Sistemas Empotrados en Tiempo Real

Una introducción basada en FreeRTOS y en el microcontrolador ColdFire MCF5282

José Daniel Muñoz Frías

ISBN: 978-84-612-9902-7
Primera edición. Febrero 2009.

A Manuela.

Índice general

Prólogo

Este libro pretende ser una introducción a la programación de sistemas empotrados basados en microcontrolador y a la programación en tiempo real. Su orientación es fundamentalmente docente, ya que el libro no es más que el fruto de varios años de experiencia del autor impartiendo la asignatura "sistemas informáticos en tiempo real" de la titulación de Ingeniería Industrial de la Escuela Técnica Superior de Ingeniería (ICAI) de la Universidad Pontificia Comillas.

La motivación para escribirlo ha sido la falta de bibliografía similar en castellano. Los únicos libros de programación en tiempo real existentes tratan de grandes sistemas informáticos programados en ADA o C++. Por otro lado, los libros centrados en programación de microcontroladores suelen pasar por alto la problemática de la programación en tiempo real.

El libro supone unos conocimientos previos del lector de programación en lenguaje C y de la arquitectura de un microcontrolador. No obstante, en el libro se estudian las técnicas de programación en bajo nivel que no se suelen estudiar en los cursos de C introductorios y también se exponen los detalles necesarios del microcontrolador usado para ilustrar los ejemplos.

La estructura del libro es la siguiente: el primer capítulo es una introducción a los sistemas en tiempo real. En él se estudia qué es un sistema en tiempo real y las distintas técnicas de programación existentes para implantarlo. De todas ellas se estudia en detalle la más simple: el bucle de *scan*. En este capítulo también se ofrece un resumen de los distintos tipos de *hardware* usados para implantar este tipo de sistemas. El segundo capítulo estudia las características del lenguaje C que son útiles para la programación en bajo nivel, ya que estas no suelen abordarse en los cursos introductorios. El tercer capítulo estudia en detalle la segunda técnica introducida en el primer capítulo para implantar un sistema en tiempo real: los sistemas *Foreground/Background*. Esta técnica es válida para implantar muchos sistemas en tiempo real de complejidad media. No obstante, cuando la complejidad del sistema aumenta, es necesario el uso de un sistema en tiempo real, lo cual se estudia en el cuarto capítulo.

Todo el libro está ilustrado con ejemplos reales. El problema de usar un ejemplo real es que es necesario decantarse por una arquitectura, que puede que no tenga nada que ver con la que ha de usar el lector. No obstante,

al programarse en alto nivel, las diferencias son mínimas. Además, si se sigue un buen estilo de programación, pueden conseguirse programas portables entre distintos microcontroladores. Para los ejemplos se ha usado el microcontrolador MCF5282 de la firma Freescale. Este microcontrolador pertenece a la familia ColdFire, la cual es bastante popular en la industria. Por otro lado, para los ejemplos basados en un sistema operativo en tiempo real se ha usado el sistema operativo FreeRTOS [Barry, 2007], el cual es un sistema operativo muy sencillo y además de *software* libre, lo que permite estudiar su código fuente.

El sistema operativo FreeRTOS se encuentra en www.freertos.org

El libro termina con dos apéndices. En el primero se resumen, a modo de guía de referencia, las funciones de FreeRTOS usadas en los ejemplos del libro. En el segundo se muestra la implantación de un autómata programable para mostrar cómo se implanta un sistema en tiempo real en la práctica. Se han realizado tres versiones: la primera basada en un bucle de *scan*, la segunda usando un sistema *Foreground/Background* y la tercera usando FreeRTOS.

Agradecimientos

Los ejemplos de este libro se han podido realizar gracias a la generosa donación de placas de desarrollo para el microcontrolador MCF5282 y del entrono de programación CodeWarrior por parte de la firma Freescale. Es por tanto justo agradecer desde aquí su apoyo y a la labor de Enrique Montero y David Luque que nos han ofrecido todo lo que les hemos pedido.

El libro se ha escrito usando herramientas de *software* libre. Para editarlo se ha usado Emacs y se ha maquetado con LaTeX. Las fotografías se han editado con Gimp y las figuras se han creado con xfig. Todos estos programas se han ejecutado en Linux. Por último, se ha usado el sistema operativo en tiempo real FreeRTOS para ilustrar los ejemplos del libro. Por ello es necesario agradecer la labor y dedicación de todos los creadores de *software* libre sin los cuales no hubiera sido posible este libro.

Por último, aunque no por ello menos importante, hay que recordar que la escritura de un libro es un sumidero de tiempo que sólo conocen los que están alrededor del autor. Por eso tengo que agradecerle a mi esposa su paciencia y comprensión durante todo el tiempo que le he dedicado a la escritura de este libro, ya que ese tiempo no se lo he podido dedicar a ella.

Convenciones usadas en el texto

Para facilitar la lectura del texto, se han usado una serie de convenciones tipográficas, las cuales se enumeran a continuación:

inglés	Las palabras que se expresan en su idioma original se han escrito en *cursiva*.
`código`	Los ejemplos de código en C se muestran con un tipo de letra `monoespaciada`.
`int`	Las palabras reservadas del lenguaje se escriben con un tipo de letra **`monoespaciada y negrita`**.
`/* Ojo */`	Los comentarios del código se escriben con un tipo de letra *`monoespaciada y cursiva`*.

Tenga en cuenta que estas convenciones tipográficas son sólo para facilitar la lectura. Cuando escriba un programa no intente cambiar el tipo de letra. No obstante, la mayoría de los compiladores actuales disponen de editores de código que colorean automáticamente las palabras clave y los comentarios.

Además se han usado una serie de notas al margen para mostrar material adicional. El tipo de información mostrado en la nota al margen se indica mediante un icono:

Documentos adicionales disponibles en la página web del libro. La dirección de la página es: `http://www.dea.icai.upcomillas.es/daniel/libroTR`

Ejercicios propuestos.

CAPÍTULO 1

Introducción

En este capítulo se introduce en primer lugar cómo se clasifican los programas en función de sus restricciones temporales. A continuación se presenta el concepto de sistema en tiempo real y el porqué son necesarias técnicas de programación especiales para programar sistemas informáticos que tengan que responder en tiempo real. Se introducirán a continuación los distintos métodos que existen para implantar un sistema en tiempo real y por último se introducirán las distintas alternativas *hardware* usadas en la industria para implantar este tipo de sistemas.

1.1. Motivación

En la industria existen multitud de sistemas con restricciones temporales. Algunos ejemplos son:

- Sistemas de control
- Sistemas multimedia
- Videojuegos

A continuación se exponen en detalle las características de cada uno de estos sistemas.

1.1.1. Sistemas de control

En la industria existen un sinfín de aplicaciones en las que es necesario controlar un sistema. Por ejemplo, en un horno es necesario controlar su temperatura, en un motor eléctrico su velocidad, etc. Antiguamente este control se realizaba o bien manualmente, o bien con sistemas mecánicos.[1] Con el desarrollo de la electrónica, estos controladores pasaron a realizarse con circuitos analógicos primero y con microprocesadores después. En

[1] El primer regulador automático de la historia es el regulador de Watt, el cual mediante un sistema mecánico regulaba la velocidad de giro de la máquina de vapor.

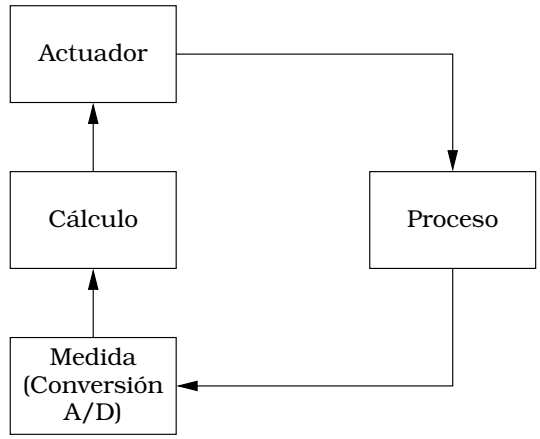


Figura 1.1: Diagrama de bloques de un sistema de control.

cualquier caso, el proceso a llevar a cabo es el mismo, el cual se muestra en la figura 1.1. Para controlar el sistema, en primer lugar es necesario medir ciertas variables de éste, como por ejemplo la temperatura en el caso de un horno, o la velocidad y la intensidad en el caso de un motor eléctrico. Para ello se usa en la mayoría de los casos un conversor analógico/digital, que convierte la magnitud medida a un valor binario. A partir de estas medidas se ejecuta un algoritmo de control. Por ejemplo, para regular la temperatura de un horno puede usarse un regulador PID. La ejecución del algoritmo de control genera unas salidas que, por medio de unos actuadores, modifican el estado del sistema. Siguiendo con el ejemplo del regulador de temperatura de un horno eléctrico, la salida del regulador será la tensión que es necesario aplicar a la resistencia, que es el actuador en este caso.

Hay que tener en cuenta que si el sistema de control se implanta con un microprocesador, el proceso completo: medidas, cálculo y actuación ha de realizarse periódicamente. A este periodo se le denomina "periodo de muestreo" y se denota como T_S. Este tiempo depende de las características del sistema a controlar. Así en un horno puede ser de decenas de segundos, pero en un motor suele ser del orden de unos pocos milisegundos. Lo importante es que este periodo de muestreo es inviolable: si el controlador no es capaz de realizar el ciclo completo de medir, calcular y actuar en dicho periodo de muestreo, el sistema quedará en bucle abierto, lo cual puede tener consecuencias catastróficas. Por ejemplo, si un circuito de control de un motor viola el tiempo de muestreo, el motor puede girar más de lo debido e incluso, si dicho motor está moviendo un robot, pueden ponerse en peligro vidas humanas.

Por tanto, en un sistema de control, las restricciones temporales son muy duras, tanto que se considera que el sistema funciona mal si no es capaz de tener los resultados a tiempo, aunque éstos sean correctos desde un punto de vista lógico.

1.1.2. Sistemas multimedia y videojuegos

Otro tipo de sistemas con restricciones temporales son los sistemas multimedia. Por ejemplo, un reproductor de DVD ha de ser capaz de leer, decodificar y representar en la pantalla un fotograma de la película cada 20 ms. Lo mismo ocurre con un videojuego: en este caso, el programa ha de leer la posición del *joystick*, recalcular la escena teniendo en cuenta la física del juego y redibujar dicha escena; todo ello en menos de 20 ms para que el jugador note un movimiento fluido. No obstante, en ambos casos, una pérdida ocasional de un fotograma no origina ninguna catástrofe, pudiendo incluso pasar desapercibido para el usuario.

Por tanto, en un sistema multimedia, aunque existen restricciones temporales, éstas no son tan duras y se pueden saltar ocasionalmente.

1.1.3. Aplicaciones ofimáticas

En el caso del resto de aplicaciones informáticas, como por ejemplo un procesador de texto o un sistema CAD, las restricciones temporales no existen: mientras el programa responda lo suficientemente rápido para que el usuario se sienta cómodo, el comportamiento del sistema se considera satisfactorio. Por ejemplo, no pasa nada si un procesador de texto hace una pequeña pausa cada 10 minutos, mientras guarda una copia de seguridad al disco por si se va la luz o se "cuelga" el programa.

1.1.4. Tipos de aplicación y sistemas operativos

Según se ha visto, existen tres tipos de aplicaciones en función del tiempo de respuesta. Ello implica que para implantar cada una de ellas habrá que usar un sistema operativo distinto.

En el caso de una aplicación ofimática, que como hemos visto ha de responder "suficientemente rápido" al usuario, pero no tiene ninguna restricción temporal; basta con usar un sistema operativo de propósito general, como Linux, Mac OS X, Windows, etc.

En el caso de una aplicación multimedia, que tiene restricciones temporales "suaves", es necesario usar un sistema operativo de tiempo real no estricto (denominados en inglés *soft real time*). Casi todos los sistemas operativos de propósito general actuales disponen de extensiones para ejecutar aplicaciones multimedia en tiempo real. En estos casos, como seguramente habrá podido observar, si el ordenador está muy sobrecargado, no se consigue el tiempo de respuesta deseado. Por ejemplo, si intentamos visualizar

un DVD mientras el ordenador está sobrecargado, se producirán "saltos" en la imagen.

Por último, en el caso de un sistema de control, las restricciones temporales son "duras", es decir, no se pueden saltar. Por ello, el sistema operativo ha de **garantizar** el tiempo de respuesta. Esto sólo lo consiguen los sistemas operativos diseñados con esta restricción en mente. A este tipo de sistemas se les denomina sistemas operativos en tiempo real estricto (*hard real time* en inglés).

1.2. Definición de sistema en tiempo real

Existen numerosas definiciones de un sistema en tiempo real. No obstante, la que mejor resume las características de dicho tipo de sistemas es la siguiente:

> Un sistema informático en tiempo real es aquel en el que la corrección del resultado depende tanto de su validez lógica como del instante en que se produce.

Es decir, en un sistema en tiempo real es tan importante la validez lógica de los cálculos como la validez temporal de los mismos. Incluso en algunos sistemas de seguridad críticos puede ser más importante el tiempo de respuesta que la validez lógica, pudiendo ser necesario elegir un método de cálculo aproximado más rápido con vistas a cumplir las restricciones temporales.

Para garantizar el tiempo de respuesta, un sistema en tiempo real necesita no solo velocidad de respuesta, sino **determinismo**. Es decir, el tiempo de ejecución de los programas de tiempo real ha de estar acotado en el caso más desfavorable, de forma que se garantice que se cumplirán siempre las restricciones temporales.

Por último, conviene destacar que un sistema de control en tiempo real no tiene por qué ser un sistema que se ejecute rápidamente. Simplemente su velocidad ha de ser acorde con la planta con la que está interactuando. Por ejemplo, el control de temperatura de un horno, al tener éste una constante de tiempo de muchos segundos, no impone una restricción temporal muy elevada. En este caso el algoritmo de control puede ejecutarse por ejemplo cada 20 segundos, con lo cual con un simple microprocesador de 8 bits es más que suficiente. Por el contrario, para poder ejecutar el último videojuego a una velocidad de refresco de pantalla mayor de 20 ms es necesario usar prácticamente un superordenador.

1.3. Tareas

Como se ha mencionado anteriormente, una de las aplicaciones típicas de tiempo real son los sistemas de control. En este tipo de sistemas es

muy frecuente encontrar un *software* dividido en varios niveles jerárquicos [Auslander et al., 1996]. Empezando por el nivel más cercano al *hardware*, éstos son:

- **Adquisición de datos y actuadores.** Este es el nivel más bajo en la jerarquía, encargado de interactuar con el *hardware* del sistema a controlar. El *software* de este nivel suele ser corto y normalmente es necesario ejecutarlo frecuentemente. La ejecución suele estar controlada por un temporizador o por una señal externa generada por el propio *hardware*. En ambos casos se usan interrupciones para detectar el fin del temporizador o la activación de la señal externa.

- **Algoritmos de control (PID).** Este nivel se corresponde con los algoritmos de control que, a partir de las medidas obtenidas por el sistema de adquisición de datos, ejecutan un algoritmo de control digital; como por ejemplo un control PID o un control adaptativo más sofisticado. El resultado de sus cálculos es el valor que hay que enviar a los actuadores de la planta. Normalmente este *software* ha de ejecutarse con un periodo de tiempo determinado (periodo de muestreo).

- **Algoritmos de supervisión (trayectorias).** Existen sistemas de control sofisticados en los que existe una capa de control a un nivel superior que se encarga de generar las consignas a los controles inferiores. Por ejemplo, pensemos en un controlador para un brazo robot. Existe un nivel superior que se encarga de calcular la trayectoria a seguir por el brazo y en función de ésta, genera las consignas de velocidad para los motores de cada articulación. Al igual que los algoritmos de control, estos algoritmos se ejecutan también periódicamente, aunque normalmente con un periodo de muestreo mayor que los algoritmos de control de bajo nivel.

- **Interfaz de usuario, registro de datos, etc.** Además del *software* de control, es normalmente necesario ejecutar otra serie de tareas como por ejemplo un interfaz de usuario para que éste pueda interactuar con el sistema; un registro de datos para poder analizar el comportamiento del sistema en caso de fallo, comunicaciones con otros sistemas, etc. Al contrario que en resto de niveles, en éste no existen restricciones temporales estrictas, por lo que se ejecuta cuando los niveles inferiores no tienen nada que hacer.

Como podemos ver en este ejemplo, es necesario realizar varias tareas en paralelo. Además la ejecución de cada tarea se realiza de forma asíncrona, respondiendo a sucesos externos al programa. Así, una tarea encargada de tomar medidas de un conversor A/D tendrá que ejecutarse cada vez que éste termine una conversión, mientras que la tarea de control lo hará cuando expire su periodo de muestreo.

Por otro lado, no todas las tareas son igual de urgentes. Normalmente cuanto más abajo están en la jerarquía mencionada anteriormente, las tareas son más prioritarias. Por ejemplo, la tarea encargada de leer el conversor A/D no puede retrasar mucho su ejecución, pues si lo hace puede que cuando vaya a leer el dato, el conversor haya realizado ya otra conversión y se pierda así el dato anterior.

En definitiva, el paralelismo es una característica inherente a los sistemas en tiempo real. Así, mientras que un programa convencional tiene una naturaleza secuencial, es decir, se divide en funciones que se ejecutan según un orden preestablecido, en un sistema en tiempo real el programa se divide en tareas que se ejecutan en paralelo, cada una de ellas con una determinada prioridad, para poder cumplir con sus restricciones temporales.

Salvo que se disponga de un sistema con varios procesadores, la ejecución en paralelo se consigue mediante la sucesión rápida de ejecuciones parciales de las tareas. Es decir, el sistema ejecuta una tarea (tarea 1) durante un periodo de tiempo (por ejemplo 5 ms), otra tarea (tarea 2) durante otro periodo de tiempo y así sucesivamente. Como el sistema conmuta rápidamente entre tarea y tarea, desde fuera da la impresión que se están ejecutando todas las tareas en paralelo. Este modo de programación presenta numerosas dificultades, que serán abordadas a lo largo de este libro. De momento cabe destacar que:

- Son necesarias técnicas para permitir la conmutación entre las diversas tareas, respetando las restricciones temporales de cada una de ellas.
- También es necesario establecer mecanismos para el intercambio de información entre tareas.
- La depuración de este tipo de sistemas es compleja, ya que su comportamiento no depende sólo de las entradas, sino también del instante en el que éstas cambian. Por tanto, es posible que un programa falle sólo si cambia una entrada en un instante determinado y no en otro. Por supuesto, conseguir reproducir un error de este estilo repetidamente para depurarlo es imposible.[2]

1.4. Métodos para implantar un sistema en tiempo real

Existen varias técnicas de programación para ejecutar un conjunto de tareas en paralelo, las cuales son, de menor a mayor complejidad de im-

[2] Por el contrario, en un programa convencional su ejecución sigue una estructura secuencial y predecible, es decir, para un mismo conjunto de entradas, el comportamiento del programa es siempre el mismo. Esto facilita mucho la depuración, pues es posible repetir un error una y otra vez hasta descubrir qué lo está originando.

plantación:

- Procesamiento secuencial (Bucle de *scan*).
- Primer plano / segundo plano (*Foreground*/*Background*).
- Multitarea cooperativa.
- Multitarea expropiativa (*preemptive*).

En las siguientes secciones de exponen cada una de estas técnicas en mayor detalle.

1.5. Procesamiento secuencial

La técnica de procesamiento secuencial consiste en ejecutar todas las tareas consecutivamente una y otra vez dentro de un bucle infinito, tal como se muestra en el siguiente código:

```
main()
{
  /* Inicialización del sistema */
  while(1){   /* Bucle infinito */
    Tarea1();
    Tarea2();
    /* ... */
    TareaN();
  }
}
```

A esta técnica se le denomina también bucle de *scan* debido a que está basada en un bucle infinito que ejecuta una y otra vez todas las tareas. Nótese que en un programa convencional, un bucle infinito se considera un error, ya que todos los programas terminan en algún momento, por ejemplo cuando el usuario elige la opción salir del menú. Por el contrario, en un sistema empotrado el programa de control no ha de terminar nunca, por lo que lo normal en este tipo de sistemas es precisamente disponer de un bucle sin fin.

Es preciso resaltar que cada una de las tareas se ejecuta desde principio a fin, ejecutándose todas ellas secuencialmente según un orden preestablecido. La característica fundamental de este tipo de sistemas es que las tareas no pueden bloquearse a la espera de un evento externo, como por ejemplo una llamada a `scanf` o un bucle de espera del tipo:

```
  while(evento_externo == FALSO);
    /* Espera un evento externo */
```

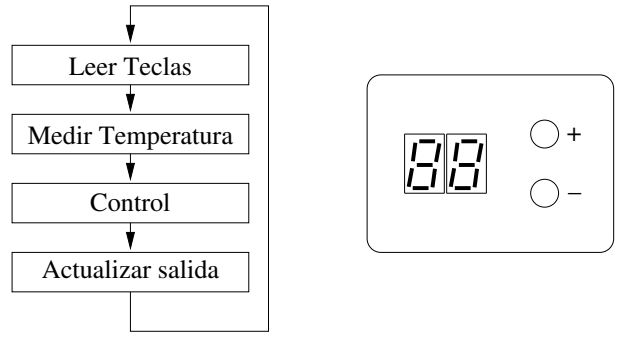


Figura 1.2: Termostato digital.

Ello es debido a que estos eventos externos son asíncronos con el funcionamiento del sistema, por lo que el tiempo que va a tardar la tarea que espera el evento no está acotado. En consecuencia no puede garantizarse el tiempo que va a tardar el bucle de *scan* completo en terminar un ciclo, violándose entonces el principio fundamental de un sistema en tiempo real, que dice que el tiempo de respuesta máximo ha de estar garantizado.

Si todas las tareas cumplen el requisito de no bloquearse, entonces el procesamiento secuencial es el método más simple para implantar un sistema en tiempo real. La única condición que debe de cumplirse, aparte del no bloqueo, es que la suma de los tiempos máximos de ejecución de todas las tareas sea inferior al periodo de muestreo necesario.

1.5.1. Ejemplo: un termostato digital

En la figura 1.2 se muestra un ejemplo sencillo de un sistema en tiempo real implantado mediante procesamiento secuencial. El sistema es un termostato para la calefacción. Como se puede observar, el sistema consta de cuatro tareas que se ejecutan continuamente:

- La primera verifica si hay alguna tecla pulsada (UP o DOWN) y en caso afirmativo modifica la consigna de temperatura. Si no hay ninguna tecla pulsada, simplemente termina de ejecutarse (no bloqueo).

