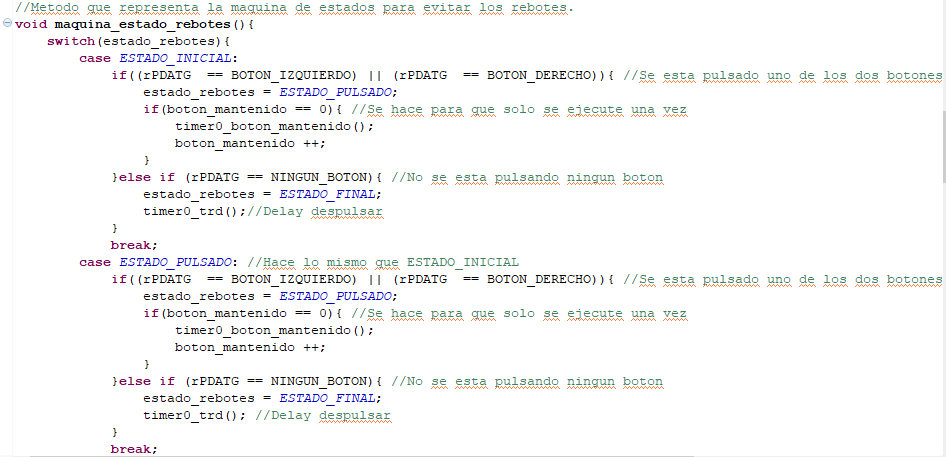
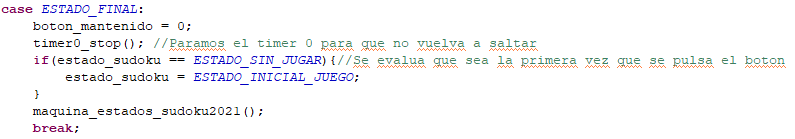


Ilustración : Máquina de estados rebotes

El código que implementa esta máquina de estados es el siguiente:





Para saber que botón se está pulsado actualmente se comprueba en el registro PDATG como se muestra en las siguientes ilustraciones:



Ilustración : NINGÚN BOTÓN PULSADO



Ilustración :IZQUIERDA



Ilustración : DERECHA

En el “**ESTADO\_INICIAL**” se comprueba a través del registro rPDATG si está pulsado el botón izquierdo o el botón derecho. Si es así, se establece la variable “estado\_rebotes” con el valor “**ESTADO\_PULSADO**” para que avance a un nuevo estado y además se utiliza la variable “botón\_mantenido” para poder activar “timer0\_boton\_mantenido” de forma que la primera vez que se compruebe esta variable se activará el timer y se incrementará el valor de “botón\_mantenido” para que en el siguiente estado no se pueda volver a activar “timer0\_boton\_pulsado”. Sin embargo, si no se está pulsando ningún botón, se establece el siguiente estado como “**ESTADO\_FINAL**” y se activa el delay de dejar de pulsar el botón.

En el “**ESTADO\_PULSADO**” se hace lo mismo que en el “**ESTADO\_INICIAL**”. Se comprueba primero si el botón derecho o izquierdo están pulsados, si es así se mantiene en el mismo estado y comprueba la variable “botón\_mantenido” para poder activar o no “timer0\_boton\_mantenido”. Si no se está pulsando ningún botón, la variable “estado\_rebotes” se establece a “**ESTADO\_FINAL**” y se activa el delay de dejar de pulsar el botón.

En este estado se pone a 0 la variable “botón\_mantenido” para que en la siguiente ejecución de la máquina se pueda volver a establecer el “timer0\_boton\_mantenido”. Después se para el timer 0 a través de “timer0\_stop”, se comprueba si aún no se está jugando al sudoku y se llama a la máquina de estados que gestiona el juego del sudoku.

