Sistemas operativos

PID_00247323

José María Gómez Cama
Màrius Montón Macián

Tiempo mínimo de dedicación recomendado: 2 horas

Universitat
Oberta
de Catalunya

Ninguna parte de esta publicación, incluido el diseño general y la cubierta, puede ser copiada, reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea éste eléctrico, químico, mecánico, óptico, grabación, fotocopia, o cualquier otro, sin la previa autorización escrita de los titulares del copyright.

Índice

Introducción 5

1. Necesidad 7

2. Tipos de gestores 8
2.1. Cooperativos 8
2.1.1. First come First serve 9
2.2. Apropiativos o *preemptive* 9
2.2.1. Round-robin 10
2.2.2. Prioridades prefijadas 10
2.2.3. Otros 11

3. Módulos comunes 12
3.1. Planificación de tareas 12
3.2. Gestión de tareas 13
3.3. Comunicación entre tareas 13
3.4. Controladores de periféricos 14
3.4.1. Información del hardware 15
3.5. Programación del controlador 21

4. Mercado actual 24

Bibliografía 25

Introducción

Tal como se ha indicado, el objetivo fundamental de un sistema empotrado (SE) es conseguir llevar a cabo una serie de tareas, sobre la base de unos requisitos, de manera eficiente y optimizando el uso de los recursos disponibles.

Conseguir este objetivo de manera genérica no es evidente, debido fundamentalmente a la innumerable cantidad de plataformas, arquitecturas y aplicaciones posibles. Ello provoca que sea virtualmente imposible dar una respuesta óptima para todos los posibles escenarios.

Sin embargo, la introducción de los sistemas operativos (SO) permite gestionar los diferentes elementos de un sistema basado en procesador de modo eficiente. A la vez, presenta al programador/usuario una máquina virtual que es equivalente, independiente de la plataforma. Esto simplifica de manera notable su uso y programación. El precio son los recursos de memoria y tiempo que requiere el propio sistema operativo.

Partiendo de lo anteriormente dicho, parece lógico pensar que el uso de un SO podría ser la manera más adecuada de lograr que un SE cumpla su objetivo fundamental. No obstante, la realidad actual muestra que esto no es así, ya que el número de sistemas empotrados que incluyen un SO es muy reducido. En general, una programación *ad hoc* suele ser la solución preferida. Los motivos pueden ser muchos, y dependen del punto de vista adoptado. Por poner ejemplos, estos podrían ser algunos teniendo como objetivo la eficiencia:

- Los recursos del sistema son muy reducidos y la inclusión de un SO los mermaría de manera notable, por lo que se requeriría una nueva plataforma con más prestaciones.

- Las tareas que se deben realizar son muy sencillas y están muy bien delimitadas, por lo que añadir un SO apenas mejoraría los resultados.

- Las tareas que se deben realizar llevan al límite una plataforma de altas prestaciones, por lo que cualquier sobrecoste debido al SO impediría alcanzar los requerimientos.

Por otro lado, desde el punto de vista contrario, podríamos tener los siguientes:

- La plataforma es tan sencilla que las funcionalidades que proporciona el SO son más complejas que la original.

- No existe una adaptación del SO a la plataforma que se va a utilizar, con lo que el sobrecoste debido a la complejidad de la arquitectura de la plataforma queda compensado por la posible dificultad de la adaptación del SO.

En estos casos, es evidente que el SO pierde frente a una programación *ad hoc*, lo que no significa que no se utilicen elementos de un SO para ayudar en la programación. A modo de ejemplo –si la aplicación es sencilla y, por lo tanto, el número de tareas que hay que realizar no es alto; la arquitectura del sistema empotrado es reducida y los periféricos no tienen una gran complejidad (ADC, GPIO, USART, I^2C, etc.)–, es muy común realizar una programación basada en un gestor de tareas en bucle.

En resumen, no existe una regla de oro a la hora de trabajar con SO sobre un SE. Y en general, su uso dependerá de:

- La aplicación.
- La plataforma.
- La experiencia del programador.

Sin embargo, es importante recalcar que las nuevas plataformas proporcionan cada vez más recursos, por lo que es previsible que de la misma manera que se ha dado el salto de la programación en ensamblador a otros lenguajes de mayor nivel, como el C, en un futuro cercano se vea como un paso natural la programación de sistemas empotrados basada en SO.

1. Necesidad

Cuando la complejidad del sistema empotrado aumenta en el número de tareas que realizar, aparece la necesidad de usar un sistema operativo para manejar dichas tareas.

De esta manera, dejamos en manos del sistema y no del desarrollador la responsabilidad de implementar la concurrencia y la multitarea para un sistema dado.

Concurrencia y multitarea

La multitarea es la capacidad de los sistemas operativos de aparentar que diversas aplicaciones están ejecutándose simultáneamente. La concurrencia es una propiedad de los sistemas informáticos de ejecutar más de un proceso a la vez y que interactúen entre ellos.

Habitualmente los sistemas operativos para sistemas empotrados son, en realidad, meros gestores de tareas, sin muchas más opciones como nos tienen acostumbrados los sistemas operativos de propósito general (que pueden tener gestores de disco, librerías para acceso a internet, librerías gráficas, etc.).

En caso de necesitar librerías de este tipo, se recurre o bien al propio fabricante del sistema operativo o bien a terceras partes que las proporcionen para el sistema operativo elegido.

2. Tipos de gestores

A diferencia de un ordenador de propósito general, un sistema empotrado está muy focalizado en dar respuesta a una determinada necesidad o problema. Dicha necesidad o problema se suele dividir en **tareas**, que son los elementos atómicos necesarios para resolver los problemas.

En este sentido, las tareas que debe realizar están definidas *a priori* y cualquier modificación de estas suele implicar una reprogramación.

> En general, podemos considerar una tarea como un elemento atómico requerido para resolver un problema. Dicha tarea precisará una información de entrada, que procesará y que le servirá para dar respuesta al problema.

Para llevar a cabo dichas tareas, es necesario que se ejecuten las instrucciones necesarias en la CPU. Para ello, se requieren también recursos de memoria y posiblemente acceso a información de determinados periféricos. Dichos recursos deben ser proporcionados de alguna manera dentro del SO.

