



Escuela Técnica Superior de Ingenieros de Telecomunicación

Core watch

Temporización y control de sistemas

Sistemas Digitales II

**Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación
Curso 2021-2022**

Profesores:

Josué Pagán Ortiz (coordinador docente) j.pagan@upm.es,
Manuel Gil Martín (coordinador administrativo) manuel.gilmartin@upm.es,
José Manuel Pardo Muñoz,
Fernando Fernández Martínez,
Luis Fernando D'Haro Enríquez,
Alberto Boscá Mojena,
Juan José Gómez Valverde,
Lucilio Cordero Grande,

Felicia Alfano, Román Cárdenas Rodríguez,
Cristina Luna Jiménez, Javier Gálvez Goicuría,
Martín Carrasco Gómez, Benito Farina

Departamento de Ingeniería Electrónica

Versión 1.0.6

Copyright 2022 – Universidad Politécnica de Madrid

Todos los derechos reservados.

Material docente. No está permitida la venta o distribución sin autorización de los autores.

Prefacio

Esta asignatura de laboratorio pretende que se familiarice con los sistemas electrónicos digitales que, nos guste o no a los *electrónicos*, tan ligados están a la programación. Programación que, en su inmensa mayoría, se hace en C o C++, todavía. Sistemas *empotrados* —o *embebidos*— que cada vez están más integrados en el Internet de las Cosas (IoT): con el aprendizaje automático, con dispositivos *wearable*, con ciudades inteligentes, coches autónomos... Independientemente de que acabe dedicándose a la electrónica, o no, como ingeniero/a haber cursado esta asignatura le dará un empuje a su formación.

Al acabar la asignatura de Sistemas Digitales II deberá saber que:

- Un sistema puede (y debería) ser descrito a través de **Finite State Machine, Máquina de Estados Finitos (FSM)**. Las FSM permiten tener un código modular, fácil de interpretar y **depurar**, y dado el caso, permiten hacer validación formal (matemática), de que el sistema funcionará como se espera cuando este esté en producción (asignatura Ingeniería de Sistemas Electrónicos (ISEL) del itinerario de Electrónica).
- La ejecución del código de nuestro sistema puede tener **interrupciones** de elementos externos (Hardware (HW), o Software (SW)) o internos (**temporizadores**). Que en un **microcontrolador** se atenderán con Interrupt Service Routine, Rutina de Servicio/Atención a la Interrupción (ISR)s interrumpiendo el flujo principal del programa, y que en un **microporcesador** se puede atender la interrupción de forma paralela mediante hilos o hebras (**threads**). Diga adiós al *pooling* o espera activa.
- Siempre tendremos relojes del sistema que nos brindan **temporizadores** para poder realizar tareas periódicas o que decidamos que sucedan en un determinado momento.
- Nuestros sistemas necesitan poder comunicarse con el mundo exterior que les rodea, y la comunicación con elementos físicos es posible gracias a unos *pines* de entrada o salida llamados **General Purpose Input/Output**,

Entrada/Salida de Propósito General (GPIO). Que estos *pines* se pueden controlar directamente, o a través de *drivers* que nos abstraen del problema.

- **Depurar** es esencial como metodología. Esto vale para la vida misma y para la asignatura 😊: *no cambiar nada y esperar que el resultado sea diferente, no va a funcionar. La magia no existe, seguro que hay algo mal. Eres resultado de tus actos.* Bromas aparte, la depuración no es exclusiva de esta asignatura, ni es exclusiva de programación, es una cuestión de ser metódico y analítico para encontrar el problema y solucionarlo.

Somos conscientes de la dificultad para algunos de enfrentarse (casi) por primera vez al lenguaje de programación C y puede que le requiera un esfuerzo extra por su parte. Por la nuestra estaremos siempre a disposición para ayudar. Además de los tutoriales de C de la propia asignatura, se le facilitan recursos y bibliografía en Moodle. Se han preparado vídeos con explicaciones de conceptos a modo “demostración”, y no clase teórica; podrá verlos enlazados al inicio de algunos capítulos.

Índice general

Prefacio	v
Lista de acrónimos	xI
1. Introducción	1
1.1. Objetivo del proyecto	1
1.2. Organización (versiones)	2
1.3. Profesorado y turnos de laboratorio	3
1.3.1. Turnos	3
1.3.2. Profesores y colaboradores docentes	3
1.4. Evaluación	4
1.4.1. Evaluación estándar continua	4
1.4.2. Evaluación estándar final	6
1.4.3. Evaluación mediante proyecto innovador	6
1.4.4. Insignias	7
1.5. Calendario	8
2. Sesión de introducción	11
3. Versión 1: proyecto base y librería reloj	13
3.1. Proyecto base: coreWatch	14
3.1.1. Proyecto coreWatch en Eclipse	14
3.2. Desarrollo librería: reloj (FSM y temporización)	17
3.2.1. Cabecera reloj.h	18
3.2.2. Código reloj.c	20
3.3. Test de la librería	26
4. Versión 2: Sistema coreWatch	29
4.1. Desarrollo del sistema: Core Watch	29
4.2. Cabecera coreWatch.h	31
4.3. Código coreWatch.c	32
4.3.1. El ThreadExploraTeclado	33

4.3.2. Primera aproximación a FSM coreWatch	37
4.3.3. Segunda aproximación a FSM coreWatch	39
4.3.4. Aproximación final a FSM coreWatch	40
5. Versión 3: teclado matricial	43
5.1. Desarrollo librería del teclado matricial	45
5.2. Test de la librería	52
6. Versión 4: LCD	55
7. Mejoras	59
7.1. Posibles mejoras	59
Bibliografía	63
Apéndices	
I Interfaces de Programación de Aplicaciones	65
A. API de referencia: coreWatch	69
Estructuras	70
Funciones de entrada de la máquina de estados	70
Funciones de salida de la máquina de estados	73
Funciones propias	81
Funciones ligadas a threads adicionales	84
B. API de referencia: reloj	85
Estructuras	85
Funciones de entrada de la máquina de estados	88
Funciones de salida de la máquina de estados	88
Subrutinas de atención a las interrupciones	89
Funciones propias	90
C. Librería LCD (HD44780U)	99
C.1. Librería LCD (HD44780U)	99
C.1.1. Instalación y uso	101
C.1.2. Funciones	102
C.1.3. Conectándolos	103

Índice de figuras

1.1.	Sistemas que integran un núcleo con temporizador, teclado y pantalla.	2
1.2.	Distribución de las calificaciones de la asignatura.	4
1.3.	Contenido de una insignia	7
1.4.	Insignias del curso.	8
1.5.	Calendario de la asignatura.	9
3.1.	Diagrama de elementos del sistema.	14
3.2.	Estructura base del proyecto en <i>Eclipse</i>	16
3.3.	Máquina de estados del <i>reloj</i>	17
3.4.	Sugerencia de plantilla genérica de una cabecera. Ejemplo para <code>reloj.h</code>	18
3.5.	Sección del fichero <code>reloj.h</code> dedicada declarar prototipos de funciones (incluye mejoras).	21
3.6.	<i>Setters</i> y <i>getters</i> sobre las variables expuestas por el <i>reloj</i> y el <i>teclado</i> . .	23
3.7.	Interpretación de la salida por consola de una compilación fallida. . . .	24
3.8.	Prototipo de la función <code>fsm_new</code> de la librería <code>fsm</code>	25
3.9.	Ejemplo de cómo mostrar por pantalla la hora y la fecha para <i>testear</i> . .	27
3.10.	Testeando la librería <code>reloj</code> desde el <code>main</code> de <code>coreWatch</code>	27
4.1.	Máquinas de estados del sistema.	30
4.2.	Ejemplos de mensajes de información durante la inicialización.	33
4.3.	Propuesta de teclas para el sistema.	35
4.4.	Depurando <code>ThreadExploraTecladoPC</code> y <code>EsNumero</code>	37
4.5.	Primera aproximación funcional de la <i>FSM coreWatch</i>	39
4.6.	Reset, set-cancel nueva hora	39
4.7.	Ejemplo de ejecución de set-cancel y <i>reset</i> de nueva hora.	40
4.8.	Procesa dígito y cambio de hora finalizado	41
4.9.	Ejemplos de ejecución de fijar nueva hora.	42
5.1.	Circuitería y esquema de un <i>teclado matricial</i>	44
5.2.	Máquina de estados del <i>teclado matricial</i>	45
5.3.	<i>Teclado matricial</i> y LCD en la placa TL04 del laboratorio.	46
5.4.	GPIOs del BCM para el <i>teclado matricial</i> en el laboratorio.	47

5.5.	Resistencia de <i>pull-down</i> a la entrada de la entrenadora.	50
5.6.	Ejemplo de 2 rebotes en una pulsación	51
5.7.	FSM de <code>coreWatch</code> para detección de comandos.	52
5.8.	Test del <i>teclado matricial</i>	53
6.1.	GPIOs del BCM para el LCD en el laboratorio.	56
7.1.	Niveles de valoración de las posibles mejoras a implementar.	60
2.	Diagrama UML de estructuras del proyecto.	67
A.1.	Diagrama de flujo de la función <code>ProcesaDigitoTime</code>	76
A.2.	Diagrama de flujo de <code>ProcesaTeclaPulsada</code> y <code>ThreadExploraTecladoPC</code> . . .	78
A.3.	Tabla ASCII	83
C.1.	Pantalla LCD de 2x16 conectada a una Raspberry Pi	100
C.2.	Circuitería y esquema de un <i>teclado matricial</i>	101

Lista de acrónimos

API	Application Programming Interface, Interfaz de Programación de Aplicaciones
ASCII	American Standard Code for Information Interchange, Código Estadounidense Estándar para el Intercambio de Información
FSM	Finite State Machine, Máquina de Estados Finitos
GCC	GNU Compiler Collection
GPIO	General Purpose Input/Output, Entrada/Salida de Propósito General
HW	Hardware
IDE	Integrated Development Interface, Entorno de Desarrollo Integrado
ISEL	Ingeniería de Sistemas Electrónicos
ISR	Interrupt Service Routine, Rutina de Servicio/Atención a la Interrupción
LCD	Liquid-Crystal Display, Pantalla de Cristal Líquido
PCB	Printed Circuit Board, Placa de Circuito Impreso
RFID	Radio Frequency Identification, Identificación por Radiofrecuencia
SoC	System on Chip
SW	Software
TBD	To Be Defined, Por Definir
UML	Unified Modeling Language, Lenguaje Unificado de Modelado

Capítulo 1

Introducción

En este capítulo se presenta la descripción general del proyecto de la asignatura y la organización de este. Aquí se podrá encontrar información sobre la evaluación y el calendario que se seguirá (**siempre que la situación sanitaria lo permita**).

La asignatura tiene una componente importante de programación que podrá completarse independiente de que se disponga del HW, o no. Los elementos HW principales que usaremos en este proyecto (*teclado matricial* y Liquid-Crystal Display, Pantalla de Cristal Líquido (LCD)) han sido emulados para poder programar y probar en vuestros propios ordenadores sin necesidad de tenerlos físicamente. **A pesar de que la asignatura ha sido programada para una situación de total presencialidad en el laboratorio, la situación sanitaria en la que nos encontramos puede dar lugar a que tenga que realizarse total o parcialmente de forma remota.** Si la situación lo permite, **el sistema debe funcionar sobre el HW del laboratorio**, no obstante, se recomienda que los alumnos usen la emulación para programar en casa e ir al laboratorio a probar el código y el HW, y preguntar dudas.

1.1. Objetivo del proyecto

El objetivo es desarrollar un sistema completamente funcional del **núcleo de temporización de cualquier otro sistema** que el alumno luego quiera desarrollar. El sistema es, sencillamente, un (i) reloj al que se le puede cambiar la hora, y reiniciarse, usando un (ii) *teclado matricial*, para mostrarse por un (iii) LCD. A este núcleo de temporización le vamos a llamar *Core Watch*, y lo podemos encontrar en casi cualquier sistema electrónico. Ver el vídeo **Demostración Core Watch**.

El alumno tiene la libertad de imaginar e implementarlo, a través de las mejoras del proyecto, en cualquier sistema que desee. Por ejemplo: en una agenda electrónica, un reloj inteligente, un mando a distancia de aire acondicionado, un termostato inteligente, cualquier dispositivo de monitorización remota de señales biométricas o físicas, *etc.*,



Figura 1.1: Sistemas que integran un núcleo con temporizador, teclado y pantalla.

como los ejemplos que se muestran en la Figura 1.1.

El sistema correrá sobre la plataforma Raspberry Pi 1 Model B+ (ver libro de tutoriales de la asignatura [1])¹. El entorno de desarrollo en el laboratorio será Eclipse.

1.2. Organización (versiones)

El desarrollo del *Core Watch* está guiado como un tutorial, por puntos. Es el proyecto básico, **los requisitos mínimos obligatorios, y supone el 80 % del total de la calificación del proyecto. Un 20 % son mejoras** (ver Sección 1.4).

Como se mencionó anteriormente, los elementos HW (*teclado matricial* y *LCD*) han sido emulados. Ya se mencionará cómo, pero teniendo dos configuraciones de proyecto (una para el laboratorio, otra para casa) y un cambio mínimo, el código se puede probar tanto con el HW real como con el emulado.

¹Es posible emplear otros modelos de Raspberry Pi. La instalación y puesta a punto de las herramientas necesarias para hacerlo funcionar son responsabilidad del alumno.

El proyecto básico lo podemos dividir en versiones o fases, a modo de guía. Las características más importantes de cada versión son:

- **Versión 1:** (i) desarrollo aislado de la librería del reloj con FSM (ii) temporización periódica, y (iii) mostrando información por la pantalla del PC. No es necesaria la Raspberry Pi. (*Estimado 2-3 semanas*)
- **Versión 2:** (i) creamos el código principal del *Core Watch* que contiene el *reloj*, (ii) introducimos la FSM del sistema y, (iii) introducimos hilos (*threads*) para uso del teclado del PC. Todavía se puede trabajar sin la Raspberry Pi, si se desea. (*Estimado 3-2 semanas*)
- **Versión 3:** (i) se elimina el hilo para leer el teclado del PC y se introduce el teclado matricial con su FSM. (ii) Pasamos a usar la Raspberry Pi. (*Estimado 2 semanas*)
- **Versión 4:** (i) se deja de usar la impresión por pantalla del PC/terminal y se introduce el LCD. (*Estimado 1-2 semanas*)
- **Mejoras:** mejoras de libre elección del alumno (nuevos dispositivos, funcionalidades, etc.). (*Estimado 5-4 semanas*)

1.3. Profesorado y turnos de laboratorio

1.3.1. Turnos

	Lunes	Martes	Miércoles	Jueves	Viernes
12:00-15:00	CLJ (<i>online</i>)	-	-	-	-
15:00-16:00	-	-	-	-	LFD,
16:00-17:00	JGV, JPO, RCR	FFM, MGM, MCG	FFM, LCG, MGM	JMP, JPO, ABM, JGG	ABM, BF/FA
17:00-18:00					
18:00-19:00					-

Cuadro 1.1: Turnos y profesores

1.3.2. Profesores y colaboradores docentes

Profesores

- **ABM**, Alberto Boscá Mojena
- **FFM**, Fernando Fernández Martínez
- **JGV**, Juan José Gómez Valverde

- **JMP**, José Manuel Pardo Muñoz
- **JPO**, Josué Pagán Ortiz (coordinador)
- **LCG**, Lucilio Cordero Grande
- **LFD**, Luis Fernando D'Haro Enríquez
- **MGM**, Manuel Gil Martín (coordinador administrativo)

Colaboradores docentes

- *BF*, Benito Farina
- *CLJ*, Cristina Luna Jiménez (*online*, a partir del 14 de febrero)
- *FA*, Felicia Alfano
- *JGC*, Javier Gálvez Goicuría
- *MCG*, Martín Carrasco Gómez
- *RCR*, Román Cárdenas Rodríguez

1.4. Evaluación

1.4.1. Evaluación estándar continua

La asignatura tiene una componente práctica muy importante, un 90 % de la nota. No obstante en esta asignatura, no solo se aplican conceptos aprendidos, sino que también se adquieren otros nuevos. El 10 % restante es el *TEST* y, aunque parezca insignificante, es donde el alumno demuestra que controla estos nuevos conceptos; mezcla preguntas cortas de programación y tipo *test*, con una nota mínima de 5. No se preocupe, pues con el dominio de la práctica dominará los conceptos teóricos. Por otro lado, aunque el proyecto se realiza por parejas, con el fin de diferenciar el trabajo de cada uno de los miembros, una parte de la calificación —un 30 % del total—, se evalúa de forma individual.



Figura 1.2: Distribución de las calificaciones de la asignatura.

La asignatura se evalúa a mitad de semestre y al final (ver Sección 1.5). Puede ver el desglose de las calificaciones en la Figura 1.2. En particular:

■ **A mitad** de semestre se evalúa:

- **Especificaciones mínimas del proyecto, versión 2 (20 %)**: por parejas. En el laboratorio, durante su turno habitual, el profesor comprueba que el avance del proyecto cumple las especificaciones (versión 2). En la siguiente sesión habrá realimentación de los problemas encontrados.
- **Test de conocimientos básicos (10 %, min. 5/10)**: individual. Examen de Moodle con preguntas test sobre contenido y/o algo de código. Se realizará en aulas (ver Sección 1.5), cada uno con un portátil². Si no se aprueba, tiene otra oportunidad al final de semestre. Tiene una duración aproximada de 45 minutos.

■ **Al final** del semestre, todo el mismo día, se evalúa:

- **Especificaciones mínimas del proyecto, hasta versión 4 (30 %)**: por parejas. Sobre el HW del laboratorio durante su turno habitual, el profesor evalúa la versión final del proyecto sobre las especificaciones obligatorias.
- **Mejoras libres sobre proyecto ($\geq 20\%$)**: por parejas. Se evalúa a la vez que las especificaciones mínimas. Las parejas que realicen mejoras entregarán una pequeña memoria adicional explicándolas (la guía estará accesible en Moodle). Con estas mejoras o montajes alternativos añadidos al proyecto básico *Core Watch* —y dependiendo de su dificultad y realización— se podrán sumar puntos hasta alcanzar la máxima nota, 10 puntos³. **Para considerar las mejoras, hay que tener una nota mínima de 5/10 en las especificaciones mínimas.**
- **Examen práctico de laboratorio (20 %, min. 5/10)**: individual. Examen práctico, en el laboratorio. Se realiza en su turno, y tiene una duración aproximada de 1 hora. Primero se examina un miembro de la pareja y luego, el otro. La prueba consiste en programar sobre un problema propuesto similar a lo que ha realizado en el laboratorio. Si no ha aprobado la parte del test, tendrá oportunidad de hacerlo tras este examen.

²En caso de no poder disponer de ordenador portátil para hacer el examen, avise al profesor, se habilitarán los puestos del laboratorio B-043 hasta completar aforo. Por otro lado, también puede solicitar prestado uno de la biblioteca.

³Los alumnos pueden hacer tantas mejoras como deseen y la suma de puntos podría exceder el 10. No obstante, la calificación máxima en actas es 10, y el exceso se considera a efectos de poder otorgar Matrículas de Honor.

1.4.2. Evaluación estándar final

El estudiante que deseé **renunciar a la evaluación continua** y optar a la evaluación por prueba final (formada por una o más actividades de evaluación global de la asignatura), deberá comunicarlo por escrito a través de una actividad del Moodle de la asignatura **antes de la séptima semana del semestre**.

La evaluación continua permite liberar conocimientos a lo largo del curso, no obstante, con respecto a la evaluación estándar final, en cuanto a alcance, no tiene diferencia. Para la fecha de evaluación ordinaria, los alumnos que opten por evaluación final tendrán que realizar:

- El examen práctico de laboratorio y un tipo test.
- El proyecto básico (las 4 versiones) funcional sobre el HW del laboratorio.
- (Si se realizan) Mejoras y la memoria correspondiente.

Los mínimos de puntuación son los mismos que para la evaluación continua.

1.4.3. Evaluación mediante proyecto innovador

Aquellos alumnos que deseen realizar un proyecto innovador basado en un problema o un diseño propio (o propuesto por un profesor), deberán hablar con alguno de los profesores de la asignatura y presentarle un listado de notas y una propuesta de proyecto donde describan, en 2 o 3 páginas:

- Objetivos del sistema propuesto.
- Recursos necesarios para llevarlo a cabo.
- Arquitecturas HW y SW propuestas para abordar el problema.

El proyecto es libre, no obstante:

- Ha de cubrir los conceptos básicos de la asignatura: (i) una base del sistema programada en C, (ii) que contenga una o varias FSM, (iii) interrupciones y (iv) temporización.
- El *lenguaje tipo Arduino* no está permitido como base del sistema, sí sobre elementos añadidos (otros microcontroladores). Cubierto lo básico en C, puede trabajar sobre el lenguaje que quiera, sea de alto nivel, o no.

Para poder abordar el proyecto será necesario contar con la aprobación de dicho profesor. No será admitido ningún proyecto (por muy complejo o perfecto que sea) que no se ajuste a estas normas. En el canal de YouTube⁴ de la asignatura puede ver vídeos de proyectos de alumnos en cursos pasados.

Solo se aceptarán 10 proyectos innovadores. Para poder evaluar las propuestas y que las parejas puedan comenzar a trabajar lo antes posible, disponen de 1 semana de plazo para presentar las propuestas.

1.4.4. Insignias

En general, las insignias, o *badges* en inglés, se usan como reconocimiento digital cuando se realiza un curso o una actividad. Constan de una imagen y una descripción con el nombre y los criterios que hay que cumplir para obtenerla, como se muestra en la Figura 1.3. Sirven para reconocer logros a lo largo del curso. Están firmadas por un profesor y permite al alumno anexarla a su *Curriculum Vitae*, o almacenarlas en *mochilas virtuales* como Badgr, u OpenBadges. Puedes conocer más sobre las insignias de Moodle UPM en el CanaltIC: Badges o insignias digitales. Puede leer más sobre insignias digitales en el manual “Insignias digitales como acreditación de competencias en la Universidad” [2].

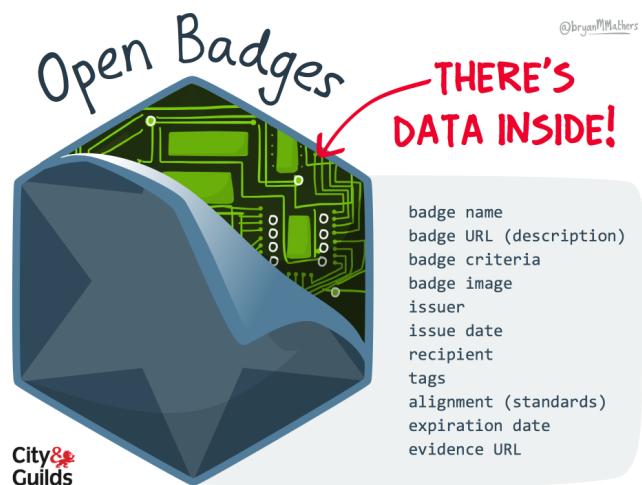


Figura 1.3: Contenido de una insignia (“Open Badges Peeled” de Bryan Mathers, con licencia CC-BY-ND License).

Los profesores, bajo criterio consensuado, podrán otorgar insignias para reconocer algún mérito destacable en la asignatura. En concreto, se contemplan las siguientes insignias (ver Figura 1.4):

⁴Canal Youtube de la asignatura: <https://www.youtube.com/channel/UCYIwg1745WMJ1n0MamDzQw/>

- Para la evaluación continua:
 - **Mejor diseño HW:** electrónica adicional, diseño de Printed Circuit Board, Placa de Circuito Impreso (PCB), integración con otras placas, *etc.*
 - **Mejor diseño SW:** funcionalidades extras significativas.
 - **Mejor diseño de producto:** entendido como un todo, se valorará la experiencia de usuario, acabado con impresión 3D, *etc.*
 - **Proyecto destacado:** proyecto que cumple funcionalidades, tiene mejoras considerables, *etc.* y a juicio de los profesores es un proyecto destacable.
- Para los proyecto innovadores:
 - **Todo proyecto innovador que lo merezca.**
 - **El mejor proyecto innovador.**

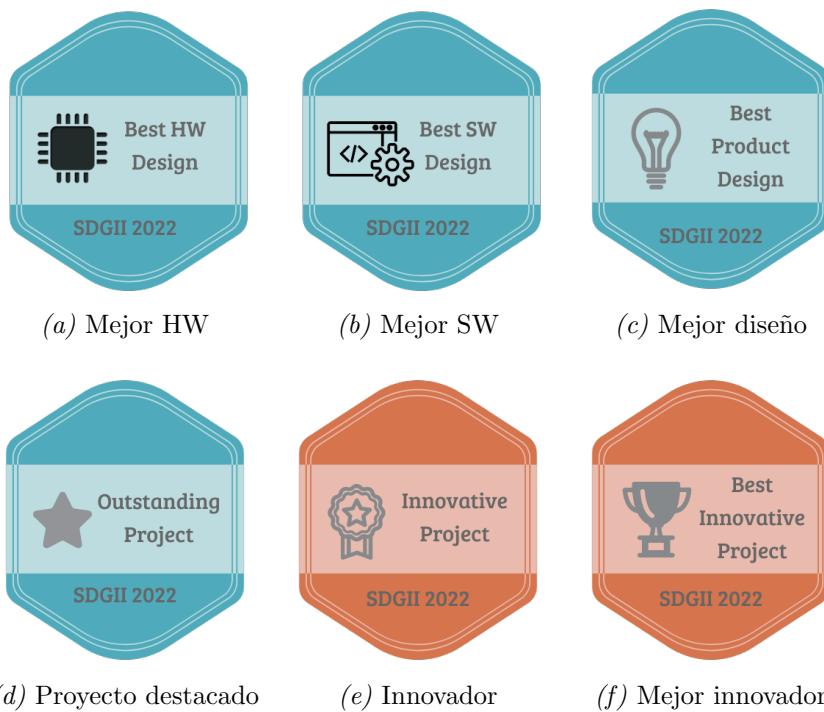


Figura 1.4: Insignias del curso.

1.5. Calendario

En la Figura 1.5 se muestra el calendario de la asignatura. A la izquierda, los días de la semana, señalando los eventos más importantes, como las evaluaciones. A la derecha un gráfico de colores para ver fácilmente qué días tiene clase cada turno.