# 7. Juego sudoku

## 7.1 Sudoku.c

La máquina de estados que se ha implementado para jugar a sudoku es la siguiente:

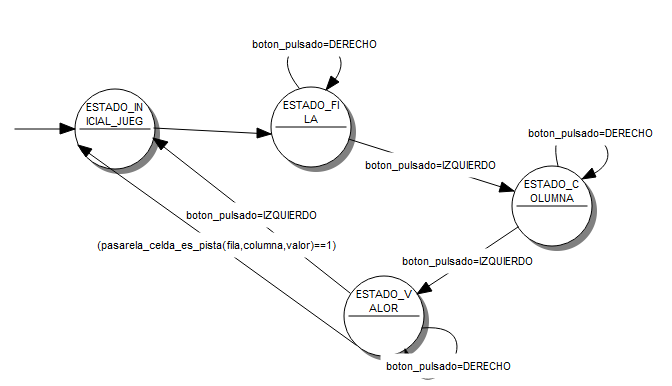


Ilustración : Máquina de estados juego sudoku

El código que implementa esta máquina de estados es el siguiente:

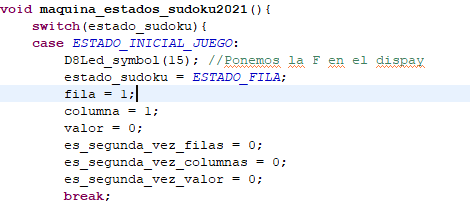


Ilustración : Estado inicial juego sudoku máquina de estados

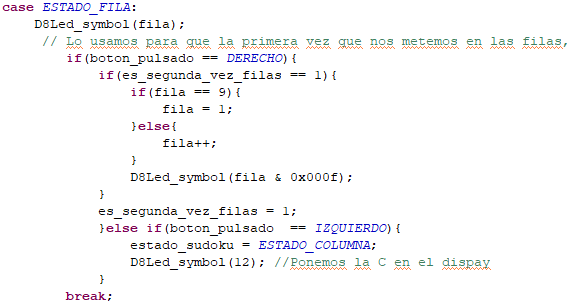


Ilustración : Estado fila juego sudoku máquina de estados

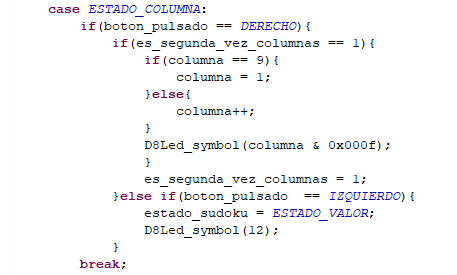


Ilustración : Estado columna juego sudoku máquina de estados

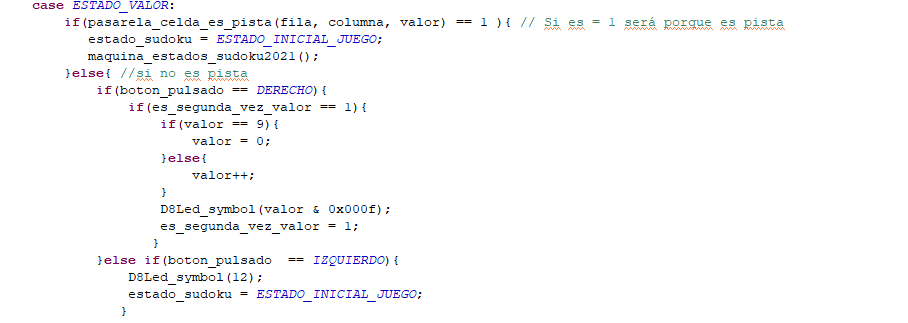


Ilustración : Estado valor juego sudoku máquina de estados (parte 1)

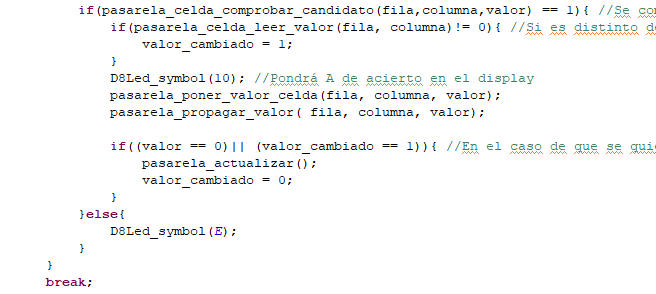


Ilustración : Estado valor juego sudoku máquina de estados (parte 2)

Los estados que se implementan son:

* **ESTADO\_INICIAL\_JUEGO**: en este estado se establecen los valores iniciales de las filas y las columnas, se establece el valor “F” en el display y se indica que el siguiente estado es “**ESTADO\_FILA**”.
* **ESTADO\_FILA**: cuando se pulse el botón derecho se comprobará la variable “es\_segunda\_vez\_filas” para que la primera vez que se entre, se pueda leer el 1 y no se incremente directamente a 2 (esta comprobación se lleva a cabo de igual forma tanto para las columnas como para el valor). Tras esto se incrementará el valor de la fila y se mostrará en el 8 led. Cuando el botón izquierdo sea pulsado se confirmará el valor, se mostrará una C en el 8led y se establecerá “**ESTADO\_COLUMNA**” como siguiente estado.
* **ESTADO\_COLUMNA**: en este caso se incrementará el valor de las columnas cada vez que se pulse el botón derecho. Al pulsar el botón izquierdo se confirmará el valor poniendo una C en el 8ledy se establecerá como siguiente estado “**ESTADO\_VALOR**”.
* **ESTADO\_VALOR**: se comprueba si la celda es una pista. En caso de que lo sea significa que esa celda no puede ser actualizada con un nuevo valor, por lo tanto, se tiene que volver a pedir una fila y una columna nueva estableciendo como siguiente estado el “**ESTADO\_INICIAL\_JUEGO**”. Si la celda no es pista, se pide al usuario introducir un valor, de forma que si se pulsa el botón derecho se incrementará el valor empezando desde el 0 y se mostrará por el 8led. Cuando se pulse el botón izquierdo se confirmará el valor poniendo una C en el 8led y se establecerá el siguiente estado como “**ESTADO\_INICIAL\_JUEGO**”. Tras esto se comprueba si el nuevo valor está dentro de los candidatos de esa celda. Después se llama a la función “pasarela\_poner\_valor\_celda” para que se establezca el valor y después se llama a la función “pasarela\_propagar\_valor” para propagar ese valor en la lista de candidatos. Además, se comprueba si el valor es 0 o si ya había un valor anteriormente y se llama a la función “pasarela\_actualizar” para volver a recalcular las listas de candidatos.

# 8. Estructura sudoku

Para poder jugar correctamente al sudoku debemos tener una estructura donde esté presente:

* La zona de juego (en este caso es un tablero).
* Métodos para actualizar y propagar el tablero.
* Métodos para modificar cada celda del tablero.

Para implementar cada una de las 3 funcionalidades descriptas hemos usado varias clases que se explicarán a continuación.

## 8.1 Zona de juego

Usando como ejemplo la práctica anterior, hemos reusado la clase “tableros.h” para almacenar el tablero del sudoku. Como en este caso solo necesitábamos un tablero hemos eliminado los demás para ahorrar memoria:

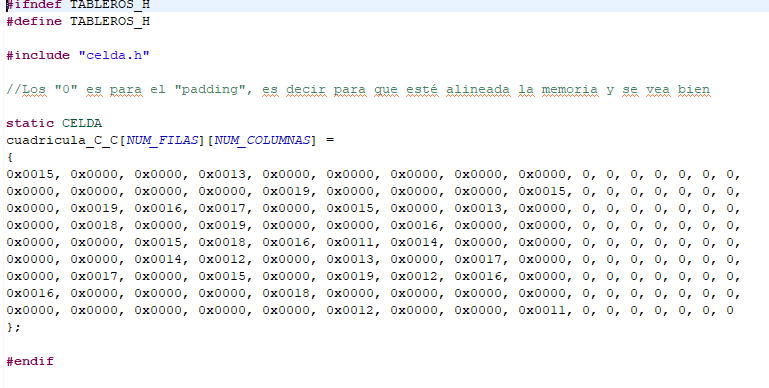


Ilustración : Clase tableros.h

Esta clase será llamada desde “sudoku\_2021.c” para instanciar y usar dicho tablero.