Scheduler o planificador

El *planificador* o *scheduler* es la parte de un sistema operativo que reparte el tiempo de ejecución en el microprocesador entre las distintas aplicaciones o procesos del sistema.

En función de los requerimientos temporales de dichas tareas, se incrementará o reducirá la complejidad del gestor de procesos. Si es muy alta, aparecerá la necesidad del planificador de tareas.

2.1. Cooperativos

El primer tipo de sistemas operativos son los cooperativos, donde son las propias tareas las que ceden el control a las demás usando algún mecanismo común. De esta manera, el programador es el que decide cuándo una tarea deja de ejecutarse para pasar la ejecución a otra.

Este tipo de diseño simplifica la implementación del sistema operativo a costa de incrementar la responsabilidad del programador en tiempo de diseño e implementación de su solución. En un sistema operativo de este tipo, en caso de un mal diseño o error en una tarea, todo el sistema puede dejar de funcionar, ya que el SO no tiene forma de controlar las tareas.

En este caso, el SO no es más que un conjunto de funciones que permiten al desarrollador ceder el control de cada tarea en un momento dado.

Gracias al aumento de prestaciones en los microcontroladores actuales, este tipo de SO están siendo sustituidos por los de tipo apropiativo.

2.1.1. First come First serve

Es quizá de los sistemas más sencillos, donde simplemente se «carga» en ejecución la primera tarea disponible. Cuando una tarea se descarga de ejecución se pone al final de la cola.

Este tipo de *scheduler* no tiene en cuenta que pueda haber tareas más prioritarias que otras, pero para cierto tipo de sistemas puede ser suficiente.

2.2. Apropiativos o *preemptive*

A diferencia de un SO cooperativo, donde el SO no tiene control real sobre qué tarea se está ejecutando en cada momento, en un SO apropiativo es el SO quien controla y gestiona la ejecución de las tareas.

Para lograrlo, un componente del SO llamado *scheduler* se ejecuta cada cierto tiempo (llamado *tick* de sistema) para evaluar el estado del sistema y decidir qué tarea debe ejecutarse a continuación.

En este caso, la tarea del desarrollador no incluye ninguna decisión explícita de dar el control al SO, sino que se deja esta tarea al *scheduler*.

En este caso, una tarea puede estar en alguno de los estados presentados en la figura 1:

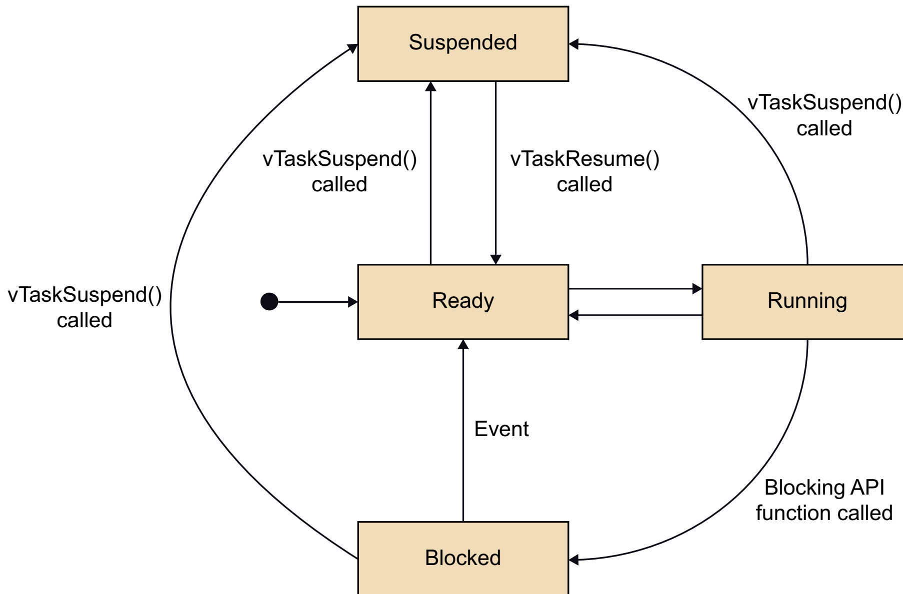
- *Ready*: una tarea en el estado *Ready* estará disponible para ser cargada en la CPU en cualquier momento.

- *Blocked*: una tarea que esté *Blocked* estará esperando algún evento y, por tanto, debe esperar a que suceda ese evento para poder volver al estado *Ready*.

- *Suspended*: es un estado especial donde la tarea e ha puesto en este estado de forma explícita para no poder cargarse en ejecución.

- *Running*: indica que esa tarea es la que está actualmente en ejecución. En caso de procesadores multihilo, más de una tarea puede estar en este estado.

Figura 1. Diagrama de estados para una tarea



La complejidad de este tipo de SO crece, ya que debe añadir los mecanismos de gestión de tareas y de control de la CPU para poder cargar y descargar tareas de ejecución según se decida.

En función del algoritmo de decisión sobre qué tareas y en qué orden deben ejecutarse, tenemos los siguientes tipos de SO.

2.2.1. Round-robin

En esta forma de arbitraje de tareas, a cada tarea se le asigna el mismo tiempo de ejecución que a las demás, y se van intercambiando siguiendo un patrón circular.

Conceptualmente es muy parecido al First come First serve, pero aquí el *scheduler* tiene la capacidad de retirar una tarea en ejecución y cambiarla por otra que esté preparada.

Es un método sencillo de implementar y con poca sobrecarga para el sistema, ya que el *scheduler* puede decir rápidamente y sin muchas operaciones qué tarea es la siguiente a ser cargada.

2.2.2. Prioridades prefijadas

En este caso, cada tarea tiene asignada desde su inicio y de manera permanente una prioridad. Esta prioridad marca su importancia respecto a las demás tareas, y es lo que usa el SO para decidir en cada momento qué tarea debe cargarse.

En este tipo de SO, a cada *tick* de sistema el SO elige la tarea que esté en el estado *Ready* de más prioridad y la carga para ejecución. También ocurre que si una tarea en algún momento pasa del estado *Blocked* al estado *Ready*, si la nueva tarea tiene mayor prioridad que la que actualmente se está ejecutando, se interrumpe inmediatamente la ejecución de la tarea en *Running* y se carga la tarea recién desbloqueada.