Orientativamente, se indica por qué versión del proyecto se debería ir cada semana. Ajústese lo más que pueda, no se confíe.

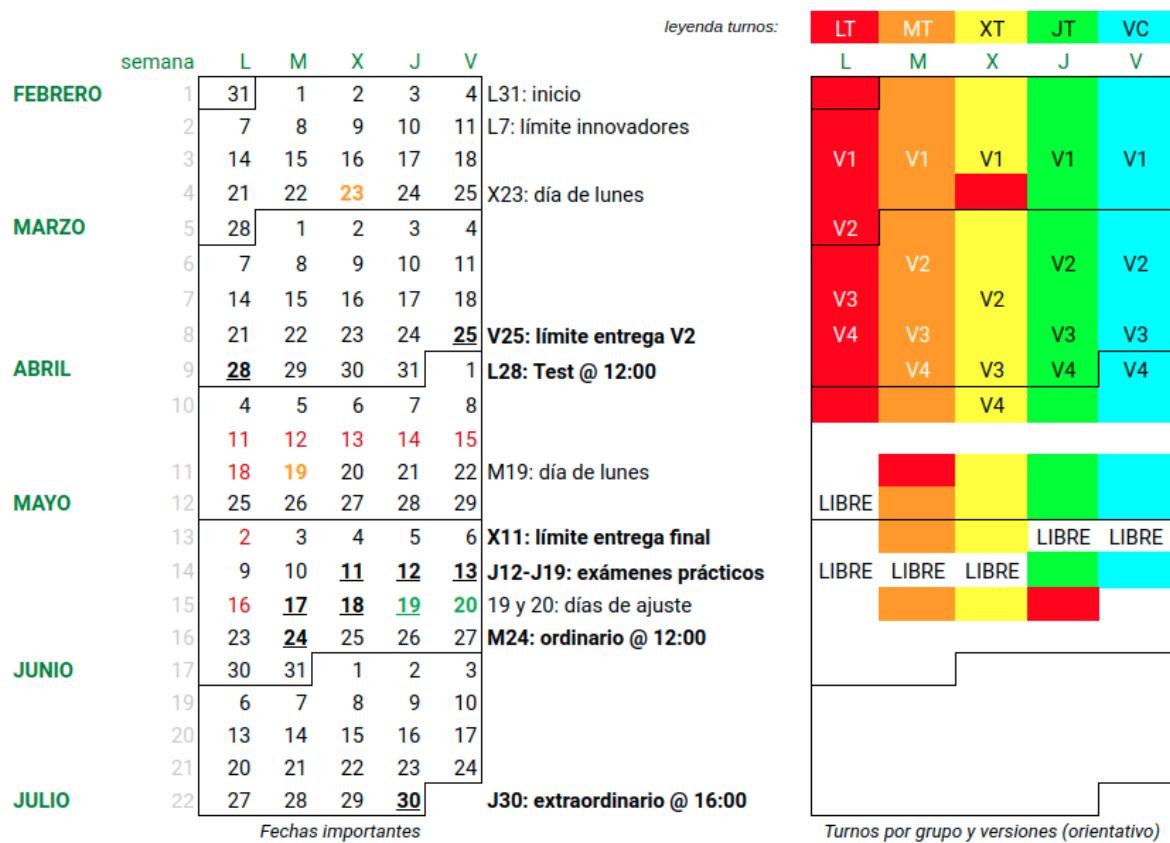


Figura 1.5: Calendario de la asignatura.

Fíjese que, debido a cómo es el calendario académico, hay alteraciones de los turnos; sobre todo, de los lunes (LT). No se preocupe, todos los turnos tienen el mismo número de horas de laboratorio. Las fechas de evaluación y entregas para quien siga evaluación continua estándar son:

- **Viernes 25 de marzo:** (**entrega obligatoria**) fecha límite de entrega en Moodle del proyecto en su versión 2 para todos los turnos. Se entrega un proyecto que compile y funcione hasta donde haya llegado. Solo se entrega un proyecto con versiones completas. Si, por ejemplo, va por la versión 3 pero todavía no funciona, suba solo el código hasta la versión 2.
- **Lunes 28 de marzo:** evaluación individual del test conceptos básicos. Será a las 12:00 en aulas reservadas para la ocasión. Todos los turnos a la vez.
- **Miércoles 11 de mayo:** (**entrega obligatoria**) fecha límite de entrega en Moodle del proyecto final para todos los turnos. Hasta versión 4 y, mejoras y memoria, si las hay.

■ **Jueves 12-jueves 19 de mayo:**

- **Jueves 12 de mayo:** examen práctico y comprobación de requisitos mínimos del proyecto para el turno del jueves (JT).
- **Viernes 13 de mayo:** *idem* para el turno del viernes (VC).
- **Martes 17 de mayo:** *idem* para el turno del martes (MT).
- **Miércoles 18 de mayo:** *idem* para el turno del miércoles (XT).
- **Jueves 19 de mayo:** *idem* para el turno del lunes (LT).

Las especificaciones mínimas del proyecto **son obligatorias**. Las dos entregas obligatorias de código (y memoria de las mejoras, si las hay) se realizarán en buzones de Moodle que se abrirán cerca de la fecha de entrega. **Solo un miembro de la pareja tiene que subir los ficheros fuente del proyecto.**

Se recomienda el uso de repositorios **privados** de código como GitHub o BitBucket para guardar el avance del proyecto cada semana. También puede usar una memoria USB. **Es responsabilidad exclusiva del alumno conservar copias de las distintas versiones del proyecto y de sus avances parciales.** No obstante, estará abierto continuamente un buzón en Moodle para que pueda subir copias de sus códigos cuando quiera. **Este buzón no se evalúa ni se atiende.**

Es responsabilidad de la pareja protegerse de copiar o ser copiados. Los códigos pasan el ***anti-copy***. Si se detecta una copia, la nota de los proyectos copia y copiado será 0. Sean honestos, inspírense, pregunten, pero nunca copien.

Capítulo 2

Sesión de introducción



Capítulos que hay que tener a mano:

- “1. Introducción al entorno de desarrollo en C para Raspberry Pi”
- “2. Prácticas de lenguaje C para Raspberry Pi”
- “3. Guion de la Primera Sesión Práctica”



Vídeos del canal de SDGII:

- Directivas de Precompilador
- Depuración Y Modificador Static
- Array, Memcpy
- Estructuras, Operador Módulo, Pow, Depuración
- Introducción C vs Java

En su primera sesión de laboratorio trabaje con los 3 capítulos de tutoriales recomendados.  Para ser pragmáticos y aprovechar el tiempo de laboratorio empiece por capítulo “3. Guion de la Primera Sesión Práctica”, donde aprenderá a configurar un proyecto para compilación cruzada sobre Raspberry Pi. Luego continúe con los otros para estar completamente familiarizado con el entorno.

Posteriormente, en casa, puede ser que le venga bien ver los videos recomendados. Tanto los documentos como los videos es importante que los tenga siempre a mano, pues tratan de conceptos fundamentales.

Capítulo 3

Versión 1: proyecto base y librería reloj

Esta versión es la que va a tener un desarrollo más largo (estimado en unas 3 semanas). Esta parte del proyecto tiene una curva de aprendizaje mayor, por eso, se le va a guiar en los pasos y la explicación será más extensa. También va a crear más código partiendo de cero, para que se familiarice y coja soltura. Al acabar, habrá creado la *librería reloj* formada por los ficheros `reloj.c` y `reloj.h`, y leerá del teclado para poder *resetear* la hora del reloj y fijar una hora predeterminada.

Debe leer los documentos referenciados y ver los vídeos sugeridos a fin de entender mejor cómo tiene que escribir el código o realizar montajes.



Capítulos que hay que tener a mano:

- “1. Introducción al entorno de desarrollo en C para Raspberry Pi”
- “3. Guion de la Primera Sesión Práctica para wiringPi”
- “4. Introducción a las máquinas de estados en C para Raspberry Pi”
- “5. Manejo de temporizadores, interrupciones y procesos con la Raspberry Pi”



Vídeos del canal de SDGII:

- Demostración Core Watch
- Directivas de Precompilador
- Depuración Y Modificador Static
- Estructuras, Operador Módulo, Pow, Depuración
- Operaciones Con Flags

3.1. Proyecto base: coreWatch

En esta sección se muestra el esquema general del sistema, y se presenta cómo ha de configurar el proyecto en *Eclipse*.

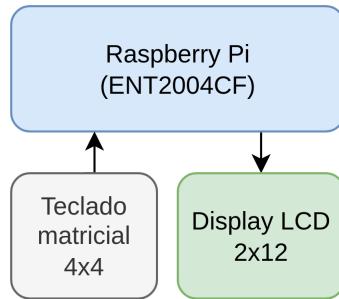


Figura 3.1: Diagrama de elementos del sistema.

La Figura 3.1 muestra los 3 elementos que conforman el sistema:

1. la unidad de control con la *Raspberry Pi*, que en el laboratorio está dentro de la caja entrenadora *ENT2004CF*.
2. El *teclado matricial 4x4* que nos servirá para controlar el sistema en su versión final (aunque en versiones intermedias usaremos el teclado del PC).
3. El display *LCD 2x12* para mostrar la salida del sistema.

3.1.1. Proyecto coreWatch en Eclipse

Vamos a ir construyendo el proyecto poco a poco. En esta primera fase/versión construiremos la librería `reloj`. Tenemos la base del proyecto sobre el que desarrollaremos: (i) primero la cabecera de la librería del *reloj*, (ii) las funciones de la librería y, (iii) por último *testearmos* su funcionamiento desde el `main` de `coreWatch`.

Para empezar a trabajar con el proyecto, siga los siguientes pasos:

1. Cree un proyecto nuevo como se indica en **los puntos 1 y 2 del capítulo “3. Guion de la Primera Sesión Práctica para wiringPi”**.

ATENCIÓN

En el laboratorio (o si trabaja con su propia Raspberry Pi) deberá trabajar con un proyecto *Eclipse* creado para compilación cruzada. Si va a trabajar en casa o en entorno emulado, el proyecto *Eclipse* compilará en nativo. **Los ficheros fuente .c y .h son independientes de que el proyecto lo haya creado para compilación nativa o cruzada, por lo que podría tener 2 proyectos Eclipse, si quiere, e ir actualizando los ficheros fuente de un proyecto a otro.**

En resumen, si va a trabajar con la Raspberry Pi siga los pasos del capítulo desde ya; lo más importante es que tiene que importar la librería *wiringPi* y compilación cruzada. Si va a empezar a programar en entorno emulado (recomendado), usaremos *pseudoWiringPi* y compilador GNU Compiler Collection (GCC) nativo, sin compilación cruzada.

2. Añada al proyecto los archivos fuente que se le han facilitado como *core Watch_proyecto_base.zip* en Moodle.
3. En las preferencias del proyecto, añada los nombres de las librerías de *pthread* y *rt* como se indica, para *wiringPi*, en el **punto 2.2-3.e del capítulo “3. Guión de la Primera Sesión Práctica para wiringPi”**. *rt* nos servirá para tener un sistema de tiempo real y poder usar las interrupciones del temporizador. *pthread* nos servirá para poder crear ejecuciones paralelas con hilos/hebras.

ATENCIÓN

Podemos trabajar en el laboratorio con HW real (Raspberry Pi, teclado, *etc.*), o en casa con los elementos emulados. Lo que nos hará poder tener un mismo código para ambos casos será poder elegir entre: (i) incluir las librerías *wiringPi.h* y *wiringPiDev.h*, o (ii) incluir las librerías *pseudoWiringPi.h* y *pseudoWiringPiDev.h*. *wiringPiDev.h* /*pseudoWiringPiDev.h* se usará para la comunicación con el LCD.

En la cabecera de configuración *ent2004cfConfig.h* se ha definido una etiqueta llamada **ENTORNO** que puede tomar los valores **ENTORNO_LABORATORIO** y **ENTORNO_EMULADO**, y que en *systemConfig.h* sirve para incluir unas librerías, u otras. Cámbiela según lo necesite.

LABORATORIO

Si trabaja en el laboratorio incluya la ruta de la librería *wiringPi* en las preferencias del proyecto como se indica en el **punto 2.2-3.e del capítulo “3. Guion de la Primera Sesión Práctica para wiringPi”**.

Elimine del proyecto todos los ficheros de emulación: *psuedoWiringPi.h* y *psuedoWiringPi.c* para evitar conflictos con las funciones de *wiringPi*.



Figura 3.2: Estructura base del proyecto en *Eclipse*.

EMULADO

Si trabaja en entorno emulado **NO** incluya la librería de `wiringPi` en las preferencias del proyecto.

Añada al proyecto los ficheros de emulación de la librería `wiringPi`: `psuedoWiringPi.h` y `psuedoWiringPi.c`. Estos ficheros, además emulan el teclado matricial y la matriz de LEDs (que no usaremos en este proyecto).

4. En el fichero de configuración de la entrenadora `ent2004cfConfig.h`, en la sección dedicada a `// CLAVES PARA MUTEX`, dé valores a las etiquetas de los mutex: (i) `STD_IO_LCD_BUFFER_KEY` como `key` del `mutex` de la *standard input-output* para imprimir por pantalla y del LCD, (ii) `RELOJ_KEY` como `key` del `mutex` de las variables globales del `reloj`, (iii) `KEYBOARD_KEY` como `key` del `mutex` de las variables globales del teclado matricial, y (iv) `SYSTEM_KEY` como `key` del `mutex` de las variables globales del sistema `coreWatch`. Para conocer qué valores pueden emplearse debe, **leer el punto 2 del capítulo recomendado: “4. Introducción a las máquinas de estados en C para Raspberry Pi”**.

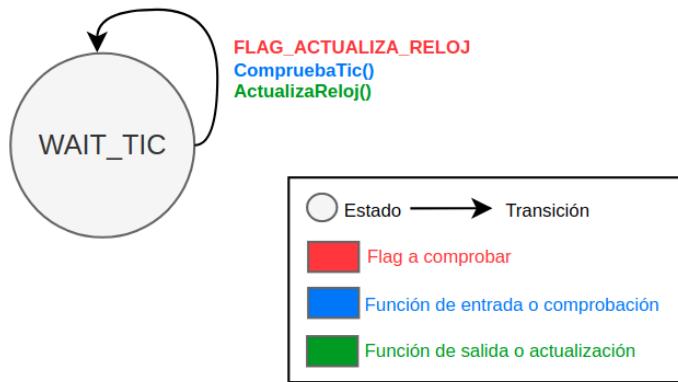


Figura 3.3: Máquina de estados del *reloj*.

ATENCIÓN

Lo mejor sería tener un *mutex* para cada elemento compartido. En nuestro caso, lo mejor sería que el LCD y la *standard input-output* (terminal) tuvieran *mutex* distintos. No obstante, la librería `wiringPi` solo permite manejar 4 *mutex* distintos, y es por ello que se propone *multiplexar* el acceso a la terminal y el LCD.

5. Ahora el proyecto debería compilar sin problemas y la estructura del mismo será como la de la Figura 3.2.

3.2. Desarrollo librería: reloj (FSM y temporización)

Vamos a crear el *reloj* del sistema, que es la parte fundamental de nuestro *Core Watch*. Más adelante, crearemos la lógica del sistema en sí, aunque ahora implementemos parcialmente funciones necesarias. Vamos a trabajar con temporizadores y máquinas de estado. **Lo dividiremos en 14 puntos**.

Estudie y practique con las máquinas de estado. Entienda cómo se definen los estados, las tablas de transiciones, y qué son las funciones de entrada/comprobación y salida/actuación. Para ello **lea los puntos 1 y 2 del capítulo recomendado: “4. Introducción a las maquinas de estados en C para Raspberry Pi”**, y vea el vídeo de **Operaciones Con Flags** en el canal de la asignatura, por si le sirve de ayuda.

El *reloj* implementa su propia máquina de estados. Se trata de una FSM mostrada en la Figura 3.3. Tiene un único estado (llamado `WAIT_TIC`) y una auto-transición. Básicamente estará comprobando que el temporizador `tmrTic` haya saltado y activado el *bit* —el *flag*— `FLAG_ACTUALIZA_RELOJ`. Esto sucederá periódicamente **cada segundo**, aunque la comprobación es más rápida, pues depende de la periodicidad con la que se

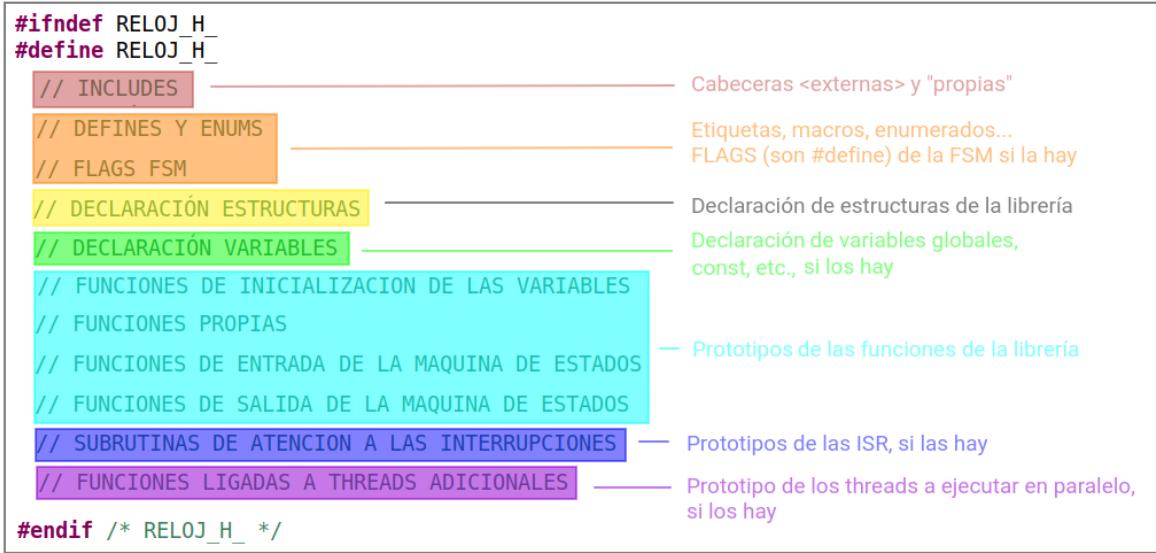


Figura 3.4: Sugerencia de plantilla genérica de una cabecera. Ejemplo para `reloj.h`

hagan las llamadas a `fsm_fire` en el bucle `while` del `main` de `coreWatch.c`. Lo veremos más adelante.

Preparemos el proyecto para poder añadir el `reloj`:

1. Cree dos ficheros fuente y llámelos `reloj.c` y `reloj.h`.
2. Si no lo ha hecho ya *Eclipse*, asegure el fichero `reloj.h` ante inclusiones múltiples mediante las directivas de pre-compilación (**ver vídeo de Directivas de Precompilador**):

```

#ifndef RELOJ_H_
#define RELOJ_H_
// Resto de código
#endif

```

3.2.1. Cabecera `reloj.h`

Vamos a completar primero la cabecera `reloj.h`. Se van a desglosar los pasos principales. **Si detecta que falta algún `#define`, o `#include`, o declaración de variable que sea necesaria o que necesite, hágalo. Lo que aquí se expone no es algo inmutable.**

La Figura 3.4 representa las secciones de una plantilla genérica de cabecera que puede utilizar a lo largo del proyecto. El orden no es un estándar, ni las secciones que ahí aparecen. No obstante, sí es muy conveniente ser ordenado y metódico en programación. **Sí es importante el orden en los siguientes casos:**

- La inclusión de cabeceras ha de ser lo primero. Es importante el orden en caso de existir dependencias entre ellas.

- Es aconsejable definir las etiquetas, macros, enumerados... justo después para que puedan ser utilizados en la declaración y definición de variables.
- Si se declara algún tipo nuevo de variable, hay que hacerlo antes de que se use en el prototipo de alguna función.

Procedamos:

1. Lo primero que debe haber son las cabeceras. Incluya las cabeceras necesarias:
 - `systemConfig.h`: porque incluye las librerías necesarias `wiringPi` (o `pseudoWiringPi`)¹, `fsm`, `tmr`, etc.
 - (*Opcional*) `util.h`: si va a hacer uso de alguna macro (como `MAX`) o función ahí definida.
2. Definimos el `enum FSM_ESTADOS_RELÓJ` con el estado `WAIT_TIC` (ver Figura 3.3).
3. Hay que definir `FLAG_ACTUALIZA_RELÓJ` y `FLAG_TIME_ACTUALIZADO`, y darles valores adecuados. `FLAG_ACTUALIZA_RELÓJ` es el que se muestra en la Figura 3.3. `FLAG_TIME_ACTUALIZADO` es un *flag* para avisar a elementos externos (en nuestro caso el sistema `coreWatch`) que el reloj se ha actualizado. Es un *flag* para comunicación entre máquinas de estado (observe el * en la Figura 4.1).
4. Inmediatamente conviene definir las etiquetas necesarias (`#define`) según las vaya necesitando. En la Application Programming Interface, Interfaz de Programación de Aplicaciones (API) del `reloj` del Apéndice B se van enunciando las más importantes. Si aún no sabe cuáles, lo verá en los siguientes pasos.
5. Declare las estructuras `TipoCalendario`, `TipoHora`, `TipoReloj` y `TipoRelojShared` según se indica en el Apéndice B. Aquí el orden en el que se declaren es importante.
Note que en la API ya han aparecido algunos `#define`. Vaya definiendo las etiquetas como se ha dicho anteriormente.
6. Declare el *array* de la tabla de transiciones de la FSM del `reloj`. No lo defina, su definición se hará en `reloj.c`.


```
||   extern fsm_trans_t g_fsmTransReloj[];
```

¹Ahora es necesaria `wiringPi` para el bloqueo y desbloqueo del *mutex* asociado a los *flags* (funciones `piLock` y `piUnlock` respectivamente).

7. Declare un *array* que le sirva para tener guardados el número de días de cada mes para años bisiestos, y no bisiestos. **A continuación se le da una idea, pero puede implementar otra:**

Declare el *array* `DIAS_MESES`. Este *array* se definirá (dará valores) luego, en el fichero `reloj.c` —donde se definen el resto de variables globales—. Puede declararse como `extern const int DIAS_MESES[2][MAX_MONTH]`; un array de valores enteros constantes (no van a cambiar), de 2 filas y 12 columnas.

Si lo prefiere, también podrían declararse dos *arrays* distintos; uno para años bisiestos y otro para no bisiestos (`DIAS_MESES_BISIESTOS[MAX_MONTH]` y `DIAS_MESES_NO_BISIESTOS[MAX_MONTH]`), por ejemplo.

También puede hacerlo de otras formas, si lo desea. Sea como fuere que lo haga, de alguna manera debe almacenar el número de días de cada mes del año, donde **febrero tiene 28 días en los años no bisiestos y 29 en los bisiestos**.

8. Por último, declare los prototipos de todas las funciones de la librería `reloj` y que están en la API del Apéndice B. El resultado tendrá una pinta como el de la Figura 3.5.

Ya hemos acabado con el encabezado —salvo por algunos `#define` que irá añadiendo más adelante—. **No se olvide de incluir en el fichero `coreWatch.h`, en la sección dedicada a `// INCLUDES` Propios:, la cabecera del `reloj` `reloj.h`.**

Compruebe que compila sin errores ni *warnings*. Si tiene alguno es porque tiene alguna dependencia incumplida, o el orden de alguna declaración no es correcto.

3.2.2. Código `reloj.c`

Vamos a proceder con la implementación de las funciones de la API del *reloj*. Deberá implementar todas las funciones de las que ya ha declarado el prototipo como se explica en el Apéndice B.

En esta sección vamos a practicar con el concepto de **depuración** (*debugging*) (ver punto 3.2.3 del capítulo “1. Introducción al entorno de desarrollo en C para Raspberry Pi” y el vídeo Depuración Y Modificador Static. La herramienta más importante y cómoda que tenemos para depurar mediante el Integrated Development Interface, Entorno de Desarrollo Integrado (IDE) que usemos; en *Eclipse* representado en el ícono . Otra forma para depurar es imprimir texto por pantalla (cuando existe). En sistemas de pocas prestaciones, o cuando no tenemos recursos, ¡incluso se depura haciendo parpadear LEDs! En general, poner mensajes temporales es de mucha ayuda (ver macro `DPRINTF` en `dprintf.h`). Incluso una vez el proyecto está funcionando en producción, es fundamental tener trazas (impresas, LEDs, ficheros de *log*...) para saber qué está pasando en caso de fallo.

```

//-
// FUNCIONES DE INICIALIZACION DE LAS VARIABLES
//-
int ConfiguraInicializaReloj (TipoReloj *p_reloj);
void ResetReloj(TipoReloj *p_reloj);

//-
// FUNCIONES PROPIAS
//-
void ActualizaFecha(TipoCalendario *p_fecha);
void ActualizaHora(TipoHora *p_hora);
int CalculaDiasMes(int month, int year);
int EsBisiesto(int year);
TipoRelojShared GetRelojSharedVar();
int SetFecha(int nuevaFecha, TipoCalendario *p_fecha);
int SetFormato(int nuevoFormato, TipoHora *p_hora);
int SetHora(int nuevaHora, TipoHora *p_hora);
void SetRelojSharedVar(TipoRelojShared value);

//-
// FUNCIONES DE ENTRADA O DE TRANSICION DE LA MAQUINA DE ESTADOS
//-
int CompruebaTic (fsm_t *p_this);

//-
// FUNCIONES DE SALIDA O DE ACCION DE LA MAQUINA DE ESTADOS
//-
void ActualizaReloj (fsm_t *p_this);

//-
// SUBRUTINAS DE ATENCION A LAS INTERRUPCIONES
//-
void tmr_actualiza_reloj_isr(union signal value);

```

Figura 3.5: Sección del fichero `reloj.h` dedicada declarar prototipos de funciones (incluye mejoras).

Podremos ayudarnos de la etiqueta `VERSION` para trabajar con distintas versiones del código sobre el mismo proyecto y activar/desactivar partes que no queramos que se compile. Por el interés que pueda tener usar estas directivas en la asignatura, será la que aquí se explique, pero es opcional. Si prefiere crear un proyecto distinto para cada versión del *Core Watch*, hágalo sin problema.