- La segunda tarea realiza una medida de la temperatura de la habitación.

- La tercera ejecuta el control. Éste puede ser un control sencillo todonada, consistente en que si la temperatura es mayor que la consigna se apaga la calefacción y si es inferior se enciende.

- Por último, la cuarta tarea se encarga de encender o apagar la calefacción en función de la salida de la tarea de control.

Un pseudocódigo de la tarea que lee el teclado es:

```
tecla = LeerTeclado(); /* no bloquea */
if(tecla == UP){
  consigna ++;
}
if(tecla == DOWN){
  consigna --;
}
```

Como se puede observar, la característica fundamental de esta tarea es que la función `LeerTeclado()` no ha de bloquearse esperando a que se pulse una tecla. Un modo de funcionamiento típico en estos casos el es siguiente: cada vez que se llame a la función, ésta comprobará si hay una tecla pulsada y en caso afirmativo devolverá un código de tecla y en caso negativo devolverá un `NULL` para indicarlo.

Una vez que se ha leído el teclado, basta con analizar si se ha pulsado la tecla `UP` o la tecla `DOWN` y actuar en consecuencia. Obviamente, si no se ha pulsado ninguna tecla la función termina sin modificar el valor de la consigna de temperatura.

La función encargada de medir la temperatura de la habitación podría implantarse según el siguiente pseudocódigo:

```
LanzarConversionAD();
while(Convirtiendo());
  /* Esperar EOC */
temp = LeerConversorAD();
```

En este pseudocódigo se ha supuesto que el termostato tiene conectado un sensor de temperatura a un conversor analógico/digital. El funcionamiento de este tipo de circuitos es el siguiente:

- En primer lugar es necesario enviarles una señal para que inicien una conversión. Si el conversor está integrado en el microcontrolador normalmente basta con escribir un valor en un registro.

- Una vez lanzada la conversión, se necesita esperar un tiempo para que ésta se realice. El conversor A/D suele indicar el final de la conversión activando una señal que puede monitorizarse desde el microcontrolador (nuevamente accediendo a algún registro de control).

- Cuando finaliza la conversión, el microcontrolador puede acceder a un registro donde se encuentra la medida obtenida.

A la vista de este pseudocódigo, probablemente se pregunte cómo es posible que se esté usando un bucle de espera:

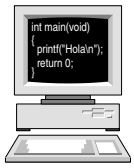
```
while(Convirtiendo());
  /* Esperar EOC */
```

si se acaba de decir que este tipo de tareas no pueden esperar un evento externo asíncrono.

La respuesta es simple: la espera al fin de la conversión no es un suceso asíncrono, pues la tarea lanza la conversión y el tiempo que tarda el conversor en finalizar está acotado, por lo que se sabe cuanto va a durar esta tarea como máximo, que es el único requisito indispensable en un sistema en tiempo real.

El pseudocódigo de la función de control es:

```
if(temp < consigna){
  calefaccion = ON;
}else if(temp > consigna + HISTERESIS){
  calefaccion = OFF;
}
```



Como puede observar, la función de control en este caso es muy simple. Lo único que se hace es comparar la temperatura medida con la consigna y decidir si hay que conectar o apagar la calefacción.

Nótese que se ha añadido una histéresis para evitar que cuando la temperatura tenga un valor cercano a la consigna, la calefacción esté continuamente encendiéndose y apagándose.

Realice el ejercicio 1.

1.5.2. Temporización del bucle de scan

En el ejemplo anterior, el programa ejecuta el bucle de *scan* a toda velocidad, lo cual puede ser lo más conveniente en muchos casos. No obstante, existen aplicaciones en las que es necesario ejecutar las tareas con un periodo de tiempo determinado y exacto. Un ejemplo de este tipo de aplicaciones son los sistemas de control digital, que han de ejecutar el ciclo medida, control y actuación cada periodo de muestreo T_s. En este tipo de sistemas, si varía el periodo T_s, las propiedades del regulador cambian, lo cual obviamente no es muy conveniente.

No obstante, es muy fácil conseguir que este tipo de sistemas ejecuten el bucle de *scan* con un periodo determinado, siempre que este periodo sea mayor que la suma de los tiempos máximos de ejecución de las tareas que componen el bucle. Para ello, basta con añadir un temporizador (*timer*) al *hardware* del sistema[3] y añadir una última tarea que espera el final del temporizador. Esta tarea se suele denominar tarea inactiva (*idle*). La tarea

[3]Por regla general, todos los ordenadores y microcontroladores disponen de varios temporizadores.

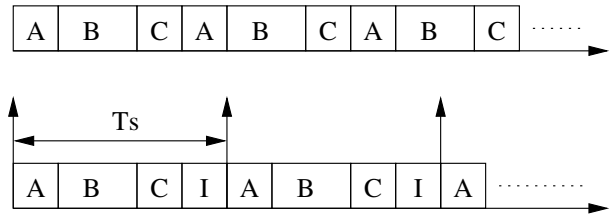


Figura 1.3: Temporización de tareas.

constará simplemente de un bucle de espera que estará continuamente comprobando si ha finalizado la cuenta del temporizador:

```
void TareaInactiva(void)
{
  while( ! fin_timer );
    /* Espera el final del periodo */

  ReinicializaTimer();
}
```

Una vez finalizada la cuenta, la tarea reinicializará el temporizador y devolverá el control al programa principal, con lo que volverá a empezar el ciclo de *scan*.

En la figura 1.3 se muestra la secuencia de tareas en ambos tipos de procesos. En la parte superior, el bucle de *scan* se repite continuamente, por lo que no se controla el periodo de muestreo. Si las tareas no tardan siempre lo mismo o si se cambia de procesador, el periodo T_s variará.

En la parte inferior se ilustra la secuencia de tareas en un sistema con control del periodo. Como puede observar, se ha añadido una tarea inactiva (I) que se queda a la espera del final del temporizador (mostrado con una flecha en la figura). De esta forma se consigue un periodo de muestreo independiente del tiempo de ejecución de las tareas y del *hardware*.

1.5.3. Tareas con distinto periodo de muestreo

Según se expuso en la sección 1.3, en las aplicaciones de control complejas existen tareas con un periodo de ejecución mayor que otras. El soporte de este tipo de aplicaciones por el procesamiento secuencial es muy básico, ya que lo único que se puede hacer es ejecutar una tarea cada n periodos de muestreo, con lo que su periodo será $n \times T_s$. Si alguna tarea ha de ejecutarse con un periodo que no es múltiplo del periodo básico T_s, entonces es necesario recurrir a un sistema *Foreground/Background* con varios temporizadores o a un sistema operativo de tiempo real.

Figura 1.4: Temporización de tareas. La tarea B tiene un periodo = $2T_s$.

En la figura 1.4 se muestra un diagrama de tiempos de un sistema con tres tareas, en el cual dos de ellas (A y C) se ejecutan cada T_s y otra tarea (B) se ejecuta cada $2T_s$.

Existen dos formas de implantar un bucle de *scan* en el que las tareas tienen distintos periodos. En primer lugar, se estudia la menos eficiente:

```
n=0;
for(;;){ /*for-ever*/
  TareaA();
  if(n%2==0){
    TareaB();
  }
  n++;
  TareaC();
  TareaInactiva();
}
```

La variable `n` se usa para llevar una cuenta del número de ejecuciones del bucle. En este ejemplo se usa el operador % (resto) para averiguar si el periodo actual es múltiplo del periodo de muestreo de la tarea B, que en este caso es $2T_s$. Este es el método más simple, aunque tiene un pequeño inconveniente: el operador % para obtener el resto tiene que dividir ambos operandos, lo cual es lento. De hecho, la mayoría de los microcontroladores de bajo coste no incluyen *hardware* para realizar la división, por lo que ésta ha de realizase mediante *software*, con la sobrecarga que ello conlleva. Por otro lado, tal como se ha implantado el programa, cuando el contador `n` rebose (pase de 11111111 a 00000000), puede ocurrir que se ejecute la tarea con un periodo distinto al estipulado. ¿Qué condición ha de cumplir el periodo de muestreo de la tarea B para que no ocurra este error?

Para evitar la realización de la división, se puede usar el algoritmo siguiente:

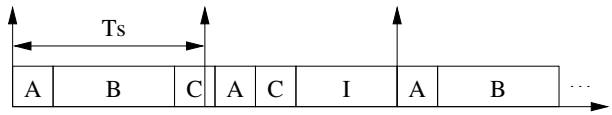


Figura 1.5: Temporización de tareas. La tarea B tiene un tiempo de ejecución demasiado largo.

```
n=2;
for(;;){ /*forever*/
  TareaA();
  if(n==2){
    TareaB();
    n=0;
  }
  n++;
  TareaC();
  TareaInactiva();
}
```

Aquí el contador se mantiene siempre entre 0 y el periodo de la tarea, volviendo a poner el contador a cero cuando éste se hace igual al periodo de la tarea. Obviamente, como no se hace ninguna división, este algoritmo es mucho más eficiente.

1.5.4. Tareas con tiempo de ejecución largo

En la figura 1.5 se muestra un caso típico en el que la aplicación del bucle secuencial empieza a dar problemas. Como se puede observar, la tarea B tiene un periodo de $2T_s$, pero tarda en ejecutarse más de la mitad del periodo de muestreo, dejando sin tiempo a la tarea C para terminar antes del final del periodo de muestreo. En el siguiente periodo, la tarea B no se ejecuta, por lo que ahora sobra un montón de tiempo.[4] Nótese que este sistema no cumple con sus restricciones temporales la mitad de los periodos de muestreo, por lo que no es válido.

Una primera solución puede consistir en cambiar el procesador por uno más rápido. Desde luego, esto es lo más cómodo, pero no lo más eficiente. La otra solución es obvia si se observa que dividiendo la tarea B en dos mitades, el sistema será capaz de ejecutar las tareas A y C y la mitad de B en cada periodo de muestreo, tal como se ilustra en la figura 1.6. No

[4]Nótese que al finalizar el primer periodo de muestreo la tarea inactiva nada más ejecutarse devolverá el control, ya que el temporizador ya habrá terminado su cuenta. Por ello no se ha mostrado su ejecución en la figura 1.5.

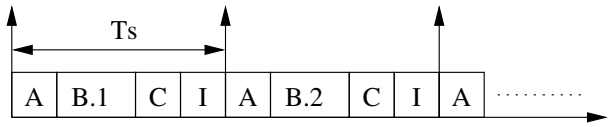


Figura 1.6: Temporización de tareas. La tarea B se ha dividido en dos partes: B.1 y B.2.

obstante, esta técnica no es tan fácil como parece, pues habrá que estimar el tiempo de ejecución de la tarea para averiguar por dónde se puede dividir. Esta estimación se complica cuando el tiempo de ejecución de la tarea es variable (por ejemplo si hay un bucle que se ejecuta un número de veces que depende de los datos de entrada).

La forma de implantar una tarea larga (B) repartida entre varios periodos de muestreo (B.1 y B.2) se muestra a continuación:

```
void TareaB(void)
{
  static int estado = 1;
  switch(estado){
  case 1:
    /* tareas B.1 */
    estado = 2;
    break;
  case 2:
    /* tareas B.2 */
    estado = 1;
    break;
  }
}
```

El bucle principal seguirá ejecutando en cada iteración del bucle todas las tareas, es decir, será algo parecido a:

```
  for(;;){
    TareaA();
    TareaB();
    TareaC();
    TareaInactiva();
  }
```

Por tanto la tarea B ha de guardar información de su estado para saber qué parte tiene que ejecutar en cada llamada, lo cual se hace mediante la variable estática `estado`. Además, si se necesita guardar una variable de

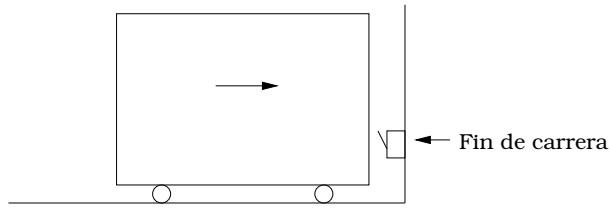


Figura 1.7: Ejemplo de sistema con requerimientos de baja latencia

un periodo de ejecución al siguiente, también tendrá que declararse como **`static`** para que mantenga su valor entre la ejecución B.1 y la B.2.

El inconveniente de este sistema es que si existen varias tareas con varios periodos de muestreo y varios tiempos de ejecución, el dividir todas las tareas para que se cumplan siempre los periodos de muestreo puede ser un problema complejo, si no imposible. Esto hace que en estos casos sea imprescindible el uso de métodos más avanzados para diseñar el sistema de tiempo real.

Realice el ejercicio 2.

1.5.5. Latencia de las tareas en el bucle de scan

La latencia en un sistema de tiempo real se define como el tiempo máximo que transcurre entre un evento (interno o externo) y el comienzo de la ejecución de la tarea que lo procesa. En la mayoría de los sistemas de control existen tareas que requieren una baja latencia. Por ejemplo, supóngase que una puerta de garaje está controlada por un microprocesador. Para detectar cuándo llega la puerta al final de su recorrido se ha instalado un fin de carrera, tal como se muestra en la figura 1.7. Cuando la puerta toque el fin de carrera, habrá que dar la orden de parada del motor inmediatemente, pues si no la puerta seguirá moviéndose y chocará contra la pared. Por tanto el tiempo que transcurra desde que se detecte la activación del fin de carrera hasta que se pare el motor ha de estar acotado.

Otro inconveniente del procesamiento secuencial es que la latencia pue-

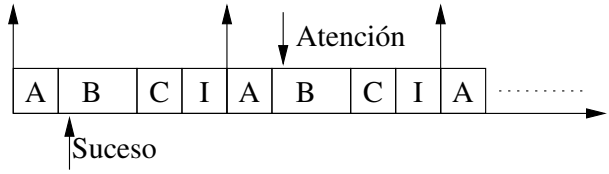


Figura 1.8: Latencia en el bucle de *scan*

de ser elevada. En un bucle de *scan*, el caso más desfavorable se da cuando el suceso externo ocurre justo después de su comprobación por parte de la tarea. En este caso tiene que pasar todo un periodo de muestreo hasta que la tarea vuelve a ejecutarse y comprobar si ha ocurrido dicho suceso, tal como se puede observar en la figura 1.8. Por tanto, la latencia de un sistema basado en procesamiento secuencial es igual al tiempo máximo que tarda en ejecutarse el bucle de *scan*.

Por tanto, si el bucle de *scan* no es lo suficientemente rápido, la dos alternativas son, o conseguir un procesador más rápido, o usar otra técnica más avanzada.

1.5.6. Ventajas e inconvenientes del bucle de scan

Como hemos podido apreciar, el procesamiento secuencial tiene como principal ventaja la facilidad de su implantación. Si ha realizado el ejercicio 1 se habrá dado cuenta de que sin tener ninguna experiencia ha sido capaz de diseñar el *software* de control de un termostato.

Por otro lado, como el orden de ejecución de las tareas es fijo y cada una de ellas se ejecuta de principio a fin, no existe ningún problema a la hora de compartir información entre tareas.

Por último, conviene destacar que este tipo de sistemas son los más eficientes, ya que no se pierde tiempo realizando cambios de contexto,[5] cosa que sí ocurre en el resto de casos.

El principal inconveniente de esta metodología es la latencia, que es igual al tiempo que tarda en ejecutarse el bucle de *scan*. Si éste bucle no es lo suficientemente rápido, la única alternativa es conseguir un procesador más rápido o usar otra técnica más avanzada.

El otro gran inconveniente del procesamiento secuencial, es la dificultad de implantar sistemas en los que existen tareas con distintos periodos de muestreo.

En conclusión, este método sólo es válido para sistemas muy sencillos en los que sólo hay que ejecutar cíclicamente una serie de tareas, todas ellas con el mismo periodo. Para sistemas más complejos, es necesario usar alguno de los métodos más avanzados que se exponen brevemente a continuación y en más detalle en el resto del texto.

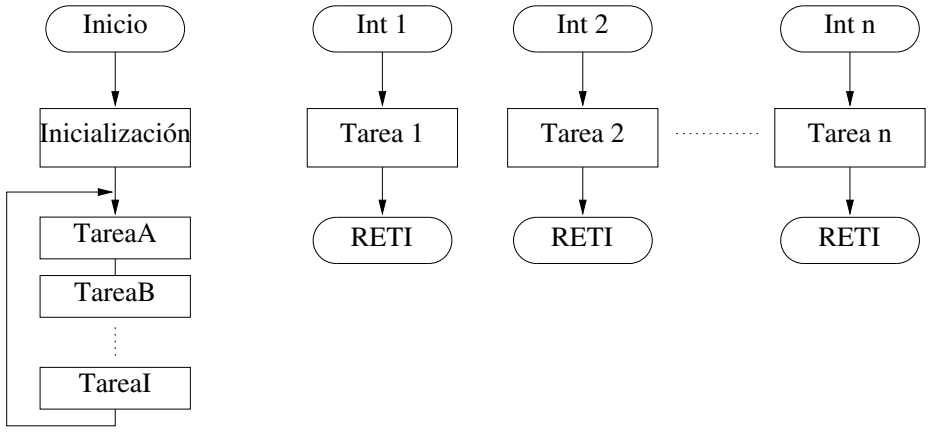


Figura 1.9: Ejemplo de sistema primer plano / segundo plano

1.6. Sistemas *Foreground/Background*

En la sección anterior se ha visto que uno de los principales inconvenientes del procesamiento secuencial, es que la latencia de todas las tareas es igual al tiempo máximo de una iteración del bucle de *scan*. Para conseguir disminuir la latencia de ciertas tareas, se pueden asociar estas tareas a interrupciones, dando lugar a un sistema como el mostrado en la figura 1.9.

A este tipo de sistemas se les denomina "tareas en 1er plano /tareas en 2º plano" (*Foreground/Background* en inglés).[6] El nombre refleja precisamente el modo de implantar este tipo de sistemas, en el cual, tal como se ilustra en la figura 1.9, existen dos tipos de tareas:

- Un programa principal que se encarga de inicializar el sistema de interrupciones y luego entra en un bucle sin fin. Dentro de este bucle sin fin se ejecutarán las tareas de 1er plano: TareaA, TareaB . . . TareaI.
- Tareas de 2º plano, encargadas de gestionar algún suceso externo que provoca una interrupción. En la figura 1.9 estas tareas son las

[5]Por contexto se entiende toda la información sobre la ejecución de una tarea: registros del microprocesador, pila, etc. En el resto de métodos para implantar un sistema en tiempo real, las tareas se ejecutan sin un orden preestablecido, con lo que en cualquier instante puede ser necesario hacer una pausa en la ejecución de una tarea, ejecutar otra más prioritaria en ese momento y luego volver a ejecutar la primera. Para ello, el cambio de una tarea a otra exige guardar el contexto de la primera y cargar el contexto de la segunda, lo cual necesita un pequeño tiempo.

[6]Otro nombre común en la literatura para este tipo de sistemas es bucle de *scan* con interrupciones.

denominadas Tarea 1, Tarea 2 . . . Tarea n.

La ventaja principal de este sistema es la baja latencia conseguida entre los sucesos externos asociados a las interrupciones y la ejecución de las tareas que atienden a dichos sucesos (Tarea 1, Tarea 2, . . . Tarea n en la figura 1.9). Además de ésto, no necesitan ningún *software* adicional, como un sistema operativo en tiempo real, por lo que no es necesario invertir en su adquisición[7] ni en su aprendizaje.

Entre los inconvenientes de este tipo de sistemas destacan:

- Se necesita soporte de interrupciones por parte del *hardware*, lo cual es lo normal en los microprocesadores actuales.
- Esta estructura sólo es válida si cada una de las tareas de tiempo real pueden asociarse a una interrupción. Si por ejemplo se tienen varias tareas que han de ejecutarse con periodos distintos y sólo hay disponible un temporizador, no queda más remedio que recurrir a un sistema operativo en tiempo real, o asociar varias tareas a una misma interrupción, lo cual complica un poco la programación.
- Este sistema es más complejo de programar y de depurar que el procesamiento secuencial, no solo porque hay que gestionar el *hardware* de interrupciones, sino porque aparecen problemas ocasionados por la concurrencia entre tareas, según se verá más adelante, y por la naturaleza asíncrona de las interrupciones.

1.6.1. Latencia en sistemas primer plano / segundo plano

La latencia en este tipo de sistemas, en una primera aproximación, es igual al tiempo máximo de ejecución de las tareas de interrupción, ya que mientras se está ejecutando una tarea, las interrupciones están inhabilitadas.[8] Por tanto, la duración de las tareas de segundo plano ha de ser lo más pequeña posible.

Veamos un ejemplo para ilustrar este aspecto:[9]

```
void TareaTemp(void) /* Tarea de segundo plano */
{
  static int ms, seg, min, hor;

  ms++;
```

[7] Existen algunos sistemas operativos de tiempo real de código abierto, aunque en estos casos se incurrirán en gastos de soporte, documentación, etc.

[8] O al menos están inhabilitadas las interrupciones de menor o igual prioridad si el procesador soporta varios niveles de interrupción.

[9] En este ejemplo se ha simplificado la sintaxis. En el capítulo 3 se estudiará cómo definir correctamente una tarea para que ésta sea lanzada al producirse una interrupción.

```
  if(ms == 1000){
    ms = 0;
    seg ++;
    if(seg == 60){
      seg = 0;
      min ++;
      if (min == 60){
        min = 0;
        hor ++;
        if (hor == 24){
          hor = 0;
        }
      }
    }
    ImprimeHora(hor, min, seg);
  }
}
```

La tarea mostrada sirve para implantar un reloj. Se ha supuesto que esta tarea está asociada a la interrupción de un temporizador, el cual se ha configurado para que interrumpa cada milisegundo.[10] Supóngase ahora que la función encargada de imprimir la hora tarda en ejecutarse 20 ms. Como por defecto, mientras se está ejecutando una interrupción, el microprocesador mantiene las interrupciones inhabilitadas, durante los 20 ms en los que se está imprimiendo la hora no se reciben interrupciones y por tanto no se actualiza el contador de milisegundos. En consecuencia, cada segundo se atrasan 20 ms, lo cual es un error inaceptable.