De esta manera, el SO garantiza que en todo momento se ejecuta la tarea de mayor prioridad que esté lista para ser ejecutada (esté en el estado *Ready*).

Esta técnica puede dar lugar a que, según los casos, ciertas tareas de baja prioridad lleguen a no ejecutarse nunca. Para estos casos, ciertos SO ofrecen técnicas de envejecimiento, que consisten en incrementar momentáneamente la prioridad de las tareas que llevan mucho tiempo sin ejecutarse.

2.2.3. Otros

Vistos los dos métodos más usuales, cabe señalar que hay multitud de variantes y métodos distintos a los presentados. En algunos de ellos se intenta mejorar el hecho de que tareas de baja prioridad no se ejecuten nunca, o tener una gestión avanzada de las tareas más prioritarias, de modo que todas se ejecuten con menores tiempos de ejecución para cada una de ellas, etc.

3. Módulos comunes

Los módulos o componentes que forman un SO para sistemas empotrados son los siguientes:

- Planificador de tareas o *scheduler*: es el encargado de decidir qué tarea pasa a ejecutarse en cada momento.
- Gestión de tareas: aquí se agrupan todas las funciones que permiten declarar, iniciar o parar las tareas en un sistema empotrado.
- Comunicación entre tareas: los SO ofrecen distintos mecanismos para comunicar y sincronizar las tareas entre ellas.
- Controlador de periféricos: algunos SO incluyen controladores para periféricos o describen cómo implementarlos por parte del desarrollador.

Veamos ahora cada uno de los componentes en detalle.

3.1. Planificación de tareas

Como ya se ha comentado en el apartado 2 de este contenido «Tipos de gestores», el *scheduler* o planificador de tareas es la parte encargada de decidir qué tareas se ejecutan, en qué orden y durante cuánto tiempo.

Es la parte más importante de un SO y la más crítica para el buen funcionamiento de todo el sistema.

Habitualmente la forma de implementar el *scheduler* es que este sea llamado periódicamente por una interrupción generada por un *timer* de sistema. El periodo de estas interrupciones se conoce como *tick* de sistema o *systick* y suele ser del orden de milisegundos.

Una vez que se ha generado la interrupción, pasa a ejecutarse el *scheduler*, esto es, la tarea que estaba en ejecución se retira de la CPU y pasa a ocuparla el planificador.

Entonces, el planificador comprueba el estado de las tareas y aplica el algoritmo de planificación para elegir la siguiente tarea que ejecutar. Una vez elegida la tarea, el mismo *scheduler* realiza la operación de cambiar de contexto la CPU para que sea la tarea seleccionada la que se ejecute en el siguiente lapso de tiempo.

Este conjunto de operaciones se realiza de manera repetitiva durante toda la vida del dispositivo. Toda esta operativa transcurre sin ninguna intervención por parte de las tareas ni del código que se está ejecutando en ellas.

3.2. Gestión de tareas

Todo SO proporciona ciertas funciones a los desarrolladores para que puedan controlar hasta cierto punto algunas características de las tareas del sistema.

Así, es común que el SO proporcione funciones como las siguientes:

- Crear y destruir tareas: las tareas se crean en tiempo de ejecución y, en algunos casos, pueden destruirse. Es habitual proporcionar el nivel de prioridad de la tarea en el momento en el que esta se crea.

- *Delay*: una tarea puede ser suspendida por un cierto tiempo, de manera que vuelve a ejecutarse una vez transcurrido ese tiempo.

- Cambiar prioridades: en algunos SO es posible que una tarea pueda cambiar su prioridad en tiempo de ejecución o, incluso, que pueda cambiar la prioridad de otra tarea.

- Suspender y resumir: en algunos casos, una tarea puede suspenderse, ya que no hace falta que se ejecute (por condiciones del sistema, por conexión o desconexión de algún componente, etc.). Con estas funciones una tarea puede cambiar su estado o el de otra tarea al estado *Suspended* y luego una tarea puede recuperar el estado *Ready* a una tarea que estuviera suspendida.

3.3. Comunicación entre tareas

Otra característica esencial de un sistema complejo es la necesidad de comunicar las tareas entre sí. Esta comunicación puede servir para sincronizar tareas o para enviarse datos entre ellas. Estos mecanismos de comunicación se dividen en estos tipos:

- Semáforos: Este método sirve para sincronizar dos tareas, existe un *token* que o bien está libre o lo tiene una de las tareas. Si la otra intenta cogerlo cuando la otra tarea lo tiene bloqueado, la primera tarea se queda en estado bloqueado hasta que la segunda tarea devuelve el *token*. En caso de que el *token* sea de tipo binario o entero, el semáforo será de este tipo.

- Colas: Este mecanismo permite enviar datos de una tarea a otra. Estas colas pueden ser de distintos tipos, pero lo más habitual es que sean de tipo FIFO, y permitan almacenar temporalmente más de un dato. De nuevo, cuando una tarea intenta sacar un dato de la cola y esta está vacía, la tarea queda bloqueada a la espera de

un dato. Lo mismo ocurre si una tarea intenta enviar un dato a la cola y esta está llena.

- Mutex: Parecido a un semáforo binario, es el mecanismo para implementar la exclusión mutua entre dos tareas. La exclusión mutua se utiliza para compartir un mismo recurso por parte de dos tareas y que no puedan usarlo simultáneamente.

Aunque cada SO tiene sus particularidades y diferencias, estos tres mecanismos básicos están disponibles en prácticamente todos los SO actuales.

3.4. Controladores de periféricos

Como se ha indicado, los controladores de dispositivos son otro de los elementos básicos de un SO, ya que proporcionan una abstracción del periférico asociado. De este modo, el periférico se convierte en una caja negra que proporciona unos determinados servicios y el programador solo tiene que interactuar con una interfaz, lo que le permite olvidarse de los aspectos internos del hardware concreto.

Si la definición de la interfaz es suficientemente genérica, simplificará su utilización para el programador. En general, todo el mundo sabe interactuar con estos botones, lo cual simplifica su uso.