Estamos implementando una librería. Nuestra librería no tiene `main` y para poder probarla necesitamos poder llamar a las funciones. Si bien podríamos crear un proyecto solo para test del *reloj*, usaremos el `main` de `coreWatch`, como veremos más adelante.

Procedamos con los `#include` y variables globales en el fichero `reloj.c`:

1. Lo primero que debe aparecer es la inclusión de cabeceras; en nuestro caso `reloj.h`
2. Seguidamente definiremos las variables globales (accesibles por todas las funciones de la librería y externas). En nuestro caso:
 - Definimos `fsm_trans_t g_fsmTransReloj[] = {{...}}` Es la tabla de

transiciones de la FSM del *reloj* (ver Figura 3.3). Leer capítulo “4. Introducción a las máquinas de estados en C para Raspberry Pi”. Note que solo tenemos una transición y un estado.

Recuerde, cada fila de la tabla de transiciones tiene la forma: `{EstadoIni, FuncCompruebaCondicion, EstadoSig, FuncAccionesSiTransicion}`. No olvide añadir la fila `{-1, NULL, -1, NULL}` que sirve a la librería `fsm.c` para detectar el fin de la tabla y empezar de nuevo a comprobar las transiciones.

- Definimos el/los *array/s* con los días de cada mes (`DIAS_MESES`), si no lo hemos hecho antes (ver punto 7 anterior).
- Declaramos la variable global `static TipoRelojShared g_relojSharedVars`. Tenga en cuenta que ha de ser `static` para que su uso esté limitado a `reloj.c`, donde está declarado [3, p. 83].

Recuerde que estamos desarrollando una librería. El *reloj* va a exponer esta variable compartida que será leída y modificada por funciones externas (Figura 3.6). En nuestro caso, leída y modificada por funciones de `coreWatch`, pero podría ser usada por cualquier otro código. Para evitar conflictos entre funciones externas accediendo al recurso, usaremos los conceptos de *getters* y *setters* —heredados de Java— tan comunes en programación y haremos la variable `static`. Se trata de funciones que serán las que (i) nos devuelvan la variable `g_relojSharedVars` cuando la queramos leer (*get*), y (ii) las que escriban cuando queramos modificarla (*set*). Además nos protegeremos con el uso de *mutex*, como veremos más adelante.

Ya podemos empezar a codificar las funciones. **Empezaremos con las funciones de inicialización de variables.** Es muy importante aquí que haya entendido bien cómo funcionan los temporizadores (puntos 1 al 4 del capítulo “5. Manejo de temporizadores, interrupciones y procesos con la Raspberry Pi” y el ejemplo 8. Apéndice sobre otro ejemplo de máquina de estados: interruptor temporizado del capítulo “4. Introducción a las máquinas de estados en C para Raspberry Pi”).

3. Codifique la función `ResetReloj` como se indica en el Apéndice B. Elija los valores por defecto que prefiera para fecha y hora. Elija `formato 24`. Recuerde que tiene que hacer uso de *mutex* cada vez que acceda a las variables *flag* de los elementos (ver uso de `piLock` y `piUnlock` en el punto 2 del capítulo “4. Introducción a las máquinas de estados en C para Raspberry Pi”).
4. Codifique la función `ConfiguraInicializaReloj` como se indica en el Apéndice B.

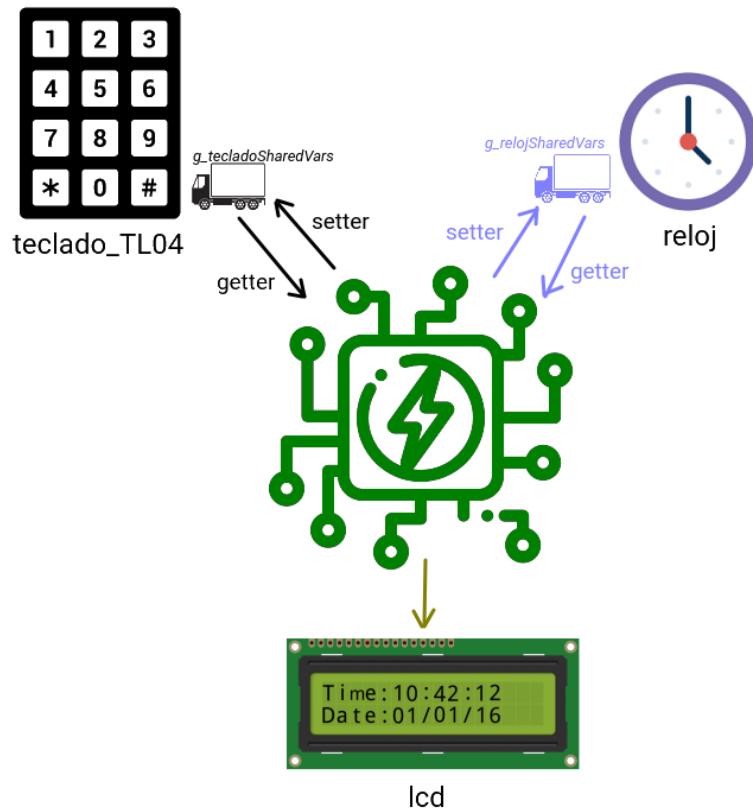


Figura 3.6: *Setters* y *getters* sobre las variables expuestas por el *reloj* y el *teclado*.

No olvide que `PRECISION_RELOJ_MS` ha de representar el valor en milisegundos.

ATENCIÓN

En el capítulo “*5. Manejo de temporizadores, interrupciones y procesos con la Raspberry Pi*” no encontrará la función `tmr_startms_periodic(tmr_t* this, int ms)`. No obstante, sí que hay un ejemplo en el **punto 4 del capítulo**. Su uso es igual que el de la función `tmr_startms(tmr_t* this, int ms)`, con la salvedad de que la primera relanza el temporizador automáticamente (como un reloj), y la segunda solo una vez (como una alarma).

Si intenta compilar, le va a pasar lo que aparece en la Figura 3.7. **Hay que leer los errores de arriba a abajo**. Atención a las explicaciones de lo que pasa. Note que, si puede irse a una terminal y quiere compilar desde ahí, debería introducir los comandos que ahí se muestran (`gcc -o "coreWatch" ... -lrt -lpthread`) y que hacen la magia².

Sigamos codificando funciones. Parecería buena idea implementar primero aquellas que estamos invocando y que nos dan error al compilar; qué menos que para que el error no sea de forma.

²La magia no existe 😊.

```

28----- // FUNCIONES DE INICIALIZACION DE LAS VARIABLES
29 -----
30 -----
31* void ResetReloj(TipoReloj *p_reloj){}
32
33* int ConfiguraInicializaReloj(TipoReloj *p_reloj){}
34

CDT Build Console [coreWatch]
gcc -o "coreWatch" ./coreWatch.o ./fsm.o ./kbhit.o ./pseudoWiringPi.o ./reloj.o ./tmr.o -lrt -lpthread
/usr/bin/ld: ./reloj.o: in function `ConfiguraInicializaReloj':
/home/josueportiz/coreWatch/Debug//reloj.c:69: undefined reference to `tmr_actualiza_reloj_isr'
/usr/bin/ld: ./reloj.o:(.data.rel+0x8): undefined reference to `CompruebaTic'
/usr/bin/ld: ./reloj.o:(.data.rel+0x18): undefined reference to `ActualizaReloj'
collect2: error: ld returned 1 exit status
make: *** [makefile:45: coreWatch] Error 1
"make all" terminated with exit code 2. Build might be incomplete.

11:18:33 Build Failed. 5 errors, 0 warnings. (took 274ms)

```

- Error de compilación. No se ha podido construir el proyecto (Build Failed). En concreto hay 5 errores que podemos ver en "Problems"

- Nos indica que hay una referencia indefinida (undefined reference), en ConfiguraInicializaReloj, porque hemos llamado a tmr_actualiza_reloj_isr y aún no la hemos codificado.

- Nos indica que hay referencias indefinidas a CompruebaTic y ActualizaReloj, porque las hemos indicado en la tabla de transiciones y aún no han sido codificadas

- Compilador GCC (nativo en PC), compila dando el nombre coreWatch al programa. Usa pseudoWiringPi (emulación, trabaja en PC) "Linka" (-l) las librerías rt y pthread que hemos indicado en propiedades del proyecto.

Figura 3.7: Interpretación de la salida por consola de una compilación fallida.

5. Codifique la función de entrada de la FSM `CompruebaTic` como se indica en el Apéndice B. Esta función de comprobación es idéntica a la del ejemplos del punto 2 del capítulo “4. Introducción a las máquinas de estados en C para Raspberry Pi”. Lea atentamente cómo dar valores a las etiquetas `flag`. No olvide proteger el acceso al `flag` con `mutex`.
6. Codifique la función de salida de la FSM `ActualizaReloj` como se indica en el Apéndice B.

Esta función de salida lo primero que hace es recuperar *los datos de usuario que tiene la máquina de estados*. Y lo hace así: `TipoDeVariable *p_miVariable = (TipoDeVariable*)(p_this->user_data);`. `TipoDeVariable` es nuestro `TipoReloj`, a `p_miVariable` le llamaremos `p_miReloj`, y `p_this` es el puntero recibido al `reloj` de nuestro `coreWatch`.

En la versión que usamos de la librería `fsm`, cuando se crea la máquina de estados (de tipo `fsm_t*`), se le puede pasar un campo `user_data`. En nuestro caso, ese `user_data` será la dirección de memoria del `reloj` (ya lo veremos más adelante). La librería `fsm` se puede usar en cualquier código—no solo en nuestro proyecto—, por eso el campo `user_data` debe ser capaz de recibir lo *que sea*. Es por ello que `user_data` es un puntero a `void`:

```
fsm_t* fsm_new (int state, fsm_trans_t* tt, void* user_data)
```

→void: puede recibir punteros a variables de cualquier tipo

Figura 3.8: Prototipo de la función `fsm_new` de la librería `fsm`.

7. Codifique la ISR del temporizador `tmr_actualiza_reloj_isr` como se indica en el Apéndice B. Es similar al ejemplo de `timer_isr` de la implementación que se hace en el **punto 8 del capítulo “4. Introducción a las máquinas de estados en C para Raspberry Pi”**, pero en nuestro caso, no se olvide de proteger el acceso a la variable `flags` con `mutex`.

Si prueba a compilar de nuevo, le dará error. Esto es porque estamos invocando a las funciones `ActualizaHora` y `ActualizaFecha`. Comente momentáneamente esas líneas para probar que el resto del código sí compila. En este punto tal vez esté teniendo problemas con el paso de direcciones de memoria, punteros, estructuras... Si es así, vuelva a ver los videos recomendados en el Capítulo 2.

ATENCIÓN

Se estará preguntando cómo ir probando todo lo que está haciendo. Y hace bien. Si no se lo ha preguntado, pregúnteselo. Bien, ahora que todo el mundo se lo ha preguntado y puesto que ya tenemos las funciones básicas de la FSM, tenga en cuenta que puede probar y depurar lo que va haciendo. Y es buen momento:

- En la función `main` de `coreWatch` cree una variable de tipo `TipoReloj`.
- Pase la dirección de memoria de esa variable a `ConfiguraInicializaReloj`.

```
int main() {
    unsigned int next;
    #if VERSION <= 1
        TipoReloj relojPrueba; // Despues va a ConfiguraInicializaSistema
        ConfiguraInicializaReloj( ); // Despues va a ConfiguraInicializaSistema
    #endif
    ...
}
```

Permite ensuciar para pruebas y que luego no nos moleste. Pero no es necesario.

- Vaya depurando sus avances de esta forma con el resto de funciones.

Ya tenemos codificadas (i) las funciones de inicialización, (ii) la función de entrada de la FSM, (iii) también la de salida, y (iv) la ISR. **Procedamos ahora a codificar las funciones restantes.**

8. Codifique la función `ActualizaFecha` como se indica en el Apéndice B. En la API se le indica cómo hacerlo haciendo uso de la función aritmética *módulo* (%). Pero si prefiere implementar la actualización de fecha de otra forma, hágalo (por ejemplo,

comprobando si se ha excedido del máximo de días del mes actual,..., aunque sea menos eficiente).

Compruebe que funciona, aunque tenga que comentar momentáneamente llamadas a funciones aún no implementadas.

9. Codifique la función `ActualizaHora` como se indica en el Apéndice B. Igualmente, si no quiere hacer uso de la función *módulo* (%), puede comprobar que no excede el máximo de minutos, horas,...
10. Codifique la función `CalculaDiasMes` como se indica en el Apéndice B.
11. Codifique la función `EsBisiesto` como se indica en el Apéndice B.
12. Codifique la función `SetHora` como se indica en el Apéndice B. Para salir de la función de forma exitosa devuelva un número entero distinto de 0.

Depure, compruebe que las funciones hacen lo que se espera. Puede llamar a las funciones directamente (para probar la ISR es mejor esperar a que salte tras lanzar el temporizador) y forzar los valores en las pestañas de depuración de y .

Para finalizar, vamos a implementar las funciones *getter* y *setter* que nos permiten leer y escribir la variable `g_relojSharedVars`.

13. Codifique la función `GetRelojSharedVar` como se indica en el Apéndice B.
14. Codifique la función `SetRelojSharedVar` como se indica en el Apéndice B.

3.3. Test de la librería

Si ha seguido los pasos, seguramente ya haya probado que funcionan las funciones que ha ido codificando. Ya podemos probar nuestro reloj casi en su totalidad: iniciararlo, *resetearlo*, y ponerle una hora. Todo lo anterior lo podemos hacer *testeando* como se ha enseñado arriba, en la función `main`.

Lo único que no podemos hacer es ver el resultado sin depurar. Como hasta la versión 2 no vamos a implementar la función `ShowTime` de `coreWatch`, podemos hacer que temporalmente imprima la hora y la fecha por la terminal. Esto puede hacerlo en `ActualizaReloj` justo antes de salir, y podría tener esta pinta:

```

VERSION 2 y 3 lo hacen en ShowTime
VERSION 4 usará el LCD
#if VERSION == 1
    // Imprime la hora y fecha por la terminal
    printf("Son las: %d:%d del %d/%d/%d\n",
           miReloj.hora.hh, miReloj.hora.mm, miReloj.hora.ss,
           miReloj.calendario.dd, miReloj.calendario.MM, miReloj.calendario.yyyy);
#endif

```

No olvide proteger las impresiones por terminal con mutex

¡OJO! Si la estructura nos la pasan por puntero accedemos con ">"

Figura 3.9: Ejemplo de cómo mostrar por pantalla la hora y la fecha para *testear*.

ATENCIÓN

Se recomienda ejecutar `fflush(stdout)`; cada vez que haga una impresión por pantalla (`printf(...)`). Esto fuerza la liberación del *buffer* de la salida estándar a que muestre todo lo que se *haya mandado a imprimir*. De otra manera, podría notar retardos indeseados en la impresión de mensajes.

Para probarlo todo ya solo nos falta (i) crear la FSM del *reloj* con `fsm_new`, y (ii) lanzar continuamente dicha máquina para que vaya comprobando si ha de realizarse una transición; esto lo que faremos con `fsm_fire` (mire el **ejemplo del punto 4 del capítulo “4. Introducción a las máquinas de estados en C para Raspberry Pi”**). Su `main` de `coreWatch` podría tener un aspecto como el de la Figura 3.10.

Compruebe, por ejemplo, que el cambio de día se produce correctamente, que cambia bien de mes de febrero a marzo en años bisiestos y no bisiestos...

```

int main() {
    unsigned int next;

#if VERSION <= 1
    TipoReloj relojPrueba; // Despues va a ConfiguraInicializaSistema
    ConfiguraInicializaReloj( ); // Despues va a ConfiguraInicializaSistema

    SetHora(1601, ); // Para pruebas Version 1 solo. Despues se quita
#endif

    fsm_t* fsmReloj = fsm_new(WAIT_TIC, g_fsmTransReloj, &(relojPrueba));

    next = millis();
    while (1) {
        fsm_fire(fsmReloj);

        next += CLK_MS;
        DelayUntil(next);
    }
}

```

Figura 3.10: Testeando la librería `reloj` desde el `main` de `coreWatch`.

¡Hemos creado nuestra primera librería! Fíjese que es *portable* a cualquier proyecto fuera de la asignatura que use la librería `fsm`³.

³Quizás lo más restrictivo es que hace uso de la librería `wiringPi` para implementar un mecanismo de *mutex*, pero es fácilmente salvable.

En la siguiente versión implementaremos el código del sistema [coreWatch](#). Volveremos a implementar una librería cuando trabajemos con el teclado matricial.

Capítulo 4

Versión 2: Sistema coreWatch

Debe leer los documentos referenciados y ver los vídeos sugeridos a fin de entender mejor cómo tiene que escribir el código o realizar montajes.



Capítulos que hay que tener a mano:

- “4. Introducción a las máquinas de estados en C para Raspberry Pi”
- “5. Manejo de temporizadores, interrupciones y procesos con la Raspberry Pi”



Vídeos del canal de SDGII:

- Demostración Core Watch
- Operaciones Con Flags

En esta sección se muestra el esquema general del sistema y los diagramas de estados de sus FSMs. Cuando acabe la versión 2 del sistema, el *Core Watch* estará implementando la máquina de estados *FSM coreWatch* de la Figura 4.1.

4.1. Desarrollo del sistema: Core Watch

La *FSM coreWatch* es la máquina principal del sistema y gestiona los eventos del mismo y su interacción con los elementos. **Es la que vamos a desarrollar en esta versión.** Consta de 3 estados: `START`, `STAND_BY` y `SET_TIME`. El detalle de cada función, *flag*, *etc.* está explicado en la API del Apéndice A. La máquina es un poco más compleja que la del *reloj*, pero la filosofía es la misma: estados, funciones de entrada que comprueban *flags*, y funciones de salida que ejecutan acciones. Cuando tenemos más

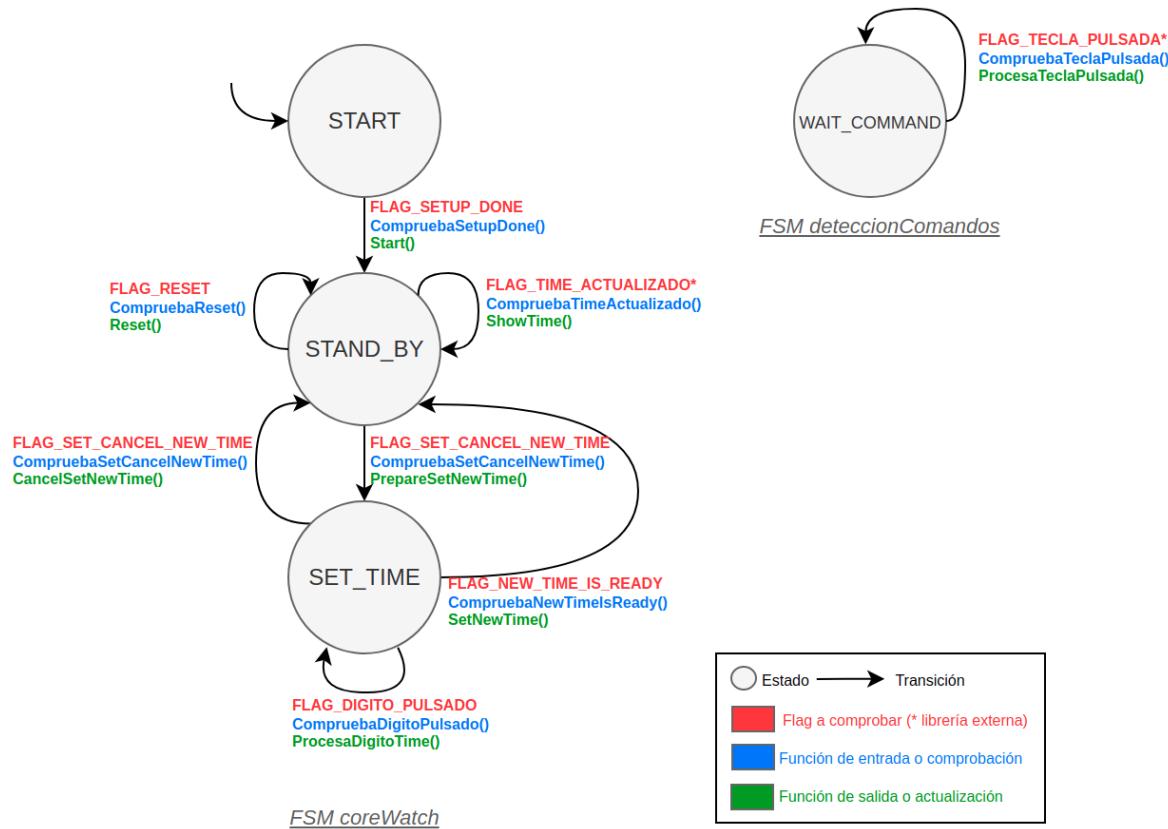


Figura 4.1: Máquinas de estados del sistema.

de un estado hay que indicar en el diagrama cuál es el estado inicial; esto se hace con la flecha *sin origen* como la que se muestra en el estado **START**. Esto se indica en el código cuando se llama la función `fsm_new` como se hizo con el *reloj* (recordar Sección 3.3).

La máquina **FSM deteccionComandos** está pendiente de que se haya pulsado una tecla—un comando—para procesarla. Está estrechamente ligada con la máquina **FSM coreWatch** porque activa gran parte de sus *flags*. Pero también está estrechamente ligada con *teclado matricial* en la `VERSION>=3`, pues es quien activa el *flag* **FLAG_TECLA_PULSADA** del que está pendiente esta máquina. Su implementación se realizará en la `VERSION 3`.

Vamos a seguir los mismos pasos que con el desarrollo de la librería `reloj`: (i) completaremos el encabezado `coreWatch.h`, (ii) codificaremos las funciones en `coreWatch.c` y, (iii) testearemos el sistema haciendo uso del teclado del PC y un `thread`. Si detecta que falta algún `#define`, o `#include`, o declaración de variable que sea necesaria o que necesite, hágalo. Lo que aquí se expone no es algo inmutable.

4.2. Cabecera coreWatch.h

En esta ocasión ya tenemos el fichero `coreWatch.h` creado con los `#include` necesarios, pero debemos rellenarlo.

Procedamos:

1. Primeramente vamos a definir los estados de la FSM que nos ataÑe en la sección dedicada a *// DEFINES Y ENUMS*. Como hicimos con el *reloj*, defina un `enum` llamado `FSM_ESTADOS_SISTEMA` con los 3 estados: `START`, `STAND_BY` y `SET_TIME`.
2. Seguidamente, definamos las etiquetas de los *flags* del sistema en la sección dedicada a *// FLAGS FSM DEL SISTEMA CORE WATCH*. Son los 6 de la *FSM coreWatch* de la Figura 4.1: `FLAG_SETUP_DONE`, `FLAG_RESET...` Deles valores adecuados ( repase los puntos 1 y 2 del capítulo recomendado: “4. Introducción a las maquinas de estados en C para Raspberry Pi” si no recuerda cómo).
3. Ya declaramos la estructura `TipoCoreWatch`. Hágalo en la sección dedicada a *// DECLARACION ESTRUCTURAS* como se indica en el Apéndice A. No introduzca todavía el `teclado` ni el `lcdId`, pues son para versiones posteriores.
4. En la sección dedicada a *// DEFINICION VARIABLES* puede definir variables globales que quiera o necesite. Por el momento, no escribimos nada.

En este punto—salvo algún `#define` que se indique en la API, o que necesite—,ya estamos preparados para definir los prototipos de las funciones de la *FSM coreWatch* (todavía no es necesario declarar los de la *FSM detectaComando*). Así pues:

5. En cada una de las secciones *// FUNCIONES...* declare los prototipos de las funciones del sistema descritas en el Apéndice A para la `VERSION 2`.
6. En este caso, no tenemos funciones ligadas a atención de interrupciones. Deje vacía la sección dedicada a *// SUBRUTINAS DE ATENCION A LAS INTERRUPCIONES*.
7. **Solo en la `VERSION 2` del sistema tendremos una función ligada a un hilo/ hebra (`thread`)**. En la sección dedicada a *// FUNCIONES LIGADAS A THREADS ADICIONALES*, declare el prototipo de la función de exploración de teclado como sigue:

```

||#if VERSION == 2
||PI_THREAD(ThreadExploraTecladoPC);
||#endif

```

ATENCIÓN

Usando las directivas de pre-compilador `#if`, `#endif` y la etiqueta `VERSION` definida en `systemConfig.h`, esta parte del código solo se compilará (ocupará espacio en memoria...) si la versión es igual a la 2.

Puede utilizar esta estrategia conforme avance en el código para tener un mismo proyecto y aislar partes del código temporales como esta, o puede generar un nuevo proyecto para cada versión independiente.

Ya hemos acabado con el encabezado. Compruebe que compila sin errores ni *warnings*. Si tiene alguno es porque tiene alguna dependencia incumplida, o el orden de alguna declaración no es correcto.

4.3. Código coreWatch.c

Vamos a proceder con la implementación de las funciones de la API del *coreWatch*. Deberá implementar todas las funciones de las que ya ha declarado el prototipo como se explica en el Apéndice A. **Lo dividiremos en 20 puntos**.

En esta sección continuaremos practicando la depuración ( repase el punto 3.2.3 del capítulo “1. Introducción al entorno de desarrollo en C para Raspberry Pi”).

Este fichero es el principal de todo el programa, y al contrario de lo que pasaba con la librería `reloj`, sí tenemos un `main`. De hecho, lo habíamos usado para *testear* la implementación de la librería.

Antes de detallar los pasos a seguir, conviene que sepa que, si para probar la librería `reloj` lo hizo metiendo el código entre directivas de pre-compilador con la etiqueta `VERSION` como en la Figura 3.10, puede dejarla por si quiere volver a probar el código de la versión anterior.

Procedamos con las variables globales del fichero `coreWatch.c`:

1. A priori, vamos a tener dos variables globales, el sistema `coreWatch` y su variable `flags`. Es conveniente evitar usar variables globales salvo que sea estrictamente necesario. Suele ser fuente de errores. De momento defina:

- `g_coreWatch` como el sistema de tipo `TipoCoreWatch`
- `g_flagsCoreWatch` como un entero `static`. Esta es la variable `flags` de nuestro sistema.