Lo que está ocurriendo es que como la latencia es igual al tiempo máximo durante el cual están inhabilitadas las interrupciones, en este ejemplo la latencia es de 20 ms. Ahora bien, como hay una tarea que necesita ejecutarse cada milisegundo, la latencia debería de ser menor de 1 ms para que todo funcionase correctamente.

Por tanto, para que un sistema primer plano / segundo plano funcione correctamente, las tareas de segundo plano han de ser muy simples para que su tiempo de ejecución sea lo más corto posible. Para conseguirlo, es necesario dejar los procesos complejos para una tarea de primer plano.

Volviendo al ejemplo anterior, se puede hacer que la tarea de segundo plano se limite a actualizar las variables `hor`, `min` y `seg`; y la tarea de primer plano se limite a imprimir la hora. El código resultante será:

[10] En la realidad, para implantar un reloj se asociaría a una interrupción con un periodo mayor para no sobrecargar al sistema. Sin embargo existen muchas situaciones en la práctica en las que se producen interrupciones con estos periodos. Tenga en cuenta que este ejemplo está pensado para ilustrar el problema de la latencia, mas que para ilustrar cómo se implantaría un reloj.

```
int seg, min, hor;

void TareaTemp(void) /* Tarea de segundo plano */
{
  static int ms;

  ms++;
  if(ms == 1000){
    ms = 0;
    seg ++;
    if(seg == 60){
      seg = 0;
      min ++;
      if (min == 60){
        min = 0;
        hor ++;
        if (hor == 24){
          hor = 0;
        }
      }
    }
  }
}

int main(void)
{
  InicializaTemporizador();
  HabilitaInterrupciones();

  for(;;){
    ImprimeHora(hor, min, seg);
    /* Resto de tareas de primer plano */
  }
}
```

Tenga en cuenta que las variables `hor`, `min` y `seg` son ahora variables globales, ya que han de compartirse entre la tarea de primer plano y la de segundo plano.

En este segundo ejemplo, la ejecución de la tarea de segundo plano será muy corta (unos cuantos μs) y en consecuencia la latencia será ahora mucho menor de 1 ms, no perdiéndose ninguna interrupción.

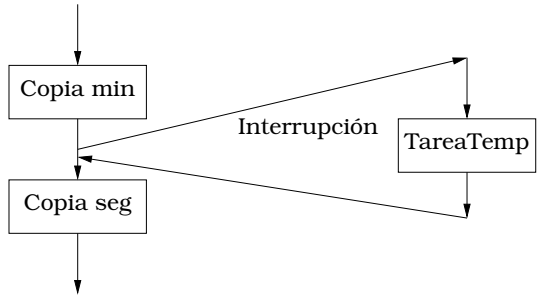


Figura 1.10: Incoherencia de datos.

1.6.2. Datos compartidos

Al trasladar los procesos lentos a la tarea de primer plano se soluciona el problema de la latencia, pero por desgracia se crea un nuevo problema: la comunicación entre tareas que se ejecutan asíncronamente.

En este ejemplo las dos tareas se comunican compartiendo tres variables globales. El problema que se origina se denomina **incoherencia de datos** y se ilustra con el siguiente ejemplo: supóngase que en el programa anterior la interrupción del temporizador se produce cuando se están copiando los argumentos de la tarea `ImprimeHora`, en concreto cuando se ha copiado el minuto de la hora actual, pero aún no se ha copiado el segundo, tal como se ilustra en la figura 1.10.[11] Además, como las leyes de Murphy se cumplen siempre, se puede esperar que en esta interrupción la variable `ms` llegue a 1000, con lo cual se actualizarán los segundos. Si además (de ello también se encargarán las leyes de Murphy) la variable `seg` es igual a 59, dicha variable pasará a valer 0 y la variable `min` se incrementará en 1. En ese momento la interrupción terminará, devolviendo el control a la tarea de primer plano que copiará el valor actualizado de `seg` antes de llamar a la tarea `ImprimeHora`. Lo que se vería en la pantalla es lo siguiente, suponiendo que la interrupción se ha producido en la llamada que imprime la segunda línea:

```
13:13:59
13:13:00
13:14:00
```

Obviamente este error tiene una baja probabilidad de ocurrir, pero seguro que ocurrirá alguna vez y, según indican las leyes de Murphy, cuando sea menos oportuno.

[11] Recuerde que en C los argumentos que se pasan a una función se copian en variables locales de ésta.

El problema que se acaba de exponer se origina al producirse una interrupción dentro de lo que se denomina una **zona crítica**. Por zona crítica se entiende toda zona de código en la que se usan recursos[12] compartidos entre dos tareas que se ejecutan de forma asíncrona, como por ejemplo entre una tarea de primer plano y una de segundo plano, o dos de segundo plano.

Para evitar incoherencias de datos es necesario conseguir que la ejecución de la zona crítica se realice de principio a fin sin ningún tipo de interrupción. Por tanto una solución podría ser inhabilitar las interrupciones durante la ejecución de la zona crítica:

```
...
for(;;){
  InhabilitaInterrupciones();
  ImprimeHora(hor, min, seg); /* Zona crítica */
  HabilitaInterrupciones();
  /* Resto de tareas de primer plano */
}
...
```

¿Es válida la solución propuesta en el ejemplo anterior? Obviamente no, pues existe una zona de código que dura 20 ms (la llamada a `ImprimeHora`) con las interrupciones inhabilitadas, por lo que la latencia es de 20 ms, tal como ocurría cuando se realizaba la impresión desde la tarea de segundo plano.

Lo que se suele hacer en estos casos, es hacer una copia de las variables compartidas con las interrupciones inhabilitadas y luego usar esas copias en el resto de la tarea. De este modo, la zona crítica se reduce al mínimo imprescindible. Teniendo esto en cuenta, la función `main` del ejemplo anterior quedaría:

```
int main(void)
{
  int chor, cmin, cseg; /* Copias */

  InicializaTemporizador();
  HabilitaInterrupciones();

  for(;;){
    InhabilitaInterrupciones();
    chor = hor;  /* Principio de la zona crítica */
    cmin = min;
    cseg = seg;  /* Final de la zona crítica */
```

[12] Por recurso aquí se entiende o una zona de memoria (variables) o un periférico (por ejemplo una pantalla).

```
    HabilitaInterrupciones();
    ImprimeHora(chor, cmin, cseg);
    /* Resto de tareas de primer plano */
  }
}
```

1.6.3. Ejecución de las tareas de primer plano

Según se ha expuesto en la sección anterior, si una tarea de segundo plano es compleja, es necesario dividirla en dos partes, dejando la parte más costosa computacionalmente para una tarea asociada de primer plano. Tal como se ilustra en la figura 1.9, las tareas de primer plano se ejecutan mediante un bucle de *scan*, por lo que su latencia será el tiempo máximo de ejecución del bucle. Por tanto, en un sistema primer plano / segundo plano, tan solo se soluciona el problema de la alta latencia si las acciones que requieren baja latencia son simples y, por tanto, pueden realizarse dentro de la rutina de interrupción.

En el capítulo 3 se estudiarán algunas técnicas que permiten mejorar un poco la latencia de las tareas de primer plano asociadas a tareas de segundo plano, aunque también se verá que estas técnicas no son muy potentes, pudiendo aplicarse tan solo a sistemas relativamente sencillos. En el resto de los casos, será necesario el uso de un sistema operativo en tiempo real, el cual se describe brevemente en la siguiente sección y en detalle en el capítulo 4.

1.7. Sistemas operativos en tiempo real

Un sistema operativo en tiempo real (SOTR) es mucho más simple que un sistema operativo de propósito general, sobre todo si dicho SOTR está orientado a sistemas empotrados basados en microcontrolador. Aunque existen una gran cantidad de SOTR en el mercado, todos ellos tienen una serie de componentes similares:

- Una serie de mecanismos para permitir compartir datos entre tareas, como por ejemplo colas FIFO.[13]
- Otra serie de mecanismos para la sincronización de tareas, como por ejemplo semáforos.
- Un planificador (*scheduler* en inglés) que decide en cada momento qué tarea de primer plano ha de ejecutarse. Las tareas de segundo plano, a las que se les denomina también en este texto rutinas de atención a interrupción, se ejecutan cuando se produce la interrupción a la que están asociadas.

[13] *First In First Out.*

Todos estos componentes se estudiarán en detalle en el capítulo 4. No obstante, estudiaremos ahora brevemente los distintos tipos de planificadores que existen y cómo consiguen que las distintas tareas se ejecuten en orden.

1.7.1. El planificador

El planificador es uno de los componentes principales de un sistema operativo en tiempo real, ya que es el encargado de decidir en cada momento qué tarea de primer plano hay que ejecutar. Para ello, el planificador mantiene unas estructuras de datos internas que le permiten conocer qué tareas pueden ejecutarse a continuación, basándose por ejemplo en si tienen datos para realizar su trabajo. Además, si existen varias tareas listas para ejecutarse, el planificador usará información adicional sobre éstas, como sus prioridades o sus límites temporales (*deadline*) para decidir cuál será la siguiente tarea que debe ejecutarse.

Otra ventaja adicional de usar un planificador es la de poder tener tareas adicionales, no asociadas a tareas de segundo plano, pero con prioridades asociadas.[14] Esto no puede realizarse con un procesamiento secuencial, ya que en este tipo de sistemas todas las tareas de primer plano, estén o no asociadas a tareas de segundo plano, comparten el mismo bucle de *scan* y, por tanto, se ejecutan secuencialmente. Además, el planificador puede configurarse para ejecutar estas tareas, o bien periódicamente, o bien cuando no tenga nada mejor que hacer. Un ejemplo del primer caso puede ser una tarea de comunicaciones que envíe información sobre el estado del sistema a una estación de supervisión remota. Un ejemplo del segundo caso sería una tarea encargada de gestionar el interfaz de usuario.

Existen dos tipos de planificadores: cooperativos (*non preemptive* en inglés) y expropiativos (*preemptive* en inglés); los cuales se estudian brevemente a continuación:

1.7.2. Planificador cooperativo

En el planificador cooperativo, son las propias tareas de primer plano las encargadas de llamar al planificador cuando terminan su ejecución o bien cuando consideran que llevan demasiado tiempo usando la CPU. En este caso se dice que realizan una cesión de la CPU (*yield* en inglés). El planificador entonces decidirá cuál es la tarea que tiene que ejecutarse a continuación. En caso de que no existan tareas más prioritarias listas para ejecutarse, el planificador le devolverá el control a la primera tarea para que continúe su ejecución. Si por el contrario existe alguna tarea con más prioridad lista para ejecutarse, se efectuará un cambio de contexto y se cederá la CPU a la tarea más prioritaria.

[14] Normalmente menores que las demás tareas.

A la vista de lo anterior, es fácil darse cuenta que el principal problema del planificador cooperativo es la falta de control sobre la latencia de las tareas de primer plano, ya que el planificador sólo se ejecuta cuando la tarea que está usando la CPU termina o efectúa una cesión. Esto hace que sea difícil garantizar la temporización de las tareas, pudiendo ocurrir que alguna de ellas no cumpla con su límite temporal (*deadline*).

Además, este tipo de planificadores obligan al programador de las tareas de primer plano a situar estratégicamente las cesiones de la CPU para minimizar la latencia. Para ello, como norma general, hay que situar cesiones de la CPU:

- En cada iteración de un bucle largo.
- Intercaladas entre cálculos complejos.

En cambio, una ventaja de este tipo de planificación, desde el punto de vista del programador de las tareas de primer plano, viene dada por la cesión explícita de la CPU: mientras una tarea se ejecuta, entre cesión y cesión, no lo hará ninguna otra tarea de primer plano. Por tanto, no existirán problemas de incoherencia de datos entre estas tareas de primer plano, aunque obviamente, si los datos se comparten con una rutina de atención a interrupción, sí seguirán existiendo estos problemas con los datos compartidos.

Otra ventaja de este tipo de planificadores es que son más simples, por lo que son más apropiados en sistemas con recursos limitados, tanto de CPU como de memoria.

1.7.3. Planificador expropiativo

Para mejorar la latencia de las tareas de primer plano puede usarse un planificador expropiativo. En este tipo de sistemas, en lugar de ser las tareas las responsables de ejecutar el planificador, éste se ejecuta periódicamente de forma automática. Para ello, se usa un temporizador que cada intervalo de tiempo (*time slice*) genera una interrupción que ejecuta el planificador. De este modo, el programador no tiene que incluir cesiones explícitas de la CPU dentro del código, ya que éstas se realizan automáticamente cada intervalo (*time slice*). Además, el intervalo de ejecución del planificador puede hacerse suficientemente bajo como para conseguir una latencia adecuada entre las tareas de primer plano. Obviamente, como la ejecución del planificador lleva su tiempo, no se puede elegir un intervalo de tiempo muy pequeño, pues entonces se estaría todo el tiempo ejecutando el planificador sin hacer nada útil. Así, los intervalos de los sistemas operativos de tiempo compartido (Linux, Mac OS X, Windows, etc.) suelen ser del orden de 10 o 20 ms; ya que en este tipo de sistemas no son necesarias latencias muy bajas. Sin embargo, en un sistema en tiempo real este intervalo puede

ser del orden de unos pocos milisegundos para conseguir bajas latencias. Por ejemplo el sistema operativo de tiempo real QNX usa un intervalo de 4 ms y el sistema operativo FreeRTOS usado en los ejemplos de este texto está configurado para un intervalo de 5 ms.

En cuanto al funcionamiento del planificador, es prácticamente igual al planificador cooperativo: verifica si hay una tarea más prioritaria que ejecutar y en caso afirmativo hace un cambio de contexto para ejecutarla.

Recursos compartidos y planificadores expropiativos

La introducción de un planificador expropiativo no viene exenta de problemas. Aparte de aumentar la complejidad del planificador, la programación de las tareas también ha de ser más cuidadosa, pues, aunque ahora no hay que preocuparse de ceder la CPU, si aparecen nuevos quebraderos de cabeza.

Un problema que hay que gestionar en este tipo de sistemas es el acceso a recursos compartidos.[15] En el siguiente ejemplo se muestran dos tareas de primer plano con la misma prioridad que acceden a una pantalla para imprimir el mensaje:

```
void TareaA()
{
  ...
  puts("Hola tío\n");
  ...
}

void TareaB()
{
  ...
  puts("Adiós colega\n");
  ...
}
```

Si se usa un planificador expropiativo, éste alternará entre las dos tareas para que se ejecuten en "paralelo". Es decir, en el primer intervalo ejecutará la tarea 1, en el siguiente la 2, luego volverá a ejecutar la 1, y así sucesivamente. Como ambas están accediendo a la pantalla, el resultado podría ser el siguiente:

```
HoAdlaió ts íoco
lega
```

[15] Este problema puede darse también en los demás sistemas si el recurso está compartido entre rutinas de atención a interrupción (segundo plano) y tareas de primer plano.

Es decir, una mezcla de ambos mensajes.[16]

La solución al problema anterior consiste en evitar el acceso simultáneo de dos o más tareas a un mismo recurso. Para ello, en el caso de tener un planificador cooperativo, lo único que habrá que hacer es no ceder la CPU mientras se está usando el recurso.[17] Ahora bien, esto originará que el resto de tareas de primer plano (no solo las que deseen usar el mismo recurso) no podrán ejecutarse, aumentándose la latencia si el acceso al recurso compartido se alarga en el tiempo.

En el caso de los planificadores expropiativos, son necesarios mecanismos más sofisticados, pues las tareas no pueden controlar cuándo van a ejecutarse las demás tareas. La solución más sencilla es inhabilitar las interrupciones mientras se está accediendo al recurso. Obviamente, esto sólo es válido si el acceso al recurso compartido dura poco tiempo, pues de lo contrario se aumentará la latencia de todo el sistema (primer y segundo plano). Otra solución consiste en evitar que el planificador realice cambios de contexto.[18] Nuevamente, pueden existir problemas de latencia, aunque ahora sólo con las tareas de primer plano. La tercera solución consiste en usar un mecanismo denominado semáforo, mediante el cual el sistema operativo se encarga de vigilar si un recurso está libre o no. Todas las tareas que deseen usar un recurso determinado, le piden permiso al sistema operativo, el cual, si el recurso está libre permitirá a la tarea que continúe con su ejecución, pero si está ocupado la bloqueará hasta que dicho recurso se libere. En el capítulo 4 se estudiarán en detalle los semáforos.

1.8. *Hardware*

Hasta ahora se han discutido las distintas técnicas utilizadas para implantar el *software* de un sistema en tiempo real. Ahora bien, de todos es bien conocido que todo *software* necesita un *hardware* en el que ejecutarse. Teniendo en cuenta que la mayoría de sistemas en tiempo real son a su vez sistemas empotrados,[19] conviene dedicar en esta introducción un poco

[16] No todos los planificadores tienen este comportamiento cuando existen dos tareas con la misma prioridad que necesitan ejecutarse. Algunos ejecutan primero una tarea y cuando ésta termina, empiezan a ejecutar la siguiente. En este tipo de planificadores no se dará este problema.

[17] Suponiendo que el recurso está compartido entre dos tareas de primer plano. Si se comparte con una rutina de atención a interrupción, la única solución es inhabilitar las interrupciones mientras se usa el recurso compartido.

[18] Para ello existirá una llamada al sistema operativo, al igual que la llamada para ceder la CPU.

[19] Un ordenador empotrado es un ordenador de propósito especial que está instalado dentro de un dispositivo y se usa para controlarlo. Por ejemplo, dentro de un automóvil existen varios ordenadores para controlar el encendido, el ABS, el climatizador, etc. Una característica de estos ordenadores es que el usuario final no tiene porqué conocer su existencia ni puede cargar programas en él, todo lo contrario que en un ordenador de propósito general. Además, al ser ordenadores diseñados para una aplicación particular,

de espacio a estudiar las distintas alternativas existentes a la hora de elegir el *hardware* adecuado para el sistema en tiempo real.

A la hora de implantar un sistema empotrado existen dos alternativas claras: usar un microcontrolador o usar un microprocesador de propósito general.

Un microcontrolador no es más que un circuito que combina en un solo chip un microprocesador junto con una serie de periféricos como temporizadores, puertos de entrada/salida, conversores A/D y D/A, unidades PWM,[20] etc. Según la potencia de cálculo necesaria se elegirá un procesador de entre 4 y 32 bits. No obstante, la potencia de cálculo de estos sistemas suele ser limitada (comparada con un microprocesador de propósito general), ya que los fabricantes de este tipo de chips buscan circuitos de bajo coste y de bajo consumo.

Si el sistema necesita una gran potencia de cálculo, la alternativa es usar un microprocesador convencional. Ahora bien, un ordenador de sobremesa no es adecuado para instalarlo en un ambiente hostil o en un espacio reducido, que es el destino de la mayoría de los sistemas empotrados. Por ello se han desarrollado estándares que permiten construir PC más robustos y de reducido tamaño, con periféricos orientados al control de sistemas (conversores A/D y D/A, puertos de entrada/salida, etc.). Dentro de estos estándares, algunos ejemplos son PC/104, VME, cPCI, etc.

1.8.1. Microcontroladores

Normalmente la mayoría de sistemas empotrados basados en microcontrolador usan una tarjeta hecha a medida, lo que permite optimizar su tamaño y sus interfaces hacia el exterior. El inconveniente de esta aproximación es el coste incurrido en diseñar la tarjeta[21] que puede ser difícil de amortizar si se necesitan pocas unidades. En estos casos, si no hay problemas de espacio, puede ser más rentable usar una placa de propósito general como la mostrada en la figura 1.11 y diseñar sólo los interfaces con el exterior.

1.8.2. Buses estándares

En la figura 1.12 se muestra un sistema basado en el bus estándar PC/104. Este tipo de sistemas usan un bus que es idéntico desde el punto de vista eléctrico y lógico a un bus ISA.[22] Sin embargo, el conector origi-

---

su potencia de cálculo estará adaptada a las tareas a realizar.

[20] Pulse Width Modulation.

[21] Lo que se denomina costes no retornables

[22] El bus ISA (*Industry Standard Architecture*) fue el bus estándar de los primeros PC. Este bus fue remplazado por completo por el bus PCI *Peripheral Component Interconnect*, más rápido y avanzado, el cual está siendo sustituido en la actualidad por el bus PCI–Express.

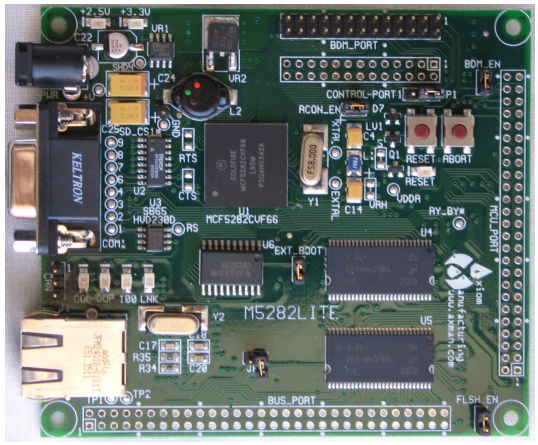


Figura 1.11: Tarjeta de desarrollo basada en microcontrolador.

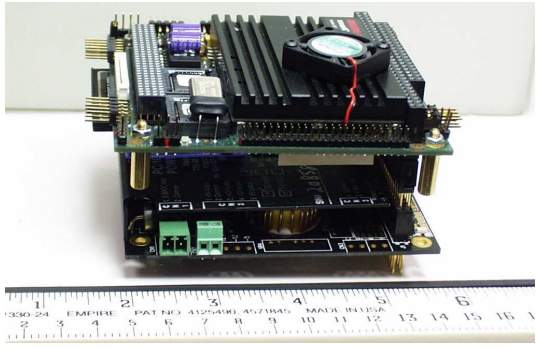


Figura 1.12: Sistema basado en bus PC/104.

nal ha sido remplazado por otro más robusto, que consta de un conector hembra por un lado y un conector macho por el otro lado, lo que permite apilar los módulos unos encima de otros, formando así un sistema bastante compacto y robusto.

La limitación principal de este bus es su velocidad relativamente baja, pues es un bus de 16 bits a 8 MHz.[23] No obstante, para la mayoría de las aplicaciones de control (conversión A/D, etc.) esta velocidad es más que suficiente. Además, la lógica necesaria para conectar un circuito a este bus es muy simple, por lo que es posible diseñar placas a medida para este

[23] El bus PCI por el contrario es de 32 bits a 33 MHz.