Ejemplo

Todos los dispositivos de música tienen un teclado semejante (reproducir, pausa, avance rápido, rebobinar).

Podemos añadir nuevas funcionalidades, como canción aleatoria, reproducir en bucle, etc. Sin embargo, si queremos que nuestro dispositivo tenga éxito, debemos asegurar que dispone de estos cuatro botones, que todo el mundo sabe identificar y, por lo tanto, utilizar.

Esto mismo sucede en los sistemas empotrados. Existe un conjunto de dispositivos que es genérico. Una interfaz tipo serie suele hacer uso de los mismos parámetros en las diferentes plataformas (velocidad de transmisión, número de bits, bits de parada, paridad, etc.). Todos estos parámetros se pueden definir de manera genérica en una interfaz, de modo que cuando un programador escribe un programa, no se ha de preocupar de saber el hardware concreto que hay debajo. Dicha interfaz se denomina abstracción de hardware. El conjunto de todas las interfaces para periféricos se conoce como capa de abstracción de hardware.

Evidentemente, puede haber peculiaridades específicas para un periférico que lo hacen diferir de la interfaz genérica. En este caso, se pueden añadir interfaces que amplíen la interfaz original, pero es importante que estas añadan nueva funcionalidad y que sean también genéricas a su vez. Para realizar estos controladores, es recomendable seguir una serie de pasos:

1) Analizar las características del sensor.
2) Si existe una interfaz genérica, analizar las funciones asociadas.
3) Si no existe, definir una.
4) Programar el código asociado a la interfaz.

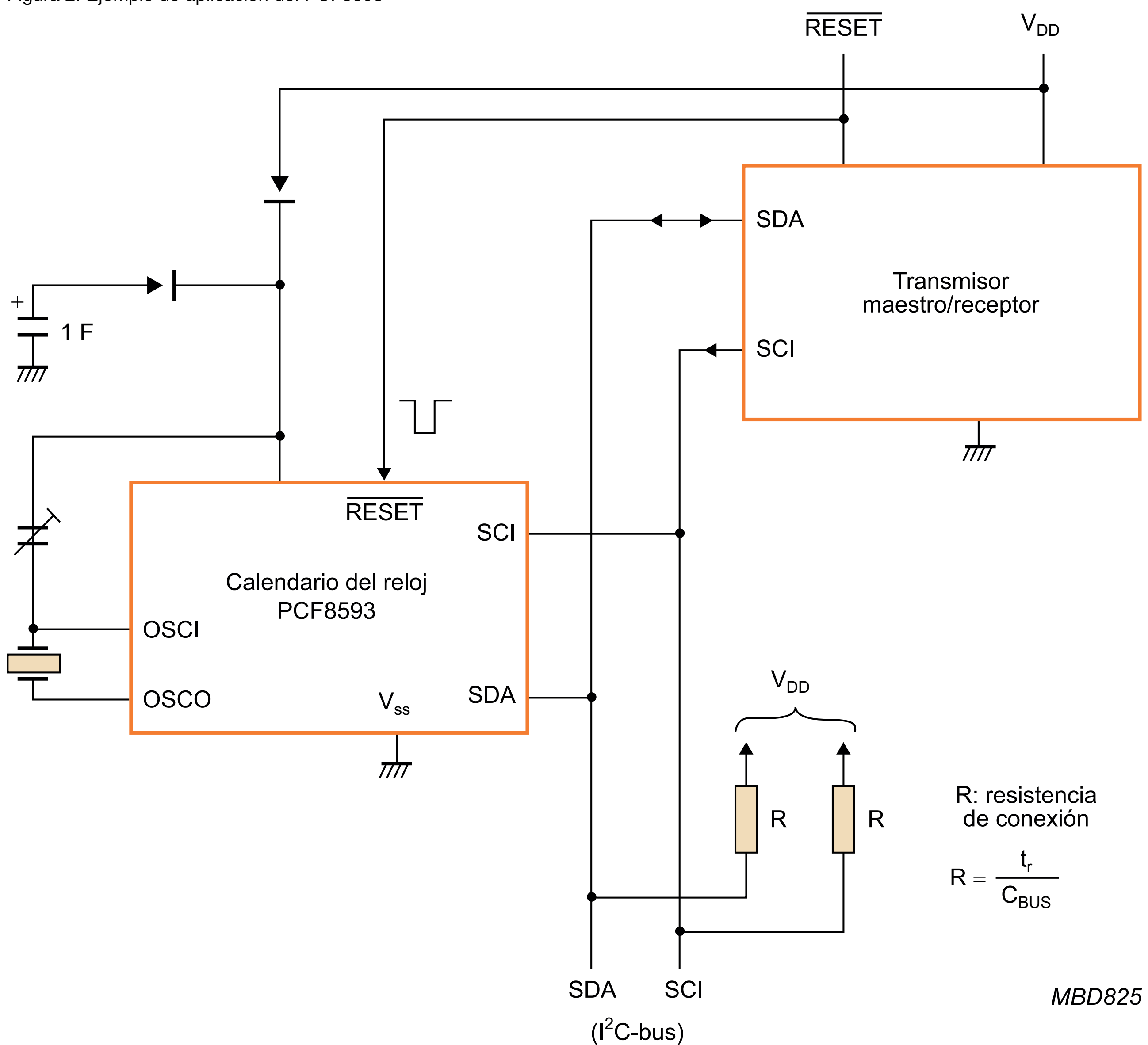
A continuación, vamos a verlos con detalle.

3.4.1. Información del hardware

Como ejemplo, vamos a analizar el caso de un reloj en tiempo real. Este tipo de dispositivos son relojes que funcionan continuamente mediante una batería. De esta manera, el microcontrolador puede conocer la hora actual consumiendo muy pocos recursos. La precisión de estos relojes se sitúa alrededor de 1 segundo.

En nuestro caso, haremos uso del circuito integrado PCF8593 de NXP. Si miramos la información recogida en la hoja de características (*datasheet*), vemos un ejemplo de aplicación (podéis verlo en la figura siguiente).