`static` da lugar a que solo pueda ser accedida por funciones de este fichero.

Siendo este el fichero principal, no es un gran problema si no se define con

este modificador, pero así evitamos la tentación de querer acceder desde elementos externos como el `teclado` o el `reloj`. 😊

ATENCIÓN

`g_flagsCoreWatch` podría estar en la estructura `TipoCoreWatch`. Por comodidad en su acceso, no está pensado así en este documento, pero siéntase libre de añadirla a `TipoCoreWatch coreWatch`, si lo desea. En tal caso, hágaselo saber a su profesor.

4.3.1. El ThreadExploraTeclado

Ya podemos empezar a codificar las funciones. **Empezaremos con las funciones de inicialización de variables.**

2. Codifique la función `ConfiguraInicializaSistema` como se indica en el Apéndice A.

Cuando se dice que *recoge el resultado de...*, quiere decir que cuando se llama a una función X su resultado se guarda en una variable, para comprobar que todo ha salido bien. Por ejemplo:

```
int resultadoDeIniciarAlgo = FuncionQueIniciarAlgo(...);
if (resultadoDeIniciarAlgo != 0){
    return CODIGO_ERROR;
}
// Sigo haciendo cosas...
```

📘 Para recordar cómo lanzar el `thread` con `piThreadCreate(...)`, repase los ejemplos del capítulo recomendado: “4. Introducción a las maquinas de estados en C para Raspberry Pi”.

Imprima mensajes útiles para el usuario. Siempre es importante saber cómo está yendo la inicialización o ejecución del código. No se olvide del `mutex` a la hora de imprimir por pantalla. Ejemplo:

```
Iniciando la configuracion e inicializacion del reloj...
...reloj configurado e inicializado correctamente.
Esta version del sistema usa el TECLADO DEL PC
Hebra (thread) creada correctamente!
Son las: 0:0:1 del 1/1/2022
Son las: 0:0:2 del 1/1/2022
```

Figura 4.2: Ejemplos de mensajes de información durante la inicialización.

Para que podamos compilar lo que vamos codificando, **vamos a implementar ahora el `thread` para procesar las pulsaciones del teclado del PC.** Para que

podamos probar el funcionamiento del sistema desde la **VERSIÓN 2**, vamos a usar el teclado del PC. En esta versión este teclado sustituye al *teclado matricial* del laboratorio y para usarlo se ha de implementar una función de ejecución paralela al programa principal, un **thread** para la exploración del teclado del PC.

El cometido de esta función es (i) leer pulsaciones de las teclas del *teclado del PC*, (ii) interpretar la tecla y activar *flags* del sistema, y (iii) avisar al usuario, si es necesario. Se implementa dentro de un **while(1)** porque su ejecución es constante.

3. Codifique la función **ThreadExploraTecladoPC** como se indica en el Apéndice A.

Empiece implementando el esqueleto del **thread** como se expone a continuación.

```
PI_THREAD (ThreadExploraTecladoPC) {
    int teclaPulsada;
    while(1) {
        delay(10); // WiringPi function: pauses program
                   // execution for at least 10 ms
        if(kbhit()) {
            teclaPulsada = kbread();
            // Logica (diagrama de flujo):
        }
    }
}
```

ATENCIÓN

Note que la variable **teclaPulsada** está definida como **int**, a pesar de que queremos leer *caracteres*.

A diferencia de otros lenguajes, los **char** en C *son* enteros. **char** es otro tipo de entero, normalmente de 8 *bits* y más pequeño que **int**, pero sigue siendo un tipo de entero.

De hecho, los caracteres literales en C, como '**a**', son de tipo **int** en lugar del tipo **char**. El valor numérico no tiene que ser estrictamente un carácter en código American Standard Code for Information Interchange, Código Estadounidense Estándar para el Intercambio de Información (ASCII), pero en la mayor parte de los casos, lo es (ver Figura A.3). Más aún, las funciones **int getchar(void)**, o **int getc(FILE * __stream)** que permiten leer pulsaciones del teclado del PC por entrada estándar **stdin** o por cualquier flujo de entrada (fichero, teclado) respectivamente, devuelven **int**.

EMULADO

Si trabajamos en emulación durante el desarrollo de la **VERSIÓN 3**, en realidad no tenemos ningún *teclado matricial* y, se usa el teclado del PC. Para nosotros será transparente y nos imaginamos que sí es un *teclado matricial*. Internamente, la librería **pseudoWiringPi** hará un **thread** para detectar las teclas. Lo veremos más adelante.



Figura 4.3: Propuesta de teclas para el sistema.

4. Seguidamente, en el apartado indicado para la *// Logica (diagrama de flujo):*, implemente la lógica que sugiere la API. Básicamente, el diagrama de flujo de la Figura A.2 se basa en una escalera de `if-else`. Esto es así por cómo está implementada la función `EsNúmero`. En otra situación, podría usar también una estructura `switch-case`.

No olvide proteger el acceso a los elementos compartidos con el *mutex* correspondiente (activación de *flags* del sistema o imprimir mensajes informativos por terminal).

Se recomienda definir las etiquetas `TECLA_...` en `coreWatch.h`. Así, si en algún momento necesita cambiar las teclas de algún comando solo ha de hacerlo ahí, en un lugar del código. Se propone usar las de la Figura 4.3. **Conviene informar al usuario de qué teclas tiene que usar, por ejemplo en el arranque del sistema.**

ATENCIÓN

Siempre que usemos la terminal de *Eclipse* para introducir pulsaciones, tendremos que pulsar **ENTER**, para que *Eclipse* lo lea y se lo pase al programa. Si ejecutamos desde la terminal de **Raspberry**, o desde una terminal del PC directamente, no.

Usando la terminal de *Eclipse*, puede dar lugar a que, si hay rebotes del teclado, nuestro programa interprete los **ENTER** como una entrada y que, tras introducir un comando válido, pase a analizar dicho **ENTER**. Si usted está mostrando un mensaje para teclas no definidas, puede sucederle que, tras pulsar una tecla *válida*, le aparezca el mensaje de una tecla *inválida*. Esto es por el **ENTER**. Si quiere diferenciar esos **ENTER** de cualquier otra tecla que sí quisiese interpretar, sustituya el **último NO** del diagrama de la Figura A.2 (el punteado) por esta condición:

```

...
// Es cualquier otra tecla (obviemos el enter '\n')
else if ((teclaPulsada != '\n') && (teclaPulsada != '\r') && (teclaPulsada != 0xA)){
    ...
}
...

```

Si todo ha ido bien e intenta compilar, le dará, al menos, un error. Y es que todavía falta por codificar la función **EsNúmero**. **A continuación vamos a codificarla**. Otro error que puede aparecerle sería con respecto a la máquina de estados del *reloj* que teníamos definida en el **main**. Si tal es el caso, lea atentamente el error, pues seguramente esté indicando que no existe **relojPrueba**; y es que ahora nuestro *reloj* es el del sistema: **g_coreWatch.reloj**.

5. Codifique la función **EsNúmero** como se indica en el Apéndice A. El concepto básico es trabajar con la representación hexadecimal del **char**. Hay múltiple formas de implementar esta función, solo que algunas serán más eficientes que otras, pero igual de válidas.

Ya podemos compilar y depurar para ver si nuestras funciones están bien.

Para poder probar nuestro código debemos ir dando forma al **main**. Para ello:

6. Llame a la función **ConfiguraInicializaSistema** con el parámetro correspondiente (dirección de memoria de nuestro sistema) y recoja el resultado.
7. Si el resultado es distinto de 0, avise al usuario de que ha habido un error y salga del programa con un **exit(0)**.

Le recomendamos que **utilizando puntos de parada, vaya comprobando cómo se lanza el **thread****, se inicializa el *reloj*, etc. Observe en la Figura 4.4 que se ha introducido la tecla **'B'** (+ENTER), y un punto de parada en la función **EsNúmero** nos permite ver el valor de la variable recibida (66 en decimal, **0x42** en hexadecimal...).

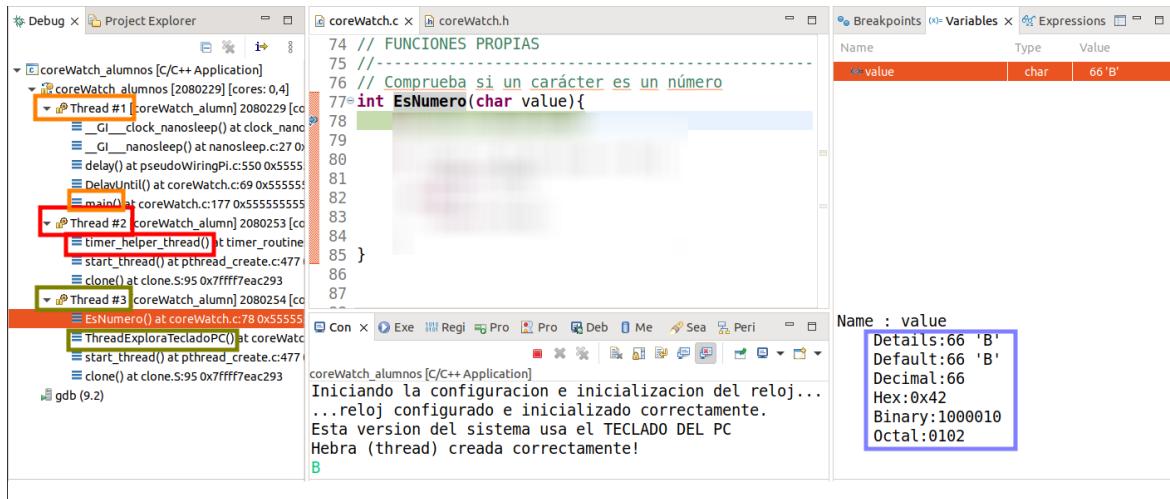


Figura 4.4: Depurando `ThreadExploraTecladoPC` y `EsNumero`.

ATENCIÓN

La Figura 4.4 contiene muchísima información. Fíjese en la parte izquierda. Observe que hay 3 `thread`: (i) el primero es nuestro programa principal, que según la pila de llamadas que cuelga está ejecutando un `delay`; (ii) el segundo el temporizador periódico de nuestro `reloj`, y (iii) el tercero es, efectivamente, nuestro `ThreadExploraTecladoPC` que ha llamado a `EsNumero`. Pinchando en cualquier punto de las pilas de llamada a cada función, podemos ver el estado de las variables.

La parte derecha de la figura está mostrando la variable `value` que nos dice que es de tipo `char`, y cuyo valor mostrado en distintas representaciones, aparece abajo.

En la parte central inferior observe que el programa no hace nada (por ahora) hasta que no se ha introducido la tecla '`B`' (+ENTER).

4.3.2. Primera aproximación a FSM coreWatch

Ahora que tiene un poco más de soltura, vamos a codificar varias funciones que no requieren mucho detalle. **Procedamos con las funciones de comprobación:**

8. Codifique las 6 funciones de entrada de la *FSM coreWatch* como se indica en el Apéndice A.

`CompruebaTeclaPulsada` la codificaremos en la `VERSION` 3. Todas las funciones son idénticas a la función de entrada que implementó para el `reloj.c`. Solo `CompruebaTimeActualizado` tiene la particularidad de que mira un *flag* externo, y necesita recuperarlo con la función que nos proporciona la librería `reloj`. **No olvide proteger el acceso a los *flags* con el *mutex* adecuado**. De nuevo en el caso de `CompruebaTimeActualizado`, no es necesario, pues la función `get` del

reloj ya protege la lectura, y podría dar lugar a inter-bloqueos si se usa el mismo *mutex*.

Llegados a este punto, codificaríamos el resto de funciones. Pero puede ser ardua la tarea. Se le propone una forma incremental de codificar el sistema, que además le permite depurar poco a poco. Atacar todo el problema de golpe puede ser más contraproducente. **Para empezar, vamos a implementar 2 funciones de salida y 2 transiciones de la máquina de estados.**

9. Codifique la función `Start` como se indica en el Apéndice A. No olvide proteger con *mutex* el acceso a *flags*.
10. Codifique la función `ShowTime` como se indica en el Apéndice A. Esta función de salida, lo primero que hace es recuperar *los datos de usuario que tiene la máquina de estados*. Es igual que lo hizo para `ActualizaReloj`. Todas las funciones de salida que *recuperan* los datos de usuario de la FSM lo hacen de la misma forma.

Cuando se imprime por pantalla la hora, se hace como en la `VERSION` 1. Si mantuvo la directiva de pre-compilador `#if VERSION==1`, ahora solo tiene que copiar ese código sin miedo a que se vaya a imprimir dos veces, pues nuestra etiqueta `VERSION` ahora vale 2.

11. Defina la tabla de transiciones parcial `fsmTransCoreWatch` de la Figura 4.5. Puede hacerlo en la función `main`, como hizo con la del *reloj*, o fuera, como variable global¹.

Cada flecha—transición—de la máquina de estados es una línea en la definición de la tabla de transiciones. Como vamos a atacar solo dos transiciones, pues implementamos dos líneas nada más. Las tablas de transiciones de las FSMs se definen siempre igual. Cada línea de la tabla representa: `EstadoOrigen`, `FuncionDeComprobacion`, `EstadoDestino`, `FuncionSiTranscion`.

12. En el `main`, como hizo con el *reloj*, cree la `fsm_t* fsmCoreWatch` usando `fsm_new`. El estado inicial es `START` y como datos de usuario le pasamos la dirección de memoria de nuestro sistema.
13. Ya solo nos queda lanzar periódicamente la máquina de estados dentro del `while(1)`. Lo hacemos bajo la misma llamada que se hace para el *reloj* (`fsm_fire(fsmReloj)`). Es importante el orden en el que se lanzan las máquinas, porque el sistema depende de que estén lanzadas previamente las de los elementos.

¡Ya hemos unido el *reloj* con nuestro *coreWatch*! Si compila y ejecuta (o depura) verá cómo se imprime por pantalla la hora. Y aunque no hagan nada, las teclas también se detectan.

¹Aunque no hay motivos para que sea global, a efectos prácticos, no nos afecta.

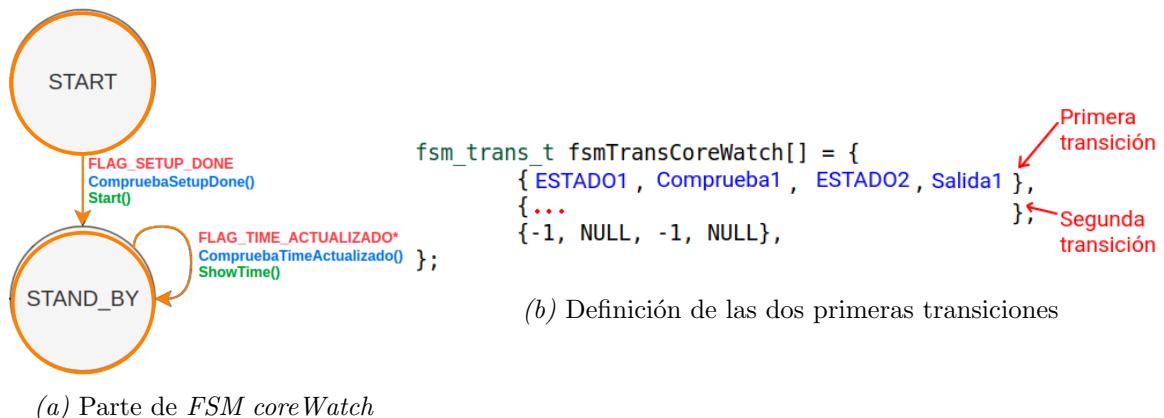


Figura 4.5: Primera aproximación funcional de la *FSM coreWatch*.

4.3.3. Segunda aproximación a **FSM coreWatch**

Vamos a continuar codificando con la estrategia de *divide y vencerás*. Ahora que ya leemos pulsaciones del teclado del PC y mostramos la hora correctamente, **vamos a implementar las funciones que resetean la hora y las que nos llevan a poner/cancelar una nueva hora.**

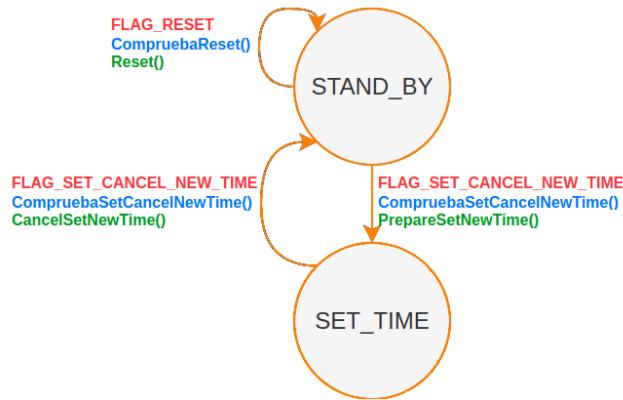


Figura 4.6: Reset, set-cancel nueva hora. Segunda aproximación funcional de la *FSM coreWatch*

Vamos a centrarnos en las tres transiciones de la Figura 4.6. Si prefiere codificar y probar de una en una, hágalo así, si se siente más seguro.

14. Codifique la función `Reset` como se indica en el Apéndice A. No olvide proteger con `mutex` el acceso a `flags`.
15. Codifique la función `PrepareSetNewTime` como se indica en el Apéndice A. No olvide proteger con `mutex` el acceso a `flags`.
16. Codifique la función `CancelSetNewTime` como se indica en el Apéndice A. No olvide proteger con `mutex` el acceso a `flags`.

Como dijimos anteriormente, cada flecha representada ha de ser una línea en la tabla de transiciones de la `fsmTransCoreWatch`.

17. Añada en `fsmTransCoreWatch` —bajo de las transiciones implementadas anteriormente— las tres transiciones de la Figura 4.6 (`EstadoOrigen`, `FuncionDeComprobacion`, `EstadoDestino`, `FuncionSiTranscion`). No olvide que, en la última línea, siempre debe quedar -1, `NULL`, -1, `NULL`, que le indica a `fsm_fire` que es el final de la tabla.

```

Iniciando la configuracion e inicializacion del reloj...
...reloj configurado e inicializado correctamente.
Esta version del sistema usa el TECLADO DEL PC
Hebra (thread) creada correctamente!
Son las: 0:0:1 del 1/1/2022
E
Son las: 0:0:2 del 1/1/2022
[SET_TIME] Introduzca la nueva hora en formato 0-24
E ↵
[SET_TIME] Operacion cancelada Pasan 3 segundos sin
Son las: 0:0:5 del 1/1/2022 volver a pulsar 'E'
Son las: 0:0:6 del 1/1/2022
Son las: 0:0:7 del 1/1/2022
Son las: 0:0:8 del 1/1/2022
F
[RESET] Hora reiniciada
Son las: 0:0:1 del 1/1/2022
Son las: 0:0:2 del 1/1/2022
Se reinicia la hora

```

Figura 4.7: Ejemplo de ejecución de set-cancel y *reset* de nueva hora.

Compile y compruebe que no tiene ningún error ni *warning*. **Vamos a probar el sistema. Depure poniendo puntos de parada en las funciones que ha implementado**, para comprobar cómo se activan y limpian los *flags*.

El ejemplo del resultado de una ejecución se muestra en la Figura 4.7. En él se muestra cómo (i) se ha pulsado la '`E`', han pasado unos segundos sin mostrar la hora (aunque sigue avanzando), (ii) se ha vuelto a pulsar la tecla '`E`' y se ha cancelado la operación, y (iii), finalmente se ha pulsado la tecla '`F`' para *resetear* la hora. Observe que cuando se reinicia la hora, no empieza en 0:0:0, sino en 0:0:1. ¿Por qué cree que es?

4.3.4. Aproximación final a FSM coreWatch

Vamos a finalizar la implementación de la *FSM coreWatch* codificando las dos últimas funciones que nos quedan. **Vamos a procesar los dígitos pulsados en un cambio de hora y actualizar el reloj**.

Vamos a centrarnos en las dos transiciones de la Figura 4.8.

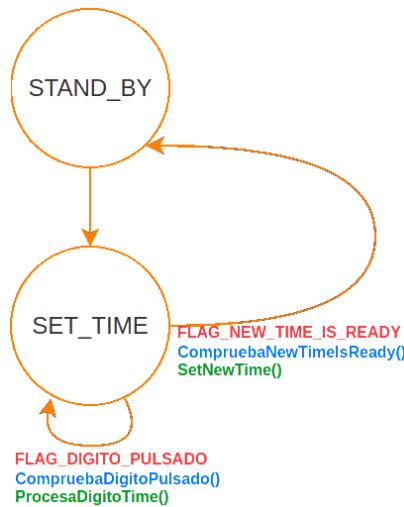


Figura 4.8: Procesa dígito y cambio de hora finalizado. Aproximación final a la *FSM coreWatch*

18. Codifique la función `ProcesaDigitoTime` como se indica en el Apéndice A. No olvide proteger con *mutex* el acceso a *flags*. **En el proyecto base es suficiente con que codifique las ramas para FORMATO 24 horas.**

Quizás se esté preguntando por qué `tempTime` o `digitosGuardados` no pueden ser variables locales `static` (que mantienen su valor cuando salen de la función), y por el contrario están en la estructura del sistema (o pudieran ser variables globales). Bien, esto es porque si fuesen locales y `static`, la función `CancelSetNewTime` no podría limpiar los valores temporales al no poder acceder a ellos.

Si se le ocurre una implementación distinta de la función, adelante. Hágaselo saber a su profesor.

19. Codifique la función `SetNuevoTiempo` como se indica en el Apéndice A. No olvide proteger con *mutex* el acceso a *flags*.

Recuerde, cada flecha del diagrama representa es una línea en la tabla de transiciones de la `fsmTransCoreWatch`.

20. Añada en `fsmTransCoreWatch` —bajo de las transiciones implementadas anteriormente— las dos transiciones de la Figura 4.8.

Antes de probar el código solo nos falta añadir un mínimo detalle. Como sabe, el bucle infinito `while(1)` de la función `main` nunca acabará y todo lo que se codifique tras él no será ejecutado. No obstante, por completitud, conviene liberar memoria de elementos que la han reservado de forma dinámica. Nos referimos a las máquinas de estado y temporizadores que, si se fija en su creación, han reservado

bloques de memoria (con `malloc`) en las funciones `fsm_new` y `tmr_new` respectivamente. Liberemos la memoria:

- Llame a `tmr_destroy` pasándole el temporizador del *reloj* `tmrTic`.
- Llame a `fsm_destroy` pasándole las máquinas de estado creadas (del *reloj* y del sistema).

Estas funciones hacen un `free` para liberar los bloques de memoria dinámica reservada.

Compile y compruebe que no tiene ningún error ni *warning*. **Vamos a probar el sistema. Depure poniendo puntos de parada en las funciones que ha implementado**, para comprobar cómo se activan y limpian los *flags*.

<pre>Iniciando la configuracion e inicializacion del reloj... ...reloj configurado e inicializado correctamente. Esta version del sistema usa el TECLADO DEL PC Hebra (thread) creada correctamente! Son las: 0:0:1 del 1/1/2022 E [SET_TIME] Introduzca la nueva hora en formato 0-24 2 [SET_TIME] Nueva hora temporal 2 0 [SET_TIME] Nueva hora temporal 20 1 [SET_TIME] Nueva hora temporal 201 3 [SET_TIME] Nueva hora temporal 2013 Son las: 20:13:0 del 1/1/2022 Son las: 20:13:1 del 1/1/2022</pre>	<pre>Iniciando la configuracion e inicializacion del reloj... ...reloj configurado e inicializado correctamente. Esta version del sistema usa el TECLADO DEL PC Hebra (thread) creada correctamente! Son las: 0:0:1 del 1/1/2022 E [SET_TIME] Introduzca la nueva hora en formato 0-24 9 [SET_TIME] Nueva hora temporal 2 9 [SET_TIME] Nueva hora temporal 23 9 [SET_TIME] Nueva hora temporal 235 9 [SET_TIME] Nueva hora temporal 2359 Son las: 23:59:0 del 1/1/2022 Son las: 23:59:1 del 1/1/2022</pre>
--	--

Figura 4.9: Ejemplos de ejecución de fijar nueva hora.

El ejemplo del resultado de dos ejecuciones se muestra en la Figura 4.9. En él se muestra cómo (i) se ha pulsado la '**E**', (ii) seguidamente, se han ido pulsando los dígitos. Observe en el ejemplo de la derecha cómo, tras pulsar cuatro veces '**9**', la hora que se ha fijado es la máxima posible en el formato *0-24*. Observe que cuando se guarda la hora, esta vez sí empieza en **XX:XX:0**, y no en **XX:XX:1**. ¿Por qué cree que es?

¡Ya hemos acabado la **VERSION 2!** Antes de continuar con la siguiente versión, **guardé una copia del proyecto que compile y sea funcional para subirla a Moodle para su evaluación.**

En la siguiente versión implementaremos la librería del *teclado matricial*, lo que quiere decir que ya empezamos a trabajar con HW.

Capítulo 5

Versión 3: teclado matricial



Capítulos que hay que tener a mano:

- “5. Manejo de temporizadores, interrupciones y procesos con la Raspberry Pi”
- “6. Iniciación al Manejo de las Entradas/Salidas del BCM 2835”



Vídeos del canal de SDGII:

- Vídeo del curso 19-20 para su versión 3

Para desarrollar esta librería no vamos a contar con una API. En esta ocasión se le facilita el esqueleto de las funciones, así como sus prototipos y las estructuras.

 **Entienda bien los puntos 6, 7 y 8.5 del capítulo “6. Iniciación al Manejo de las Entradas/Salidas del BCM 2835”** y eche un vistazo al  vídeo recomendado, le puede ayudar. En nuestra versión de la librería existen diferencias con respecto a la del ejemplo y el vídeo, no obstante, se le hará notar más adelante.

Si tiene la oportunidad de abrir en casa cualquier sistema que tenga un teclado o botonera¹, seguramente encuentre una circuitería como la de la Figura 5.1. La goma gris es la cara interna de los botones, que está sobre la PCB verde. Las almohadillas negras que ve son contactos metálicos que, cuando se pulsa el botón, se cortocircuitan con el metal de la PCB y cierran el circuito².

La idea de colocar los botones así, haciendo una rejilla, es muy inteligente.

¹No nos hacemos responsables de posibles daños que pueda ocasionar 😊.