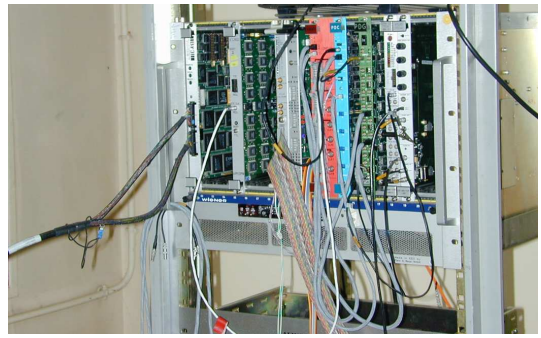


Figura 1.13: Sistema basado en rack.

bus que realicen una interfaz con el sistema a controlar. Por el contrario, otros buses como VME o cPCI son mucho más complejos, lo que dificulta el diseño de placas a medida.

La ventaja de este tipo de sistemas es que al estar basados en un procesador IA-32, no son necesarias herramientas de desarrollo especiales como compiladores cruzados o emuladores, ni hace falta familiarizarse con otra arquitectura, ya que estos sistemas no son más que un PC adaptado para sistemas empotrados.

1.8.3. Sistemas basados en rack

Cuando se necesita un sistema empotrado de altas prestaciones, la alternativa es usar un sistema basado en *rack* como VME o cPCI. Estos sistemas constan de un bus trasero (*backplane*) al que se conectan una serie de tarjetas para formar un ordenador a medida. Las tarjetas pueden ser CPU, memoria, periféricos de entrada/salida, discos, etc. Un ejemplo de este tipo de sistemas es el mostrado en la figura 1.13

1.9. Ejercicios

1. Diseñe el programa de control del termostato usando lenguaje C. Suponga que dispone de una librería con las siguientes funciones, definidas en el archivo `Termostato.h`:

   - **`int`** `LeerTeclado()`. Devuelve 1 si se ha pulsado la tecla UP, 2 si se ha pulsado DOWN y 0 si no se ha pulsado ninguna.
   - **`void`** `LanzarConversionAD()`. Arranca un conversión.
   - **`int`** `Convirtiendo()`. Devuelve un 1 si el conversor está aún realizando la conversión y un 0 si ha finalizado.

- **int** `LeerConversorAD()`. Devuelve el valor del conversor A/D. El conversor es de 12 bits y está conectado a un sensor de temperatura que da 0 V cuando la temperatura es de 0^o C y 5 V cuando la temperatura es de 100^o C. El fin de escala del conversor A/D es precisamente 5 V.
- **void** `ArrancaCalefaccion()`. Conecta la calefacción.
- **void** `ApagaCalefaccion()`. Desconecta la calefacción.

Tenga en cuenta que:

- Será necesario un cierto trasiego de datos entre las cuatro tareas.
- La temperatura de consigna al iniciarse el programa será de 20 grados.

2. Al programa diseñado en el ejercicio anterior le falta una tarea para mostrar el valor de la temperatura en el *display*. Para ello suponga que dispone de la función:

```
void ImprimeDisplay(char *pcadena);
```

La cual imprime una cadena de caracteres en el display del termostato. Para imprimir la temperatura se usará la función `sprintf` para formatear el texto en una cadena[24] y a continuación se imprimirá la cadena en el display con la función `ImprimeDisplay`. No obstante, después de escribir el código descubre que la ejecución de esta tarea es demasiado lenta y ha de partirse en dos, tal como se ha descrito en la sección 1.5.4. Puesto que no hay ningún problema en que esta tarea se ejecute con un periodo de muestreo inferior al de las tareas de control, diseñe la tarea para que su periodo de ejecución sea $2T_s$.Escriba la tarea para que en una primera ejecución se formatee el mensaje en la cadena y en una segunda ejecución se envíe la cadena al *display*.

[24]La función `sprintf` funciona igual que `printf`, salvo que en lugar de imprimir en la pantalla "imprime" en una cadena. Por ejemplo, para escribir la variable entera `var_ent` en la cadena `cad` se ha de ejecutar `sprintf(cad, "var = %d\n", var_ent);`. Por supuesto la cadena `cad` ha de tener una dimensión suficiente para almacenar los caracteres impresos.

CAPÍTULO 2

Lenguaje C para programación en bajo nivel

La mayoría de sistemas en tiempo real han de interactuar directamente con el *hardware*. Para ello suele ser necesario manipular bits individuales dentro de los registros de configuración de los dispositivos. Por otro lado, este tipo de sistemas suelen ser también sistemas empotrados basados en microcontroladores, en los cuales, al no existir unidades de coma flotante, es necesario trabajar en coma fija. En estos casos es necesario conocer las limitaciones de los tipos de datos enteros: rango disponible, desbordamientos, etc.

En este capítulo se van a estudiar precisamente estos dos aspectos, ya que éstos no suelen ser tratados en los libros de C de nivel introductorio o, si acaso, son tratados muy superficialmente.

2.1. Tipos de datos enteros

El lenguaje C dispone de 2 tipos de enteros básicos: **`char`** e **`int`**. El primero, aunque está pensado para almacenar un carácter, también puede ser usado para almacenar un entero de un byte, tanto con signo como sin signo.[1] Recuerde que, para el ordenador, un byte en la memoria es simplemente eso, un byte. Hasta que no se le dice al compilador que lo imprima como un carácter, dicho byte no se tratará como tal, realizando la traducción número–símbolo usando la tabla ASCII.

Otro aspecto a resaltar es la indefinición del tamaño de los tipos enteros en C. El lenguaje sólo especifica que el tipo **`int`** ha de ser del tamaño "natural" de la máquina. Así, en un microcontrolador de 16 bits, como los registros internos del procesador son de 16 bits, el tipo **`int`** tendrá un tamaño de 16 bits. En cambio, en un procesador de 32 bits como el ColdFire, el PowerPC o el Pentium, como el tamaño de los registros es de 32 bits, los **`int`** tienen un tamaño de 32 bits. Además, en un procesador de 8 bits podríamos encontrarnos con que un **`int`** tiene un tamaño de 8 bits, si el diseñador del compilador así lo ha decidido. Por si esto fuera poco, los ta-

[1] Obviamente, en este último caso se declarará la variable como **`unsigned char`**.

maños de **short int** y de **long int** tampoco están determinados. Lo único a lo que obliga el lenguaje es a que el tamaño de un **short int** ha de ser menor o igual que el de un **int** y éste a su vez ha de tener un tamaño menor o igual que el de un **long int**.

Cuando se programa un sistema empotrado, una característica muy importante es la portabilidad.[2] Si se está escribiendo un programa de bajo nivel en el que se necesitan usar variables de un tamaño determinado, en lugar de usar los tipos básicos como **int** o **short int**, es mejor definir unos nuevos tipos mediante la sentencia **typedef**, de forma que si se cambia de máquina, baste con cambiar estas definiciones si los tamaños de los tipos no son iguales. Si no se hace así, habrá que recorrer todo el programa cambiando las definiciones de todas las variables, lo cual es muy tedioso y, además, propenso a errores.

Para definir los nuevos tipos, lo más cómodo es crear un archivo cabecera (al que se puede llamar por ejemplo `Tipos.h`) como el mostrado a continuación, e incluirlo en todos los programas.

```
#ifndef TIPOS_H
#define TIPOS_H

typedef unsigned char       uint8;
typedef unsigned short int  uint16;
typedef unsigned long int   uint32;

typedef signed char         int8;
typedef signed short int    int16;
typedef signed long int     int32;

#endif
```

El archivo `Tipos.h`[3] no es más que una serie de definiciones de tipos en los que se menciona explícitamente el tamaño. Si se incluye este archivo en un programa, podrán usarse estos tipos de datos siempre que se necesite un tamaño de entero determinado. Por ejemplo si se necesita usar un entero de 16 bits sin signo bastará con hacer:

```
#include "Tipos.h"
```

[2] Se dice que un programa es portable cuando su código fuente está escrito de manera que sea fácil trasladarlo entre distintas arquitecturas, como por ejemplo pasar de un microcontrolador de 16 bits a otro de 32 bits. Por ejemplo, un programa escrito en C que use sólo funciones de la librería estándar, será fácilmente portable entre distintos tipos de ordenadores sin más que recompilarlo.

[3] Este archivo está basado en las definiciones de tipos usadas por el entorno de desarrollo CodeWarrior para el microcontrolador ColdFire MCF5282. Si se usa este entorno no es necesario crear el archivo puesto que estas definiciones de tipos se incluyen ya en el archivo `mcf5282.h`.

Bits	Rango sin signo	Rango con signo
8	$0 \leftrightarrow 255$	$-128 \leftrightarrow 127$
16	$0 \leftrightarrow 65\,535$	$-32\,768 \leftrightarrow 32\,767$
32	$0 \leftrightarrow 4\,294\,967\,296$	$-2\,147\,483\,648 \leftrightarrow 2\,147\,483\,647$

Cuadro 2.1: Rangos de enteros con y sin signo de 8, 16 y 32 bits.

```
...

int main(void)
{
  uint16 mi_entero_de_16_bits_sin_signo;

  ...
}
```

2.1.1. Rangos de las variables enteras

Uno de los factores a tener en cuenta para elegir un tipo de dato entero es su rango. En general, el rango de un entero sin signo de n bits viene dado por la expresión:

$$R = 0 \leftrightarrow 2^n - 1$$

y el rango de un entero con signo codificado en complemento a 2, que es como se codifican estos números en todos los ordenadores, es:

$$R_s = -2^{n-1} \leftrightarrow 2^{n-1} - 1$$

Por tanto, teniendo en cuenta estas ecuaciones, los rangos de los enteros de 8, 16 y 32 bits serán los mostrados en el cuadro 2.1.

Para elegir un tipo de entero u otro es necesario hacer un análisis del valor máximo que va a tomar la variable. Por ejemplo, si una variable va a almacenar la temperatura de una habitación, expresada en grados centígrados, será suficiente con una variable de 8 bits con signo.[4] Ahora bien, si es necesario realizar operaciones con esta temperatura, puede que sea necesario usar un tipo con mayor número de bits. Por ejemplo, en el siguiente fragmento de código:

```
int8 t;
t = LeeTemperaturaHabitacion();
t = t * 10;
```

[4]El elegirla con signo es para tener en cuenta temperaturas bajo cero.

Si la temperatura de la habitación es por ejemplo 30 grados centígrados, el resultado de `t * 10` será 300, que obviamente no cabe en una variable de 8 bits, produciéndose entonces un desbordamiento. El problema es que en C, para conseguir una mayor eficiencia, no se comprueba si se producen desbordamientos, con lo cual el error pasará desapercibido para el programa, pero no para el sistema que esté controlando.

Es por tanto muy importante hacer una estimación de los valores máximos que puede tomar una variable, no solo en el momento de medirla, sino durante toda la ejecución del programa. De esta forma se podrá decidir qué tipo usar para la variable de forma que no se produzca nunca un desbordamiento.

Por último, decir que en caso de usar un procesador de 32 bits, no se ahorra prácticamente nada en usar una variable de 8 o de 16 bits, por lo que en vistas a una mayor seguridad, es mejor usar siempre variables de 32 bits. Si por el contrario el microprocesador es de 16 bits, en este caso si que es mucho más costoso usar variables de 32 bits, por lo que sólo se deberán usar cuando sea necesario.

2.2. Conversiones de tipos

Las conversiones automáticas de C pueden ser peligrosas si no se sabe lo que se está haciendo cuando se mezclan tipos de datos. Normalmente las conversiones por defecto son razonables y funcionan en la mayoría de los casos. No obstante pueden darse problemas como el mostrado a continuación:

```
int main()
{
  uint16 u;
  ...
  if (u > -1){
    printf("Esto se imprime siempre.\n");
  }
}
```

En este ejemplo se compara un entero sin signo (la variable `u`) con otro con signo (la constante `-1`). El lenguaje C, antes de operar dos datos de tipos distintos, convierte uno de ellos al tipo "superior" en donde por superior se entiende el tipo con más rango y precisión. Si se mezclan números con y sin signo, antes de operar se convierten todos a números sin signo. No obstante, la conversión no implica hacer nada con el patrón de bits almacenado en la memoria, sino sólo modificar cómo se interpreta. Así, en el ejemplo anterior la constante `-1` queda convertida a `65535` (`0xFFFF`)[5] que,

[5]Recuerde que -1 codificado en complemento a 2 en 16 bits es precisamente `0xFFFF`.

según se puede ver en el cuadro 2.1, es el mayor número dentro del rango de `uint16`, por lo que la condición del **`if`** será **siempre falsa**.

En conclusión, no se deben de mezclar tipos, salvo que se esté completamente seguro de lo que se está haciendo.

2.2.1. Categorías de conversiones

Las conversiones de tipos en C pueden clasificarse en dos categorías:

- Promociones. Son aquellas conversiones en las que no hay pérdidas potenciales de precisión o rango. Un ejemplo es la conversión de un entero a un número en coma flotante:

  ```
  double d = 4;
  ```

  Nótese que en este ejemplo sí que será necesario realizar un cambio en el patrón de bits que representa el número, ya que ambos se codifican de distinta manera. Esto obviamente conlleva una pequeña pérdida de tiempo. Si por el contrario se promociona un entero de 16 bits a otro de 32, sólo se añaden los bits más significativos.[6]

- Degradaciones. Son aquellas en las que se produce una pérdida de precisión, como por ejemplo la asignación de un número en coma flotante a un entero (**`int`** `i = 4.3;`). En este caso se pierde la parte decimal. Otro ejemplo de degradación es la asignación de un número de 32 bits a uno de 16:

  ```
  int32 i_largo;
  int16 i_corto;
  ...
  i_corto = i_largo; /* Posible error */
  ```

  En este caso, se pierden los 16 bits más significativos, por lo que si el valor almacenado en `i_largo` es mayor que `32767` o menor que `-32768`, se producirá un error al salirse el valor del rango admisible por un `int16`.

Por último, conviene recordar que es posible forzar una conversión mediante el operador *cast*.[7] Este operador es necesario, por ejemplo, al asignar memoria dinámica para convertir el puntero genérico (**`void`** `*`), devuelto por la función `calloc`, al tipo de la variable a la que se le asigna memoria:

[6] Si el número es sin signo, se añaden ceros a la izquierda hasta completar el tamaño de la nueva palabra. Si por el contrario el número es con signo, se extiende el bit de signo para no modificar el valor del número. La extensión de signo consiste en copiar en los bits más significativos que se añaden el mismo valor que tiene el bit de signo.

[7] El operador *cast* consiste en el tipo al que se desea convertir el operando encerrado entre paréntesis.

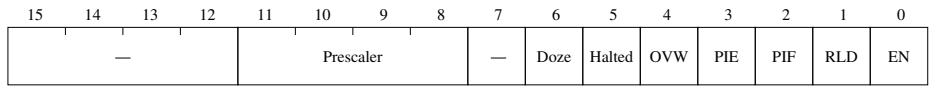


Figura 2.1: Registro PCSR (*PIT Control and Status Register*).

```
double *pd;
...
pd = (double *)calloc(20, sizeof(double));
```

También es necesario el uso del operador *cast* al realizar una degradación si se desea evitar que el compilador genere un aviso, pues de esta forma se le dice al compilador que se está seguro de lo que se está haciendo (o al menos eso se espera):

```
int32 i_largo;
int16 i_corto;
...
i_corto = (int16) i_largo; /* Estamos seguros que */
 /* i_largo contiene un valor que está dentro del */
 /* rango de int16 */
```

2.3. Manipulación de bits

En la programación de sistemas empotradas es muy frecuente encontrarse con variables que, en lugar de un número, contienen una serie de campos de bit en los que se agrupa información diversa. Un ejemplo típico son los registros de configuración de los periféricos. Por ejemplo, en la figura 2.1 se muestra el registro de configuración de los temporizadores PIT[8] del ColdFire MCF5282. Este registro de 16 bits contiene un campo de 4 bits (Prescaler) y 7 campos de 1 bit (Doze, Halted, OVW, PIE, PIF, RLD y EN).[9] Obviamente, para manejar los temporizadores es necesario poder acceder a cada uno de estos campos por separado.

Otro ejemplo típico consiste en empaquetar varios datos en una sola variable, lo cual sólo tiene sentido si la memoria del microcontrolador es escasa; ya que el acceso a cada una de las variables empaquetadas es más complejo. Por ejemplo, si tenemos 8 variables lógicas, en lugar de usar 8 bytes para almacenarlas, podemos usar un sólo byte y asignar un bit para cada variable lógica.

[8] *Programmable Interrupt Timer*. Existen 4 temporizadores en el ColdFire MCF5282, denominados PIT0, PIT1, PIT2 y PIT3.

[9] El resto de bits están reservados para futuros usos y han de dejarse a cero.

Dado que el lenguaje C se pensó para realizar programas de bajo nivel, es decir, programas que interactuasen directamente con el *hardware*, se definieron varios métodos para manipular variables a nivel de bit; los cuales se estudian en las siguientes secciones.

2.3.1. Operadores a nivel de bit

El lenguaje C define los siguientes operadores a nivel de bit:

- Operadores de desplazamiento. Los operadores << y >> permiten desplazar el valor de una variable un número de bits hacia la izquierda o la derecha respectivamente. Por ejemplo:

  ```
  b = a << 4;
  ```

  desplaza el valor almacenado en a 4 bits hacia la izquierda[10] y almacena el resultado en b.

- Operadores lógicos a nivel de bit de dos operandos. Los operadores &, | y ^ realizan las operaciones AND, OR y XOR bit a bit entre sus dos operandos. Así, si se inicializan a y b con los valores:

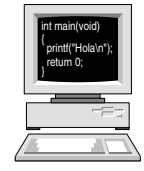
  ```
  a = 01101101
  b = 10101001
  ```

  Entonces el resultado de hacer:

  ```
  c = a & b;
  ```

  será `00101001`

- El operador ~ (NOT) invierte cada uno de los bits de su operando. Por ejemplo si `a=0xf0`, `~a` valdrá `0x0f`



Realice el ejercicio 1.

Por último, destacar que los operadores lógicos y los operadores lógicos a nivel de bit no son equivalentes, como se demuestra en el ejercicio 1.

Manipulación de bits individuales

Los operadores de desplazamiento y de nivel de bit son útiles para manipular bits individuales dentro de una palabra. Los casos típicos son:

[10] Recuerde que desde el punto de vista aritmético, desplazar n bits a la izquierda equivale a multiplicar por 2^n. De la misma forma desplazar n bits a la derecha equivale a dividir por 2^n.

- Verificar el estado de un bit de una variable. Para ello basta con hacer un AND entre la variable y una **máscara** con todos los bits a cero excepto el bit que se quiere comprobar. Así, para verificar el estado del bit PIF (bit 2) del registro PCSR del temporizador PIT0 del microcontrolador MCF5282 (`MCF_PIT0_PCSR`), hay que escribir:

  ```
  if (MCF_PIT0_PCSR & (1<<2)) /* ¿bit 2 a 1? */
  ```

  Como se puede apreciar, aunque podría haberse calculado el valor de la máscara (`0x0004`), es mucho más fácil utilizar el operador desplazamiento, ya que así se aprecia directamente el número del bit a verificar. Además, la operación de desplazamiento se evalúa en tiempo de compilación ya que sus dos argumentos son constantes. En consecuencia, el programa final se ejecutará igual de rápido.

- Para poner un bit a 1 hay que hacer una OR entre la variable y una máscara con todos los bits a cero, excepto el bit que se quiere poner a 1. El método para obtener la máscara es idéntico al caso anterior. Así, el siguiente código pone a uno el bit PIE (bit 3) del registro PCSR del temporizador PIT0:

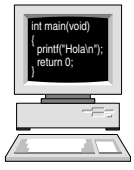
  ```
  MCF_PIT0_PCSR |= (1<<3); /* bit 3 a 1 */
  ```

- Para poner un bit *n* a cero, la máscara ha de construirse con todos los bits a 1 excepto el bit *n* y hacer un AND entre la variable y la máscara. Para ello, primero se crea una máscara con todos los bits a cero excepto el bit *n* y luego se invierte la máscara con el operador NOT. Por ejemplo, si se desea poner a cero el bit PIE (bit 3) del registro PCSR del temporizador PIT0, basta con hacer:

  ```
  MCF_PIT0_PCSR &= ~(1<<3); /* bit 3 a 0 */
  ```

- Para invertir un bit se hace una XOR con una máscara igual a la usada para ponerlo a 1:

  ```
  MCF_PIT0_PCSR ^= (1<<3); /* bit 3 se invierte */
  ```



Realice los ejercicios 2, 3 y 4.

Nota Importante: Tenga en cuenta que, salvo para el primer ejemplo, el bit que se desea cambiar ha de ser de lectura/escritura, cosa que no ocurre en todos los registros *hardware* de los periféricos.

Manipulación de campos de bits

La manipulación de campos de varios bits dentro de una misma palabra es parecida a la manipulación de bits sueltos. Los casos típicos son:

- Extraer un campo de bits para analizarlo. Para ello, los pasos a seguir son:
  - Desplazar el dato a la derecha para llevar el campo al bit 0.
  - Hacer una AND con una máscara para eliminar el resto de bits.

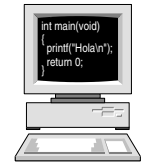
  Por ejemplo, si se desea obtener el valor del campo Prescaler del registro PCSR del temporizador PIT0, habrá que desplazar el dato 8 bits para trasladar el campo del bit 8 al bit 0. A continuación, habrá que hacer una AND con la máscara `0x000F`, para poner los bits 15 a 4 a cero y dejar así sólo el campo Prescaler. La instrucción para llevar esto a cabo es:

  ```
  pres = (MCF_PIT0_PCSR>>8) & 0x000F;
  ```

- Escribir un campo de bits. En este caso, el proceso es:
  - Borrar el campo de bits.
  - Eliminar los bits más significativos que sobren del dato a escribir.[11]
  - Desplazar el dato a escribir para alinearlo con el campo destino.
  - Hacer una OR entre el dato a escribir y la variable destino.