Figura 2. Ejemplo de aplicación del PCF8593



Podemos observar dos aspectos fundamentales:

1) La comunicación con el microcontrolador se basa en un protocolo I^2C.*
2) Mediante una capacidad grande, puede continuar funcionando en caso de que se pierda la alimentación en un momento determinado.

* ***I2C-bus specification and user manual*, NXP. http://www.nxp.com/documents/user_manual/UM10204.pdf.**

Evidentemente, es necesario estudiar el formato de trama que enviará el I^2C para poder establecer cómo se van a enviar y recibir los datos. Para ello, miramos la hoja de características y vemos que tiene los siguientes registros que podemos modificar (podéis ver la figura siguiente).

Figura 3. Registros del PCF8593

Clock modes		Event counter		
Control/status		Control/status		00
Hundredths of a second 1/10 s	1/100 s	D1	D0	01
Seconds 10 s	1 s	D3	D2	02
Minutes 1 min	1 min	D5	D4	03
Hours 10 h	1 h	Free		04
Year/date 10 day	1 day	Free		05
Weekday/month 10 month	1 month	Free		06
Timer 10 day	1 day	Timer T1	T0	07
Alarm control		Alarm control		08
Hundredths of a second 1/10 s	1/100 s	Alarm D1	Alarm D0	09
Alarm seconds		D3	D2	0A
Alarm minutes		D5	D4	0B
Alarm hours		Free		0C
Alarm date		Free		0D
Alarm month		Free		0E
Alarm timer		Alarm timer		0F

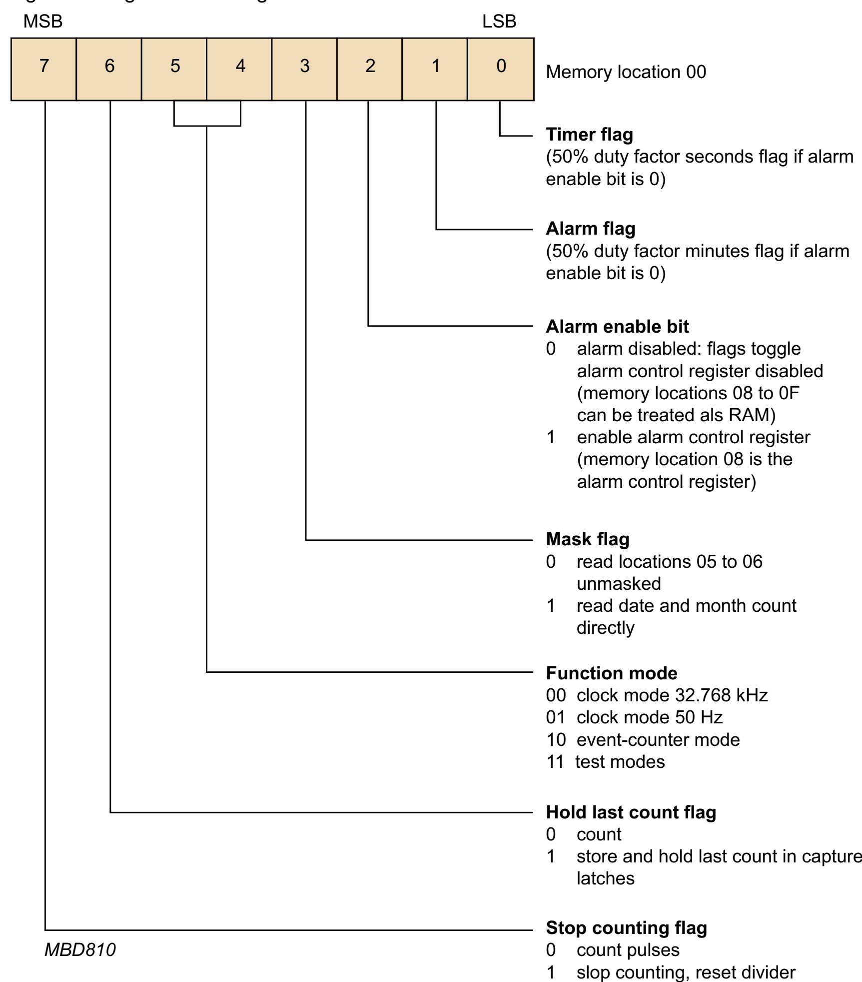
El dispositivo tiene 16 registros. El cero es de configuración y estado. En función de la configuración, puede trabajar en diferentes modos:

- Reloj basado en un oscilador a 32.768 Hz.
- Reloj basado en un oscilador a 50 Hz.
- Contador de eventos.

En nuestro caso, vamos a trabajar en el modo del reloj, por lo que nos concentraremos en este. Vemos que los modos de reloj del registro 1 al 7 indican el momento actual en decimales codificados en binario (*binary coded decimal*, BCD). Del 8 al 15 permiten establecer la alarma, que nosotros no vamos a utilizar.

Para configurarlo, hemos de tener en cuenta el registro 0. Los diferentes bits de dicho registro permiten establecer los modos de trabajo. La descripción de estos se muestra en la figura siguiente.