²Para un mejor contacto, el circuito de la imagen son dos pistas de cobre en zig-zag que se cortocircuitan al pulsar el botón. Lo más simple sería una cruz que no se toca, pero su contacto es menos fiable.

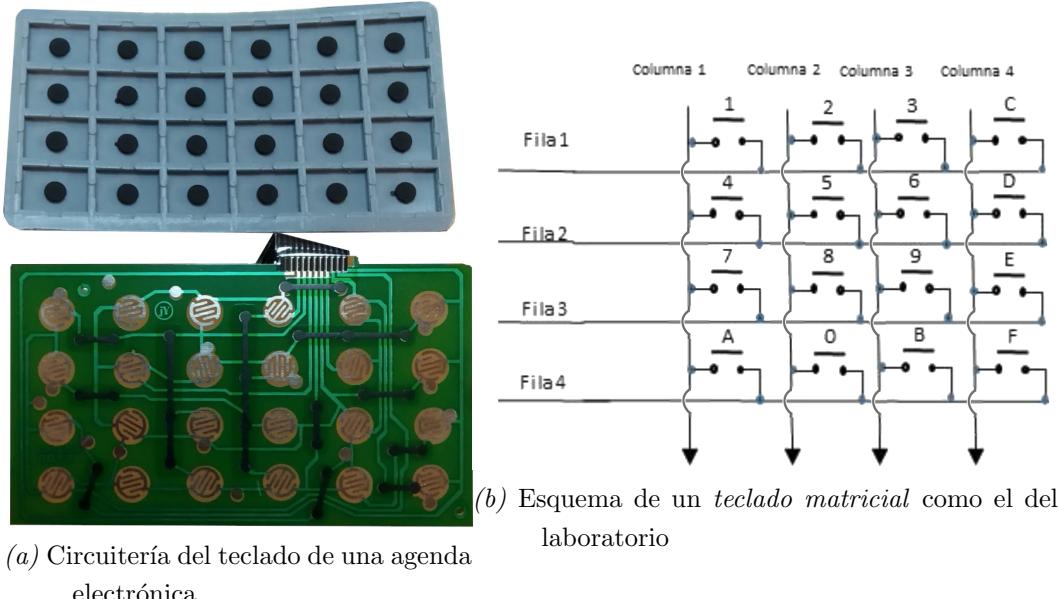


Figura 5.1: Circuitería y esquema de un *teclado matricial*.

Haciendo un enrejillado no es necesario tener un *cable* para cada botón, porque un cable por cada botón implicaría tener un *pin* de entrada en nuestro microcontrolador por cada uno (si no se usan multiplexores, claro), y los *pines* no es algo que sobre, generalmente, en los encapsulados de los *chip*. Por ejemplo, en el teclado de la izquierda de la Figura 5.1 se usan 10 cables para 24 botones (ahorro del 58,3 % de conexiones/ *pines*), y en el del teclado del laboratorio se usan 8 conexiones para 16 botones (ahorro del 50,0 %).

Pero esta idea no sale gratis. A cambio de reducir el HW necesario, tenemos que complicar el SW un poquito. Como habrá leído en el tutorial (punto 6 del capítulo “6. Iniciación al Manejo de las Entradas/Salidas del BCM 2835”), debemos ir excitando —poniendo tensión— las columnas de forma cíclica para poder detectar qué botón se ha pulsado (lo hacemos leyendo las filas). Fíjese que solo podemos tener una columna con tensión a la vez porque, de otro modo, no seríamos capaces de distinguir entre los botones de una misma fila. Si hace un dibujo, lo verá fácilmente.

No se preocupe, la gestión de columnas tiene fácil solución si trabajamos con las máquinas de estado; pues para controlar la excitación de las columnas del *teclado matricial* tenemos la FSM de la Figura 5.2.

Esta máquina de estados más la máquina *FSM deteccionComandos* (Figura 4.1) son las mismas que las de los ejemplos del tutorial, pero están separadas. La principal diferencia con respecto a los documentos de ayuda es que en nuestro proyecto, la interpretación de las teclas pulsadas la hace el sistema y no el teclado. No se preocupe, veremos cómo hacerlo más adelante.

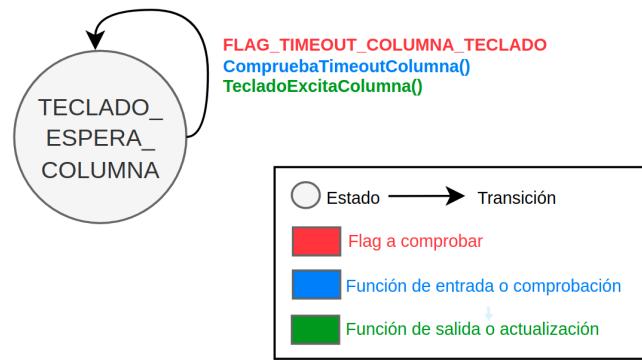


Figura 5.2: Máquina de estados del *teclado matricial*.

Recuerde que, como hicimos con el *reloj*, estamos implementando una librería. El *teclado matricial* no tiene por qué saber nada de las acciones que hace el sistema cuando se pulsa una tecla. Es por eso que en nuestro proyecto, el teclado, se encargará solo de guardar la tecla pulsada y de avisar de que ha habido una pulsación. El sistema que use esta librería —sea en este proyecto u otro— deberá comprobar la estructura de datos que expone el *teclado matricial* con un *get*. La idea es exactamente la misma que la que implementó en el *reloj*.

5.1. Desarrollo librería del teclado matricial

Vamos a proceder con la implementación de las funciones del *teclado matricial*. Deberemos implementar todas las funciones que tenemos en el fichero `teclado_TL04.c`. También deberemos hacer modificaciones en el fichero del sistema `coreWatch.c`. Lo dividiremos en 16 pasos.

Preparemos el proyecto de *Eclipse*:

1. Ya no vamos a usar el teclado del PC. Puede crear un proyecto nuevo partiendo del anterior y comentar o borrar todo lo relativo al `thread`. O también puede encerrarlo entre directivas de pre-compilador `#if VERSION == 2` y `#endif`, y actualizar el valor de la etiqueta `VERSION` a 3.
2. Descargue de Moodle los ficheros de la librería `teclado`: `teclado_TL04.h` y `teclado_TL04.c` y añádalos a su proyecto.

Todos los prototipos de funciones están definidos, y el código está dispuesto para que pueda compilar sin errores. Si compila, verá que todo son *warnings* por definición de variables que no se usan (todavía).

Nota: si en las opciones de compilación del proyecto marcó la opción de que todos los *warning* sean considerados error (opción `-Werror` *Warnings as errors (-Werror)*),

desmárquela para poder codificar poco a poco esta librería.

3. Incluya en `coreWatch.h` la cabecera de la librería, igual que hizo para el *reloj*.
4. Añada a nuestra estructura de `TipoCoreWatch` la variable que llamaremos `teclado` de tipo `TipoTeclado` (ver Figura 2).

ATENCIÓN

Igual que en las versiones anteriores, si está trabajando en emulación, use la librería `pseudoWiringPi`. Si está trabajando en el laboratorio, para programar la Raspberry Pi, quite los ficheros `pseudoWiringPi` del proyecto y trabaje con la librería `wiringPi`.

Procedamos con la implementación. Para poder ir completando la librería de forma incremental, le recomendamos seguir estos pasos, pero puede atacar el problema de otra manera si lo desea.



Figura 5.3: Teclado matricial y LCD en la placa TL04 del laboratorio.

Cuando vamos a conectar cualquier HW que nos compremos, antes de ponernos a codificar su librería *per se*, debemos saber con qué conexiones va a trabajar nuestro *micro*, y configurarlo. En nuestro caso, en qué GPIOs de la Raspberry Pi va a estar conectado el *teclado matricial*³. Afortunadamente, en el laboratorio estas conexiones

³A veces usaremos indistintamente Raspberry Pi, en genérico, como plataforma, otras veces, *BCM*. El *BCM2835* es el microprocesador System on Chip (SoC) de la *Raspberry Pi* en los modelos A, B y B+. Pero los puertos, las GPIOs, son los mismos: salen del microprocesador *BCM* y la plataforma los hace accesible por los pines.

están preestablecidas. La fajas de cables de la TL04 que conectan con las entradas y salidas digitales de la entrenadora *ENT2004CF* (fajas de la izquierda y derecha respectivamente en la Figura 5.3) van a las GPIOs de la Raspberry Pi según la tabla de la Figura 5.4.

ENTRADAS			
Placa TL-04 conexión entradas entrenador		Número del puerto BCM sólo utilizable como entrada	ENT2004CF LEDS ROJOS
Pin			
1	Teclado Fila 1	5	LRE0
2	Teclado Fila 2	6	LRE1
3	Teclado Fila 3	12	LRE2
4	Teclado Fila 4	13	LRE3

SALIDAS			
Placa TL-04 conexión salidas entrenador		Número del puerto BCM sólo utilizable como salida	ENT2004CF LEDS VERDES
	Pin		
Teclado Columna 1	1	0	LVS0
Teclado Columna 2	2	1	LVS1
Teclado Columna 3	3	2	LVS2
Teclado Columna 4	4	3	LVS3

Figura 5.4: GPIOs del microprocesador BCM2835 para el *teclado matricial* en el laboratorio.

4. En la cabecera dedicada a la entrenadora, `ent2004cfConfig.h`, defina correctamente los valores de las etiquetas `GPIO_KEYBOARD_COL_...` y `GPIO_KEYBOARD_ROW_...` que indican las GPIO de las columnas y las filas a las que está conectado nuestro *teclado matricial*.
5. En el código del sistema (`coreWatch.c`), tenemos que decirle a la librería de nuestro *teclado matricial* dónde lo hemos conectado. Lo haremos en `ConfiguraInicializaSistema`, tras configurar el *reloj*. Ello podemos hacerlo la función `memcpy`:

```
// Copia en memoria N bytes de SRC a DEST:
// memcpy(DEST, SRC, N)
memcpy(p_sistema->teclado.filas, arrayFilas, sizeof(arrayFilas));
memcpy(p_sistema->teclado.columnas, arrayColumnas, sizeof(arrayColumnas));
```

Para ello, cree dos *arrays* en los que guarde los valores de las GPIO de las filas y las columnas. Utilice la función `memcpy` para hacer una copia de los *arrays* creados a los *arrays* `filas` y `columnas` del teclado del sistema.

ATENCIÓN

En C **NO** se pueden copiar *arrays* así:

```
p_sistema->teclado.filas = arrayFilas;
```

Tampoco podríamos **asignar** así los valores:

```
p_sistema->teclado.filas = {...};
```

Esta notación es solo posible en la **inicialización** de un *array*, como cuando ha creado los *arrays* con las GPIO, o como:

```
TipoTeclado teclado = { .filas ={...}, .columnas={...} };
```

Por lo que, para copiar *arrays* en C debemos usar `memcpy` o guardar elemento a elemento con un bucle `for`.

6. En `ConfiguraInicializaSistema`, seguidamente, llame a `wiringPiSetupGpio()` y recoja el *int* que devuelve para comprobar que la inicialización del *driver* es correcta. En caso contrario salga del programa devolviendo algún valor distinto de 0.

ATENCIÓN

Para comunicarnos con los *pines* GPIO, utilizaremos la librería `wiringPi` o `pseudoWiringPi`, si estamos en emulación, y que ya ha venido usando para el uso de *mutex*. Para acceder a los registros de entrada/salida donde los *pines* están reflejados en el microprocesador es necesario cargar el *driver*. Esto lo hacemos llamando a `wiringPiSetupGpio()`.

■ Lea el punto 2 del capítulo “6. Iniciación al Manejo de las Entradas/Salidas del BCM 2835”.

7. Ya podemos llamar a `ConfiguraInicializaTeclado` pasándole la dirección de memoria de nuestro teclado. Informe al usuario por pantalla de vez en cuando. Pruebe a compilar y vea que no hay errores.

Empecemos ya con la implementación de la librería. Por ejemplo, podemos empezar con las funciones más sencillas:

8. Complete las funciones `GetTecladoSharedVar` y `SetTecladoSharedVar` de la misma forma que lo hizo para el *reloj*. Tenga en cuenta que el *mutex* que protege al teclado es distinto que el del *reloj*.

Si compila de nuevo, verá que le han desaparecido algunos *warning* puesto que la variable `g_tecladoSharedVars` ya está siendo usada.

9. Complete la función de entrada de la máquina de estados. Esta función comprueba el valor del *bit* que representa el `FLAG_TIMEOUT_COLUMNAS_TECLADO` de la variable `flags` de nuestra estructura de tipo `TipoTecladoShared`.

A continuación **vamos a codificar la función que configura e inicializa el teclado matricial.**

10. Complete la función `ConfiguraInicializaTeclado` como indican los comentarios del fichero. Esta función, al igual que hicimos con el *reloj* y el sistema, se encarga de (i) inicializar la variable global `g_tecladoSharedVars`, (ii) configurar los *pines*, y (iii) configurar el temporizador de la FSM del *teclado matricial*.

En los tutoriales tiene toda la información necesaria. Asegúrese de leer atentamente. No obstante, algunas cosas a tener en cuenta:

- Proteja en todo momento la configuración del teclado y la variable compartida con el *mutex correspondiente*.
- **Inicialización de `g_tecladoSharedVars`:**
 - El tiempo de guarda del anti-rebotes es, ahora, 0.
 - La `columnaActual` representa la columna que va a ser excitada. No es crítico el valor que ponga al principio, pero más adelante tenga en cuenta cómo afecta la implementación de la función `TecladoExcitaColumna` si quiere que empiece por la primera.
 - Para tener control sobre la variable `teclaDetectada`, puede darle un valor inicial distinto al de cualquiera de las teclas del *teclado matricial*; así le será más fácil depurar al principio.
- **Configuración de *pines*:**
 - Recuerde que el **valor** de los *pines* de **salida** (los de las columnas) **los controla usted** y debe darle un valor adecuado.
 - **Los valores** de los *pines de entrada* (los de las filas), **no pueden ser asignados**, pero como buenos *telecos* que somos 😊, sabemos que si dejamos **un pin** (que es un *cable*) **al aire, hará de antena**, por lo que estaríamos leyendo ruido electromagnético acoplado. Es por ello que se usan las resistencias de *pull-up* y *pull-down* para fijar valores conocidos ante la falta de un estímulo. En la Figura 5.5 se muestra la configuración que hay en el laboratorio. Observe que, a falta de excitación, la *tierra virtual* fuerza 0V en la GPIO. En este caso **tenemos una resistencia de *pull-down***, pero podríamos tener una de *pull-up*; es por ello que **debemos especificar en qué flanco** (subida, bajada, indiferente) **vamos a leer la pulsación y qué rutina se ejecutará** cuando se interrumpa.
 - Si le es más cómodo, puede crear un array con punteros a las ISR para recorrerlo en la inicialización de las filas,  **como se hace en el**

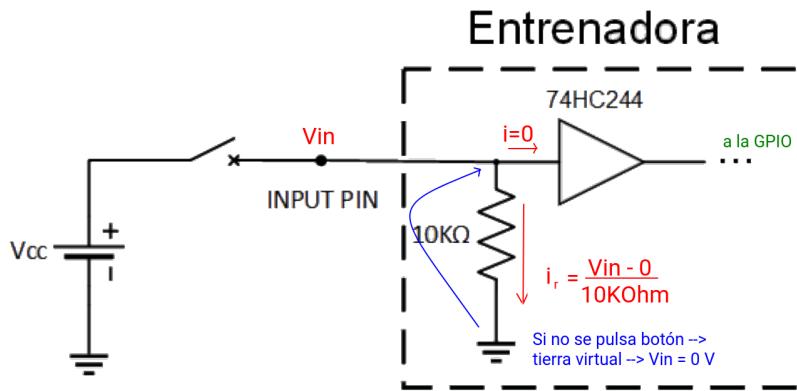


Figura 5.5: Resistencia de *pull-down* a la entrada de la entrenadora.

punto 8.5 del capítulo “6. Iniciación al Manejo de las Entradas/Salidas del BCM 2835”.

- A diferencia del *reloj* en el que la temporización ha de ser estricta y convenía definir el temporizador como periódico, aquí, no. Cree el temporizador no periódico y láncelo justo antes de salir de configurar el teclado. **Más adelante habrá que volverlo a lanzar nuevamente.**

La función que acaba de implementar tiene como novedad la configuración de elementos HW. Tanto si está trabajando en el laboratorio con `wiringPi`, como en emulación con `pseudoWiringPi`, le recomendamos que compruebe que la inicialización es correcta haciendo depuración.

Ahora que están asignadas las ISR a las GPIO de las filas, es buen momento para codificarlas:

11. Codifique las ISR de las filas del *teclado matricial* `teclado_fila_X_isr`. Estas rutinas tienen 4 tareas principales:
 - a) Implementar un mecanismo anti-rebotes SW como el ejemplo del tutorial.
 - b) Guardar la `teclaDetectada`. Para ello, ya se sabe en qué `FILA_X` está (la que da nombre a la ISR), y junto con el valor de la `columnaActual` puede recogerlo del array `tecladoTL04` definido al inicio del fichero.
 - c) Avisar de que se ha leído una tecla.
 - d) Actualizar el tiempo de guarda del mecanismo anti-rebotes.
12. El tiempo de guarda que ayuda a eliminar la detección de rebotes (anti-rebotes, *debounce*) hay que definirlo en milisegundos en la etiqueta `DEBOUNCE_TIME_MS`.

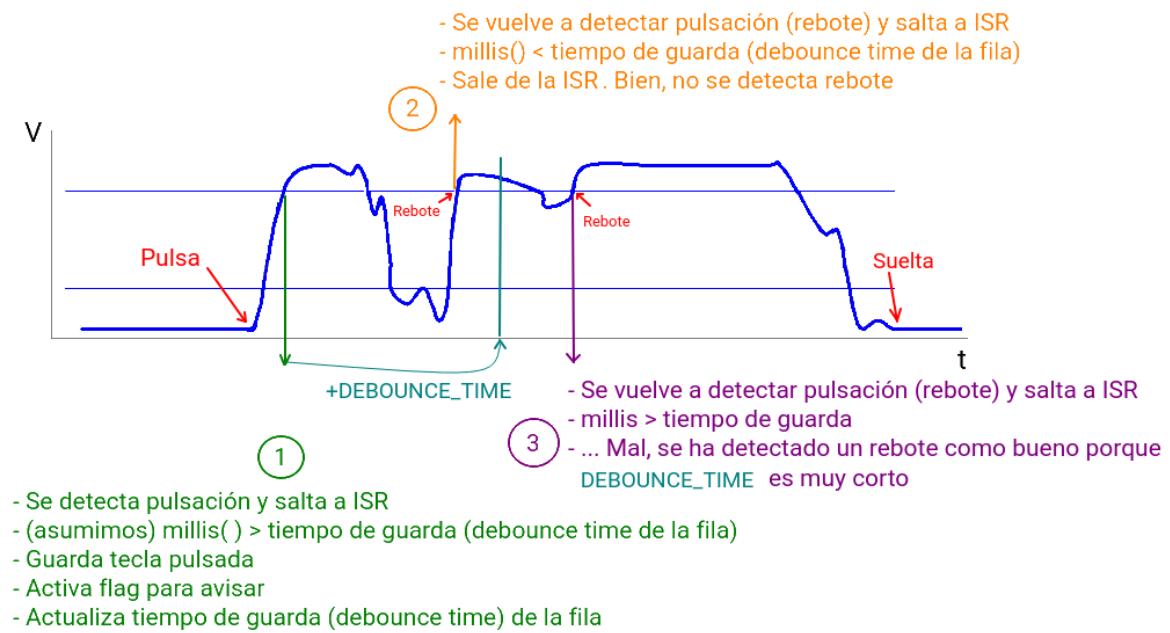


Figura 5.6: Ejemplo de 2 rebotes en una pulsación con tiempo de anti-rebotes bajo.

Fíjese en el supuesto de la Figura 5.6. Hay una pulsación y la mecánica del botón hace que haya 2 rebotes bien definidos. El primer rebote no se detecta porque está dentro del tiempo de guarda. No obstante, este no es suficiente para que sí *se nos cuele* el segundo.

Salvo que tenga mucha experiencia jugando a videojuegos 😊, no seremos muy rápidos marcando teclas. Dejando un tiempo de guarda entre 100 – 200 ms debería ser suficiente, pero quizás deba ajustarlo haciendo pruebas en el laboratorio . Lea el punto 5.6 del capítulo ‘5. Manejo de temporizadores, interrupciones y procesos con la Raspberry Pi’ . Note que en el ejemplo del capítulo hay un reinicio de `debounceTime` que no se contempla en la Figura 5.6.

13. Definamos también la ISR del temporizador de excitación de columnas `timer_duracion_columna_isr` cuya única tarea es levantar el *flag* que indica que ha pasado el tiempo (`timeout`) para activar la siguiente columna.

Ya solo nos queda **implementar la función de salida de la FSM del teclado matricial**.

14. Codifique la función `TecladoExcitaColumna` como se indica en los comentarios del código. No se olvide de proteger las operaciones con el *mutex* correspondiente.
15. Codifique la función `ActualizaExcitacionTecladoGPIO` como se indica en los comentarios del código. Esta función es auxiliar a la anterior y simplemente se

encarga de encender y apagar columnas. Tenga cuidado con el uso de *mutex* para no provocar inter-bloqueos.

16. Si no lo ha hecho ya, asigne valores a las etiquetas `FLAG_TIMEOUT_COLUMNAS_TECLADO` y `FLAG_TECLA_PULSADA`.

¡Ya ha acabado la implementación de la librería del *teclado matricial*! Compile y compruebe que no tiene errores ni *warnings*. El siguiente paso es probar la librería con el sistema `coreWatch`.

5.2. Test de la librería

Ya hemos adelantado algo de trabajo en la sección anterior. Como haríamos con cualquier HW que comprássemos y quisiésemos conectar, hemos (i) elegido a qué GPIOs conectamos el *teclado matricial*, (ii) desarrollado su librería, (iii) añadido el elemento `teclado` a nuestra estructura y (iv) realizado la llamada a la función de configuración. Ahora solo nos queda unir la máquina de estados de la librería con la del sistema.

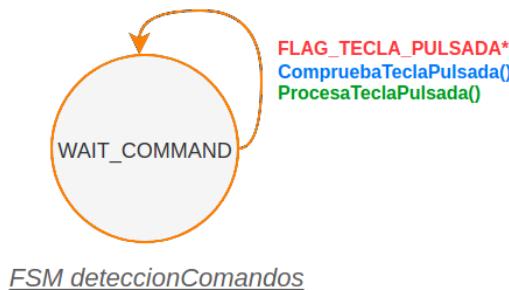


Figura 5.7: FSM de `coreWatch` para detección de comandos.

Primeramente vamos a implementar la segunda máquina de estados del sistema y que habíamos dejado. Para ello:

1. Añada en `coreWatch.h` los prototipos de las funciones `CompruebaTeclaPulsada` y `ProcesaTeclaPulsada` como se definen en la API del Apéndice A.
2. Codifique la función `CompruebaTeclaPulsada`.
3. Codifique la función `ProcesaTeclaPulsada`. No vuelva a programar la lógica de la Figura A.2, cópiela del `thread ThreadExploraTecladoPC`.
4. En `coreWatch.h`, donde definió `FSM_ESTADOS_SISTEMA`, defina el enumerado `FSM_DETECCION_COMANDOS` con el único estado de nuestra máquina `WAIT_COMMAND`.

5. Donde haya definido la tabla de transiciones `fsmTransCoreWatch`, defina la de la *FSM deteccionComandos*: `fsm_trans_t fsmTransDeteccionComandos`, con su única transición. No se olvide del `{-1, NULL, -1, NULL}`.

Ya solo nos queda (i) inicializar la máquina de estados de la *FSM deteccionComando*, (ii) inicializar la máquina de estados del *teclado matricial*, y (iii) ejecutarlas de forma cíclica. Para ello:

4. En la función `main` cree la máquina de estados `fsm_t* deteccionComandosFSM` para detectar teclas pulsadas. Pásele la dirección de memoria de nuestro sistema *coreWatch*.
5. Seguidamente, cree la máquina de estados `fsm_t* tecladoFSM` pasándole la dirección de memoria del `teclado` del sistema. La tabla de transiciones nos la da la librería del teclado en su variable global `g_fsmTransExcitacionColumnas`.
6. No olvidemos lanzar (`fsm_fire`) ambas máquinas en el bucle `while(1)`. Hágalo antes de lanzar la del sistema, puesto que depende de ellas.

No se olvide igualmente de destruir el temporizador `tmr_duracion_columna` del *teclado matricial* y las nuevas máquinas de estado para liberar memoria tal y como hizo al final de la anterior versión.

```
Iniciando la configuracion e inicializacion del reloj...
...reloj configurado e inicializado correctamente.
[pseudoWiringPi][wiringPiSetupGpio para setup de laboratorio seleccionado: Teclado matricial y LCD 2x12]
Iniciando la configuracion e inicializacion del teclado matricial...
[pseudoWiringPi][pinMode][pin 5][OUTPUT]
[pseudoWiringPi][pinMode][pin 6][OUTPUT]
[pseudoWiringPi][pinMode][pin 12][OUTPUT]
[pseudoWiringPi][pinMode][pin 13][OUTPUT]
[pseudoWiringPi][pinMode][pin 5][INPUT]
[pseudoWiringPi][pullUpDnControl][pin 5][PUD_DOWN]
[pseudoWiringPi][pinMode][pin 6][INPUT]
[pseudoWiringPi][pullUpDnControl][pin 6][PUD_DOWN]
[pseudoWiringPi][pinMode][pin 12][INPUT]
[pseudoWiringPi][pullUpDnControl][pin 12][PUD_DOWN]
[pseudoWiringPi][pinMode][pin 13][INPUT]
[pseudoWiringPi][pullUpDnControl][pin 13][PUD_DOWN]
...teclado configurado e inicializado correctamente.
Son las: 0:0:1 del 1/1/2022
Son las: 0:0:2 del 1/1/2022
B
Tecla EXIT pulsada!
Saliendo del programa... BYE BYE!
```

Mensaje exclusivo en EMULACION

Mensajes de WiringPi y pseudoWiringPi cuando se llama a wiringPiSetupGpio()

Figura 5.8: Test del *teclado matricial*. Detalle del *setup* del *driver* de control de GPIOs.