## 8.2 Métodos para modificar la zona de juego

En el enunciado esta descrito que debemos usar los métodos de actualizar más óptimos por lo que usamos el método “actualizar\_c” (implementado en “sudoku\_2021.c”) y “propagar\_arm” (implementado en “propagar.asm”) porque fueron los métodos más óptimos en la práctica anterior como se puede ver en la siguiente ilustración:

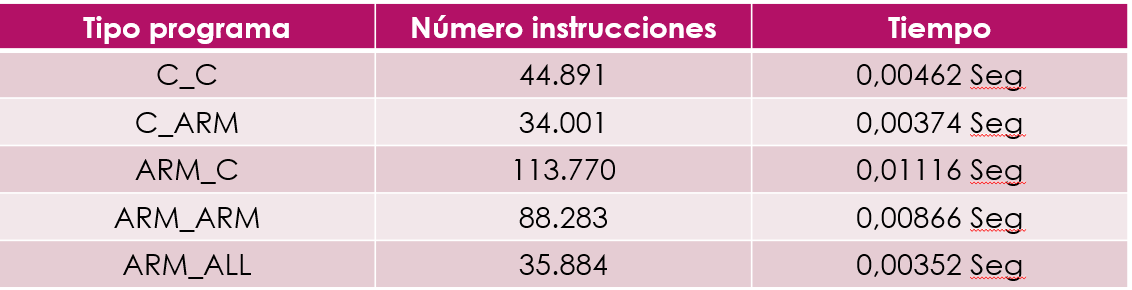


Ilustración : Tiempos práctica 1

Como no tiene sentido volver a explicar dichos métodos ya que están explicados minuciosamente en la memoria de la práctica 1, vamos a ver las diferencias de la clase “sudoku\_2021.c”.

### 8.2.1 sudoku\_2021.c

Al implementar esta clase hubo un gran problema y es que debe ser llamada desde la clase “maquinas\_estados.c”, en dicha clase no existe el tablero por lo que debemos crear pasarelas entre la clase “sudoku\_2021.c” y “maquina\_estados.c”.

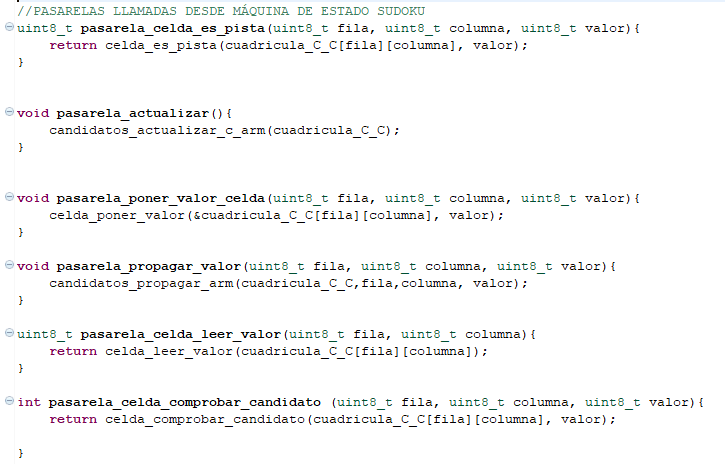


Ilustración 43: Métodos pasarelas en sudoku\_2021.c

## 8.3 Métodos para modificar las celdas del tablero

Manteniendo la estructura de la práctica 1, hemos conservado la clase “celda.h” donde están todos los métodos para modificar o leer los datos dada una celda. Sin embargo, hemos necesitado añadir ciertos métodos para dotar de nueva funcionalidad a esta clase:

### 8.3.1 celda\_es\_pista

Este nuevo método comprueba, dada una celda y un valor, si es pista o no. Este método es el siguiente:

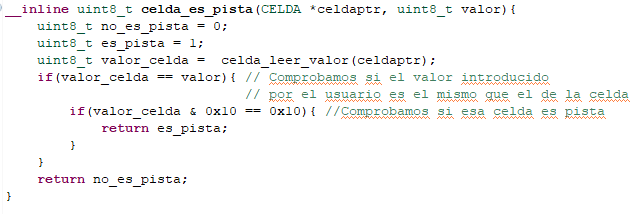


Ilustración : Método celda\_es\_pista() de la clase celda.h

En este método el usuario introduce un valor y se comprueba primeramente si ese valor corresponde con el valor que hay almacenado en la celda. En caso de que sea el mismo, se comprueba si esa celda es pista. Para ello y sirviéndonos de la ilustración 44 realizamos una operación lógica AND con el valor de la celda y 1000 (valor en hexadecimal 0x10) para comprobar si ese bit está activado y por consiguiente es pista. En caso de que sea pista devolverá un acierto. En caso contrario un error