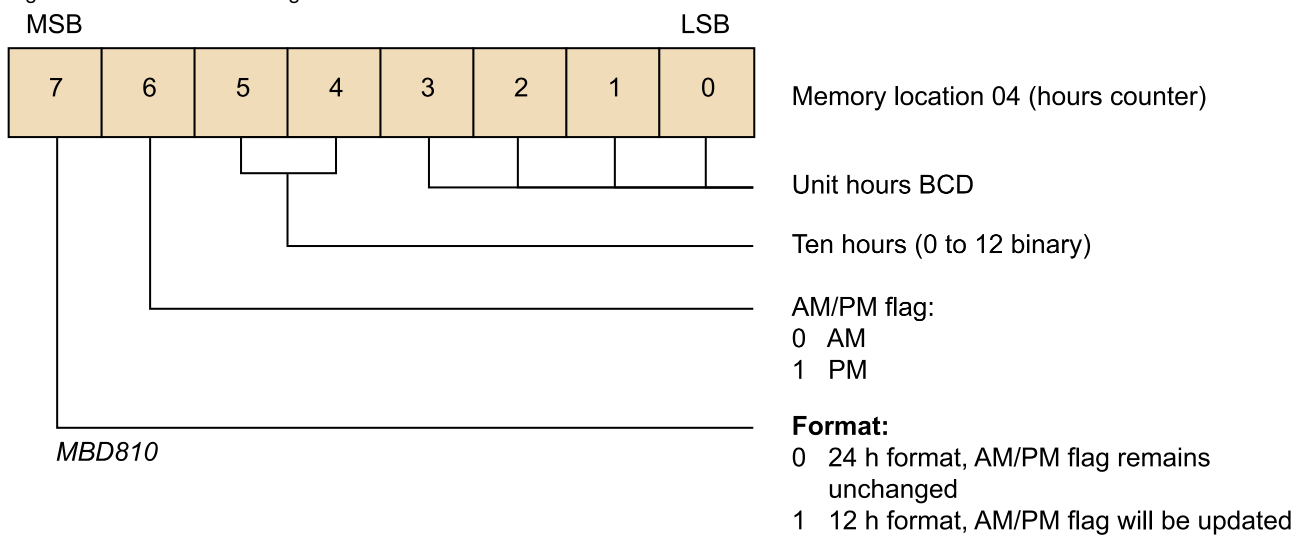
Figura 4. Registro de configuración del PCF8593



Los bits 0 y 1 indican el estado del temporizador (*timer*) o de la alarma (*alarm*). Vemos que el modo de funcionamiento lo definen los bits 4 y 5. Suponiendo que trabajamos con un reloj de 32.768 Hz, vemos que hemos de poner sendos bits a 0. También vemos que si no queremos hacer uso de la alarma, hemos de poner el bit 2 a 0. El bit de máscara nos permite decidir si vemos toda la información de los registros 5 y 6 o únicamente la información del día y el mes. Teniendo en cuenta que queremos toda la información, lo pondremos a 0. Lo mismo haremos con el bit de mantener, el bit de stop.

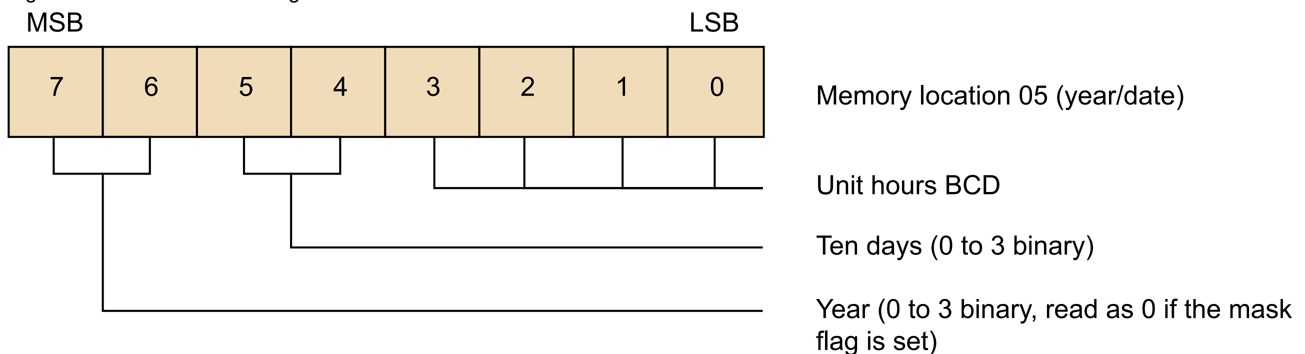
Para poner en hora el integrado, hemos de tener en cuenta las características especiales de los registros 4, 5 y 6 (podéis ver las siguientes tres figuras, respectivamente).

Figura 5. Funcionalidad del registro 4 del PCF8593



Se puede modificar el formato del registro entre 12 y 24 horas cambiando el bit 7. En este caso, trabajaremos en formato 24, por lo que el bit deberá ser puesto a 0. En este formato, el bit 6 no modificará su estado, y el registro lo podremos leer y escribir como un registro BCD.

Figura 6. Funcionalidad del registro 5 del PCF8593

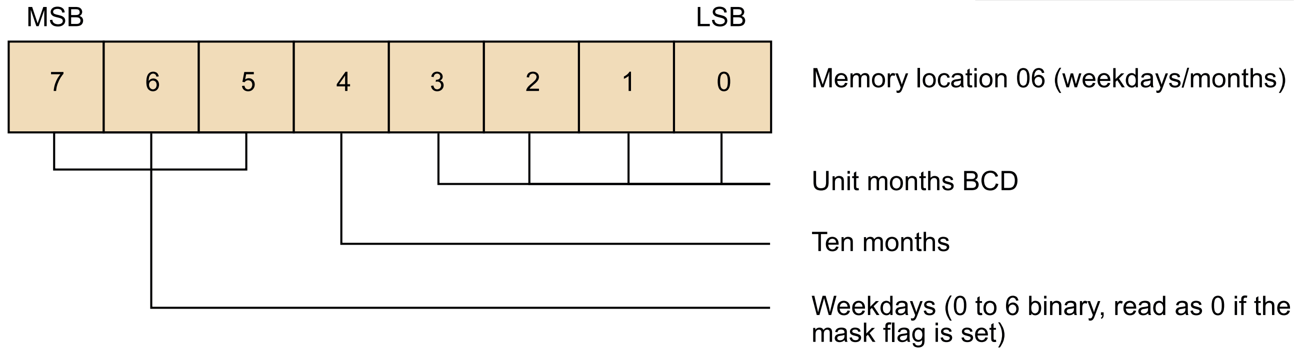


Este registro proporciona el día del mes en los bits 0-5 en BCD. En los dos bits más altos, encontramos el valor del año dentro de un ciclo de año bisiesto. Esto permite determinar el número de días del mes de febrero. A la hora de inicializar el valor de este registro, deberemos tenerlo en cuenta.

Ejemplo

Si es el año 2010, podemos calcular el módulo 4 del año, que nos da el valor 2. En este caso, pondremos un 1 en el bit 7 y un 0 en el 6.