Compile y depure. Compruebe que no tiene errores.

Antes de continuar con la siguiente versión, guarde una copia del proyecto que compile y sea funcional. En la siguiente versión implementaremos utilizaremos el LCD sustituyendo la impresión por la terminal.

Capítulo 6

Versión 4: LCD



Capítulos que hay que tener a mano:

- “1. Introducción al entorno de desarrollo en C para Raspberry Pi”
- “8. Manejo de Display LCD con Raspberry Pi B+”



Vídeos del canal de SDGII:

- Demostración Core Watch

En esta última versión obligatoria del proyecto base no vamos a desarrollar ninguna librería, ni máquina de estados, pues se deja para mejoras (ver Capítulo 7). **En esta versión (i) configuraremos las GPIOs para la conexión con el LCD y (ii) sustituiremos las impresiones por terminal por impresiones por el LCD.**

El *display* de la placa TL04 del laboratorio consta de (i) un controlador muy conocido, el *HD44780* de la empresa Hitachi, y (ii) la pantalla LCD propiamente dicha de 2 filas y 16 columnas (ver Figura 5.3).

En el LCD podemos posicionarnos en una fila o columna concreta, imprimir un carácter, *mandar* a imprimir un texto más largo, borrar la pantalla, parpadear el cursor... Aquí se le va a guiar por lo más básico, como ya habrá visto en algunas funciones de las APIs.

 **Lea el capítulo “8. Manejo de Display LCD con Raspberry Pi B+”, en especial, los puntos 1.4, 1.5 y 2.** Compréndalo. Aquí le guiaremos por los principales pasos. **Lea con atención los apartados Initialisation and Usage y Functions** que son una suerte de API y contenido suficiente para poder implementar los cambios necesarios en el proyecto. Se le facilita una traducción en el Apéndice C.

Procedamos configurando el proyecto.

1. En el fichero de configuración del sistema `systemConfig.h`:

LABORATORIO

Incluya la ruta de la librería `wiringPiDev` en las preferencias del proyecto como se indica en el [punto 2.2 3.e “1. Introducción al entorno de desarrollo en C para Raspberry Pi](#).

En el `#if` de pre-compilación `ENTORNO==ENTORNO_LABORATORIO`, incluya la cabecera de sistema `lcd.h`.

EMULADO

Añada al proyecto los ficheros de emulación del LCD: `pseudoWiringPiDev.h` y `pseudoWiringPiDev.c`.

En el `#if` de pre-compilación `ENTORNO==ENTORNO_EMULADO`, incluya la cabecera `pseudoWiringPiDev.h`.

2. De nuevo, afortunadamente para nosotros, en el laboratorio las conexiones del *display* están preestablecidas. Según la configuración del laboratorio solo podemos escribir en el LCD, y no leer de él, por lo que la faja de cables de la TL04 que nos interesa es la que conecta con las salidas digitales de la entrenadora *ENT2004CF* (faja de la derecha en la Figura 5.3). Dichas salidas están conectadas a los GPIOs de la Raspberry Pi según la tabla de la Figura 6.1.

SALIDAS			
ENT2004CF LEDS VERDES	Número del puerto BCM sólo utilizable como salida	Placa TL-04 conexión salidas entrenador	
			Pin
LVS6	08 (SPI_CS0)	LCD_RS	7
LVS7	10 (SPI_MOSI)	LCD_Enable	8
LVS8	11 (SPI_CLK)	LCD_Bit 0	14
LVS9	14 (UART_TXD)	LCD_Bit1	15
LVS10	17	LCD_Bit 2	16
LVS11	18	LCD_Bit 3	17
LVS12	22	LCD_Bit 4	18
LVS13	23	LCD_Bit 5	19
LVS14	24	LCD_Bit 6	20
LVS15	25	LCD_Bit 7	21

Figura 6.1: GPIOs del microprocesador BCM2835 para el LCD en el laboratorio.

En la cabecera dedicada a la entrenadora, `ent2004cfConfig.h`, defina correctamente los valores de las etiquetas `GPIO_LCD_D...`, `GPIO_LCD_RS` y `GPIO_LCD_EN` que conectan las GPIO con los *pines* de datos, *reset* y *enable* de nuestro LCD.

3. Incluya en la estructura `TipoCoreWatch` el campo `int lcdId` que será el *handle* a nuestro *display*, como se muestra en el Apéndice A.

Ya tenemos la configuración del proyecto. Ahora **procedemos a integrar el LCD**.

4. En el código del sistema ([coreWatch.c](#)) tenemos que decirle a la librería `lcd` (o `pseudoWiringPiDev` si está en emulación) dónde hemos conectado el LCD. Lo haremos en `ConfiguraInicializaSistema`, tras configurar el *teclado matricial* (repase lo relativo a la `VERSION` 4 en la API para `ConfiguraInicializaSistema`).

Llame a la función de inicialización del LCD y recoja el resultado. El resultado es el *handle*, identificador, del *display*. Guárdelo en `lcdId`, pues será el que utilicemos en todas las llamadas a las funciones de la librería. Ya sabe que el número de filas de nuestro *display* es 2 y 12 son las columnas. Puede definir etiquetas para ello. Trabajaremos en *modo 8 bits*. Si más adelante necesita hacer uso de las 4 GPIOs de los *bits* de datos menos significativos (`d0-d3`), use *modo 4 bits*. El resto de parámetros de entrada son las GPIOs que ya hemos definido.

```
p_sistema->lcdId = lcdInit(rows, cols, bits, rs, strb, d0, d1, d2, d3, d4, d5, d6, d7);
```

Compruebe que el *handle* no es -1 y se ha inicializado correctamente. En caso contrario, salga de la función devolviendo un código distinto de 0.

Compruebe que compila correctamente.

5. Por último, **sustituya todas las impresiones por pantalla que se hacen después de la configuración del sistema** por impresiones sobre el LCD. Siga las indicaciones dadas en el Apéndice A para las funciones de salida del *coreWatch* para la `VERSION` 4. Son orientativas, si encuentra una forma más elegante o diferente de interactuar con el usuario, hágalo, mientras no se pierda el objetivo que persigue cada mensaje: `CancelSetNewTime`, `PrepareSetNewTime`, `ProcesaDigitoTime`, (*opcional*) `ProcesaTeclaPulsada`, `Reset` y `ShowTime`.

¡Ya hemos acabado las especificaciones mínimas obligatorias del proyecto! Recuerde que debe funcionar sobre el montaje del laboratorio aunque le funcione en emulación. A partir de aquí, puede avanzar cuanto quiera con la inclusión de mejoras opcionales.

Capítulo 7

Mejoras



Vídeos del canal de SDGII:

- SDGII Proyectos TOP 20/21
- SDGII Proyectos TOP 19/20

En las últimas sesiones puede poner a punto la versión final del sistema e incluir diversas mejoras a su elección. Con ello podrá alcanzar la máxima nota. **Recuerde que (i) el proyecto es el 70 % de la nota total, y (ii) que el proyecto propuesto con especificaciones mínimas obligatorias tiene un máximo de 8 puntos.** Las mejoras le permitirán alcanzar la máxima nota de la parte del proyecto.

7.1. Posibles mejoras

Las sugerencias que se muestran son orientativas. La valoración de la dificultad (de diseño y de implementación) también. No obstante, se valorarán especialmente aquellas mejoras que conlleven algún tipo de esfuerzo por parte del alumno a un mayor número de niveles. Así, se tendrán en cuenta los 3 niveles de la arquitectura típica de cualquier sistema embebido: (i) nivel SW o de aplicación (nivel superior en verde de la pila presentada en la Figura 7.1, nivel HW (nivel inferior en rojo), y nivel de SW de sistema o nivel dedicado a la integración HW-SW (nivel intermedio en naranja), mediante el uso de *drivers* (*i.e.* controladores) SW para el manejo de dispositivos.

Así, el impacto de la mejora se valorará según las escalas de la Figura 7.1, y podría puntuar, orientativamente, como se indica:

- **Solo el primer nivel: BAJA valoración**, por tratarse de una mejora que solo requiere modificar código. **Hasta 0,5 puntos.**

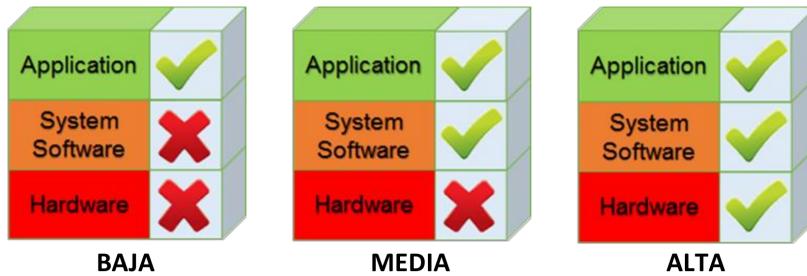


Figura 7.1: Niveles de valoración de las posibles mejoras a implementar.

- **Solo los dos primeros niveles: valoración MEDIA,** por tratarse de una mejora que, además de modificar el código, requiere del uso de algún nuevo recurso de la Raspberry Pi (como por ejemplo otro temporizador, FSM) o de una configuración alternativa de los ya usados (bien sean HW o SW). **Entre 0,5 y 1,0 puntos.**
- **Los tres niveles: valoración ALTA,** por afectar a todos los niveles de diseño del sistema, por lo que es fundamental que se incluya nuevo HW al sistema. **Entre 1,0 y 1,5 puntos.**

En cualquier caso, las mejoras aquí propuestas o las que usted sugiera, pueden implementarse de distintas formas más o menos **complejas** y con distintos grados de perfección (**calidad**) por lo que la calificación final dependerá de ambos aspectos. Igualmente, la **originalidad** o la **novedad** de sus propuestas, será tenida en cuenta para la evaluación. **Hacer una mejora no implica necesariamente conseguir toda la puntuación. Del mismo modo, podrían conseguirse más puntos de los previstos si se merece.**

A continuación se muestra algunas posibles ideas para mejorar el sistema o convertirlo en parte de otro proyecto mayor.

Si tiene dudas sobre la implementación o valoración de estas u otras mejoras, no dude en ponerse en contacto con cualquiera de los profesores de la asignatura.

Descripción	Dificultad
Mejorar FSM sistema: desde <code>SET_TIME</code> poder dar ' <code>F</code> ' y reiniciar	BAJA
Que todos los números se representen con dos dígitos (cero delante cuando son 0-9)	BAJA
Mostrar al usuario el tiempo transcurrido desde que se inició el sistema	BAJA
Mostrar al usuario el día de la semana si se pulsa una tecla	BAJA
Añadir un cronómetro: que se pueda pausar y reanudar, que almacene el tiempo y calcule la media de varias vueltas	BAJA
Considerar AM y PM para el formato de 12 horas, y poder cambiar de formato (<code>SetFormato</code>)	BAJA
Poner un tiempo máximo a las operaciones de <code>SET_TIME</code> para que si pasado un tiempo no se ha introducido un dígito nuevo, vuelva al estado <code>STAND_BY</code>	MEDIA
Permitir modificar la fecha (<code>SetCalendario</code>)	MEDIA
Añadir una alarma que salte dada la hora y minutos (<code>SetAlarma</code>)	MEDIA
Añadir una calculadora	MEDIA
Meter FSM para el LCD	MEDIA
Integrar sensores (movimiento, ritmo cardíaco, temperatura, humedad...), sonido...	ALTA
Integrar otros microcontroladores o placas de desarrollo (tipo Arduino)...	ALTA
Integrar control remoto infra-rojos para cambiar TV, por ejemplo	ALTA
Integrar lector Radio Frequency Identification, Identificación por Radiofrecuencia (RFID)	ALTA
Nuevos esquemas y montajes PCBs	ALTA
Interfaces de visualización alternativa (web, móvil), comunicación con la nube...	ALTA

Cuadro 7.1: Ejemplos de posibles mejoras

Bibliografía

- [1] J. Agustín Sáenz, A. Boscá Mojena, L. Cordero Grande, R. de Córdoba Herralde, Á. D’Haro Enríquez, Luis Fernando Fernández Herrero, F. Fernández Martínez, M. Gil Martín, J. J. Gómez Valverde, C. López Martín, Juan Antonio Luna Jiménez, P. J. Malagón Marzo, J. M. Montero Martínez, J. M. Moya Fernández, J. Pagán Ortiz, J. M. Pardo Muñoz, A. Rodríguez Domínguez, F. Romero Izquierdo, and R. S. Segundo Hernández, *Tutoriales de Sistemas Digitales II*. Departamento de Ingeniería Electrónica, Universidad Politécnica de Madrid, 2022.
- [2] O. Borrás Gené, “Insignias digitales como acreditación de competencias en la universidad,” 2017.
- [3] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk, *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

Parte I

Interfaces de Programación de Aplicaciones (APIs)

Este apéndice contiene las interfaces de programación (APIs) de los principales elementos del sistema: sus funciones y estructuras. Úselas para desarrollar el código según vaya necesitando. El diagrama Unified Modeling Language, Lenguaje Unificado de Modelado (UML) de estructuras del proyecto se muestra en la Figura 2. Este diagrama le sirve para ver la dependencia de estructuras, así como las variables globales *expuestas* por los elementos **reloj** y **teclado** a través de las funciones *getter* y *setter* que implementa cada uno.

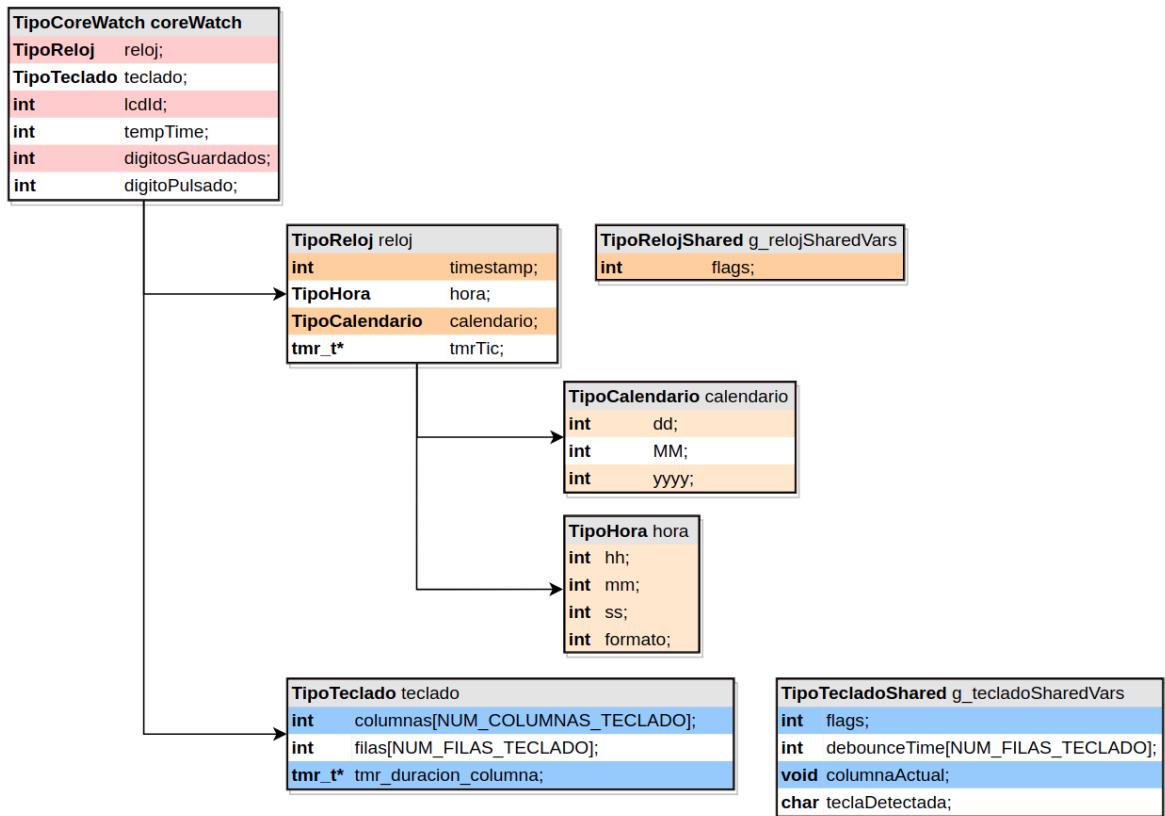


Figura 2: Diagrama UML de estructuras del proyecto.

Apéndice A

API de referencia: coreWatch

Estructuras

[TipoCoreWatch](#)

Funciones de entrada o transición de la máquina de estados

[CompruebaDigitoPulsado](#)
[CompruebaNewTimeIsReady](#)
[CompruebaReset](#)
[CompruebaSetCancelNewTime](#)
[CompruebaSetupDone](#)
[CompruebaTeclaPulsada](#)
[CompruebaTimeActualizado](#)

Funciones de salida o acción de la máquina de estados

[CancelSetNewTime](#)
[PrepareSetNewTime](#)
[ProcesaDigitoTime](#)
[ProcesaTeclaPulsada](#)
[Reset](#)
[SetNewTime](#)
[ShowTime](#)
[Start](#)

Funciones propias

[ConfiguraInicializaSistema](#)
[DelayUntil](#)
[EsNumero](#)

Funciones ligadas a threads adicionales

[ThreadExploraTecladoPC](#)

Estructuras

Se muestran a continuación las principales estructuras que **deben definirse** en el fichero `coreWatch.h`, así como la explicación de algunos `#define`, también necesarios. Podrá añadir más `#define` si los necesita. **Evite poner valores constantes en el código; use etiquetas para ello.**

TipoCoreWatch

`TipoCoreWatch` es la estructura del sistema. Su estructura básica para la `VERSION` 2 no debe incluir el `teclado` ni el `lcdId`, que serán añadidos en las versiones 3 y 4 respectivamente.

```
||  typedef struct {
    TipoReloj reloj;
    TipoTeclado teclado;
    int lcdId;
    int tempTime;
    int digitosGuardados;
    int digitoPulsado;
} TipoCoreWatch;
```

- `reloj`: reloj del sistema.
- `teclado`: teclado matricial del sistema.
- `lcdId`: identificador de nuestra pantalla LCD. Es un *handle* que devuelve la función `lcdInit` de la librería `wiringPiDev` (o `pseudoWiringPiDev` en su defecto) cuando se inicializa y configura una pantalla LCD. La librería puede manejar hasta 8 pantallas distintas.
- `tempTime`: variable auxiliar que guarda la nueva hora de forma temporal mientras el usuario está introduciendo los dígitos de la nueva hora.
- `digitosGuardados`: variable auxiliar que lleva la cuenta de la cantidad de dígitos marcados por el usuario durante un cambio de hora o fecha.
- `digitoPulsado`: variable auxiliar para guardar el último número que haya marcado el usuario.

Funciones de entrada de la maquina de estados

CompruebaDigitoPulsado

```
||  int CompruebaDigitoPulsado(fsm_t* p_this);
```

Función que comprueba si el *flag* (*bit*) `FLAG_DIGITO_PULSADO` de la variable `g_flagsCoreWatch` ha sido activado tras pulsar un número del teclado.

Parámetros:

`p_this: fsm_t*`

No lo usamos. Puntero al contenido de `user_data` de la máquina de estados del *coreWatch*.

Salida:

`result: int`

Valor entero que indica con 0 que no se ha activado el (*flag*) `FLAG_DIGITO_PULSADO`; distinto de 0 si sí se ha activado.

CompruebaNewTimeIsReady

```
||     int CompruebaNewTimeIsReady(fsm_t* p_this);
```

Función que comprueba si el *flag* (*bit*) `FLAG_NEW_TIME_IS_READY` de la variable `g_flagsCoreWatch` ha sido activado tras haber leído los 4 dígitos de una nueva hora.

Parámetros:

`p_this: fsm_t*`

No lo usamos. Puntero al contenido de `user_data` de la máquina de estados del *coreWatch*.

Salida:

`result: int`

Valor entero que indica con 0 que no se ha activado el (*flag*) `FLAG_NEW_TIME_IS_READY`; distinto de 0 si sí se ha activado.

CompruebaReset

```
||     int CompruebaReset(fsm_t* p_this);
```

Función que comprueba si el *flag* (*bit*) `FLAG_RESET` de la variable `g_flagsCoreWatch` ha sido activado, lo que sucede cuando se pulsa la tecla `TECLA_RESET` 'F'.

Parámetros:

`p_this: fsm_t*`

No lo usamos. Puntero al contenido de `user_data` de la máquina de estados del *coreWatch*.

Salida:

`result: int`

Valor entero que indica con 0 que no se ha activado el (*flag*) `FLAG_RESET`; distinto de 0 si sí se ha activado.

CompruebaSetCancelNewTime

```
||     int CompruebaSetCancelNewTime(fsm_t* p_this);
```

Función que comprueba si el *flag* (*bit*) `FLAG_SET_CANCEL_NEW_TIME` de la variable `g_flagsCoreWatch` ha sido activado, lo que sucede cuando se pulsa la tecla `TECLA_SET_CANCEL_TIME` 'E'. La misma tecla sirve tanto para indicar que se quiere hacer un cambio de hora, como para cancelar la operación.

Parámetros:

p_this: `fsm_t*`

No lo usamos. Puntero al contenido de `user_data` de la máquina de estados del *coreWatch*.

Salida:

result: `int`

Valor entero que indica con 0 que no se ha activado el (*flag*) `FLAG_SET_CANCEL_NEW_TIME`; distinto de 0 si sí se ha activado.

CompruebaSetupDone

```
||     int CompruebaSetupDone(fsm_t* p_this);
```

Función que comprueba si el *flag* (*bit*) `FLAG_SETUP_DONE` de la variable `g_flagsCoreWatch` ha sido activado. El flag se activa al finalizar la configuración e inicialización del sistema.

Parámetros:

p_this: `fsm_t*`

No lo usamos. Puntero al contenido de `user_data` de la máquina de estados del *coreWatch*.

Salida:

result: `int`

Valor entero que indica con 0 que no se ha activado el (*flag*) `FLAG_SETUP_DONE`; distinto de 0 si sí se ha activado.

CompruebaTeclaPulsada

```
||     int CompruebaTeclaPulsada (fsm_t* p_this);
```

Función de la FSM `deteccionComandos` para la versión 3 del proyecto. Comprueba si el *flag* (*bit*) `FLAG_TECLA_PULSADA` de la variable `flags` de la estructura `g_tecladoSharedVars` del *teclado* ha sido activado tras pulsar una tecla. Esta función está comprobando el valor de una variable de la librería `teclado_TL04`, por lo que (i) accede a ella a través

de la función *get* del *teclado matricial* y, (ii) comprueba el valor como igual que hacen el resto de funciones de comprobación.

Parámetros:

p_this: `fsm_t*`

No lo usamos. Puntero al contenido de `user_data` de la máquina de estados del *coreWatch*.

Salida:

result: `int`

Valor entero que indica con 0 que no se ha activado el (*flag*) `FLAG_TECLA_PULSADA` del *teclado matricial*; distinto de 0 si sí se ha activado.

CompruebaTimeActualizado

```
||     int CompruebaTimeActualizado(fsm_t* p_this);
```

Función que comprueba si el *flag* (*bit*) `FLAG_TIME_ACTUALIZADO` de la variable `flags` de la estructura `g_relojSharedVars` del *reloj* ha sido activado por el temporizador del mismo. Esta función está comprobando el valor de una variable de la librería `reloj`, por lo que (i) accede a ella a través de la función *get* del *reloj* y, (ii) comprueba el valor como hacen el resto de funciones de comprobación.

Parámetros:

p_this: `fsm_t*`

No lo usamos. Puntero al contenido de `user_data` de la máquina de estados del *coreWatch*.

Salida:

result: `int`

Valor entero que indica con 0 que no se ha activado el (*flag*) `FLAG_TIME_ACTUALIZADO` del *reloj*; distinto de 0 si sí se ha activado.

Funciones de salida de la máquina de estados

CancelSetNewTime

```
||     void CancelSetNewTime(fsm_t* p_this);
```

Función que cancela la lectura de una nueva hora que se está introduciendo por parte del usuario; por ejemplo, si nos hemos equivocado. Para ello:

1. Recupera el *coreWatch* de tipo `TipoCoreWatch` haciendo un *cast* del argumento `p_this` recibido. Al sistema *coreWatch* recuperado podemos llamarle `*p_sistema`, por ejemplo.

2. Limpia las variables auxiliares que hayan podido ser modificadas: `tempTime` y `digitosGuardados`.
3. Limpia el *flag* `FLAG_SET_CANCEL_NEW_TIME` que le ha hecho llegar hasta aquí.
4. Informa al usuario de que se ha cancelado la operación haciendo uso del *mutex* correspondiente. Dependiendo de la versión:
 - `VERSION < 4`: Imprime por la pantalla de la terminal "`[SET_TIME] Operacion cancelada\n`".
 - `VERSION >= 4`:
 - a) Limpia el LCD.
 - b) Coloca el cursor en la primera columna de la segunda fila.
 - c) Imprime "`CANCELADO`" y espera durante `ESPERA_MENSAJE_MS` milisegundos antes de limpiar de nuevo el LCD.

Parámetros:

`p_this: fsm_t*`

Puntero al contenido de `user_data` de la máquina de estados del *coreWatch*.

Salida:

-

PrepareSetNewTime

```
|| void PrepareSetNewTime(fsm_t* p_this);
```

Es una función *de paso*, de transición. Solo informa al usuario de que debe introducir una nueva hora y el formato (12 o 24) en el que se espera.

1. Recupera el *coreWatch* de tipo `TipoCoreWatch`. Para ello hace un *cast* del argumento `p_this` recibido. Al sistema *coreWatch* recibido podemos llamarle `*p_sistema`, por ejemplo.
2. Recupera el `formato` de la `hora` del `reloj` del sistema recuperado.
3. Por seguridad, limpia el *flag* `FLAG_DIGITO_PULSADO` para asegurar que no esté activo por haber presionado algún número previamente.
4. Limpia el *flag* `FLAG_SET_CANCEL_NEW_TIME` que le ha hecho llegar hasta aquí.
5. Informa al usuario de que ha de introducirse una nueva hora y el formato esperado (haciendo uso del *mutex* correspondiente). Dependiendo de la versión:

- `VERSION < 4:` Imprime por la pantalla de la terminal "`[SET_TIME] Introduzca la nueva hora en formato 0-%d\n`". Donde `%d` imprimirá el `formato` recogido.
- `VERSION >= 4:`
 - a) Limpia el LCD.
 - b) Coloca el cursor en la primera columna de la segunda fila.
 - c) Imprime "`FORMAT: 0-%d`". Donde `%d` imprimirá el `formato` recogido.