  Por ejemplo, para escribir el valor de la variable `pres` en el campo Prescaler del registro PCSR del temporizador PIT0, hay que ejecutar las siguientes instrucciones:

  ```
  MCF_PIT0_PCSR &= ~(0x0F << 8); /* Puesta a cero */
  MCF_PIT0_PCSR |= (pres & 0x000F) << 8;
  ```



Realice el ejercicio 5

Uso de máscaras

Para conseguir una mayor claridad del código, pueden definirse **máscaras** para acceder a bits individuales, y darles a estas máscaras los nombres de los bits. Por ejemplo, para poder modificar el estado de los bits `PIE` y `PIF` se definen las máscaras:

```
#define PIE (1<<3)
#define PIF (1<<2)
```

[11] Si se está seguro de que el valor a escribir tiene los bits que no forman parte del campo a cero, este paso puede ser innecesario. Un ejemplo típico es la escritura de una constante. Si no se está seguro, es mejor aplicar la máscara para una mayor robustez del programa.

Entonces, para poner a 1 el bit `PIE` del registro `MCF_PIT0_PCSR` se hará un OR con su máscara:

```
MCF_PIT0_PCSR |= PIE;
```

Y para ponerlo a 0, se hará una AND con su máscara negada:

```
MCF_PIT0_PCSR &= ~PIE;
```

La ventaja de este método es que las definiciones pueden situarse en un archivo cabecera (por ejemplo `mcf5282.h`). A partir de entonces, no se tendrá que volver a mirar el manual para averiguar en qué posición está el bit: sólo habrá que recordar el nombre del bit, lo cual es siempre muchísimo más fácil.

Si se desean activar o desactivar varios bits a la vez, pueden combinarse varias máscaras con el operador `|`. Por ejemplo para poner a 1 los bits `PIE` y `PIF`:

```
MCF_PIT0_PCSR |= PIE | PIF;
```

Y para ponerlos a cero:

```
MCF_PIT0_PCSR &= ~(PIE | PIF);
```

Por último, conviene tener en cuenta que algunos microcontroladores, como por ejemplo el Infineon 167, disponen de direccionamientos a nivel de bit que permiten acceder a bits individuales sin necesidad de usar máscaras. No obstante, si se busca un programa portable es mejor usar máscaras y operadores a nivel de bit, ya que éstos están soportados por todas las arquitecturas y compiladores.

2.3.2. Campos de bits

A pesar de usar nombres para las máscaras, el acceso a campos de bit dentro de una palabra usando operadores lógicos y desplazamientos no es muy intuitivo. Para facilitar la escritura del código y su legibilidad, el lenguaje C incluye un método para acceder a estos campos de bits usando una sintaxis similar a la de las estructuras. La única diferencia entre una estructura normal y una estructura con campos de bits radica en que cada variable se puede dividir en una serie de campos de bits de tamaños arbitrarios, siempre y cuando estos tamaños sean inferiores al del dato sobre el que se declaran. En el ejemplo siguiente se define un tipo compuesto por una palabra de 8 bits dividida en dos campos de 4 bits para almacenar un número BCD de dos dígitos.[12]

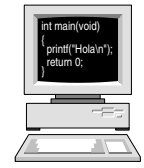
[12]Un número BCD, como indican sus siglas en ingles, *Binary Coded Decimal*, es una codificación en binario de cada uno de sus dígitos. Así, el número 27 se codificará con 8 bits, usándose los cuatro bits más significativos para codificar el dígito 2 y los cuatro menos significativos para codificar el 7. Por tanto, el 27 se representa en BCD como

```
typedef struct{
  uint8 digito0  :4, /* Ojo se termina con una coma
        digito1  :4; /* Ojo el final es un ; */
}BCD2;
```

Nótese que los campos se separan con una coma, mientras que el final de la variable se indica con un punto y coma. Esta sintaxis es así para poder incluir varias variables con campos de bits dentro de una estructura, como por ejemplo:

```
typedef struct{
  uint8 digito0  :4,
        digito1  :4;
  uint8 digito2  :4,
        digito3  :4;
}BCD4;
```

El acceso a los elementos de una estructura de campos de bits se realiza del mismo modo que el acceso a elementos de estructuras normales: usando el operador punto o el operador flecha (->) si se dispone de un puntero a la estructura. En el ejemplo siguiente, se muestra un programa que define un número BCD de dos dígitos y que a continuación lo inicializa al valor 27.



Realice el ejercicio 6

```
int main (void)
{
  BCD2 numero;

  numero.digito0 = 2;
  numero.digito1 = 7;
  ...
}
```

Es necesario volver a recalcar que la sintaxis de las estructuras con campos de bits es exactamente la misma que la de las estructuras normales. Así, una estructura de campos de bits también puede inicializarse al declararla. Por ejemplo, el código anterior también puede escribirse como:

```
int main (void)
{
  BCD2 numero = {2, 7};
```

0010 0111. La ventaja de este método de codificación es la facilidad de conversión. Sus inconvenientes son el ocupar para un mismo número un mayor número de bits y que el *hardware* para realizar operaciones con este tipo de números es más complejo que el usado para realizar operaciones con números codificados en binario puro. Por ello, los ordenadores de propósito general no soportan operaciones con números de este tipo, aunque algunos disponen de instrucciones especiales para facilitar un poco su manejo.

```
  ...
}
```

En este caso, el primer campo definido en la estructura (`digito0`) se inicializa con el primer valor (2) y el segundo campo (`digito1`) con el segundo valor (7).

También pueden crearse vectores de estructuras de campos de bit e inicializar dichos vectores:

```
int main (void)
{
  BCD2 numeros[] = {2, 7, 4, 0};

  ...
}
```

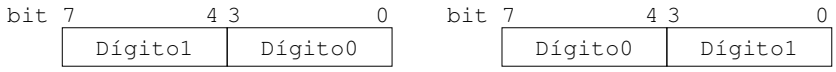
En este caso el primer elemento del vector se inicializará con el número BCD 27 y el segundo con el 40.

Además de para trabajar con datos definidos por el programador, como en el ejemplo anterior de números BCD, también se puede usar una estructura con campos de bit para acceder a los registros de configuración de los periféricos de un microcontrolador. Para ello sólo hay que definir adecuadamente los campos. El lenguaje C permite incluso definir campos vacíos para tener en cuenta los bits no usados dentro del registro. Así, en el siguiente ejemplo se muestra la declaración de una estructura de campos de bit para acceder cómodamente al registro `MCF_PIT0_PCSR`.

```
typedef struct{
  uint16            :4, /* No usado */
         Prescaler  :4,
                    :1, /* No usado */
         Doze       :1,
         Halted     :1,
         OVW        :1,
         PIE        :1,
         PIF        :1,
         RLD        :1,
         EN         :1; /* Ojo es ; */
}MCF_PIT0_PCSR_campos; /* Campos de bits */
```

Inconvenientes del manejo de campos de bits de C

Aunque el uso de campos de bits mejora la legibilidad del código, presenta dos inconvenientes importantes, los cuales hacen que en la práctica su uso no esté muy extendido.



a) Formato Little Endian (IA−32) b) Formato Big Endian (Coldfire)

Figura 2.2: Distintos modos de ordenar los campos de bit en función de la arquitectura.

El primer inconveniente es la eficiencia, pues para acceder a cada campo es necesario realizar las operaciones de desplazamiento y enmascaramiento expuestas en la sección 2.3.1. Esto impide realizar optimizaciones como poner a uno o a cero varios bits a la vez, tal como se mostró en los ejemplos de la página 42. También es más eficiente acceder a una variable estándar (**`int`**, **`char`**, etc.) que a un campo de bit. Por tanto, su uso para almacenar variables, tal como se ha mostrado en los ejemplos anteriores que trabajan con números en BCD, sólo estará justificado si la memoria del ordenador está limitada.

El segundo inconveniente, más grave si cabe, es que el orden en el que se colocan los campos dentro de la palabra no está definido por el lenguaje. Así, en los ejemplos anteriores con los números BCD, un compilador puede colocar los campos declarados en primer lugar en los bits menos significativos y otro los puede colocar en los más significativos. Por ejemplo, el compilador `gcc` para IA-32 en Linux coloca en los bits menos significativos los primeros campos en definirse, tal como se muestra en la figura 2.2.a. En cambio, el compilador CodeWarrior para ColdFire coloca el primer campo de la variable en los bits más significativos, tal como se muestra en la figura 2.2.b.[13]

Esto no tiene importancia si el uso de los campos es sólo para conseguir un almacenamiento de los datos más compacto, tal como se ha hecho con los números BCD. Sin embargo, si se definen campos de bits para acceder a registros internos del microcontrolador, tal como el ejemplo mostrado para acceder al registro `MCF_PIT0_PCSR`, el orden de los campos es obviamente muy importante.

[13]La razón de este desaguisado radica en que el estándar de C no obliga a ningún orden en especial [Kernighan and Ritchie, 1991]. Por tanto, los diseñadores del compilador hacen que el orden de los campos de bits, siga al de los bytes dentro de una palabra. Los procesadores de la familia IA-32 almacenan el byte menos significativo de la palabra, en la posición baja de la zona de memoria en la que se almacena dicha palabra. A este tipo de arquitectura se le denomina *Little Endian*. En este tipo de arquitecturas, el compilador coloca los primeros campos en definirse en los bits menos significativos de la palabra. En cambio, el ColdFire es una arquitectura *Big Endian*, lo cual quiere decir que el byte más significativo se coloca en la posición de memoria más baja. Por ello, en esta arquitectura el compilador coloca los primeros campos en definirse, en los bits más significativos de la palabra.

2.4. Acceso a registros de configuración del microcontrolador

Los periféricos que incluyen los microcontroladores se configuran y se controlan por medio de una serie de registros. Estos registros están situados en posiciones fijas de memoria[14] definidas por el diseñador del *hardware*. Cuando se define una variable en C, el compilador asigna un espacio de memoria para almacenarla y luego usa su dirección internamente para acceder a la variable y usar su contenido. Sin embargo, en el caso de los registros de configuración, el espacio de memoria ya está asignado, por lo que sólo es necesario definir un puntero a la posición de memoria del registro. Por ejemplo, si se necesita acceder en un programa al registro PCSR del microcontrolador MCF5282, en primer lugar habrá que consultar el manual del microcontrolador para saber en qué posición de memoria está dicho registro. Como se puede ver en la página 19-4 del manual [FreeScale, 2005], dicho registro está en la dirección 0x40150000.[15] A continuación se muestra un ejemplo para esperar el final de la cuenta del temporizador, comprobando para ello el bit `PIF` (bit 2) del registro. Dicho bit se pone a 1 al finalizar la cuenta del temporizador.

```
#define PIF (1<<2)
...

void EsperaFinTemp(void)
{
  volatile uint16 *p_pcsr = (volatile uint16 *)0x40150000;

  while( (*p_pcsr & PIF) == 0)
    ; /* Espera fin temporizador */
}
```

En primer lugar conviene destacar que la variable `p_pcsr` es un puntero a una variable de tipo `uint16`, ya que el registro PCSR es de 16 bits. Dicho puntero se ha inicializado a la dirección del registro, con lo cual, usando el operador * podremos acceder al contenido del registro, tal como se muestra en la línea 8. Nótese que la dirección del registro es una constante entera, por lo que es necesario usar un *cast* para convertirla a un puntero y evitar así un aviso (*warning*) del compilador.

[14] En el caso del ColdFire. Existen arquitecturas como la IA–32 en las que los registros de control de los periféricos están situados en otro espacio de direcciones, siendo necesarias instrucciones especiales (`in` y `out`) para su acceso.

[15] Los registros de configuración de periféricos en el ColdFire MCF5282 están todos situados a partir de una dirección base, denominada en el manual IPSBAR (*Internal Peripheral System Base Address Register*). Dicha dirección se inicializa en el reset a 0x40000000. Por tanto, en los ejemplos de este libro se supondrá que IPSBAR = 0x40000000.

Por otro lado, hay que aclarar el porqué se ha usado la palabra clave **volatile** en la declaración de la variable. Si se declara la variable p_pcsr como una variable normal, el compilador para optimizar el código cargará la variable en un registro antes de entrar en el bucle y luego, en lugar de volver a leer la variable de memoria en cada iteración, usará la copia del registro, que es mucho más eficiente. El problema radica en que cuando el temporizador termine y cambie el bit 2 del registro MCF5282_PIT0_PCSR, el programa no se enterará, puesto que estará comprobando el valor antiguo que guardó en el registro. El programa por tanto se quedará en un bucle infinito. Para evitar esto, mediante la palabra clave **volatile** se informa al compilador que la variable puede cambiar por causas externas a la ejecución del programa. El compilador entonces se verá obligado a leer siempre la palabra de la memoria, en lugar de usar una copia guardada en un registro interno.

2.4.1. Acceso a registros internos sin usar una variable de tipo puntero

Los pasos seguidos en el ejemplo anterior para acceder a un registro de configuración, pueden realizarse sin necesidad de usar una variable auxiliar de tipo puntero para almacenar la dirección del registro. La misma función puede escribirse así:

```
#define PIF (1<<2)
...

void EsperaFinTemp(void)
{
  while( (*(volatile uint16 *)0x40150000 & PIF) == 0)
    ; /* Espera fin temporizador */
}
```

No obstante, en estos casos es mejor definir una constante para mejorar la legibilidad del código:

```
#define PIF (1<<2)
#define PCSR_PIT0 (*(volatile uint16 *)0x40150000)
...

void EsperaFinTemp(void)
{
  while( (PCSR_PIT0 & PIF) == 0)
    ; /* Espera fin temporizador */
}
```

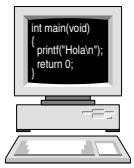
Esta es la alternativa usada en el entorno de desarrollo CodeWarrior para acceder a los registros del microcontrolador, tal como puede verse en el

archivo `mcf5282.h`.[16]

2.5. Uniones

Una unión es similar a una estructura salvo que en lugar de reservarse una zona de memoria para cada miembro de la estructura, se reserva una sola zona de memoria a compartir por todos los miembros de la unión. En el ejemplo siguiente, se reserva una sola palabra de 16 bits.

```
typedef struct{
  uint8 byte1;
  uint8 byte2;
}DOSBYTES;

typedef union{
  uint16 palabra;
  DOSBYTES bytes;
}U_WORD_BYTE;
```



Realice los ejercicios 7 y 8

A esta zona de memoria se puede acceder o bien como una palabra de 16 bits o bien como una estructura formada por dos bytes. Esto permite escribir en la unión una palabra para luego poder acceder a los dos bytes que la componen, lo cual es útil si se desea invertir el orden de ambos bytes. Por ejemplo, la siguiente función invierte el orden de los dos bytes de una palabra, lo cual es necesario si se desea enviar una palabra desde una máquina *Little Endian* a una máquina *Big Endian*.

```
uint16 swap(uint16 ent)
{
  U_WORD_BYTE uwb;
  uint8 temp;

  uwb.palabra = ent;
  temp = uwb.bytes.byte1;
  uwb.bytes.byte1 = uwb.bytes.byte2;
  uwb.bytes.byte2 = temp;

  return uwb.palabra;
}
```

[16] Las definiciones de constantes para acceder a los registros en este archivo son un poco más complejas, ya que la dirección base de los registros (IPSBAR) puede cambiarse. No obstante, la idea principal es la misma que se ha expuesto aquí.

2.6. Extensiones del lenguaje

Los compiladores ofrecen ciertas extensiones al lenguaje que permiten, entre otras cosas, un acceso al *hardware* que no se previó en el estándar. En esta sección se van a discutir dos extensiones que son imprescindibles en los programas de bajo nivel: escritura de instrucciones en ensamblador y soporte de interrupciones.

El problema de estas extensiones es que no son estándares, por lo que cada compilador implanta las que sus diseñadores creen más convenientes. No obstante, todos los compiladores suelen tener las extensiones que se van a discutir aquí, aunque con distinta sintaxis.

2.6.1. Uso de ensamblador en C

Cuando se realizan programas de bajo nivel, hay situaciones en las que no hay más remedio que usar instrucciones en código máquina. Por ejemplo, en el ColdFire, tal como se verá en el capítulo siguiente, para habilitar las interrupciones hay que usar una instrucción de código máquina especial que copia un valor en el registro de estado de la CPU (denominado SR). Para facilitarle la vida al programador, la mayoría de los compiladores disponen de mecanismos que permiten introducir instrucciones en ensamblador dentro de una función en C. En el caso de CodeWarrior, basta con usar la directiva `asm`, encerrando entre llaves el código en ensamblador; tal como se muestra en el siguiente ejemplo:

```
void EnableInt(void)
{
  asm{
    move.w  #0x2000,SR
  }
}
```

Incluso es posible acceder a las variables definidas en la función desde el código en ensamblador, tal como se muestra en el siguiente ejemplo para CodeWarrior:

```
long square(short a)
{
   asm {
     move.w  a,d0    // Copia la variable a al registro
     mulu.w  d0,d0   // lo eleva al cuadrado
   }
  return;
  /* Por convención las funciones devuelven el resultado
     en el registro D0. Como ya se ha puesto el resultado
     en D0 en el ensamblador, no hace falta poner nada en
```

```
    el return. */
}
```

Conviene destacar que en estos casos el compilador se encarga de generar el código para usar los argumentos, crear variables locales, salir de la función, etc. Es por ello que en lugar de usar la instrucción de código máquina `rts` para salir de la función desde el código en ensamblador, se ha usado la instrucción de C **`return`** que hace que el compilador genere el código necesario para salir de la función ordenadamente.

2.6.2. Soporte de interrupciones

El estándar ANSI C 1999 no contempla el soporte de interrupciones, ya que éstas dependen del procesador que se esté usando. Por ejemplo, hay procesadores como el MIPS que cuando se produce una interrupción saltan siempre a una posición de memoria determinada, mientras que otros como el ColdFire o el Infineon 167, saltan a una posición de memoria en función de la interrupción producida. En estos casos se dice que el procesador tiene un sistema de interrupciones vectorizadas, ya que se construye en la memoria un vector de direcciones de forma que cuando se produce la interrupción número n se salta a la posición n del vector. Por ejemplo el 167 salta a la posición 0x20 del vector de interrupciones cuando se produce una interrupción del temporizador 0.

También conviene tener en cuenta que una rutina de atención a interrupción es distinta de una función normal. Para empezar, como puede ser llamada en cualquier instante, ha de guardar cualquier registro que vaya a modificar. Además, la instrucción de código máquina para retornar de la rutina de interrupción es distinta de la de una función normal.[17] Por último, conviene recordar que las rutinas de atención a interrupción no devuelven ni reciben ningún valor.

Afortunadamente, la mayoría de los compiladores disponen de extensiones para el soporte de interrupciones. Así, en el compilador de Keil para Infineon 167, la definición de una rutina de atención a interrupción se realiza añadiendo la palabra clave `interrupt` seguida del número del vector de interrupción al que se asociará la función, tal como se muestra a continuación:

```
void InterruptTimer0(void) interrupt 0x20
{
  /* Rutina de atención a la interrupción del timer 0 */
  /* Compilador Keil para 167                         */
}
```

[17] En el salto a una función normal sólo se guarda el contador de programa. Sin embargo, cuando se salta a una rutina de interrupción ha de guardarse además el registro de estado como mínimo.

En cambio, el compilador CodeWarrior para ColdFire sólo permite definir una función como de atención a interrupción. Para ello se precede su definición con la directiva `__declspec(interrupt)`, tal como se muestra en el siguiente ejemplo. Sin embargo, al contrario que el compilador Keil, deja al programador la labor de inicializar el vector de interrupción correspondiente, para que apunte a la rutina de atención a interrupción:

```
__declspec(interrupt) void InterruptPIT0(void)
{
  /* Rutina de atención a la interrupción de PIT0 */
  /* Compilador CodeWarrior para ColdFire         */
}
```

Por último, a continuación se muestra otro ejemplo de declaración de rutina de atención a interrupción, en este caso para el compilador de *software* libre gcc:

```
void __attribute__((interrupt("IRQ")))
                                SYS_kbd_irq_handler(void)
{
  /* Rutina de atención a la interrupción del teclado */
  /* Compilador gcc para IA-32 (Linux)                */
}
```

Este compilador, al igual que CodeWarrior, deja al programador la labor de inicializar el vector de interrupción.

El que CodeWarrior y gcc no inicialicen el vector de interrupción no se debe a que sus autores sean torpes. Es debido a que estos compiladores están orientados a microprocesadores más complejos que permiten situar el origen de los vectores de interrupción en una posición arbitraria de memoria, por lo que el compilador no puede saber a priori dónde ha de colocar la dirección de la rutina de interrupción. Por el contrario, en el 167 la tabla de vectores de interrupción está siempre en el mismo sitio, con lo que el compilador puede encargarse de esta tarea. En el capítulo 3 se mostrará como inicializar el vector de interrupción en el ColdFire MCF5282.

OJO: En ambos casos: `__declspec(interrupt)` y `__attribute__` las palabras clave están precedidas por **dos** guiones bajos.

2.7. Ejercicios

1. Evalúe el resultado de las dos expresiones:
   - `(5 || !3) && 6`
   - `(5 | ~3) & 6`
2. Suponga que necesita poner a 1 el bit 4 de un registro denominado `MCF_REG_SOLO_ESCRITURA`, que como su propio nombre indica, es un

registro de sólo escritura; es decir, que si se lee de su dirección se obtendrán valores aleatorios, pero no el último valor que se escribió.[18] Escriba una función que realice esta tarea.

Pista: Use una variable para almacenar el último valor escrito en el registro.

3. Siguiendo con el ejercicio anterior, escriba una función para modificar el bit 4 del registro `MCF_REG_SOLO_ESCRITURA`. La función tendrá un único argumento que será el valor a escribir en dicho bit.

4. Modifique la función del ejercicio anterior para que se pueda especificar el número de bit a modificar y el valor que se desea escribir en el registro.

5. Escriba las sentencias necesarias para modificar los campos del registro MCF5282_PIT0_PCSR tal como se indica:

   a) Poner el campo Prescaler a 0110

   b) Poner a 1 los bits PIE, RLD y EN.

   c) Poner a 0 el bit Doze.

   Para mayor comodidad, el registro PCSR se vuelve a mostrar a continuación.

6. Utilizando el tipo `BCD2` definido en la página 43, realice un programa que sume dos números en BCD. En una primera versión, los dos números se inicializarán en el código y en una segunda versión se pedirán al usuario (dígito a dígito). Recuerde que el algoritmo para sumar números en BCD es el siguiente:

   - Los números se suman dígito a dígito, empezando por los dos dígitos menos significativos.
   - Si el resultado de sumar dos dígitos es mayor de 9 o si se produce un acarreo al siguiente dígito, se suma 6 a dicho resultado. En cualquiera de estos dos casos, es necesario sumar 1 (acarreo) a los dos dígitos siguientes.