Figura 7. Funcionalidad del registro 6 del PCF8593



Finalmente, el registro 6 indica a la vez el mes, bits 0-4, en BCD y el día de la semana, bits 5-7. Si queremos grabar la fecha «Domingo, 2011-05-29 19:52:23.87», tendremos que escribir los valores indicados en la tabla siguiente:

Tabla 1. Valores de inicialización de los registros del PCF8593

Valor	Registro	Bits	Valor hexadecimal	Valor binario	Comentarios
Domingo	6	7-5	7	111	
2011	5	7-6	3	11	2011 %4
05	6	4-0	05	0 0101	
29	5	5-0	29	10 1001	
19	4	5-0	19	01 1001	
52	3	7-0	52	0101 0010	
23	2	7-0	23	0010 0011	
87	1	7-0	87	1000 0111	

Para calcular el valor final del registro 5, podemos hacerlo de manera sencilla con la siguiente fórmula:

$$\text{Registro 5} = (\text{Valor}_{\text{Año}} << 6)|(\text{Valor}_{\text{Dia}});$$

En este caso, el resultado del registro 5 es 1110 1001 en binario, o E9 en hexadecimal. A partir de estos valores, podemos inicializar el RTC enviando la siguiente trama I^2C.

Tabla 2. Trama I^{2}C inicialización del PCF8593

Binary (BCD)		HEX	Registro	Comentarios
				Generar condición de inicio I^{2}C
1010	0010	A2		Dirección del esclavo I^{2}C, escritura
0000	0000	00		Dirección 0, el resto de los bytes son datos
0000	0000	00	00	Control/estatus 1, se dejan todos los bits a 0
1000	0111	87	01	87 centésimas de segundo
0010	0011	23	02	23 segundos
0101	0010	52	03	52 minutos
0001	1001	19	04	19 horas
1110	1001	E9	05	Tercer año del ciclo bisiesto, día 29
1100	0101	C5	06	Domingo, mes 5
0000	0000	00	07	Deshabilitamos la alarma y el temporizador
				Generar condición de parada I^{2}C

Una vez enviada esta trama, el integrado se ha inicializado adecuadamente. Si queremos leer los valores de minutos y horas, simplemente deberemos enviar la cabecera correspondiente de I^{2}C.

Tabla 3. Trama I^{2}C inicialización del PCF8593

Binary (BCD)		HEX	Registro	Comentarios
				Generar condición de inicio I^{2}C
1010	0010	A3		Dirección del esclavo I^{2}C, lectura
0000	0000	03		Dirección 03, el resto de los bytes son datos
XXXX	XXXX	XX	03	Minutos
XXXX	XXXX	XX	04	Horas
				Generar condición de parada I^{2}C

Como la trama es de lectura, se indica con X los valores que se recibirán, pero que no se conocen. Suponiendo que hayan pasado 15 minutos desde que programamos el integrado, recibiremos el valor hexadecimal 20 para la hora y 07 para los minutos.

Una vez conocido el modo de trabajar con el dispositivo, debemos ver cómo comunicarnos con él. Para ello, hemos de mirar los periféricos del microcontrolador disponibles. Si tiene un controlador de I^{2}C, el proceso se simplifica. En caso contrario, se pueden programar las entradas y salidas de propósito general para lograrlo, aunque se incrementa ligeramente la complejidad.

Uno de los parámetros importantes que se debe tener en cuenta es la velocidad máxima permitida por el RTC para comunicarnos a través del I^{2}C. En la hoja de características, podemos ver que esta se denomina fSCL y que su valor máximo es 100 kHz. Por lo tanto, hemos de limitar la transmisión a esta velocidad si queremos que el circuito reciba correctamente la información.

3.5. Programación del controlador

A partir de lo anterior, podemos escribir el programa para comunicarnos con el RTC. En este caso, hemos de tener en cuenta que trabajamos con dos capas: la de menor nivel es el controlador para el I^2C, mientras que la segunda es la del RTC. Asumimos que el RTC solo lo vamos a utilizar en casos concretos, dado que en el microcontrolador ya tenemos un reloj de sistema, que puede ir incrementando la fecha. El RTC lo utilizaremos cuando se vuelva a iniciar el sistema porque haya sido detenido o por cualquier otro motivo que pueda dejar el reloj del sistema desconfigurado. En este sentido, el acceso al RTC no va a hacer uso de interrupciones y, por lo tanto, lo podemos programar como un conjunto de métodos para ser utilizados por la aplicación.

Empezamos creando la estructura internalClk, donde guardaremos los valores del reloj, en centésimas de segundo, aunque podría ser mayor si nos interesara y el reloj del sistema lo permitiera.

```
struct date_t{
        uint8_t cs;
        uint8_t sec;
        uint8_t min;
        uint8_t hour;
        uint8_t day;
        uint8_t month;
        uint16_t year;
        uint8_t weekday;
} internalClk;
```

Seguidamente, creamos un método que permite convertir el contenido de la estructura del reloj interno al formato de trama I^2C para poder enviar los datos al RTC. Suponemos que al método le llega un vector del mismo tamaño que registros tiene el RTC.

```
#define REGISTER_NUMBER ((uint8_t) 0x10)
void generateFrameData(struct date_t* date, uint8_t values[REGISTER_NUMBER]) {
        values[0] = 0;
        values[1] = ((date->cs / 10) << 4)   (date->cs
                    % 10);
        values[2] = ((date->sec / 10) << 4)   (date->sec
                    % 10);
        values[3] = ((date->min / 10) << 4)   (date->min
                    % 10);
        values[4] = ((date->hour / 10) << 4)
                              (date->hour % 10);
        values[5] = ((date->day / 10) << 4)   (date->day
                    % 10)   ((date->year % 4) << 6);
        values[6] = ((date->month / 10) << 4)
                      (date->month % 10)   (date->weekday << 5);
}
```