Parámetros:

`p_this: fsm_t*`

Puntero al contenido de `user_data` de la máquina de estados del `coreWatch`.

Salida:

-

ProcesaDigitoTime

```
|| void ProcesaDigitoTime(fsm_t* p_this);
```

Función que implementa la lógica de recoger los dígitos marcados para acomodarlos a un `int` (`tempTime` de la estructura recibida) con la forma *HHMM* (*HH*: horas, *MM*: minutos). Para ello:

1. Recupera el `coreWatch` de tipo `TipoCoreWatch` haciendo un *cast* del argumento `p_this` recibido. Al sistema `coreWatch` recuperado podemos llamarle `*p_sistema`, por ejemplo.
2. (*Opcional*) Recupera las variables auxiliares `tempTime` y `digitosGuardados` y las guarda en variables locales para trabajar de forma más cómoda.
3. Por comodidad, puede copiar en la variable local `int ultimoDigito` el valor de la estructura del sistema `digitoPulsado`. En esa variable está el último dígito marcado (números, no letras). Dependiendo de la versión:
 - `VERSION == 2:` la última tecla pulsada la habrá guardado el `thread ThreadExploraTecladoPC` en la variable de la estructura `digitoPulsado` como un `int`.
 - `VERSION >= 3:` la última tecla pulsada la habrá guardado la función `ProcesaTeclaPulsada`, que la ha recogido de la estructura compartida `g_tecladoSharedVars` del `teclado`. El acceso a `g_tecladoSharedVars` se hace con el `get` de la librería `teclado_TL04`.
4. Limpia el `flag FLAG_DIGITO_PULSADO` que le ha hecho llegar hasta aquí.

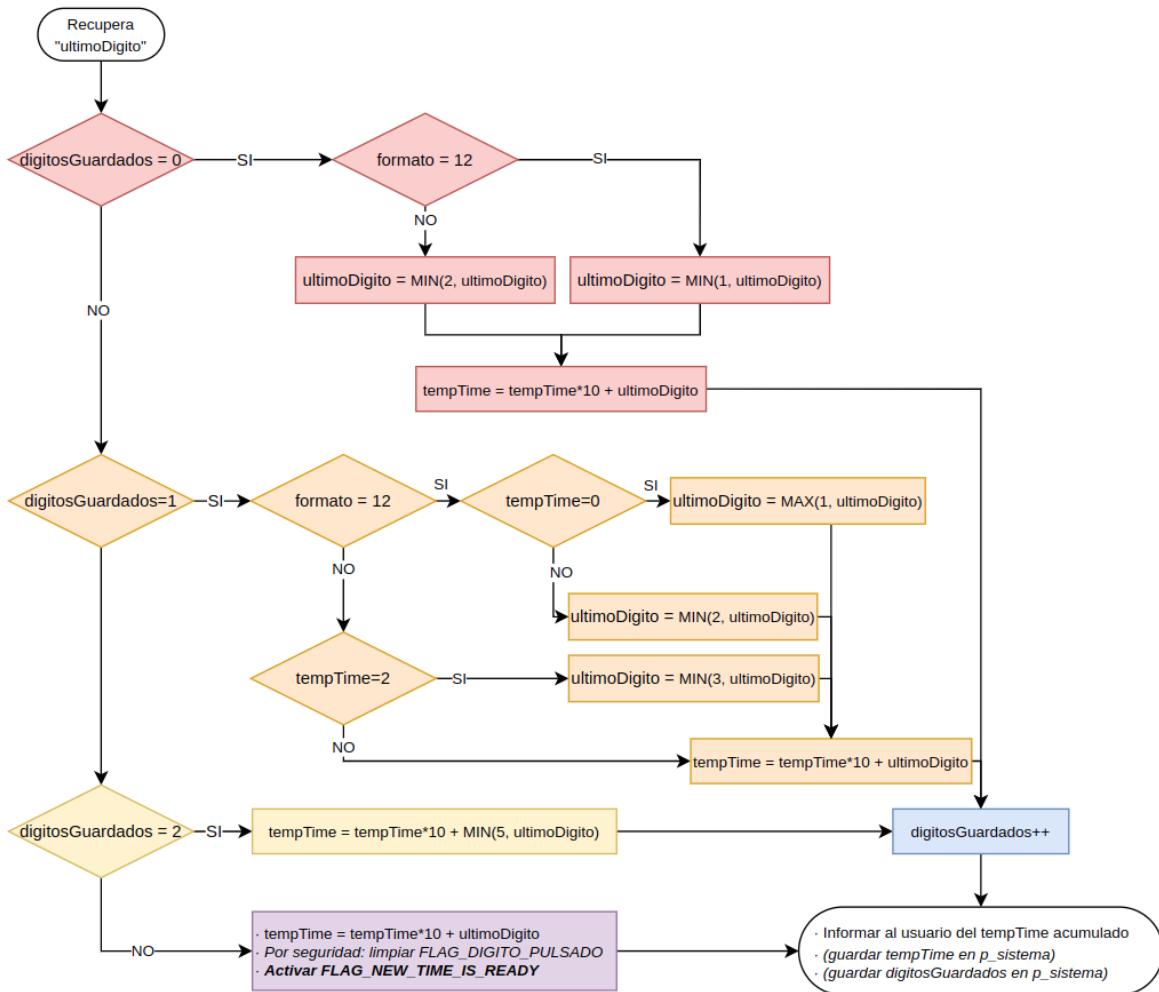


Figura A.1: Diagrama de flujo de la función [ProcesaDigitoTime](#).

5. Implementa el diagrama de flujo de la Figura A.1 para guardar la hora temporalmente (`tempTime`):

- Por seguridad, limpiamos el *flag* `FLAG_DIGITO_PULSADO` por si, en caso de estar procesando el último dígito, hubiese habido un rebote mecánico que haya levantado de nuevo el *flag*. De otra manera, al estar el *flag* activo, entraría de nuevo en la función y: (i) o estaría esperando hasta acumular 8 dígitos, o (ii) `SetNewTime` pondría una hora de un solo dígito. *Basado en la heurística*.
- Activa el *flag* `FLAG_NEW_TIME_IS_READY` para avisar a la FSM y salir del estado `SET_TIME`.
- (*Opcional*) *Si por causalidades del rebote de las teclas del teclado matricial, o cualquier motivo, `tempTime` tiene más de 4 dígitos, se van desplazando a la izquierda los dígitos ya guardados. Ejemplo:*

```

|| if (digitosGuardados < 3) {
    ||   if (tempTime > 2359) {

```

```

    tempTime %= 10000;
    tempTime = 100* MIN((int)(tempTime/100), 23) + MIN(
        tempTime%100, 59);
}
}

```

- Para informar al usuario de lo que va marcando y que tenga una realimentación visual, se le informa, como indica el diagrama de flujo. Usando los *mutex* correspondientes y dependiendo de la versión:
 - **VERSION < 4:** Imprime por la pantalla de la terminal "**[SET_TIME] Nueva hora temporal %d\n**". Donde **%d** imprimirá el tiempo acumulado **tempTime**.
 - **VERSION >= 4:**
 - a) Limpia la primera línea del LCD (imprime espacios en blanco).
 - b) Imprime en la primera línea: "**SET: %d**" . Donde **%d** imprimirá el tiempo acumulado **tempTime**.
- Si se está trabajando con una copia local de **tempTime** y **digitosGuardados** (como se indica opcionalmente en el punto 2), hay que guardar estos valores locales en la estructura del sistema **p_sistema**.

Parámetros:

p_this: fsm_t*

Puntero al contenido de **user_data** de la máquina de estados del *coreWatch*.

Salida:

-

ProcesaTeclaPulsada

```

|| void ProcesaTeclaPulsada (fsm_t* p_this);

```

Función de la FSM **deteccionComandos** para la **VERSION 3** del proyecto. Implementa la misma lógica que el **thread** de la **VERSION 2**, que es la de la Figura A.2. Esta función:

1. Recupera el *coreWatch* de tipo **TipoCoreWatch** haciendo un *cast* del argumento **p_this** recibido. Al sistema *coreWatch* recuperado podemos llamarle ***p_sistema**, por ejemplo.
2. Limpia el *flag* **FLAG_TECLA_PULSADA** de la librería del *teclado matricial*, para lo que (i) recoge la variable **g_tecladoSharedVars** mediante *get*, (ii) limpia el *flag*, lo guarda de nuevo mediante *set* de la librería **teclado_TL04**.
3. De la variable recogida **g_tecladoSharedVars** guarda el valor de la **teclaDetectada** en una variable local de tipo **char** que llamamos **teclaPulsada**.

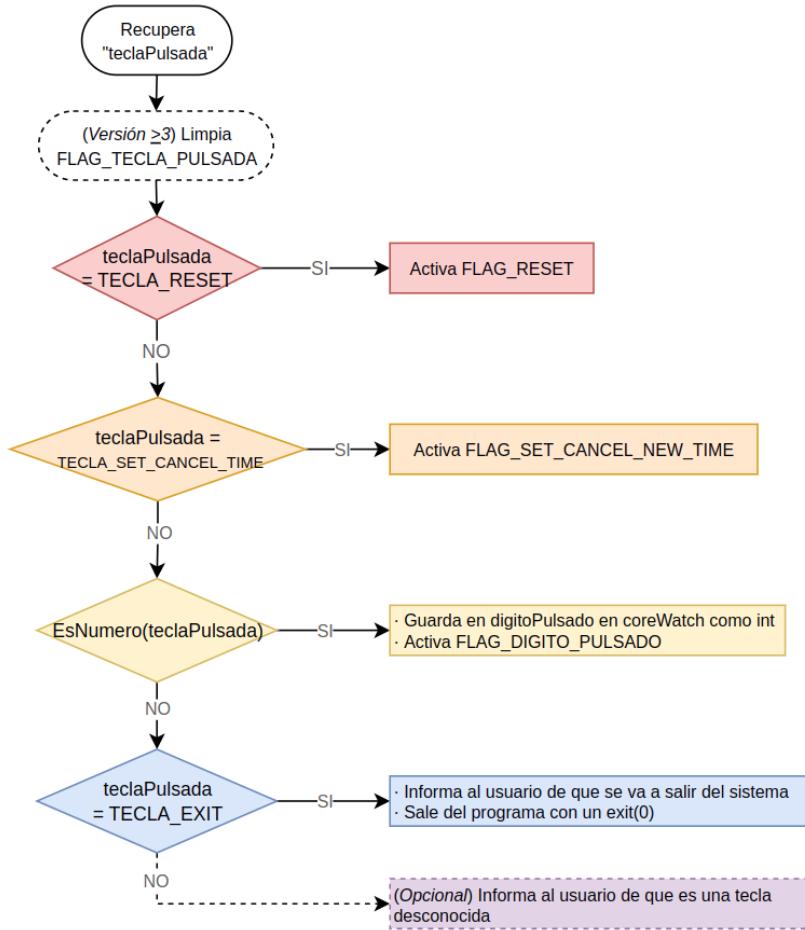


Figura A.2: Diagrama de flujo de la función `ProcesaTeclaPulsada` y el `thread ThreadExploraTecladoPC`.

4. Implementa toda la misma lógica que el `thread` de la `VERSION 2`.

Parámetros:

`p_this: fsm_t*`

Puntero al contenido de `user_data` de la máquina de estados del `coreWatch`.

Salida:

-

Reset

```
|| void Reset(fsm_t* p_this);
```

Función que *resetea* el `reloj` del sistema. Para ello:

1. Recupera el `coreWatch` de tipo `TipoCoreWatch` haciendo un *cast* del argumento `p_this` recibido. Al sistema `coreWatch` recuperado podemos llamarle `*p_sistema`, por ejemplo.

2. Resetea la hora a los valores por defecto. Para ello solo ha de llamar a `ResetReloj` pasándole la dirección de memoria del `reloj` del sistema recibido.
3. Limpia el flag `FLAG_RESET` que le ha hecho llegar hasta aquí.
4. Informa al usuario de que se ha realizado un *reset* correctamente. Haciendo uso del *mutex* correspondiente y dependiendo de la versión:
 - `VERSION < 4`: Imprime por la pantalla de la terminal "`[RESET] Hora reiniciada \n`".
 - `VERSION >= 4`:
 - a) Limpia el LCD.
 - b) Coloca el cursor en la primera columna de la segunda fila.
 - c) Imprime "`RESET`" y espera durante `ESPERA_MENSAJE_MS` milisegundos antes de limpiar de nuevo el LCD.

Parámetros:

`p_this: fsm_t*`

Puntero al contenido de `user_data` de la máquina de estados del `coreWatch`.

Salida:

-

SetNewTime

```
||     void SetNewTime(fsm_t* p_this);
```

Función que guarda en el `reloj` del sistema la hora temporal que ha introducido el usuario. Para ello:

1. Recupera el `coreWatch` de tipo `TipoCoreWatch` haciendo un *cast* del argumento `p_this` recibido. Al sistema `coreWatch` recuperado podemos llamarle `*p_sistema`, por ejemplo.
2. Limpia el flag `FLAG_NEW_TIME_IS_READY` que le ha hecho llegar hasta aquí.
3. Guarda la nueva hora. Para ello llama a la función `SetHora` pasándole la hora temporal `tempTime` y la dirección de memoria del `reloj` del sistema `p_sistema`.
4. Por seguridad, se limpian (ponen a 0) las variables auxiliares `tempTime` y `digitosGuardados`.

Parámetros:

p_this: `fsm_t*`

Puntero al contenido de `user_data` de la máquina de estados del *coreWatch*.

Salida:

-

ShowTime

```
|| void ShowTime(fsm_t* p_this);
```

Función que muestra la hora y fecha al usuario. Para ello:

1. Recupera el *coreWatch* de tipo `TipoCoreWatch`. Para ello hace un *cast* del argumento `p_this` pasado como `user_data` de la FSM. Al sistema *coreWatch* recibido podemos llamarle `*p_sistema`, por ejemplo.
2. Limpia el *flag* que le ha hecho llegar hasta aquí. El *flag* ha sido activado por la FSM del *reloj* en la variable compartida `g_relojSharedVars`. No olvide proteger con *mutex* el acceso a *flags*. Para limpiar el *flag*:
 - a) Recoge la estructura `g_relojSharedVars` de tipo `TipoRelojShared` llamando a la función `GetRelojSharedVar`.
 - b) Modifica la variable `flags` recibida en la estructura.
 - c) Guarda la estructura modificada llamando a la función `SetRelojSharedVar`.
3. Muestra la hora al usuario. Haciendo uso del *mutex* correspondiente y dependiendo de la versión:
 - `VERSION < 4`: Imprime por la pantalla el mismo mensaje que aparece en la función `ActualizaReloj` para la `VERSION 1`.
 - `VERSION >= 4`:
 - a) Recupera el LCD del sistema (el identificador o *handler* `lcdId`) en una variable local.
 - b) Limpia el LCD.
 - c) Imprime hora, minutos y segundos separados por ":" : "%d: %d: %d" en la primera fila.
 - d) Coloca el cursor en la primera columna de la segunda fila.
 - e) Imprime día, mes y año separados por "/" : "%d/ %d/ %d".
 - f) Devuelve el cursor a la primera fila y primera columna.

Parámetros:

p_this: `fsm_t*`

Puntero al contenido de `user_data` de la máquina de estados del *core Watch*.

Salida:

-

Start

```
|| void Start(fsm_t* p_this);
```

Función de auxiliar que simplemente limpia el *flag* `FLAG_SETUP_DONE` que la función `ConfiguraInicializaSistema` levanta para indicar que el sistema se ha iniciado y configurado correctamente.

Parámetros:

p_this: `fsm_t*`

No lo usamos. Puntero al contenido de `user_data` de la máquina de estados del *core Watch*.

Salida:

-

Funciones propias

ConfiguraInicializaSistema

```
|| int ConfiguraInicializaSistema (TipoCoreWatch *p_sistema);
```

Función que se encarga de inicializar los elementos y variables del sistema. Según la versión:

■ `VERSION >= 2:`

1. Inicializa el valor de la variable global `g_flagsCoreWatch` a 0.
2. Inicializa los valores de las variables auxiliares `tempTime` y `digitosGuardados` del sistema recibido a 0.
3. Inicializa el `reloj`, para lo que llama a la función `ConfiguraInicializaReloj` pasándole la dirección de memoria del `reloj` del sistema `p_sistema` recibido. Recoge el resultado de la inicialización.
4. Comprueba el resultado de la inicialización del `reloj` y en caso de ser distinto de 0 sale de la función (`return`) con algún código (`int`) distinto de 0.
5. Lanza el `thread` de exploración del teclado llamando a `piThreadCreate(ThreadExploraTecladoPC)` y recoge el resultado de la llamada.

6. Comprueba el resultado de la inicialización del `thread` y en caso de no ser 0 sale de la función (`return`) con algún código (`int`) distinto de 0.

- **VERSION >= 3:**

7. Inicializa el `teclado`, para lo que llama a la función `ConfiguraInicializaTeclado` pasándole la dirección de memoria del `teclado` del sistema `p_sistema` recibido.

- **VERSION >= 4:**

8. Inicializa el `lcd`, para lo que llama a la función `lcdInit` de la librería `lcd` y guarda el valor devuelto del `handle` en la variable `lcdId` del sistema `p_sistema` recibido. Para la configuración utiliza los siguientes parámetros:

- Número de filas: `NUM_ROWS` 2
- Número de columnas: `NUM_COLS` 12
- Número de bits de la comunicación: `NUM_BITS` 8
- GPIOs: acordes a las conexiones que representan

```

int rs = GPIO_LCD_RS;
int strb = GPIO_LCD_EN;
int d0 = GPIO_LCD_D0;
int d1 = GPIO_LCD_D1;
int d2 = GPIO_LCD_D2;
int d3 = GPIO_LCD_D3;
int d4 = GPIO_LCD_D4;
int d5 = GPIO_LCD_D5;
int d6 = GPIO_LCD_D6;
int d7 = GPIO_LCD_D7;

```

Las etiquetas de los GPIOs han de definirse según las conexiones del laboratorio con la Raspberry Pi, según están se muestran en la Figura 6.1 y que están definidas en el fichero de configuración de la entrenadora `ent2004cfConfig.h`.

- Por último, en todas las versiones levanta el *flag* `FLAG_SETUP_DONE` para avisar de que la inicialización es correcta y poder pasar del estado `START` a `STAND_BY`.

Aprovecha en cada punto para imprimir mensajes por pantalla sobre la evolución de la inicialización (haciendo de *mutex*).

Parámetros:

p_this: `fsm_t*`

Es un puntero a la dirección de memoria de nuestro sistema *coreWatch* de tipo `TipoCoreWatch`.

Salida:

result: int

Entero que indica que la inicialización ha sido exitosa (0), o no (otro valor). Esto permite hacer detección de errores, si se considera.

DelayUntil

```
|| void DelayUntil(unsigned int next);
```

Función que realiza la espera activa (`delay`) hasta el *timestamp* indicado en milisegundos.

Parámetros:

next: unsigned int

Valor entero sin signo que representa el instante de tiempo (*timestamp*) hasta el que hay que esperar.

Salida:

-

EsNúmero

```
|| int EsNúmero(char value);
```

Función que comprueba si un valor pasado como `char` es un número (del 0 al 9), o no.

Una forma muy sencilla para comprobar que un `char` es un número, es fijarse en sus valores en hexadecimal. Para ello nos fijamos en la tabla de códigos ASCII de la Figura A.3. Observe que el valor hexadecimal del carácter 0 es `0x30` (48 en decimal), y que el valor del carácter 9 es `0x39` (57 en decimal), por lo que con simples operaciones podemos saber si el `char` recibido hace referencia a un número, o no.

Dec	Hex													
0	00	NUL	16	10	DLE	32	20	48	30	0	64	40	@	
1	01	SOH	17	11	DC1	33	21	49	31	1	65	41	A	
2	02	STX	18	12	DC2	34	22	50	32	2	66	42	B	
3	03	ETX	19	13	DC3	35	23	51	33	3	67	43	C	
4	04	EOT	20	14	DC4	36	24	52	34	4	68	44	D	
5	05	ENQ	21	15	NAK	37	25	53	35	5	69	45	E	
6	06	ACK	22	16	SYN	38	26	54	36	6	70	46	F	
7	07	BEL	23	17	ETB	39	27	55	37	7	71	47	G	
8	08	BS	24	18	CAN	40	28	56	38	8	72	48	H	
9	09	HT	25	19	EM	41	29	57	39	9	73	49	I	
10	0A	LF	26	1A	SUB	42	2A	*	58	3A	:	74	4A	J
11	0B	VT	27	1B	ESC	43	2B	+	59	3B	;	75	4B	K
12	0C	FF	28	1C	FS	44	2C	,	60	3C	<	76	4C	L
13	0D	CR	29	1D	GS	45	2D	-	61	3D	=	77	4D	M
14	0E	SO	30	1E	RS	46	2E	.	62	3E	>	78	4E	N
15	0F	SI	31	1F	US	47	2F	/	63	3F	?	79	4F	O
												95	5F	_
												111	6F	o
												127	7F	DEL

Figura A.3: Tabla ASCII en la que se resaltan los códigos de los números del 0 al 9.

Parámetros:**value:** `char`Valor que contiene el `char` a comprobar.**Salida:****result:** `int`

Entero que indica con un 0 que el carácter recibido no es un número, y con cualquier otro valor indica que sí lo es.

Funciones ligadas a threads adicionales

ThreadExploraTecladoPC

```
|| PI_THREAD(ThreadExploraTecladoPC);
```

Esta función se ejecuta de forma paralela al programa principal en un hilo (`thread`). Su cometido es (i) leer pulsaciones de las teclas del *teclado del PC*, (ii) interpretar la tecla y activar *flags* del sistema, y (iii) avisar al usuario, si es necesario. Se implementa dentro de un `while(1)` porque su ejecución es constante.

Su uso es exclusivo de la versión 2. No obstante, toda la lógica y (casi) toda su implementación es la misma que la de la función `ProcesaTeclaPulsada` de la versión 3.

La lógica que implementa es la del diagrama de flujo de la Figura A.2. Para ello, en un `while(1)`:

1. Hace un `delay` de 10 milisegundos, necesario según indicación de la librería `wiringPi`, que gestiona la concurrencia.
2. Comprueba si se ha pulsado una tecla. Para ello, mira si el valor devuelto por la función `kbhit()` (de la librería `kbhit`) es distinto de 0.
3. Si se ha pulsado una tecla, recupera la tecla pulsada en la variable `int teclaPulsada`.
4. Implementa la lógica de la Figura A.2.

Tenga

Parámetros:**Salida:**

Apéndice B

API de referencia: reloj

Estructuras

[TipoCalendario](#)

[TipoHora](#)

[TipoReloj](#)

[TipoRelojShared](#)

Funciones de entrada o transición de la máquina de estados

[CompruebaTic](#)

Funciones de salida o acción de la máquina de estados

[ActualizaReloj](#)

Subrutinas de atención a las interrupciones

[tmr_actualiza_reloj_isr](#)

Funciones propias

[ActualizaFecha](#)

[ActualizaHora](#)

[CalculaDiasMes](#)

[ConfiguraInicializaReloj](#)

[EsBisiesto](#)

[GetRelojSharedVar](#)

[ResetReloj](#)

[SetFecha](#)

[SetFormato](#)

[SetHora](#)

[SetRelojSharedVar](#)

Estructuras

Se muestran a continuación las principales estructuras que **deben definirse** en el fichero `reloj.h`, así como la explicación de algunos `#define`, también necesarios. Podrá

añadir más `#define` si los necesita. **Evite poner valores constantes en el código; use etiquetas para ello.**

TipoCalendario

`TipoCalendario` es una estructura que contiene tres `int` con el día, mes, y año actuales. Debe definirse como:

```
typedef struct {
    int dd;
    int MM;
    int yyyy;
} TipoCalendario;
```

- `dd`: día del mes. El valor más alto que puede tomar depende del mes en el que estemos (atención con el mes de febrero en años bisiestos, ver `EsBisiesto`). El valor más bajo es `MIN_DAY` 1.
- `MM`: mes del año. El valor más alto que puede tomar es `MAX_MONTH` 12. El valor más bajo es `MIN_MONTH` 1.
- `yyyy`: año representado con 4 dígitos. El valor más bajo que elegimos guardar es `MIN_YEAR` 1970¹.

`MIN_DAY`, `MAX_MONTH`, `MIN_MONTH`, y `MIN_YEAR` nos permiten comprobar que los valores introducidos al fijar una fecha, tienen sentido.

TipoHora

`TipoHora` es una estructura que representa *el tiempo* actual (hora, minutos y segundos), así como el formato de representación de la hora. Ha de ser definida como:

```
typedef struct {
    int hh;
    int mm;
    int ss;
    int formato;
} TipoHora;
```

- `hh`: hora del reloj. Los valores mínimos y máximos (`MIN_HOUR`, `MAX_HOUR`) son respectivamente: (1,12) si `formato`=12, y (0,23) si `formato`=24.

¹El 1 de enero de 1970 a las 00:00:00 UTC se conoce como el Unix epoch. Unix eligió esa fecha de forma arbitraria para establecer una fecha uniforme como inicio del tiempo, y el día de año nuevo de 1970 pareció el más conveniente.

- **mm**: minutos del reloj. El valor más alto que puede tomar es **MAX_MIN** de 59.
- **ss**: segundos del reloj. El valor más alto que puede tomar es 59.
- **formato**: es un entero que toma los valores 12 o 24 y que habrá que definir en las etiquetas **TIME_FORMAT_12_H** y **TIME_FORMAT_24_H**.

MIN_HOUR, **MAX_HOUR**, y **MAX_MIN** nos permiten comprobar que los valores introducidos al fijar una hora, tienen sentido.