   A continuación se muestra un ejemplo:

[18] Esto es muy frecuente en la práctica, pues se simplifica el *hardware*, aunque a costa de complicarle la vida el programador.

```
   1    1
0001 1001 0111
0010 1000 0100
--------------
0100 0010 1011
     0110 0110
     ---- ----
     1000 0001
```

En la suma del primer dígito (7 + 4) se produce un resultado mayor que 9, por lo que se suma 6 a la cifra obtenida y se acarrea un 1 a las siguientes dos cifras (9 y 8). En la suma de estas dos cifras se produce un acarreo a la tercera cifra, por lo que también ha sido necesario sumar 6.

7. Escriba de nuevo la función `swap` mostrada en la sección 2.5 (página 48) usando desplazamientos y máscaras en lugar de una unión.

8. Escriba una función para convertir una palabra de 32 bits de formato *Little Endian* a formato *Big Endian*.

CAPÍTULO 3

Sistemas *Foreground/Background*

3.1. Introducción

En este tema se van a estudiar en mayor profundidad los sistemas basados en interrupciones (*Foreground/Background*):

En primer lugar se va a estudiar el soporte de interrupciones de la familia de microcontroladores ColdFire de Freescale. A continuación se verán los métodos para evitar problemas de incoherencia de datos y por último se estudiará la planificación de las tareas de primer plano.

3.2. Soporte de interrupciones en ColdFire

El microcontrolador ColdFire no es más que una CPU de la familia 68000[1] a la que se le han añadido una serie de periféricos integrados en el mismo chip. La familia 68000 fue muy popular en los años 80 usándose, por citar algunos ejemplos, en los primeros Apple Mac, en las estaciones de trabajo de Sun Microsystems, en el Atari ST y en el Commodore Amiga. Debido a que estaba dirigida a este segmento, se añadieron funcionalidades para dar soporte al sistema operativo. Precisamente por esto, dispone de dos modos de funcionamiento: el modo supervisor y el modo usuario. En el primero están accesibles todos los recursos de la máquina y es el modo que usa el sistema operativo para su ejecución. En el modo usuario hay ciertas partes de la máquina inaccesibles al programa para evitar en lo posible que un comportamiento anómalo de éste deje bloqueado al sistema operativo. Una de las partes inaccesibles es el registro de estado (SR) en el que se controla la habilitación e inhabilitación de interrupciones, ya que, como veremos más adelante, son las interrupciones las que permiten que el sistema operativo tome el control sobre los programas de usuario, además de facilitar la interacción con los distintos periféricos, lo cual es también labor del sistema operativo.

[1] El nombre 68000 se debe al número de transistores que contenía el primer chip de la familia.

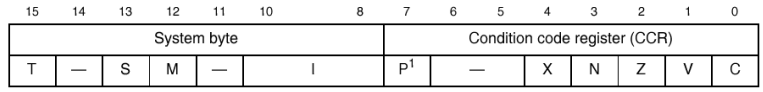


Figura 3.1: Registro de estado del ColdFire

El modo de funcionamiento, usuario o supervisor, se controla mediante el bit 13 del registro de estado de la CPU (bit S). Si está a 1 la CPU trabaja en modo supervisor y si está a cero trabaja en modo usuario. Cuando la CPU arranca, lo hace en modo supervisor, ya que en el arranque es el sistema operativo el que toma el control de la máquina. Normalmente el sistema operativo cambia al modo usuario al lanzar los programas, aunque algunos sistemas como el Apple Mac trabajaban siempre en modo supervisor. En el campo de los sistemas empotrados se suele trabajar también en modo supervisor, pues los programas que se realizan suelen interactuar directamente con el *hardware* y con las interrupciones, no existiendo una clara distinción entre sistema operativo y tareas de usuario.

3.2.1. Niveles de interrupción

El ColdFire dispone de 7 niveles de interrupción. Cuando se está ejecutando una rutina de atención a interrupción, en el campo I del registro de estado SR, mostrado en la figura 3.1, se almacena el nivel de dicha interrupción, de forma que la CPU sólo atenderá interrupciones de un nivel superior. Esto permite gestionar las latencias de los dispositivos asociando los niveles de interrupción más prioritarios a los periféricos que necesiten menor latencia. El nivel 7 presenta una excepción a esta regla, pues aunque el campo I esté a 7, la interrupción se atenderá. Se dice en estos casos que es una interrupción no enmascarable. Este nivel de interrupción se reserva en la práctica para aquellos periféricos que no puedan esperar. Un ejemplo típico es colocar un supervisor de tensión, que genere una interrupción si la tensión empieza a bajar, para que la CPU guarde su estado en memoria no volátil. Obviamente, esta interrupción ha de atenderse de inmediato, pues de lo contrario la tensión caerá por completo y ya no se podrá guardar nada.

Como se dijo antes, el registro SR sólo es accesible en modo supervisor, para evitar que un programa de usuario inhabilite las interrupciones por error.

3.2.2. Habilitación e inhabilitación de interrupciones

Para habilitar e inhabilitar las interrupciones basta con escribir un 0 o un 7 respectivamente en el campo I del registro SR. Además, en ambos

casos hay que dejar el bit de supervisor (bit 13) a 1. Como el acceso al registro SR se realiza mediante una instrucción especial no hay más remedio que usar el ensamblador. A continuación se muestran dos funciones para realizar esta labor:

```
void Enable(void)
{
  asm{
    move.w #0x2000,SR
  }
}

void Disable(void)
{
  asm{
    move.w #0x2700,SR
  }
}
```

Ambas funciones se han escrito para el compilador CodeWarrior para ColdFire. Tal como se ha mostrado en la sección 2.6.1, para incluir instrucciones en ensamblador en este compilador basta con encerrarlas dentro de la directiva `asm{}`.

3.2.3. Controladores de interrupciones en el ColdFire MCF5282

El MCF5282 dispone de dos controladores de interrupciones: INTC0 e INTC1. El controlador INTC1 se usa para las interrupciones del bus CAN, que no va a ser usado en este texto. El INTC0 se usa para el resto de periféricos. De todas formas, el principio de funcionamiento de ambos controladores es el mismo, por lo que todo lo expuesto para el INTC0 es válido para el INTC1.

Cada controlador gestiona hasta 63 fuentes de interrupción, permitiendo activar sólo aquellas que interesen en cada momento mediante un registro de máscaras. En el arranque todas las interrupciones están enmascaradas, para que si no se inicializan los periféricos no se produzcan interrupciones indeseadas. Eso sí, si se olvida desenmascarar la interrupción del periférico que se desea usar, no se conseguirá que éste interrumpa.

Cada fuente de interrupción dispone de un registro, denominado `ICRxx` en donde `xx` es el número de la fuente de interrupción, para definir su nivel y su prioridad. La prioridad de cada fuente de interrupción es totalmente programable. Como la CPU sólo soporta 7 niveles de interrupción, el controlador de interrupciones permite asignar cada fuente de interrupción a un nivel determinado, y dentro de ese nivel permite asignar prioridades para que, en caso de que dos dispositivos interrumpan a la vez, se procesen sus peticiones en orden. Conviene destacar que en este caso se procesan

las interrupciones completamente, es decir, hasta que no termina la rutina de atención del primer periférico no se atenderá la del siguiente del mismo nivel. Por el contrario, si mientras se está procesando la interrupción de un nivel llega una petición con un nivel más alto, se dejará de procesar la interrupción de menor nivel para pasar a procesar la de alto nivel. Cuando esta última termine se reanudarla el proceso de la interrupción de nivel inferior.

Para cada fuente de interrupción la CPU salta a una rutina cuya dirección ha de estar almacenada en la tabla de vectores de interrupción. La correspondencia entre la fuente de interrupción y su índice en la tabla es:

- Para INTC0: 64 + Nº fuente interrupción.
- Para INTC1: 128 + Nº fuente interrupción.

Para más información puede consultar el capítulo 10 del manual del microcontrolador MCF5282 [FreeScale, 2005].

Ejemplo de configuración del controlador de interrupciones

A continuación se muestra el proceso a seguir para configurar el controlador de interrupciones, de forma que las interrupciones generadas por un periférico lleguen a la CPU. Como ejemplo se usará el temporizador PIT0 (*Programmable Interrupt Timer 0*), aunque para el resto de periféricos el proceso será idéntico.

En primer lugar es necesario definir con qué nivel y con qué prioridad va a interrumpir el periférico. Como el PIT0 es la fuente de interrupción número 55, tal como puede verse en la tabla 10-13 del manual [FreeScale, 2005], habrá que configurar el registro ICR55. Para facilitar la vida al programador, en el archivo de cabecera `"mcf5282.h"` se definen las direcciones de estos registros con el nombre `MCF_INTC0_ICRxx`, siendo `xx` el número de la fuente de interrupción.[2] En dichos registros, los 3 bits menos significativos almacenan la prioridad y los 3 siguientes el nivel. Así, para asignar a la interrupción del PIT0 el nivel 1 y la prioridad 0 habrá que escribir en el registro ICR55 un 001 000, que en hexadecimal es un 0x08.

Una vez definidos el nivel y la prioridad, es necesario desenmascarar la interrupción para que ésta llegue a la CPU. Para ello el controlador de interrupciones dispone de un registro de 64 bits en el que el bit número *n* enmascara la fuente de interrupción número *n* (un 1 impide la interrupción y un 0 la permite). Así, para permitir las interrupciones del PIT0, como éste es la fuente de interrupción número 55 habrá que poner a cero el bit 55 de

[2] En realidad, en la versión 7 de CodeWarrior, el archivo `"mcf5282.h"` a su vez incluye varios archivos `.h` con la definición de los registros relacionados según su función. Por ejemplo, las direcciones de los registros `MCF_INTC0_ICRxx` están en realidad en el archivo `"mcf5282_INTC.h"`.

dicho registro. Como el ColdFire sólo puede acceder directamente a palabras de 32 bits, el registro de máscaras se divide en dos, el IMRH para los 32 bits más significativos y el IMRL para los 32 menos significativos. Además en el archivo "mcf5282.h" están definidas unas máscaras para acceder a cada uno de los bits. Estas máscaras tienen un 1 en el bit correspondiente a su fuente de interrupción. Así por ejemplo la máscara para la fuente nº 7 se denomina MCF_INTC_IMRL_INT_MASK7 y es igual a 10000000_{Bin}.[3] Por otro lado, la máscara para la fuente 37 se llama MCF_INTC_IMRH_INT_MASK37 y vale 00100000_{Bin}. Según lo anterior, para desenmascarar una interrupción hay que hacer una AND con la máscara negada para así poner el bit correspondiente a cero. Para enmascararla habrá que hacer una OR con la máscara para ponerlo a 1, pero siempre teniendo en cuenta que los bits del 0 al 31 van al registro IMRL y los del 32 al 63 al IMRH. Es conveniente volver a resaltar que en el arranque todos estos bits están a 1, con lo que todas las interrupciones están enmascaradas.

El código por tanto quedaría como:

```
MCF_INTC0_ICR55 = 0x08; /* Nivel 1 Prioridad 0 */
MCF_INTC0_IMRH &= ~MCF_INTC_IMRH_INT_MASK55; /* PIT0
                                        Desenmascarada */
```

Con las dos instrucciones anteriores se consigue que cuando el periférico interrumpa, a la CPU le llegue la señal de interrupción. La respuesta de la CPU ante la interrupción consiste en dejar lo que esté haciendo en ese momento[4] y ejecutar una rutina específica para atenderla. Es por tanto necesario establecer una correspondencia entre interrupción y rutina de atención. En el ColdFire esta correspondencia se realiza mediante una tabla de vectores de interrupción. Cada interrupción tiene un número asociado que se usa para localizar en la tabla de vectores la dirección de la rutina que la atiende. El ColdFire dispone de una tabla con 256 elementos. Como los elementos de la tabla son direcciones, cada elemento ocupará 4 Bytes.

La tabla de vectores puede situarse en cualquier posición de la memoria, siempre que ésta sea múltiplo de 1 MB. Dicha posición se almacena en el registro VBR, accesible sólo en modo supervisor. En el sistema M5282LiteES usado en los ejemplos de este texto, el registro VBR se inicializa por defecto a la posición 0x20000000.

De los 256 vectores, los 64 primeros se usan para atender sucesos internos a la CPU como divisiones por cero, reset, etc. Los 64 siguientes se usan para los periféricos asociados al controlador de interrupciones INTC0. Los otros 64 están asociados al INTC1 y el resto no se usan en este microcontrolador.

---

[3] El valor de la máscara es de 32 bits, aunque para simplificar sólo se han mostrado aquí los 8 bits menos significativos, ya que los más significativos están a 0.

[4] Salvo que esté ya ejecutando otra interrupción de mayor prioridad.

Siguiendo con el ejemplo anterior, a continuación se muestra cómo inicializar el vector de interrupciones para atender la interrupción del temporizador PIT0. Se supone en primer lugar que la función de atención a la interrupción se denomina `IntPIT0`. Lo primero que hay que tener en cuenta es que la tabla de vectores empieza en la posición 0x20000000. Como el PIT0 está asociado al controlador de interrupciones INTC0 y, según se ha mencionado antes, es la fuente de interrupción número 55; la entrada en la tabla en la que hay que copiar la dirección de su rutina de atención será 64 + 55. Para obtener la dirección de memoria en la que copiar la dirección de la función `IntPIT0`, habrá que sumar a la dirección base de la tabla el número de vector (64 + 55) multiplicado por 4, porque cada entrada en la tabla ocupa 4 Bytes. Por último, la dirección obtenida es la dirección de un puntero a una función, es decir una dirección de una dirección. Por ello hay que convertirlo mediante un *cast* a un puntero doble. En el contenido de dicho puntero doble, que será la entrada en la tabla, se escribe la dirección de la función, previamente convertida a puntero **`void`**. Recuerde que el nombre de la función es su dirección, al igual que el nombre de un vector es la dirección de su primer elemento. Así pues el código quedaría como:

```
*(void**)(0x20000000+(64+55)*4)=(void*)IntPIT0;
```

Nótese que se han usado punteros genéricos **`void`** en lugar de punteros a funciones para simplificar la nomenclatura, aunque lo más correcto sería lo segundo. No obstante, como en este caso se sabe perfectamente lo que se está haciendo, tampoco es muy grave engañar un poco al compilador.

3.3. Datos compartidos

Para reducir la latencia, las rutinas de atención a interrupción (ISR)[5] deben de tardar poco tiempo en ejecutarse. Por ello deben limitarse a realizar el trabajo estrictamente necesario para atender al *hardware* y dejar todo el proceso de los datos para las tareas de primer plano. Esto obliga a que exista una comunicación entre las rutinas de atención de interrupción y las tareas de primer plano. La manera más fácil de realizar esta comunicación es mediante el uso de una variable global (o varias) compartida entre ambas tareas.

Ahora bien, para evitar problemas de coherencia de datos como el ilustrado en la introducción, la tarea de primer plano ha de usar los datos compartidos de manera atómica. Se dice que un trozo de programa es atómico si éste no puede ser interrumpido. Por tanto, para que un trozo de programa sea atómico, éste ha de ser, o bien una sola instrucción en código máquina[6] o bien un trozo de programa que se ejecute mientras las interrupciones estén inhabilitadas.

[5] Del inglés *Interrupt Service Routine*.

[6] Existen algunas excepciones a esta regla. Por ejemplo, en los procesadores de la

Para inhabilitar y habilitar las interrupciones lo mejor es usar dos funciones (o macros), una para habilitarlas, que se suele denominar `Enable()` y otra para inhabilitarlas (`Disable()`). El hacerlo así permite una mayor portabilidad del código, ya que si se cambia de plataforma sólo habrá que cambiar estas dos funciones (o macros). En la sección 3.2.2 se ha mostrado cómo implantar estas funciones en un ColdFire usando el compilador CodeWarrior.

3.3.1. Ejemplo

A continuación se muestra un ejemplo en el que se comparte un dato entre una rutina de interrupción y una tarea de primer plano. En todos los sistemas el tiempo es una variable fundamental: permite generar retardos, sincronizar tareas, etc. Una manera fácil de gestionar el tiempo es programar un temporizador para que genere interrupciones de forma periódica y en la rutina de atención a la interrupción de dicho temporizador ir incrementando una variable, a la que se suele denominar `ticks`. Accediendo a dicha variable se podrá saber el tiempo que lleva encendido el ordenador, calcular el tiempo que se tarda en realizar un proceso, esperar un número de "ticks" determinado, etc.

Cuando se realizan programas de cierta envergadura, es necesario dividir el programa en módulos. Estos módulos han de estar lo más aislados posible del resto del programa. Para ello se les dota de un interfaz que permite aislar a dicho módulo del resto. Este ejemplo también se va a aprovechar para introducir este concepto. Para que la variable `ticks` no esté accesible a todo el programa, de forma que cualquier función por error pueda modificarla, se va a crear una función que devuelve su valor, a la que se denominará `TicksDesdeArr`. El código del módulo, que se guardará en un archivo denominado `temporizador.c`, es el siguiente:

```
static uint32 ticks = 0; /* Per. reloj desde arranque */

__declspec(interrupt) IntPIT0(void)
{
  ticks++;
}

uint32 TicksDesdeArr(void)
{
  return ticks;
}
```

familia IA-32 existen instrucciones de copia de cadenas de caracteres que, como pueden tardar mucho tiempo en ejecutarse si las cadenas son largas, pueden ser interrumpidas en mitad del proceso. No obstante, este tipo de instrucciones no pueden interrumpirse en cualquier punto, sino sólo cada vez que terminan un ciclo de acceso a la memoria.

Nótese que la variable `ticks` se ha definido fuera de las funciones, por lo que será global. No obstante, se ha precedido de la palabra clave **`static`**. Con esto se consigue que la variable sea accesible como global a todas las funciones que estén dentro del módulo `temporizador`,[7] pero no será accesible al resto de las funciones del programa. La forma de conocer el número de "ticks" de reloj transcurridos por el resto de módulos es mediante una llamada a la función `TicksDesdeArr`. Para que esta función pueda ser llamada desde otros módulos es necesario que dichos módulos conozcan su prototipo. Para ello, junto con el archivo `temporizador.c` se crea un `temporizador.h` en el que se especifica el interfaz del módulo. Este archivo en este caso sería:

```
#ifndef TEMPORIZADOR_H
#define TEMPORIZADOR_H

uint32 TicksDesdeArr(void);

#endif
```

Cabe preguntarse ahora si existirán problemas de coherencia de datos en este programa. Como se ha mencionado antes, si el trozo de programa en el que se usan los datos compartidos es atómico no habrá ningún problema. En este ejemplo sólo hay una variable compartida, `ticks` y ésta sólo se usa en la sentencia C **`return`** `ticks;`. Ahora bien, el que esta sentencia C sea atómica depende de cómo la traduzca el compilador, lo cual a su vez depende del microprocesador. Así por ejemplo si se usa gcc para un Pentium en Linux, la sentencia se traducirá en:

```
mov EAX, ticks
```

Que es atómica. Si se usa en cambio el compilador de Keil (o el gcc) para 167, que es un microcontrolador de 16 bits, la misma sentencia se traducirá en:

```
mov R0, ticks
mov R1, ticks+2
```

Que obviamente no es atómica. Por tanto, si se programa en C, es mejor asegurarse la atomicidad de las zonas críticas inhabilitando las interrupciones durante la zona crítica. Para ello, lo que **no puede hacerse** es:

```
static uint32 ticks=0; /* Per. reloj desde arranque */

__declspec(interrupt) void IntPIT0(void)
{
```

[7] Decir todas las funciones que estén dentro del módulo `temporizador` es equivalente a decir todas las funciones que estén dentro del archivo `temporizador.c`.

```
  ticks++;
}

uint32 TicksDesdeArr(void)
{
  Disable();
  return ticks;
  Enable();
}
```

Obviamente el código anterior está mal, pues se retorna de la función antes de volver a habilitar las interrupciones, con lo que se parará el sistema al no volver a atenderse las interrupciones nunca más. La forma correcta de hacerlo es realizar una copia de la variable compartida con las interrupciones inhabilitadas, de forma que dicha copia sea atómica:

```
uint32 TicksDesdeArr(void)
{
  uint32 c_ticks;

  Disable();
  c_ticks = ticks;
  Enable();
  return c_ticks;
}
```

En general, para evitar problemas al compartir variables, es conveniente copiar las variables compartidas a variables locales con las interrupciones inhabilitadas y luego usar las copias en la función. Además, así se minimiza el tiempo durante el cual están inhabilitadas las interrupciones, con lo que se mejora la latencia del sistema.

Obviamente este método conlleva una pequeña pérdida de rendimiento, ya que se pierde un poco de tiempo al copiar las variables. No obstante, salvo que las variables compartidas tengan un tamaño elevado, como por ejemplo un vector o una estructura; la pequeña pérdida de rendimiento compensa el aumento de fiabilidad del sistema.

3.3.2. Anidamiento de zonas críticas

Cuando se usan variables compartidas hay que ser especialmente cuidadoso, ya que se pueden cometer errores que son bastante difíciles de encontrar y que sólo se manifestarán cuando se produzca una interrupción durante una zona crítica no protegida. En el listado siguiente se muestra un ejemplo de uso de la función `TicksDesdeArr` mostrada en el apartado anterior:

```
void UnaFuncion(void)
```

```
{
  uint32 t;

  Disable();
  /* Hacemos algo */
  t = TicksDesdeArr();
  /* Hacemos más cosas */
  Enable();
}
```

En el ejemplo anterior se llama a la función `TicksDesdeArr` desde una zona crítica, lo cual al parecer es algo inocente. Sin embargo, como se acaba de ver, la función `TicksDesdeArr` tiene una zona crítica, con lo que inhabilita las interrupciones, que no es grave, pero luego las vuelve a habilitar, lo cual se convierte en una bomba de relojería, ya que la zona de código `/* Hacemos más cosas */` se ejecutará con las interrupciones habilitadas y por tanto dejará de ser atómica. Por supuesto, según las leyes de Murphy, seguro que el sistema funcionará sin problemas hasta que esté instalado en las máquinas del cliente y además sólo fallará cuando usted no esté presente.