Del mismo modo creamos un método para hacer el paso inverso.

```
void restoreFrameData(uint8_t values[REGISTER_NUMBER], struct date_t* date) {
        uint8_t hour12;
        uint8_t temp;

        temp = ((values[1] >> 4) * 10);
        temp += (values[1] & 0x0F);
        date->cs = temp;

        temp = ((values[2] >> 4) * 10);
        temp += (values[2] & 0x0F);
        date->sec = temp;

        temp = ((values[3] >> 4) * 10);
        temp += (values[3] & 0x0F);
        date->min = temp;
```

```
        if ((values[4] & 0x80) != 0) {
                hour12 = values[4] & 0x1F;

                temp = ((hour12 >> 4) * 10);
                temp += (hour12 & 0x0F);

                if ((values[4] & 0x40) != 0) {
                        temp += 11;
                } else {
                        temp -= 1;
                }
                date->hour = temp;

        } else {
                temp = ((values[4] >> 4) * 10);
                temp += (values[4] & 0x0F);
                date->hour = temp;
        }

        temp = (((values[5] & 0x30) >> 4) * 10);
        temp += (values[5] & 0x0F);
        date->day = temp;

        temp = (values[5] >> 6);
        date->year = temp;
        temp = (((values[6] & 0x10) >> 4) * 10);

        temp += (values[6] & 0x0F);
        date->month = temp;
        temp = (values[6] >> 5);

        date->weekday = temp;
}
```

Para el método de envío de la trama, se sigue un esquema semejante al de la tabla «Trama I^{2}C inicialización del PCF8593». En primer lugar, se realiza la conversión del formato reloj interno al de datos del RTC con generateFrameData. Seguidamente, empezamos la transmisión de la trama. Para ello, enviamos en primer lugar una marca de inicio de I^{2}C. A continuación, enviamos la dirección de escritura del RTC y la dirección del primer registro al que queremos acceder (en este caso, el 0). Posteriormente enviamos los datos para programar el reloj. Finalmente, enviamos una marca de finalización.

```
void writeClk(struct date_t* date) {
        uint8_t data[REGISTER_NUMBER];
        int i;

        generateFrameData(date, data);

        startCondition();
        putByte(0xA2);
        putByte(0x00);

        for (i = 0; i < REGISTER_NUMBER; i++) {
                putByte(data[i]);
        }
        stopCondition();
}
```

En el *main*, estamos suponiendo que estamos emulando en un sistema Posix. En primer lugar, inicializamos el I^{2}C. Seguidamente, miramos la hora actual del sistema (si fuera un microcontrolador, esta línea no se utilizaría por motivos evidentes). El siguiente paso es inicializar el reloj interno. Imprimimos el contenido del reloj en pantalla y enviamos los datos al RTC por I^{2}C. Una vez hecho esto, realizamos un bucle de espera de unos diez segundos. Teniendo en cuenta que es posible que nos despierten antes de que pasen los diez segundos, comprobamos el tiempo que ha pasado y si no ha llegado, se vuelve a esperar.

Una vez efectuada esta acción, leemos el contenido del RTC, que mostramos en pantalla, y comparamos el resultado con el del reloj interno. Una vez finalizado, se acaba la aplicación.

```
int main(void) {
        time_t startTime, endTime;

        i2cInit();
        startTime = time(NULL);

        internalClk.cs = 0;
        internalClk.sec = 23;
        internalClk.min = 52;
        internalClk.hour = 19;
        internalClk.day = 29;
        internalClk.month = 5;
        internalClk.year = 2011;
        internalClk.weekday = 6;

        printf("%d,  %d/%d/%d  %d:%d:%d.%d\n", internalClk.weekday,
                        internalClk.year, internalClk.month, internalClk.day,
                        internalClk.hour, internalClk.min, internalClk.sec,
                             internalClk.cs);

        writeClk(&internalClk);

        do {
                sleep(2);
                endTime = time(NULL);
        } while((endTime-startTime) < 10);

        readClk(&internalClk);

        printf("%d,  %d/%d/%d  %d:%d:%d.%d\n", internalClk.weekday,
                        internalClk.year, internalClk.month, internalClk.day,
                        internalClk.hour, internalClk.min, internalClk.sec,
                             internalClk.cs);

        printf("Start: %ld\n", startTime);
        printf("End: %ld\n", endTime);
        printf("Delta: %ld\n", endTime-startTime);

        return EXIT_SUCCESS;
}
```

4. Mercado actual

El mundo de los sistemas operativos para sistemas empotrados es muy grande, con multitud de soluciones dadas por distintos fabricantes o proyectos de código libre.

Entre los SO más utilizados tenemos [Quinnell (2015)]:

- FreeRTOS: Proyecto de código libre de un SO de tiempo real y multiplataforma. Actualmente está portado a más de veinte arquitecturas de microcontroladores distintos. Está diseñado para ser simple y eficiente, y que consuma muy pocos recursos.

- Contiki: Proyecto originario de una universidad de Suecia y que fue diseñado y pensado sobre todo para resolver temas de conectividad y redes. Integra un simulador, de manera que probar distintas soluciones de comunicación se convierte en una tarea relativamente sencilla.

- Micrium (uc/OS-II y uc/OS-III): Sistema operativo multitarea comercial que también se ha portado a múltiples arquitecturas diferentes. La funcionalidad que ofrece es parecida a FreeRTOS y al común de los RTOS (manejo de tareas, comunicación entre ellas, etc.). Las dos versiones son muy parecidas; la versión III fue una mejora de la anterior, sobre todo en el manejo de las prioridades de las tareas.

- RTX: De la compañía Keil, diseñado para la familia Cortex de ARM. Configurable mediante una herramienta gráfica sencilla, permite, por ejemplo, elegir qué tipo de *scheduler* se desea.

- VxWorks: SO de la compañía Wind River y que soporta microcontroladores de varios fabricantes. Es el más avanzado y complejo de los aquí presentados, ya que soporta *multi-core* (esto es, más de una CPU en el sistema), interfaz gráfica de usuario, *stack* de red (TCP/IP), etc.

De todos modos, hay muchísimos SO más y siguen apareciendo nuevos año tras año. La elección de cuál usar, como todo, será un compromiso entre las características técnicas del proyecto y del SO, de los conocimientos y la experiencia del equipo desarrollador y, probablemente, de variables de mercado, como el precio, el tipo de soporte ofrecido, etc.

Bibliografía

Quinnell, R. (2015). «2015 Embedded Markets Study». Report técnico, ARM, EDN network.