TipoReloj

TipoReloj es una estructura que representa el *reloj* del *Core Watch* y contiene el calendario, hora, temporizador, y *timestamp* del sistema. Ha de ser definida como:

```
||| typedef struct {
|||     int timestamp;
|||     TipoHora hora;
|||     TipoCalendario calendario;
|||     tmr_t* tmrTic;
||| } TipoReloj;
```

- **timestamp**: es un entero que guarda la información de segundos que han pasado desde que se arrancó el sistema. Como el tamaño de un **int** es de 32 bits, puede almacenar hasta $2^{32} = 136$ años, aproximadamente.
- **hora**: es una estructura que sirve para almacenar las horas, minutos y segundos.
- **calendario**: es una estructura que sirve para almacenar el día, mes y año.
- ***tmrTic**: es un puntero a un temporizador; temporizador que avisa de que es momento de actualizar la hora. La precisión está condicionada por **PRECISION_RELOJ_MS**. Está en la estructura y no como variable interna de la librería para que pueda ser destruido (**tmrDestroy**) cuando se acabe el programa principal del sistema.

La etiqueta **PRECISION_RELOJ_MS** indica, **en milisegundos**, el tiempo de actualización de nuestro reloj: **1 segundo**. Ha de ser definida en **reloj.h**.

TipoRelojShared

TipoRelojShared es una estructura que contiene variables que expone la librería **reloj** para comunicar *eventos* o *el estado* mediante las funciones **GetRelojSharedVar** y **SetRelojSharedVar**. La variable que se defina de este tipo ha de ser global, porque:

(i) estas variables han de ser accedidas por la rutina de atención a la interrupción del reloj `tmr_actualiza_reloj_isr` a la que, por su definición, no podemos pasarle otros parámetros, o (ii) se comparte entre elementos del sistema mediante *getters* y *setters*. Se define como:

```
||     typedef struct {
||         int flags;
||     } TipoRelojShared;
```

- `flags`: variable entera de la que se usarán sus *bits* para notificar eventos del reloj.

Según lo anterior, tendremos una variable global `g_relojSharedVars` de tipo `TipoRelojShared` definida en `reloj.c`.

Funciones de entrada de la máquina de estados

CompruebaTic

```
||     int CompruebaTic(fsm_t* p_this);
```

Función que comprueba si el *flag* (*bit*) `FLAG_ACTUALIZA_RELOJ` de la variable `flags` de la estructura `g_relojSharedVars` ha sido activado por el temporizador. Hace uso del *mutex* correspondiente del *reloj*.

Parámetros:

`p_this: fsm_t*`

No lo usamos. Puntero al contenido de `user_data` de la máquina de estados del *reloj*.

Salida:

`result: int`

Valor entero que indica con 0 que no se ha activado el (*flag*) `FLAG_ACTUALIZA_RELOJ`; distinto de 0 si sí se ha activado.

Funciones de salida de la máquina de estados

ActualizaReloj

```
||     void ActualizaReloj (fsm_t* p_this);
```

Función que actualiza la hora y la fecha del *reloj* del *Core Watch*. Hace uso del *mutex* correspondiente del *reloj* cuando consulta y modifica su variable global. Esta función realiza, estas acciones:

1. Recupera el *reloj* de tipo `TipoReloj`. Para ello hace un *cast* del argumento `p_this` pasado como `user_data` de la FSM. Al *reloj* recibido podemos llamarle `*p_miReloj`, por ejemplo.
2. Aumenta **en 1** el `timestamp` del reloj recibido.
3. Llama a la función `ActualizaHora` pasándole la dirección de memoria de la `hora` del reloj recibido.
4. Llama a la función `ActualizaFecha`, si procede. Solo se llama si la hora, minutos y segundos de la `hora` del reloj son 0 tras su actualización.
5. Limpia el flag `FLAG_ACTUALIZA_RELOJ` que le ha hecho llegar hasta aquí.
6. Avisa al sistema de que se ha actualizado la hora. Ello se hace activando el flag `FLAG_TIME_ACTUALIZADO` que más tarde leerá `coreWatch`.
7. *Si el temporizador se ha definido periódico, no hay que hacer nada más. Si el temporizador se ha creado para que solo salte una vez, este es el punto en el que hay que lanzarlo de nuevo* (ver [ConfiguraInicializaReloj](#)).

Parámetros:

`p_this: fsm_t*`

Puntero al contenido de `user_data` de la máquina de estados del *reloj*.

Salida:

-

Subrutinas de atención a las interrupciones

tmr_actualiza_reloj_isr

```
||     void tmr_actualiza_reloj_isr (union sigval value);
```

Subrutina de atención a la interrupción del temporizador periódico que marca el *tic* de **1 segundo** del *reloj*. Activa el flag (*bit*) `FLAG_ACTUALIZA_RELOJ` de la variable `flags` de la estructura `g_relojSharedVars`. Hace uso del *mutex* correspondiente del *reloj*.

Parámetros:

`value: union sigval`

`value` es una variable de tipo `union sigval` y contiene el valor asociado a la señal (`SIGEV_SIGNAL`), o argumento para el *thread* (`SIGEV_THREAD`) que ha hecho saltar al temporizador y llegar hasta aquí. Este valor no lo usaremos, generalmente².

²Esta unión está definida en la cabecera del sistema `_sigval_t.h`

Salida:

-

Funciones propias

ActualizaFecha

```
|| void ActualizaFecha(TipoCalendario *p_fecha);
```

Actualiza la fecha (día (`dd`), mes (`MM`), y año (`yyyy`)) de la estructura recibida `p_fecha`, según los siguientes pasos:

1. Llama a la función `CalculaDiasMes` con el mes y el año actual (variables `MM` y `yyyy`) para calcular cuántos días tiene el mes actual.
2. Actualiza el día con *módulo (%) días del mes que estemos*. Para ello:
 - a) Toma el *día actual+1* (variable `dd`).
 - b) Le aplica el módulo *días del mes que estemos+1*.
 - c) Se queda con el máximo entre el resultado anterior y 1. Puede ayudarse de la macro `MAX` definida en `util.h`.

Ejemplo: qué pasa del 30 al 31 de enero:

Día actual: 30
 Días del mes actual: 31
 Módulo *días del mes que estemos + 1*: $31 \% 32 = 31$
 Máximo: $\text{MAX}(1, 31) = 31$

Ejemplo: qué pasa del 31 de enero al 1 de febrero:

Día actual: 31
 Días del mes actual: 31
 Módulo *días del mes que estemos + 1*: $32 \% 32 = 0$
 Máximo: $\text{MAX}(1, 0) = 1$

- d)* Guarda el nuevo día en la estructura recibida (variable `dd` de `p_fecha`).
3. Si el nuevo día es 1, actualiza el mes de forma análoga a como se hace con los días. Para ello:
 - a) Toma el *mes actual+1*.
 - b) Le aplica el módulo `MAX_MONTH+1`.
 - c) Se queda con el máximo entre el resultado anterior y 1.
 - d)* Guarda el nuevo mes en la estructura recibida (variable `MM` de `p_fecha`).

4. Si el nuevo día es 1, y el nuevo mes es 1, actualiza el año. Para ello:
 - a) Suma 1 al año actual.
 - b) Guarda el nuevo año en la estructura recibida (variable `yyyy` de `p_fecha`).

Parámetros:

`p_fecha: TipoCalendario*`

Es un puntero al calendario de nuestro *reloj* de tipo `TipoCalendario`.

Salida:

-

ActualizaHora

```
|| void ActualizaHora(TipoHora *p_hora);
```

Actualiza la hora (`hh`), minutos (`mm`), y segundos (`ss`) de la estructura recibida `p_hora`, según los siguientes pasos:

1. Actualiza el segundero. Para ello:
 - a) A los segundos actuales le suma 1.
 - b) Le aplica módulo *60*.
 - c) Guarda los segundos en la estructura recibida (variable `ss` de `p_hora`).
2. Si los nuevos segundos son 0, actualiza el minutero de forma análoga a los segundos.
3. Si los nuevos segundos y los nuevos minutos son 0, actualiza la hora. Para ello:
 - a) A la hora actual le suma 1.
 - b) Si el `formato` es `TIME_FORMAT_12_H`, le aplica módulo `formato+1` (para que el mínimo sea 1 y el máximo 12). Pero si el `formato` es `TIME_FORMAT_24_H`, le aplica módulo `formato` solo (para que el mínimo sea 0 y el máximo 23).
 - c) Guarda la hora en la estructura recibida (variable `hh` de `p_hora`).

Parámetros:

`p_hora: TipoHora*`

Es un puntero a la hora (tiempo) de nuestro *reloj* de tipo `TipoHora`.

Salida:

-

CalculaDiasMes

```
||     int CalculaDiasMes(int month, int year);
```

Comprueba si un el año indicado (`year`) es bisiesto, para devolver el número de días del mes indicado (`month`).

`CalculaDiasMes` hace:

1. Llamada a `EsBisiesto` pasándole el año actual.
2. Devuelve el número de días del mes solicitado para el año solicitado.

Parámetros:

month: `int`

Mes actual del *reloj* de nuestro *Core Watch*.

year: `int`

Año actual del *reloj* de nuestro *Core Watch*.

Salida:

diasMes: `int`

Número de días del mes actual.

ConfiguraInicializaReloj

```
||     int ConfiguraInicializaReloj(TipoReloj *p_reloj);
```

Función de inicialización del reloj a valores conocidos de fecha y hora durante el arranque. Para ello:

1. Llama a `ResetReloj`.
2. Crea un nuevo temporizador de tipo `tmr_t*` cuya función ISR es `tmr_actualiza_reloj_isr`.
3. Guarda el temporizador en la variable `tmrTic` de la estructura `p_reloj` recibida.
4. Lanza el temporizador en **modo periódico** con un periodo de `PRECISION_RELÓJ_MS` milisegundos.

Parámetros:

p_reloj: `TipoReloj*`

Es un puntero a la dirección de memoria de nuestro *reloj* de tipo `TipoReloj`.

Salida:

result: `int`

Entero que indica que la inicialización ha sido exitosa (0), o no (otro valor). Esto permite hacer detección de errores, si se considera.

EsBisiesto

```
||     int EsBisiesto(int year);
```

Función que comprueba si un año es bisiesto, o no. Para ello, se realiza el siguiente algoritmo:

1. Si el año es divisible por 4 (*módulo 4 es igual 0*), vaya al paso 2. Si no, vaya al paso 5.
2. Si el año es divisible por 100 (*módulo 100 es igual 0*), vaya al paso 3. Si no, vaya al paso 4.
3. Si el año es uniformemente divisible por 400 (*módulo 400 es igual 0*), vaya al paso 4. Si no, vaya al paso 5.
4. El año es bisiesto (tiene 366 días). Devuelve 1.
5. El año no es bisiesto (tiene 365 días). Devuelve 0.

Parámetros:

year: `int`

Año actual del *reloj* de nuestro *Core Watch*.

Salida:

esBisiesto: `int`

Si el año es bisiesto, devuelve 1. Si no lo es, devuelve 0.

GetRelojSharedVar

```
||     TipoRelojShared GetRelojSharedVar();
```

Getter de variable global. Función que devuelve la variable `g_relojSharedVars`. Esta es la variable global que la librería *reloj* expone al resto de programas a través de las funciones `GetRelojSharedVar` y `SetRelojSharedVar`.

1. Simplemente devuelve `g_relojSharedVars`. Para evitar conflictos, hace una copia local de `g_relojSharedVars` protegido por *mutex*, y la devuelve.

Parámetros:

-

Salida:

g_relojSharedVars: `TipoRelojShared`

Variable global que contiene el estado de los *flags* del *reloj*.

ResetReloj

```
|| void ResetReloj(TipoReloj *p_reloj);
```

Inicializa el *reloj* a valores conocidos de fecha y hora. Para ello:

1. Define la variable `calendario` de tipo `TipoCalendario` e inicializa día (`dd`), mes (`MM`), y año (`yyyy`) a valores conocidos. Estos valores han de ser definidos en las etiquetas `DEFAULT_DAY`, `DEFAULT_MONTH`, y `DEFAULT_YEAR` respectivamente, en el fichero `reloj.h`.
2. Asigna al *reloj* pasado por puntero `p_reloj` la variable `calendario` recién creada.
3. Asigna a la estructura `hora` del *reloj* `p_reloj`, la hora por defecto: hora (`hh`), minutos (`mm`), y segundos (`ss`). Estos valores han de ser definidos en el fichero `reloj.h`, en las etiquetas `DEFAULT_HOUR`, `DEFAULT_MIN`, y `DEFAULT_SEC` respectivamente.
4. Asigna el `formato` a la estructura `hora` del *reloj* `p_reloj`. El formato de hora por defecto ha de ser definido en `DEFAULT_TIME_FORMAT`.
5. Inicializa la variable `timestamp` del *reloj* `p_reloj` a 0.
6. Inicializa `flags` de la variable global `g_relojSharedVars` a 0. **Recuerde siempre proteger el acceso a variables compartidas con mecanismos de exclusión mutua**; en el caso del *reloj*, con la *key* `RELOJ_KEY` definida en `ent2004cfConfig.h`.

Parámetros:

p_reloj: `TipoReloj*`

Es un puntero a la dirección de memoria de nuestro *reloj* de tipo `TipoReloj`.

Salida:

-

SetFecha

To Be Defined, Por Definir (TBD) por parte del alumno si se realiza como mejora.

Parámetros: TBD

Salida: TBD

SetFormato

TBD por parte del alumno si se realiza como mejora.

Parámetros: TBD

Salida: TBD

SetHora

```
||     int SetHora(int horaInt, TipoHora *p_hora);
```

Establece la hora deseada haciendo previamente una conversión de los datos recibidos. Para ello:

1. Se comprueba que el valor recibido `horaInt` es positivo. En caso contrario, se sale de la función (`return`) con un valor distinto de 0³.

2. Se cuenta el número de dígitos de `horaInt`. Para contar el número de dígitos, se puede hacer, por ejemplo, dividiendo entre 10 iterativamente con este algoritmo basado en `do-while`:

```
1: numero_digitos ← 0
2: auxiliar ← hora_recibida
3: do
4:   auxiliar se actualiza dividiendolo entre 10
5:   numero_digitos aumenta en 1
6: while auxiliar ≠ 0
```

3. Se comprueba que tiene los dígitos necesarios: 1, 2, 3, o 4. En caso contrario, se sale de la función con un valor distinto de 0. Como un `int` no puede tener ceros delante, podemos encontrar estas situaciones:

- Que `horaInt` tiene 1 dígito: asumimos que se ha querido introducir las 00 horas, y los minutos son menores de 10, por lo que `horaInt` solo representa los minutos del 0 al 9.
- Que `horaInt` tiene 2 dígitos: asumimos que se ha querido introducir las 00 horas, por lo que `horaInt` solo representa los minutos.
- Que `horaInt` tiene 3 dígitos: asumimos que se ha querido introducir una hora entre las 01 y las 09.
- Que `horaInt` tiene 4 dígitos: se ha querido introducir una hora entre las 10 y las 23.

4. Se extrae la hora de `horaInt`. Si es mayor que `MAX_HOUR`, se pone este valor.

5. Si es necesario, se convierte la hora extraída al `formato` que se indique `p_hora`. Si la hora extraída es 0 y el `formato` es de 12 horas, se fuerza la hora a que sea 12 (no se distingue entre AM y PM).

Por ejemplo, si la hora introducida es 2319 y el formato es de 12 horas:

³Es posible que ya haya visto que el valor 0 se devuelve siempre que una llamada a función ha sido exitosa, y en caso contrario, un valor distinto. Si quiere organizarse, puede definir etiquetas de error para los distintos errores que usted quiera definir, así será más fácil depurar o saber dónde ha fallado.

```

horaInt: 2319
formato de p_hora: TIME_FORMAT_12_H
hora extraída: 23
hora actualizada al formato: 23-12 = 11

```

Por ejemplo, si la hora introducida es (00)35 y el formato es de 12 horas:

```

horaInt: 35
formato de p_hora: TIME_FORMAT_12_H
hora extraída: 0
hora actualizada al formato: 0+12 = 12

```

6. Se extraen los minutos de **horaInt**. Si son mayor que **MAX_MIN**, se pone este valor.
7. Se guardan los valores de hora y minutos extraídos en la estructura **p_hora**. El valor de los segundos, se pone a 0.
8. Sale de la función devolviendo 0.

Parámetros:

horaInt: `int`

Valor entero que representa la hora y minutos a los que se quiere poner el reloj. Los dos primeros dígitos representan la hora, y los dos últimos los minutos (*hhmm*). Se puede ver **horaInt** como:

```
horaInt = hh*100+mm
```

p_hora: `Tipohora`

Es un puntero a la hora (tiempo) de nuestro *reloj* de tipo `Tipohora`.

Salida:

result: `int`

Entero que indica que la actualización de la hora ha sido exitosa (0), o no (otro valor). Esto permite hacer detección de errores, si se considera.

SetRelojSharedVar

```
||     void SetRelojSharedVar(TipoRelojShared value);
```

Setter de variable global. Función que actualiza el valor de la variable **g_relojSharedVars**.

1. Simplemente guarda el valor recibido **value** en **g_relojSharedVars**. Hace uso del *mutex* correspondiente del *reloj*.

Parámetros:

value: `TipoRelojShared`

Estructura que contiene los *flags* de estado del `reloj` actualizados.

Salida:

-

Apéndice C

Librería LCD (HD44780U)

Esta sección es una traducción al español de la página web LCD Library (HD44780U). Todos los contenidos pertenecen a su autor Gordon Henderson.

C.1. Librería LCD (HD44780U)

La *wiringPi* LCD devLib le permite controlar la mayoría de las populares pantallas LCD de 1, 2 y 4 líneas que están basadas en el controlador **Hitachi HD44780U** o en controladores compatibles.

Le permite conectar múltiples pantallas a una sola Raspberry Pi. Las pantallas pueden ser conectadas directamente al GPIO de la Pi o a través de los muchos chips de expansión GPIO soportados por wiringPi —por ejemplo, el expansor GPIO MCP23017 I2C (por ejemplo, utilizado en algunas de las placas de Adafruit).

Los siguientes diagramas de Fritzing describen cómo conectar las pantallas directamente al GPIO de la placa de una Raspberry Pi en los modos de 8 y 4 *bits*:

La librería es fácil de usar, sin embargo, el cableado de las pantallas puede ser un reto, así que tenga cuidado.

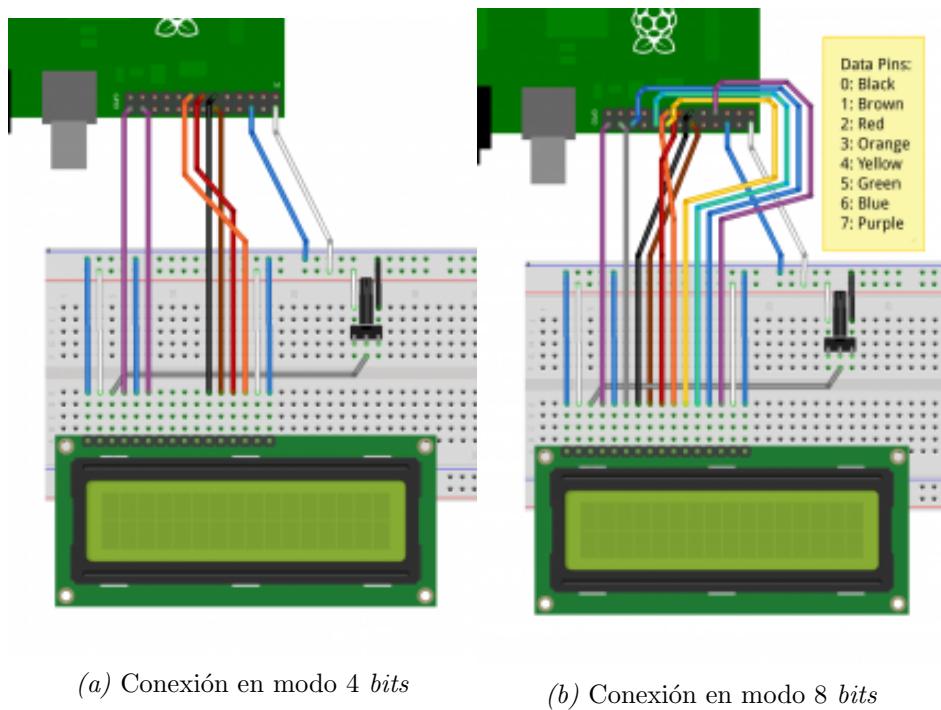
Es posible conectar más de una pantalla. En el modo de 8 *bits*, la primera pantalla necesita 10 *pines* GPIO y cada pantalla adicional necesita solo 1 *pin* más, por lo que con un máximo de 17 *pines* GPIO, puede haber 8 pantallas. Si usas el modo de 4 *bits* (trivial en el código), entonces puedes tener 4 pantallas más —¡12 LCDs! Sin embargo, sospecho que el resto del cableado puede ser difícil... El cableado se describe al final de esta página.

La pantalla LCD puede trabajar a 5V o a 3,3V, pero si estamos usando una pantalla de 5V entonces debemos asegurarnos de que la pantalla nunca puede escribir datos a la Raspberry Pi, de lo contrario pondrá 5V en los *pines* GPIO de la Pi y podría destruirla. En el mejor de los casos destruirás los drivers de los *pines*, en el peor destruirás tu Pi.

Cuando uses una pantalla de 5V, asegúrate de conectar siempre el *pin R/W* de la pantalla a tierra para forzar que la pantalla sea de solo lectura



Figura C.1: Pantalla LCD estándar de 2x16 conectada directamente a una Raspberry Pi y una placa LCD RGB retroiluminada de Adafruit con botones de control. Ver esta página para más detalles de la configuración de la pantalla Adafruit usando wiringPi.



(a) Conexión en modo 4 bits

(b) Conexión en modo 8 bits

Figura C.2: Circuitería y esquema de un *teclado matricial*.

para el host. Si no, la pantalla podría, potencialmente, poner 5V de nuevo en la Pi, lo que es potencialmente dañino.

C.1.1. Instalación y uso

Para utilizar la librería LCD, necesitarás esto al principio de tu programa:

```
#include <wiringPi.h>
#include <lcd.h>
```

Primero, necesita inicializar *wiringPi* de la forma que desee. La librería LCD llamará a las funciones *pinMode* según sea necesario.

```
int lcdInit(int rows, int cols, int bits, int rs, int strb, int d0
           , int d1, int d2, int d3, int d4, int d5, int d6, int d7) ;
```

Esta es la función principal de inicialización y debe ser llamada antes de utilizar cualquier otra función de la pantalla LCD.

Rows y **cols** son las filas y columnas de la pantalla (por ejemplo, 2,16 o 4,20). **bits** es el número de *bits* de ancho en la interfaz (4 u 8). Los parámetros **rs** y **strb** representan los números de *pin* del *pin RS* del display y del *pin Strobe (E)*. Los parámetros **d0** a **d7** son los números de *pin* de los 8 *pines* de datos conectados desde la Pi al display. Solo se utilizan los 4 más altos si se trabaja en modo de 4 *bits*.

El valor de retorno es el 'handle' que se utilizará en todas las llamadas posteriores a la librería LCD cuando se trate de ese LCD, o -1 para indicar un fallo. (Generalmente

por parámetros incorrectos).

En los diagramas anteriores, el de 4 *bits* se inicializaría con:

```
|| fd = lcdInit(2, 16, 4, 11,10 , 0,1,2,3,0,0,0,0);
```

y el de 8 *bits* con:

```
|| fd = lcdInit(2, 16, 8, 11,10 , 0,1,2,3,4,5,6,7);
```

C.1.2. Funciones

```
|| lcdHome (int handle)
|| lcdClear (int handle)
```

Estas funciones ponen el cursor en al inicio y borran la pantalla, respectivamente.

```
|| lcdDisplay (int fd, int state) ;
|| lcdCursor (int fd, int state) ;
|| lcdCursorBlink (int fd, int state) ;
```

Estas funciones encienden o apagan la pantalla, encienden o apagan el cursor y encienden o apagan el parpadeo del cursor. El parámetro `state` es `True` o `False`. Los ajustes iniciales son: pantalla encendida, cursor apagado y parpadeo del cursor apagado.

```
|| lcdPosition (int handle, int x, int y);
```

Pone la posición del cursor para introducir texto posteriormente. `x` es la columna y 0 es el borde de la izquierda. `y` es la fila y 0 es la fila superior.

```
|| lcdCharDef (int handle, int index, unsigned char data [8]) ;
```

Esta función le permite re-definir uno de los 8 caracteres definibles por el usuario en la pantalla. `data` es un array de 8 *Bytes* que representan el carácter desde la fila superior a la inferior. Tenga en cuenta que los caracteres son en realidad 8x5, por lo que solo se utilizan los 5 *bits* menos significativos. `index` va de 0 a 7 y después podrá imprimir el carácter definido mediante la llamada a `LCDPutchar()`.

```
|| lcdPutchar (int handle, unsigned char data) ;
|| lcdPuts (int handle, const char *string) ;
|| lcdPrintf (int handle, const char *message, ...) ;
```

Estas funciones imprimen un solo carácter ASCII, una cadena, o a una cadena formateada utilizando los comandos habituales de formato `printf`.

Por el momento, no hay un desplazamiento inteligente de la pantalla, pero las líneas largas continúan por la siguiente fila, si es necesario.

Vea el programa de ejemplo *lcd.c* en el directorio de ejemplos dentro de la distribución del software **wiringPi**.

C.1.3. Conectándolos

La conexión de las pantallas es relativamente sencilla. Se eligen 4 u 8 *pines* GPIO para el bus de datos, y luego 2 *pines* más para el control. Los LCDs tienen 2 cables de control etiquetados como RS y E. El *pin* E es al que nos referíamos como *pin* estroboscópico en la sección anterior.

Cuando uses una pantalla de 5V, asegúrate de conectar siempre el *pin* R/W de la pantalla a tierra para forzar que la pantalla sea de solo lectura para el host. Si no, la pantalla podría poner potencialmente 5V en la Pi, lo que podría dañarla. Mira los diagramas anteriores del LCD y úsalos para conectar los *pines* en a las GPIO. Utilice el este diagrama para ayudarse a tener un control de los *pines* GPIO que está utilizando.

Para una segunda (o tercera, *etc.*) pantalla, se conectan en paralelo, conectando todos los mismos *pines* con la excepción del *pin* E. Cada pantalla necesita su propio *pin* E conectado a un *pin* GPIO diferente.