La solución para evitar este tipo de errores es obviamente documentar claramente qué funciones contienen una zona crítica y no llamarlas desde otra zona crítica. Si además prevemos que una función de este tipo puede ser llamada desde una zona crítica, lo mejor es prepararse para ello. Una solución es la mostrada en el siguiente listado:

```
void TicksDesdeArr(void)
{
  uint32 copia_ticks;
  uint32 est_ant; /* Estado anterior de interrupciones */

  est_ant = Disable();
  copia_ticks = ticks;
  if(est_ant){
    Enable();
  }
  return copia_ticks;
}
```

Tal como se puede apreciar, se hace que la función `Disable()` devuelva el estado anterior de las interrupciones (0 inhabilitadas, 1 habilitadas). Si estaban inhabilitadas, es señal de que la función ha sido llamada desde una zona crítica y por tanto no ha de volver a habilitar las interrupciones. Si por el contrario estaban habilitadas, la zona crítica corresponde sólo a la función y por tanto ha de volver a habilitar las interrupciones al terminar su zona crítica.

Para obtener una mayor seguridad de que el código no contiene *bugs* escurridizos como el mostrado en la transparencia anterior, puede usarse el esquema propuesto en esta transparencia para proteger **todas** las zonas críticas. El inconveniente será el sacrificar las prestaciones del sistema pues se complica un poco el código, pero a cambio se obtiene una mayor fiabilidad, lo cual es siempre lo más importante en un sistema en tiempo real.

3.3.3. Habilitación e inhabilitación de interrupciones de forma segura

Para implantar el mecanismo propuesto en la sección anterior que resuelve el problema del anidamiento de zonas críticas, es necesario que la función `Disable` devuelva el estado de las interrupciones. En el ColdFire, dado que este estado está almacenado en los bits 10 a 8 del registro de estado, es necesario recurrir de nuevo a la programación en ensamblador para obtenerlo; tal como se muestra en el siguiente listado:

```
#ifndef INTERRUPCIONES_H
#define INTERRUPCIONES_H
#include "mcf5282.h"

inline void Enable(void)
{
 asm
 {
   move.w #0x2000,SR
 }
}

inline uint32 Disable(void)
{
  uint32 ret;
  asm
  {
    move.l d3, -(a7)  // Guarda d3 en la pila
    move.w SR, d3     // Lee el SR
    asr    #8, d3     // Pasamos el campo I al bit 0
    andi   #7, d3     // lo dejamos a él solito
    move.l d3, ret    // y se copia a la variable de retorno
    move.l (a7)+, d3  // Se restablece el d3
    move.w #0x2700,SR // Y se inhabilitan Interrupciones
  }

  return !ret; // Si vale 0 es que estaban habilitadas
```

```
                // pero la función ha de devolver 1 en
                // este caso
}
#endif
```

A la vista del listado, es necesario hacer énfasis en lo siguiente:

- Como ambas funciones son muy cortas y se usan muy frecuentemente en los programas se han declarado como `inline`. Esto hace que el compilador se limite a sustituir el código de la función en lugar de realizar una llamada a ésta. Así, si tenemos el código:

  ```
  a = b;
  Enable();
  c= d;
  ```

  El código generado será:

  ```
  move.w b, a
  move.w #0x2000,SR
  move.w c, d
  ```

  En lugar de:

  ```
  move.w b, a
  jsr Enable
  move.w c, d
  ```

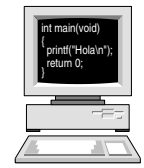
  Esto permite una mayor eficiencia del programa al ahorrar la sobrecarga de la llamada a la función, pero a cambio obtenemos un programa con más instrucciones[8] que, por tanto, necesitará más memoria.

- El inconveniente de las funciones declaradas como `inline` es que se resuelven en tiempo de compilación, por lo que tienen que estar en el mismo archivo desde donde se llaman. Otra opción más elegante es escribir las funciones `Enable` y `Disable` dentro de un archivo de cabecera (`interrupciones.h` por ejemplo) e incluir dicho archivo cabecera en los archivos en los que se necesite usar estas funciones.

- A la hora de escribir la función `Disable`, no es posible copiar el registro de estado SR directamente a una variable en memoria, por lo que ha de usarse un registro intermedio para ello. Dicho registro ha de guardarse en la pila y restaurarse después de usarlo, pues no se puede saber desde la función `Disable` si está libre o por el contrario está siendo usado por el programa para almacenar alguna variable.

[8] Salvo en el caso particular del ejemplo mostrado para `Enable`, en el que la función consta de una sola instrucción.

- Para aislar el campo I del registro SR, ha sido necesario en primer lugar desplazarlo 8 bits a la derecha, para llevarlo al principio de la palabra; usando la instrucción `asr`. A continuación, se le ha aplicado una máscara con el número 7 (0111 en binario), para quedarse sólo con los tres bits del campo I; usando la instrucción `andi`. Recuerde que este campo será 0 si las interrupciones están habilitadas o 7 si están inhabilitadas.[9]



Realice el ejercicio 1.

3.3.4. Métodos para compartir variables sin inhabilitar interrupciones

Existen ciertas técnicas para compartir variables sin necesidad de inhabilitar las interrupciones. La finalidad de estos métodos es la de disminuir la latencia del sistema. El problema de estas técnicas es que suponen códigos más elaborados, en los cuales es fácil equivocarse y por tanto dar lugar a errores. Además estos errores serán difíciles de encontrar, pues sólo se manifestarán cuando se produzca una interrupción en una zona crítica desprotegida. Por tanto, salvo que tengamos problemas de latencia, lo más fiable a la hora de compartir variables es hacer una copia de éstas con las interrupciones inhabilitadas. Para los casos en los que esto no sea posible se muestran a continuación varios métodos para compartir variables sin inhabilitar las interrupciones. Tenga en cuenta que estos códigos funcionan correctamente tal cual están: un pequeño cambio del código puede hacer que no sean válidos.

Lecturas sucesivas

Un primer método consiste en la lectura de una variable compartida dos veces seguidas, dando por válido el valor leído si en ambas ocasiones se ha leído el mismo valor. En el siguiente listado se muestra cómo usar esta técnica para implantar la función `TicksDesdeArr`:

```
volatile static uint32 ticks=0;

uint32 TicksDesdeArr(void)
{
  uint32 copia_ticks;
  copia_ticks = ticks;
  while(copia_ticks != ticks){
```

[9] Nótese que este esquema sólo es válido si a la función `Enable` se le llama desde las tareas de primer plano. Si se tuviese una zona crítica dentro de una interrupción por compartirse datos entre dos rutinas de interrupción; al llamar a la función `Enable` se habilitarían todas las interrupciones, no sólo las de nivel superior a la interrupción en curso. En este caso si se produce una interrupción de nivel inferior, ésta interrumpiría la ejecución de la interrupción en curso, lo cual en ciertos casos puede ser problemático.

```
    copia_ticks = ticks;
  }
  return copia_ticks;
}
```

Esta técnica se basa en que si dos lecturas sucesivas de la variable compartida dan el mismo resultado es porque no se ha producido una interrupción durante la lectura y por tanto el valor ha de ser válido. Obviamente para que el método sea válido el bucle **while** ha de ejecutarse más rápido que el periodo de la interrupción, lo cual ocurrirá siempre.

Nótese que para evitar problemas con las optimizaciones del compilador la variable `ticks` se ha declarado como **volatile**. Si no se hace así, el compilador verá que se ha copiado un valor en una variable y a continuación se verifica si la variable y la copia son distintas. Lo más probable es que el compilador "piense" que el programador es un poco inútil y elimine por completo el bucle, con lo que ya no se tiene ninguna protección para evitar la incoherencia de datos. Al declarar la variable `ticks` como **volatile** se informa al compilador que la variable puede cambiar por causas ajenas al programa en curso, ya sea por una interrupción o por la intervención de un periférico externo. El compilador entonces leerá siempre la variable `ticks` de memoria, es decir, no eliminará lecturas o escrituras en la memoria al optimizar el código.

Este método sólo es apropiado cuando hay una o dos variables compartidas, pues de lo contrario la condición del bucle while se complica en exceso y las prestaciones del sistema empeoran al tener que leer dos veces consecutivas cada variable [Simon, 1999].

Doble *Buffer*

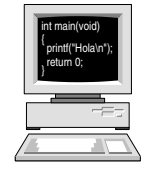
Otra técnica para compartir datos sin necesidad de inhabilitar las interrupciones es la de usar un **doble *buffer***. Esta técnica consiste en crear dos conjuntos de variables compartidas (*buffers*) de forma que mientras la tarea de primer plano usa un *buffer*, la tarea de interrupción usa el otro. Para que cada una sepa qué *buffer* ha de usar, se crea una bandera para arbitrar el acceso. Para ilustrar este método, se han vuelto a implantar tanto la función `TicksDesdeArr`, como la rutina de atención a interrupción del temporizador PIT0 para que usen un doble *buffer* para comunicarse:

```
static uint32 ticks=0;
static uint32 buf_ticks[2];
static uint8 tarea_usa_buf0 = FALSO;

__declspec(interrupt) IntPIT0(void)
{
  ticks++;
```

```
  if(tarea_usa_buf0){
    buf_ticks[1] = ticks;
  }else{
    buf_ticks[0] = ticks;
  }
}

uint32 TicksDesdeArr(void)
{
  uint32 copia_ticks;
  if(tarea_usa_buf0){
    copia_ticks = buf_ticks[0];
  }else{
    copia_ticks = buf_ticks[1];
  }
  tarea_usa_buf0 = !tarea_usa_buf0;
  return copia_ticks;
}
```

En este ejemplo, el doble *buffer* está formado por el vector `buf_ticks[2]` y la bandera para arbitrar el acceso es `tarea_usa_buf0`, la cual será cierta cuando la tarea de primer plano esté usando el *buffer* 0 (`buf_ticks[0]`). La rutina de atención de la interrupción del temporizador 0 se limita a copiar el valor de `ticks` en el *buffer* que la tarea de primer plano no esté usando en ese momento. Como se puede apreciar, la tarea de primer plano usa el *buffer* que le corresponde y sólo cuando ha terminado de usarlo invierte el valor de la bandera (`tarea_usa_buf0`) para que la siguiente vez que se ejecute use el otro *buffer* que la rutina de atención a la interrupción habrá actualizado.

A la vista de este código es fácil ver algunos inconvenientes de esta técnica:



Realice el ejercicio 2.

- La tarea de primer plano usará un valor que no está actualizado, ya que el más actual lo habrá escrito la rutina de atención a la interrupción en el otro buffer.
- Si la tarea de primer plano se ejecuta más rápido que la rutina de interrupción, es fácil darse cuenta que estará usando alternativamente un valor antiguo y otro más nuevo.
- Se pueden perder datos si ocurren dos interrupciones sin que se ejecute la tarea de primer plano.

El uso de esta técnica tiene sentido cuando el tamaño de las variables compartidas es grande y la tarea de primer plano hace cálculos complejos con estos valores, de forma que si se inhabilitan las interrupciones o

bien mientras se copian las variables compartidas o bien mientras se están usando; la latencia se hace demasiado grande.

Cola o *Buffer circular*

Uno de los inconvenientes del doble *buffer* mencionado en la sección anterior es su limitada capacidad de almacenamiento, ya que sólo puede guardar un nuevo valor mientras se está procesando el antiguo en la tarea de primer plano. Por tanto, si mientras dicha tarea de primer plano se está ejecutando llegan dos interrupciones, el dato generado por la primera interrupción se perderá para siempre.

En el ejemplo anterior, como la tarea de primer plano sólo necesita conocer el valor actual de la variable `ticks`, el doble `buffer` es una solución adecuada. Sin embargo, hay sistemas más complejos en los que han de leerse **todos** los datos proporcionados por la rutina de interrupción. Por ejemplo, supóngase que un sistema empotrado ha de comunicarse por un puerto serie con un ordenador. Mediante esta línea de comunicación se podrán enviar comandos al sistema, los cuales consistirán en una serie de caracteres terminados con un retorno de carro.

Para implantar este sistema, lo más lógico es dividir el trabajo entre una rutina que atienda la interrupción del puerto serie y una tarea de primer plano. La rutina de interrupción se limitará a leer el carácter recibido y, en una primera aproximación, guardarlo en la memoria usando una variable compartida o un doble *buffer*. La tarea de primer plano se encargará de ir leyendo estos caracteres y guardarlos en una cadena para formar el mensaje recibido e interpretarlo cuando llegue el retorno de carro. Obviamente, si en algún instante mientras la tarea de primer plano se está ejecutando llegan varios caracteres seguidos (con su correspondientes interrupciones),[10] se perderán caracteres. En este caso, la solución es usar un *buffer* mayor que haga de "colchón" entre la rutina de atención a interrupción y la tarea de primer plano. De esta forma, mientras se cumpla que por término medio la tarea de primer plano es capaz de procesar los caracteres que le envía la rutina de interrupción, el sistema funcionará perfectamente.

En este tipo de situaciones, lo más apropiado es usar una cola, cuya implantación se muestra a continuación. En primer lugar se muestra la rutina de interrupción, la cual lee un carácter de la UART[11] y lo deposita en la cola.

```
#define TAM_COLA 100
```

[10] Esta situación se dará si el procesador no es muy potente y los mensajes son complejos de interpretar.

[11] UART son las siglas de *Universal Asincronous Receiver Transmitter* y no es más que un circuito que se usa para enviar y recibir caracteres por un puerto serie. Estas UART se pueden configurar para que generen una interrupción cuando reciben un carácter, el cual almacenan en un registro; que en el caso del ColdFire MCF5282 se denomina `MCF_UART0_URB`.

```
static char cola[TAM_COLA];
static uint8 icabeza=0; /*índice para añadir*/
static uint8 icola=0;   /*índice para leer*/

__declspec(interrupt) InterrSerie0(void)
{
  if( (icabeza+1 == icola) ||
      (icabeza+1 == TAM_COLA && icola == 0)){
    /* La cola está llena */
  }else{
    cola[icabeza] = MCF_UART0_URB; /* Lee carácter del
                                      puerto serie */
    icabeza++;
    if(icabeza == TAM_COLA){
      icabeza = 0;
    }
  }
}
```

La cola se construye a partir de un vector de un tamaño suficientemente grande como para que en el caso más desfavorable no se llene[12] y dos índices dentro de dicho vector: `icabeza` que apunta al elemento en el que se guardará el siguiente carácter a añadir a la cola e `icola` que apunta al elemento que se leerá (y se retirará) de la cola.[13] Si ambas variables son iguales la cola estará vacía y si `icabeza` vale `icola-1`, la cola estará llena.

En la rutina de atención a la interrupción del puerto serie se verifica en primer lugar si la cola está llena. El qué hacer en este caso depende de la aplicación. Como en la mayoría de sistemas empotrados no hay disponible una pantalla en la que imprimir un mensaje de error, hay que buscar soluciones alternativas. La más simple consiste en no hacer nada, pero entonces si el sistema falla no se puede saber qué ha ocasionado el fallo. También se puede encender un LED para indicar el error, pero entonces si existen varias fuentes de error; o se llena el aparato de LEDS o si se pone uno solo no se sabrá qué demonios ha pasado. En este último caso, para poder saber qué errores se han producido en el sistema se puede mantener un registro de errores en una memoria no volátil como una EEPROM o una RAM alimentada con baterías para su posterior descarga y análisis en un PC.

[12] Para ello habrá que estimar cuál es el tiempo máximo que la rutina de primer plano estará sin retirar caracteres de la cola y cuántos caracteres añadirá la rutina de interrupción mientras tanto.

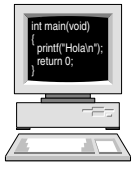
[13] Como todas estas variables se comparten entre la rutina de interrupción y la tarea de primer plano, han de ser globales; aunque se han declarado **`static`** para que sólo sean visibles dentro de este módulo.

Si la cola no está llena se escribirá el dato (leído del *buffer* de recepción del puerto serie) y se avanzará el índice `icabeza`, haciendo que vuelva a cero si se llega al final del vector. De esta forma cuando se llega al final del vector se continua por el principio. Ésta última característica hace que a este tipo de estructura de datos se le denomine comúnmente *buffer* circular.

Como se ha mencionado antes, la tarea de primer plano ha de retirar los caracteres que ha depositado la rutina de interrupción en la cola, tal como se muestra en el siguiente listado:

```
#define TAM_MENS 100
void ProcesaSerie()
{
  static char mensaje[TAM_MENS];
  static uint8 indice=0;
  if(icola != icabeza){ /* Hay datos nuevos */
    mensaje[indice] = cola[icola];
    icola++;
    if(icola == TAM_COLA){
      icola=0;
    }
    if(mensaje[indice] == '\n'){
      mensaje[indice] = '\0';  /* Se termina la cadena */
      ProcesaMensaje(mensaje); /* y se procesa */
      indice = 0;
    }else{
      indice++;
    }
  }
}
```

Nótese que cada vez que se ejecuta la tarea de primer plano se retira un carácter de la cola si éste está disponible y se copia en una cadena de caracteres donde se almacena el mensaje recibido. Cuando se recibe el retorno de carro se pasa a procesar el mensaje y una vez procesado se vuelve a repetir el proceso.



Realice los ejercicios 3 y 4.

Tenga en cuenta que al igual que la rutina de interrupción, el índice `icola` se incrementa cada vez que se retira un carácter de la cola, volviendo a 0 cuando se llega al final del vector.

Por último, destacar que el código mostrado sólo es válido en el caso de que exista una sola rutina de interrupción que inserta datos en la cola y una tarea de primer plano que los retira. Para casos en los que existen varias rutinas de atención a interrupción que insertan datos en la cola, es necesario proteger las zonas críticas de dicha inserción para evitar problemas de incoherencia de datos; los cuales se producirán cuando ocurra una interrupción de nivel superior mientras una interrupción de nivel inferior

está introduciendo datos en la cola.

3.4. Planificación

Según se ha visto antes, las rutinas de interrupción han de limitarse a atender el *hardware*, dejando todo el trabajo complejo para tareas de primer plano. El problema que origina esta estrategia es el cómo organizar la ejecución de estas tareas de primer plano. En los siguientes apartados se van a estudiar dos técnicas para conseguirlo.

3.4.1. Bucle de scan

El método más fácil consiste en usar la técnica del bucle de *scan* para gestionar la ejecución de estas tareas. En este caso, para mejorar la eficiencia del sistema o para evitar procesar el mismo dato varias veces, es necesario ejecutar las tareas de primer plano que dependen de datos proporcionados por tareas de interrupción sólo cuando se generen nuevos datos. Para ello, si la comunicación de datos es por medio de una cola FIFO, lo único que hay que hacer es verificar si hay datos nuevos antes de ejecutar la tarea de primer plano, tal como se ha hecho en el ejemplo anterior. Si el método de comunicación es por medio de variables compartidas o doble *buffer* será necesario usar una bandera para anunciar cuando hay un dato nuevo. Esta bandera se pondrá a 1 en la tarea de interrupción y a cero en la tarea de primer plano, tal como se muestra en el ejemplo siguiente.

```
static uint8 bandera=0;
__declspec(interrupt) IntPIT0(void)
{
  /*Atención al hardware y bla bla bla*/
  bandera = 1;
}
void ProcesaTiempo()
{
  if(bandera){
    /*ProcesoComplicado*/
    bandera = 0;
  }
}
```

En este ejemplo se ha supuesto que tenemos que realizar un proceso complejo disparado por una interrupción del temporizador 0. La rutina de interrupción `IntPIT0()` se limitará a reiniciar el temporizador y a incrementar una cuenta del tiempo. Para indicarle a la rutina de primer plano que tiene que ejecutarse activa la `bandera`. El resto del trabajo (representado por `/*ProcesoComplicado*/`) se realiza en la tarea de primer plano

`ProcesaTiempo()`, la cual sólo realiza este trabajo si la `bandera` está activa. Obviamente es muy importante no olvidarse de desactivar la `bandera` para no volver a ejecutar el */*ProcesoComplicado*/*. Nótese también que la variable `bandera` sólo se usará dentro de este módulo, por lo que se ha declarado como **`static`** para hacerla invisible al resto de módulos.

En ciertas situaciones han de ejecutarse varias tareas de primer plano cada vez que se ejecute una interrupción. En este caso, una alternativa es usar una bandera para cada tarea:

```
static uint8 bandera_p=0, bandera_i=0;
__declspec(interrupt) IntPIT0(void)
{
  bla bla bla;
  bandera_p = 1;
  bandera_i = 1;
}
void ProcesaTiempo()
{
  if(bandera_p){
    /*ProcesoComplicado*/
    bandera_p = 0;
  }
}
void ImprimeTiempo()
{
  if(bandera_i){
    /*Se imprime el tiempo*/
    bandera_i = 0;
  }
}

---------- Módulo main.c ----------
int main(void)
{
  ...
  while(1){
    ProcesaTiempo();
    ImprimeTiempo();

    ProcesaOtraCosa();
    ...
  }
}
```

En la parte superior del ejemplo (hasta la línea `--- Módulo main.c ---`) se muestra parte del módulo temporizador, que estará almacenado en el archivo `temporizador.c`, mientras que en la parte inferior se muestra el módulo del bucle de *scan*, almacenado en el archivo `main.c`. Como se puede apreciar, desde el bucle de *scan* se están ejecutando continuamente las tareas `ProcesaTiempo` e `ImprimeTiempo`, aunque éstas sólo harán algo útil si la rutina de interrupción se ha ejecutado y ha puesto las banderas a 1.

Otra alternativa es gestionar la bandera desde el bucle de *scan* y llamar desde allí a ambas tareas, tal como se muestra a continuación:

```
uint8 bandera=0;
__declspec(interrupt) IntPIT0(void)
{
  bla bla bla;
  bandera = 1;
}

---------- Módulo main.c ----------
int main(void)
{
  extern uint8 bandera;
  ...
  while(1){
    if(bandera){
      ProcesaTiempo();
      ImprimeTiempo();
      bandera = 0;
    }
    ProcesaOtraCosa();
    YOtraMas();
  }
}
```

En este caso el programa es más eficiente, al usarse sólo una bandera y sólo realizarse la llamada a las funciones cuando realmente es necesario. El inconveniente es que el bucle de *scan* es menos elegante que en el caso anterior.

Además, en este caso la bandera tendrá que compartirse entre el módulo del bucle de *scan* (`main.c`) y el módulo del temporizador. Por ello ahora la variable global `bandera` ha de ser visible también fuera del módulo del temporizador, por lo que **`no`** se ha declarado como **`static`**. Por otro lado, para poder usar en un módulo una variable global definida en otro módulo distinto, dicha variable ha de declararse como **`extern`**, tal como se ha hecho en el módulo del bucle de *scan* `main.c`.