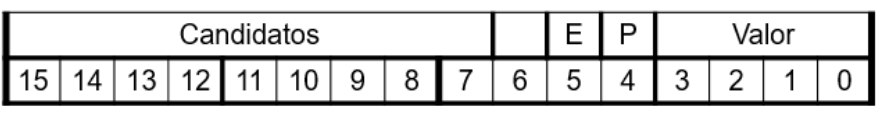


Ilustración : Codificación de cada celda

### 8.3.2 celda\_comprobar\_candidato

Este método comprueba, dada una celda y un valor, si es candidato. Para implementarlo hemos tomado el ejemplo del método “celda\_eliminar\_candidato” donde dado una celda y un valor elimina el candidato:

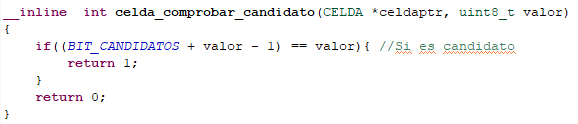


Ilustración : Método celda\_comprobar\_candidato

# 9. Tiempos

Los tiempos con respecto a la práctica anterior son mayores ya que las prestaciones de la placa Embest S3CEV40 son mucho peores que un PC convencional.

Los tiempos son los siguientes:

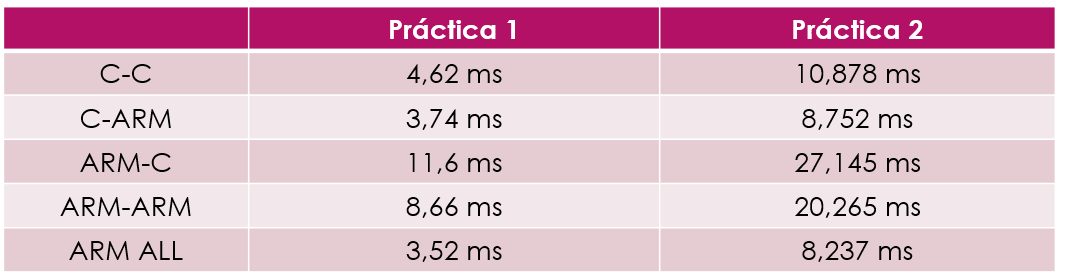


Ilustración : Tiempos práctica 1 y 2

# 10. Problemas durante la práctica

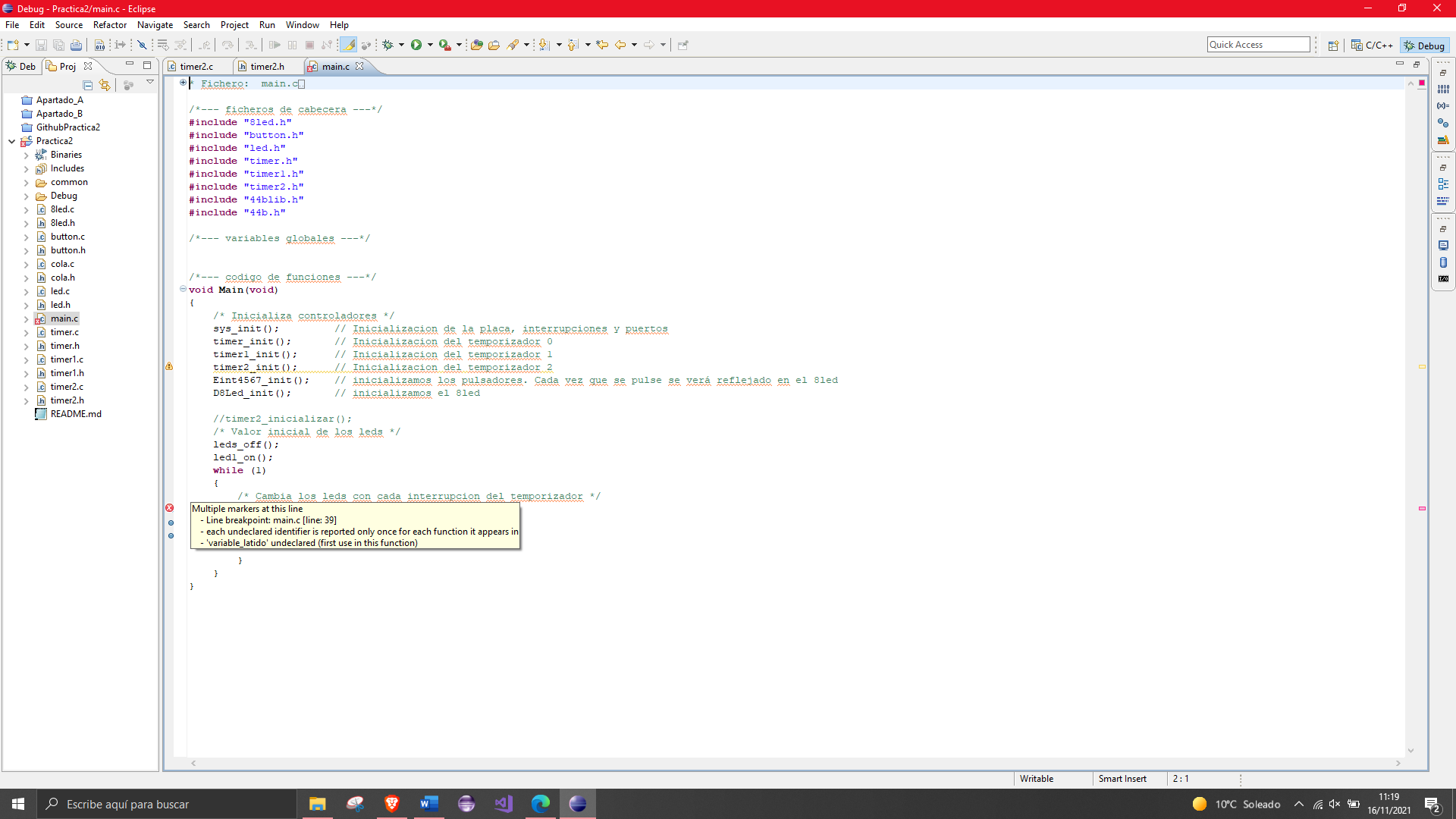
1º A la hora de usar una variable creada en un timer desde main nos daba el siguiente error:  


Ilustración 48: Error en main al usar una variable global de Timer 2

Para solucionarlo vimos que debíamos hacer dos cosas:

* Importar la clase.h del timer que tenía la variable que queríamos usar en el main.
* Cambiar la cabecera del fichero.h que define el comportamiento del timer en cuestión de la siguiente manera:

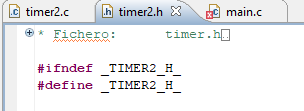


Ilustración 49:Ejemplo cabecera Timer 2

2º Ver registros del sistema

Al intentar ver el registro del sistema nos salía el siguiente error:

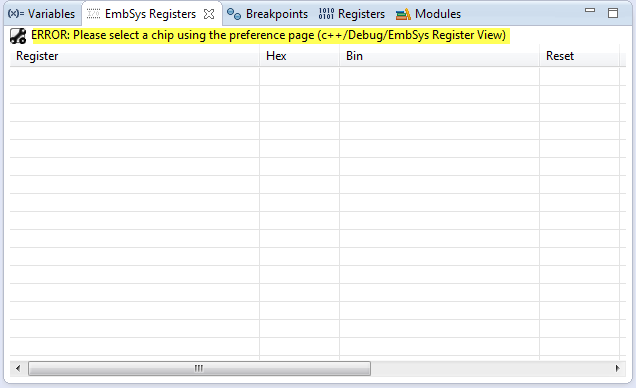


Ilustración 50: Error al ver los registros del sistema

Buscando en internet en el enlace 2 de la bibliografía vimos que debíamos seguir el siguiente camino:

Window->Prefecentes->C/C++->Debug->EmbSys Register View y realizar la siguiente configuración:

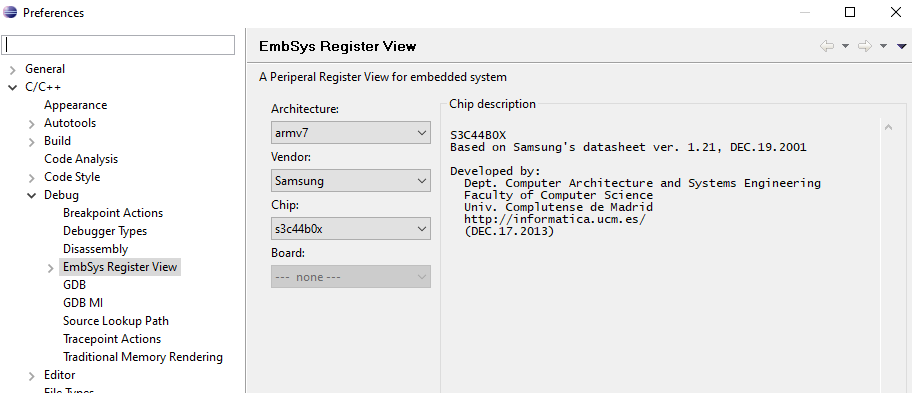


Ilustración 51: Configuración para ver los registros de la placa

3º Incorporar el código de la práctica 1

Al intentar incorporar el código de celda.h y tableros.h nos daban muchísimos errores como los que se muestran en la siguiente ilustración:

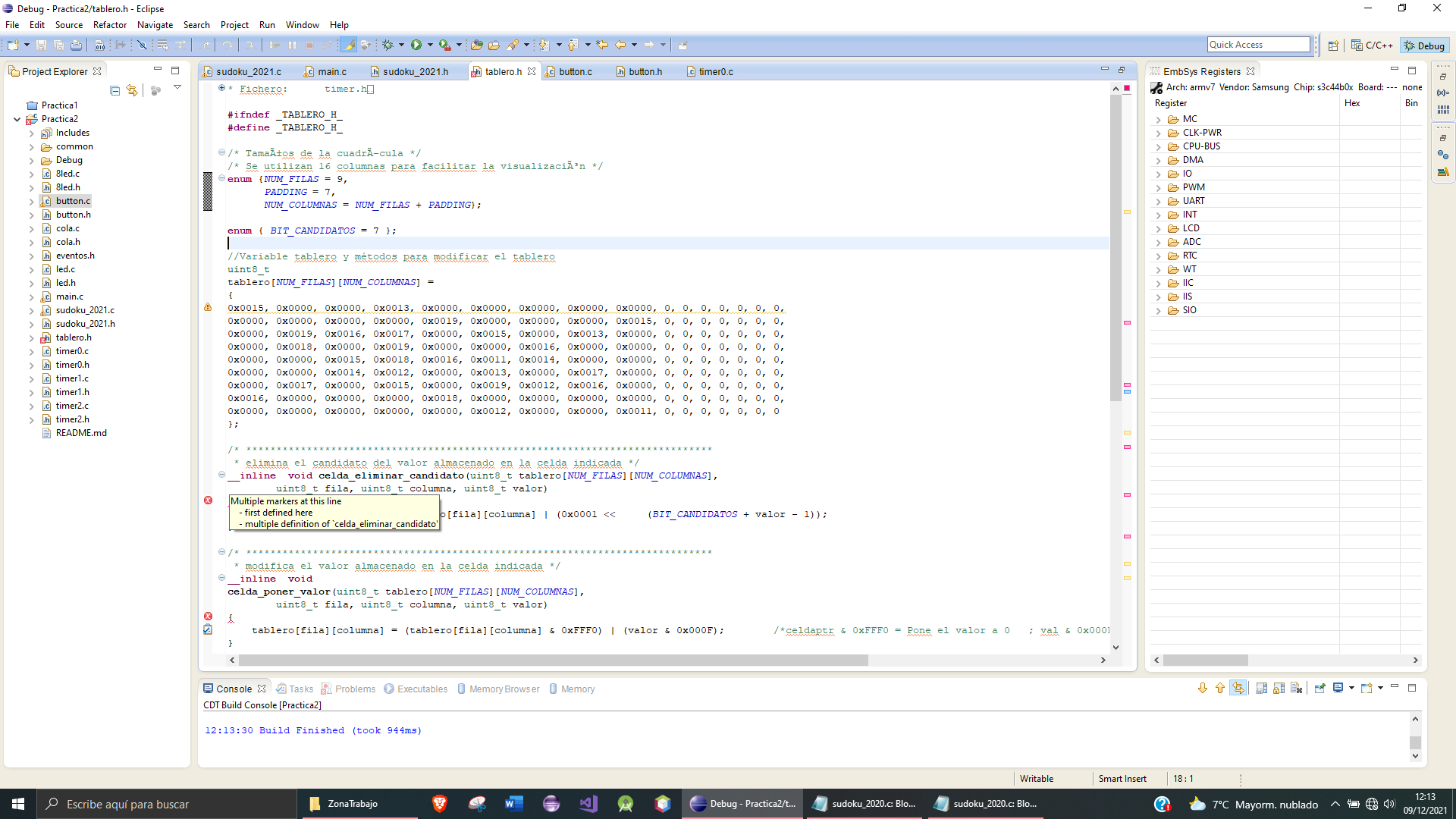


Ilustración 52: Error en métodos de tablero.h

Intentamos juntar Celda.h y tableros.h en una única clase llamada tablero.h donde estuviera el tablero y los métodos que lo modifican. Como no funcionaba intentamos cambiar la naturaleza del tablero de tipo CELDA a tipo uint16\_t y pasar los métodos de Celda.h a tablero.h para eliminar la clase Celda.h. Esto provocó los errores que se muestran en la ilustración anterior. Para solucionar los problemas volvimos a crear el proyecto y empezar de 0. En este intentó nos funcionó.

4º Tablero uint8\_t

A la hora de diseñar el tablero pensamos que sería buena idea implementarlo como un entero sin signo de 8 bits pero a la hora de visualizarlo vimos que no entendía bien cómo se puede mostrar en la siguiente ilustración:

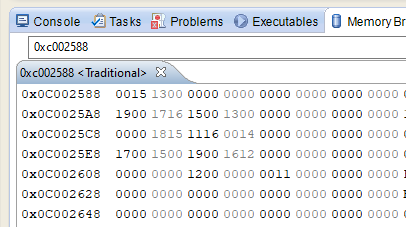


Ilustración : Tablero sudoku uint8\_t

Por consiguiente, decidimos implementar el tablero sudoku como en la práctica 1.

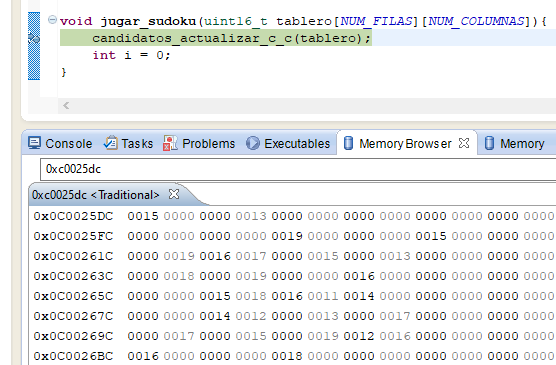


Ilustración 54:Tablero sudoku

# 11. Conclusiones

Tras realizar la práctica hemos visto el funcionamiento real de una placa física donde los componentes no funcionan como esperábamos y debíamos configurarlos para solucionar dichos errores.

Gracias a esta práctica hemos obtenido un conocimiento mayor sobre la parte hardware de un computador.

# 12. Bibliografía

1º Entrada/salida from: <https://moodle.unizar.es/add/pluginfile.php/4871328/mod_resource/content/3/Entradasalida.pdf>

2º How to add register details view in Eclipse from: <https://mcuoneclipse.com/2014/05/29/how-to-add-register-details-view-in-eclipse/>

3º Bitwise Operators in C Programing from: <https://www.programiz.com/c-programming/bitwise-operators>

4º Apuntes de arquitectura y organización de computadores 1 y 2 del grado de ingeniería informática.

5º Apuntes de proyecto hardware del grado de ingeniería informática.