El elegir una u otra técnica dependerá de los recursos que dispongamos en el sistema. Si el sistema dispone de muy poca memoria o las prestaciones del microprocesador son muy pobres, podría ser mejor usar la segunda opción. Sin embargo, es poco probable que un sistema esté tan ajustado, por lo que será siempre mejor verificar las banderas dentro de la función, ya que así se encapsulan mejor los datos al no ser necesario exportar las banderas fuera del módulo.

El método de planificación propuesto para las tareas de primer plano tiene como única ventaja su simplicidad. Lo demás son inconvenientes:

- Se pierde tiempo comprobando las banderas, aunque la verdad es que el tiempo perdido será despreciable.
- La latencia de las tareas de primer plano es igual al tiempo de ejecución del bucle de *scan*. Esto puede hacer que este método sea inservible si existen tareas que necesitan una latencia menor que el tiempo máximo de ejecución del bucle de *scan*.
- No se pueden tener en cuenta las prioridades. Si existen tareas más prioritarias que otras, no pueden ejecutarse antes que las menos prioritarias. Se pueden hacer algunas chapuzas como las mostradas en el capítulo 1, pero que no dejan de ser eso, CHAPUZAS.

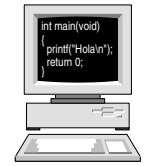
3.4.2. Planificación mediante cola de funciones

Si el método anterior no consigue planificar adecuadamente las tareas de primer plano, principalmente por problemas de latencia; una solución intermedia, sin necesidad de recurrir a un sistema operativo en tiempo real, es usar un planificador mediante cola de funciones. Este planificador usa una cola similar a la discutida para compartir datos entre tareas (pag. 70). Sin embargo, en lugar de almacenar datos, la cola se usa para almacenar los punteros a las tareas de primer plano que hay que ejecutar.

El mecanismo consiste en que las rutinas de interrupción introducen en la cola los punteros a sus funciones de primer plano y en el bucle de *scan* se retiran estos punteros de la cola y se llama a las funciones a las que apuntan estos punteros. Obviamente, cuando la cola esté vacía no se llamará a nadie y se ejecutarán las tareas de primer plano no asociadas a interrupciones.

Si no se incluye ningún sistema de prioridad, las rutinas de primer plano se ejecutan en el mismo orden en el que se producen las interrupciones a las que están asociadas, lo cual puede ser apropiado en muchos casos. En este caso la latencia máxima será igual al tiempo de ejecución de la tarea de primer plano no asociada a interrupción más larga, más el tiempo de ejecución de todas las tareas asociadas a interrupción que hayan podido activarse durante este tiempo. Si esta latencia no es aceptable, será

necesario establecer un mecanismo de prioridad, por lo que en este caso la latencia será igual al tiempo de ejecución de la tarea de primer plano más larga (más el tiempo de ejecución de todas las rutinas de atención a interrupción que puedan producirse en ese tiempo).



Realice el ejercicio 5.

Módulo de gestión de punteros a funciones

Para gestionar mejor la cola de punteros a funciones, es muy útil crear un módulo adicional encargado de ello. El módulo consistirá en una cola para almacenar los punteros y dos funciones, una para introducir un puntero a función en la cola y otra para extraerlo:

```
#define TAM_COLA 150

typedef void (*PUNT_FUN)(void);
static PUNT_FUN vfun[TAM_COLA];

static int icab = 0; /* índice para añadir */
static int icol = 0; /* índice para leer */

void Encola(PUNT_FUN pfun)
{
  if( (icab+1 == icol) ||
      (icab+1 == TAM_COLA && icol == 0)){
    /* La cola está llena */
  }else{
    vfun[icab]=pfun; /* Introduce la función en la cola */
    icab++;
    if(icab == TAM_COLA){
      icab = 0;
    }
  }
}

PUNT_FUN DesEncola(void)
{
  PUNT_FUN ret;

  if(icol != icab){ /* Hay funciones nuevas */
    ret = vfun[icol];
    icol++;
    if(icol == TAM_COLA){
      icol=0;
    }
  }else{ /* No hay que llamar a nadie */
```

```
    ret = (PUNT_FUN) NULL;
  }
  return ret;
}
```

El módulo consta en primer lugar de la definición de la cola de punteros a funciones, la cual se realiza usando una definición de tipos para que sea más clara y cómoda. Así, lo primero que se hace es definir el tipo `PUNT_FUN` que será un puntero a una función. A continuación se crea un vector de punteros a funciones y dos índices para gestionar la cola.

Para añadir una función a la cola de una forma más elegante, se ha escrito la función **void** `Encola(PUNT_FUN pfun)`, la cual introduce el puntero a función `pfun` en la cola, siempre y cuando ésta no esté llena. Como puede observarse, el proceso es idéntico al seguido en la página 70 para insertar caracteres dentro de la rutina de interrupción, sólo que ahora se ha creado una función para gestionar el proceso.

Para retirar los punteros a funciones de la cola se sigue un proceso análogo al seguido en la página 72, salvo que ahora se realiza el proceso en la función `PUNT_FUN DesEncola(void)`. La función verifica si hay algún puntero en la cola y en caso afirmativo lo devuelve. Si la cola está vacía devuelve un NULL para indicarlo.

Nótese que el manejo del puntero a la función en ambas funciones es idéntico al manejo de una variable de tipo carácter realizado en la cola para almacenar los caracteres del puerto serie mostrado en las páginas 70 y 72. Lo único que ha cambiado es el tipo de la variable.

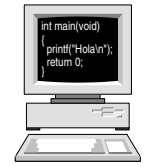
El uso de este módulo por parte de las rutinas de interrupción y del bucle de *scan* se muestra en el siguiente ejemplo:

```
---------- Módulo temporizador.c ----------

#include "ColaFun.h"

__declspec(interrupt) IntPIT0(void)
{
  ...
  Encola(ProcesaTiempo);
}

---------- Módulo main.c ----------

#include "ColaFun.h"

int main(void)
{
  PUNT_FUN pfun;
```

```
  ...
  while(1){
    pfun = DesEncola();
    if(pfun != NULL){
      (*pfun)();
    }
    TareaNoAsociadaAInter();
    OtraTareaNoAsocAInter();
    ...
  }
}
```

En la parte superior del listado se muestra el módulo de la interrupción de tiempo. Como se puede apreciar, la rutina de interrupción, después de atender al *hardware* y de realizar sus cometidos, como por ejemplo actualizar la hora; llamará a la función `Encola` para añadir a la cola el puntero a su función asociada `ProcesaTiempo`. Nótese que el nombre de la función es traducida por el compilador como su dirección, al igual que ocurre con los nombres de los vectores y matrices.

En la parte inferior del listado se muestra el bucle de *scan* del sistema. En él se llama a la función `DesEncola()` para obtener el puntero de la siguiente función a ejecutar. Como recordará, la función `DesEncola()` devuelve un NULL si la cola está vacía, por lo que antes de ejecutar la función a la que apunta el puntero `pfun` hay que verificar que este no sea nulo.[14]



Realice los ejercicios 6 y 7.

Planificación mediante cola de funciones con prioridades

Con el método mostrado en la sección anterior, las rutinas de de primer plano asociadas a interrupciones se ejecutan en el mismo orden en el que se producen sus interrupciones correspondientes. Esto puede dar lugar a problemas si hay alguna rutina que tiene una limitación temporal drástica y además su latencia es menor que la latencia máxima del sistema. En este caso, la solución es establecer un mecanismo de prioridad de forma que se ejecuten en primer lugar las tareas de primer plano con limitaciones temporales más rígidas. Para ello existen dos opciones:

- Si hay pocos niveles de prioridad se puede mantener una cola para cada nivel. Así la función `Encola()` recibirá además del puntero a la función un valor de prioridad e insertará el puntero en la cola correspondiente. La función `DesEncola()` verificará en primer lugar si hay alguna función para ejecutar en la cola más prioritaria y en caso de que no la haya, verificará si hay alguna función en la siguiente cola y

[14] En algunos sistemas si se ejecuta la función almacenada en la dirección cero se provoca un reset del sistema.

así sucesivamente hasta llegar a la cola menos prioritaria. Obviamente, si existen más de dos o tres prioridades llenaremos el sistema de colas y complicaremos el código.

- Si hay muchos niveles, la opción más sensata es añadir un campo de prioridad en la cola de punteros. Para ello se creará una estructura que contenga el puntero a la función y la prioridad de dicha función, formándose una cola con estas estructuras. Para gestionar las prioridades, habrá que modificar la función `Encola` para que inserte las funciones de forma que se mantenga la cola ordenada. No obstante, en este caso esta función será mucho más compleja, lo cual no será una buena idea teniendo en cuenta que ha de ejecutarse dentro de una rutina de interrupción. Otra alternativa sería modificar la función `Desencola` para que extrajese las funciones según su nivel de prioridad. En cualquier caso, si se necesitan gestionar prioridades será mucho más apropiado usar un sistema operativo en tiempo real para realizar la planificación.

3.5. Ejercicios

1. Modifique la función `Enable` para que acepte como parámetro un nivel de interrupción, de forma que se habiliten sólo las interrupciones de nivel superior.

2. Razone si el siguiente código para usar el doble *buffer* es correcto. La rutina de atención a la interrupción es la mostrada en la página 68. Indique también si este código tiene alguna ventaja frente al mostrado en la página 68

```
uint32 TicksDesdeArr(void)
{
  tarea_usa_buf0 = !tarea_usa_buf0;
  if(tarea_usa_buf0){
    return buf_ticks[0];
  }else{
    return buf_ticks[1];
  }
}
```

3. Modifique el programa para el manejo de colas expuesto en las páginas 70 y 72 para almacenar la cola y sus índices en una estructura de datos.

4. En el programa para el manejo de colas expuesto en las páginas 70 y 72 ¿existe alguna zona de este programa que tenga que ser atómica?

Busque para ello alguna variable que se use a la vez en la rutina de interrupción y la tarea de primer plano.

5. En un sistema en tiempo real planificado con cola de funciones con prioridad se tienen las siguientes tareas:

   - 4 tareas de primer plano no asociadas a interrupción con tiempos de ejecución de 1, 4, 3 y 2 milisegundos
   - 3 tareas de primer plano asociadas a interrupción con tiempos de ejecución de 0.5, 1 y 1.5 milisegundos.
   - 3 rutinas de atención a interrupción con periodos de 1, 20 y 40 ms. Las tres rutinas se ejecutan en 20 microsegundos. Cada una de estas rutinas de interrupción disparan una tarea de primer plano: la interrupción de 1 ms de periodo dispara la tarea de 0.5 ms de duración, la interrupción de 20 ms dispara la tarea de 1 ms y la de 40 ms dispara la tercera tarea.

   Calcule la latencia de las tareas asociadas a cada una de las interrupciones.

6. En el listado de la página 78, indique la latencia de la tarea de primer plano asociada a la interrupción de tiempo. Indique también cómo modificaría el código para mejorarla.

7. Modifique el listado de la página 78 para que en cada iteración del bucle de *scan* se ejecuten todas las tareas asociadas a interrupción que estén pendientes de ejecución; las cuales estarán almacenadas en la cola.

CAPÍTULO 4

Sistemas operativos en tiempo real

En los capítulos anteriores se ha puesto de manifiesto que cuando la aplicación en tiempo real es relativamente compleja, es necesario recurrir a un sistema operativo en tiempo real que gestione la ejecución de las distintas tareas. En este capítulo se estudian las características de este tipo de sistemas, que son bastante distintas a las de los sistemas operativos convencionales. Se analizan también los distintos mecanismos de sincronización de tareas, de comunicación de datos y de gestión de tiempo que ofrecen estos sistemas operativos. Para ilustrar los ejemplos de este capítulo se va a usar el sistema operativo FreeRTOS [Barry, 2007]. Éste es un sistema operativo de código abierto y que está portado a numerosos microcontroladores. Además es muy simple, lo cual lo hace muy adecuado para este texto introductorio.

4.1. Introducción

Un sistema operativo en tiempo real (S.O.T.R) o *Real Time Operating System (R.T.O.S)* en ingles, es muy diferente a un sistema operativo convencional de tiempo compartido, como Linux, Mac OS X o Windows.

Un sistema operativo de tiempo compartido, nada más arrancar el ordenador toma el control de éste y luego ejecuta los programas de aplicación, cargándolos de disco según las órdenes de los usuarios. Por el contrario, un sistema operativo en tiempo real es más parecido a una librería de funciones que se enlazan con la aplicación, de forma que al arrancar el ordenador es la aplicación la que toma el control del ordenador e inicializa el sistema operativo, para luego pasarle el control. Esto permite eliminar las partes del sistema operativo que no se usen para ahorrar memoria, que suele estar limitada en los sistemas empotrados.

Otra característica importante de los sistemas operativos en tiempo real es que éstos no se protegen frente a errores de las aplicaciones. En un sistema en tiempo compartido, cada aplicación funciona en su espacio de memoria virtual y es constantemente vigilado, de forma que si intenta acceder a una zona de la memoria que no le pertenece, el sistema operativo

dejará de ejecutarlo para que no corrompa al resto del sistema. En la mayoría de los sistemas operativos en tiempo real esto no ocurre para simplificar el diseño y el hardware necesario.[1] Además, dado que los sistemas empotrados habitualmente sólo ejecutan una aplicación, si esta se cuelga ya da igual que se cuelgue todo el sistema con ella. La consecuencia de todo esto es que hay que ser aún más cuidadoso al diseñar los programas de tiempo real.

Por último, aunque en sistemas operativos de tiempo compartido el mercado ofrece pocas elecciones, en el caso de los sistemas operativos en tiempo real existe una amplia oferta, tanto de sistemas de software libre como propietarios. Algunos ejemplos son: FreeRTOS, μC/OS-II, RTAI, VxWorks, QNX, LynxOS, y un largo etc.

4.2. Tareas

El bloque básico de un programa basado en un sistema operativo de tiempo real es la tarea.

Una tarea no es más que una función en C, aunque esta función obviamente puede llamar a otras funciones. La única condición que ha de cumplir una función para convertirse en una tarea es que no termine nunca, es decir, ha de contener un bucle infinito en el que realiza sus acciones.

La tarea se inicializa mediante una llamada al sistema operativo en tiempo real, especificándose en dicha llamada la prioridad de la tarea, la memoria que necesita, la función que la implanta (denominada punto de entrada), etc. En la sección 4.3.2 se muestra cómo se crea una tarea.

Los sistemas operativos en tiempo real pueden ejecutar un número arbitrario de tareas, estando limitado dicho número sólo por la memoria disponible en el sistema. Lo que sí suele estar limitado es el número de prioridades disponibles. Por ejemplo en FreeRTOS el número máximo es configurable y se recomienda ponerlo lo más bajo posible, ya que cada nivel de prioridad que se añade consume memoria para almacenar las estructuras de datos de control para gestionarla. El número máximo por defecto es de 5. Dicho número puede cambiarse editando el archivo `FreeRTOSConfig.h` y el máximo está tan solo limitado por el tipo de dato **`unsigned long`** de la máquina. Por ejemplo, en la versión de FreeRTOS para ColdFire, la prioridad se almacena en un entero sin signo de 32 bits, con lo que el valor máximo es $2^{32} - 1$. Por el contrario, en μC/OS-II, aunque el número máximo de prioridades es configurable y por defecto es 32, el método usado para gestionar las prioridades limita este valor a 64. Además, en este sistema operativo no pueden existir dos tareas con la misma prioridad, por

[1] Para poder usar memoria virtual es necesario el uso de una unidad de manejo de memoria (MMU) que sólo está disponible en los microprocesadores de altas prestaciones (Pentium, Power PC), pero no en los microcontroladores usados habitualmente en sistemas empotrados.

lo que se limita el número máximo de tareas a 64. No obstante dicho valor es más que suficiente para el rango de aplicaciones para las que están pensados estos sistemas operativos. No olvide que los dos sistemas operativos mencionados están diseñados para sistemas empotrados basados en microcontroladores de bajo coste.

4.2.1. Estados de una tarea

Tal como se ilustra en la figura 4.1, cada tarea puede estar en uno de los siguientes tres estados:[2]

- **Ejecución (*Running*)**: El microprocesador la está ejecutando. Sólo puede haber una tarea en este estado.
- **Lista (*Ready*)**: La tarea tiene trabajo que hacer y está esperando a que el procesador esté disponible. Puede haber un número cualquiera de tareas en este estado.
- **Bloqueada (*Blocked*)**: No tiene nada que hacer en este momento. Está esperando algún suceso externo. Puede haber un número cualquiera de tareas en este estado.

4.3. El planificador

El planificador es la parte del sistema operativo en tiempo real que controla el estado de cada tarea y decide cuándo una tarea pasa al estado de ejecución. El principio que sigue para ello es bien simple: exceptuando las tareas que están bloqueadas, la tarea con mayor prioridad es la que estará en el estado de ejecución.

En los sistemas operativos de propósito general como Linux o Windows las tareas de menor prioridad reciben un poco de tiempo de CPU de vez en cuando para que sus usuarios no se desesperen, aunque haya tareas de mayor prioridad ejecutándose. Para ello sus planificadores ejecutan algoritmos más o menos elaborados. Por el contrario, los planificadores de los sistemas operativos en tiempo real son bastante más "brutos", por lo que si una tarea de mayor prioridad se apropia del procesador durante un largo periodo de tiempo, las tareas de menor prioridad tendrán que esperarse. Por tanto la elección de la prioridad ha de hacerse con un buen juicio.

[2]Esta figura es una simplificación de la realidad. La mayoría de los sistemas operativos incluyen estados adicionales, pero que son similares al estado "Bloqueada". Por ejemplo, en FreeRTOS las tareas pueden ponerse en el estado "Suspendida" mediante una llamada al sistema operativo. En este estado la tarea no se ejecuta nunca. La tarea estará en este estado hasta que otra tarea la despierte mediante otra llamada al sistema operativo. Entonces pasará al estado de "Lista".

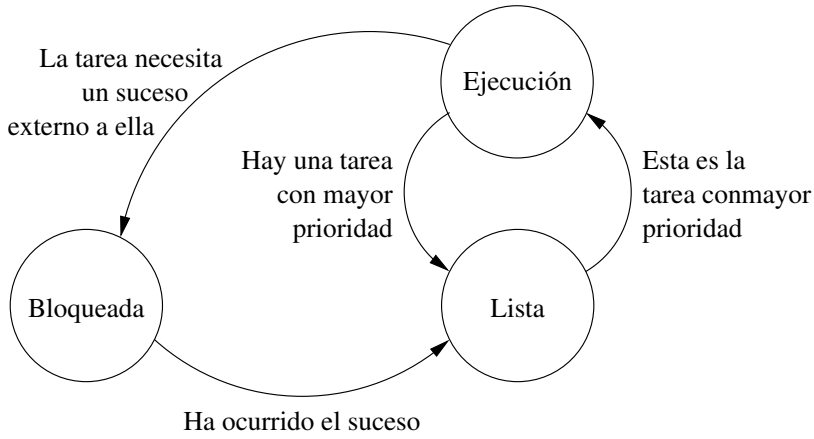


Figura 4.1: Estados de una tarea.

Por ejemplo, los cálculos tediosos deberán de realizarse en tareas con baja prioridad para no estropear la latencia del resto del sistema.

En la figura 4.1 se muestran las transiciones de estado de las tareas realizadas por el planificador. De la figura se desprenden las siguientes consecuencias:

- Una tarea sólo puede bloquearse cuando esté ejecutándose, ya que será entonces cuando llegue a un punto en el que necesite algún dato proporcionado por otra tarea (por ejemplo por una interrupción) o necesite esperar un determinado periodo de tiempo. Por ejemplo, si la tarea necesita leer datos de una cola, realizará una llamada al sistema operativo, que es el encargado de gestionar las colas. Si la cola tiene datos, la llamada al sistema operativo en tiempo real devolverá dichos datos, pero si la cola está vacía, el sistema operativo bloqueará la tarea, la cual no volverá al estado de "lista" hasta que lleguen datos a la cola.

- Para que una tarea bloqueada pase a lista, otra tarea debe despertarla. Siguiendo con el ejemplo anterior, la tarea que se bloqueó esperando datos de una cola no se despertará hasta que otra tarea deposite un dato en la cola.

- Una vez que una tarea está en el estado "lista", su paso al estado de "ejecución" depende sólo del planificador. Sólo cuando esta tarea sea la de mayor prioridad pasará a ejecutarse.

4.3.1. Preguntas típicas

Algunas preguntas que puede estar haciéndose son:

- ¿Cómo sabe el planificador que una tarea ha de bloquearse? Existen una serie de llamadas al sistema operativo en tiempo real que pueden bloquear a la tarea que las realiza. Por ejemplo, si la tarea necesita datos de una cola, al llamar a la función del sistema operativo que retira los datos, si éstos no están disponibles la tarea se bloqueará y no volverá a estar lista hasta que alguien (otra tarea) introduzca datos en la cola.

- ¿Qué pasa si todas las tareas están bloqueadas? El sistema operativo ejecuta un bucle sin fin (*idle task*). Obviamente en algún momento una interrupción tendrá que despertar alguna tarea, pues si no el sistema no será nada útil.

- ¿Qué pasa si dos tareas con la misma prioridad están "listas"? Depende del sistema operativo. Algunos sistemas operativos en tiempo real no permiten que existan dos tareas con la misma prioridad. Otros eligen una de las dos para ejecutarse y cuando la primera termine ejecutan la otra. Otros más sofisticados ejecutarán ambas tareas en paralelo, ejecutando un trozo de una durante un intervalo de tiempo (*time-slice*) y luego un trozo de otra.

- Si mientras una tarea está ejecutándose otra de mayor prioridad se desbloquea, ¿qué pasa? Si el sistema operativo en tiempo real es expropiativo, la tarea en ejecución pasa al estado "lista" y la tarea recién desbloqueada para al estado de ejecución. Si el sistema operativo es colaborativo, hasta que la tarea que está en ejecución no ceda el control (*yield*) no se empezará a ejecutar la tarea más prioritaria.

4.3.2. Ejemplo

Antes de continuar conviene ver un ejemplo, aunque muy simplificado, de cómo se programa un sistema de tiempo real usando un sistema operativo en tiempo real. El ejemplo elegido va a ser el autómata programable diseñado en el apéndice B, el cual incluye una serie de interruptores horarios. La gestión de los interruptores horarios en el diseño *background/foreground* del ejemplo mostrado en la sección B.3 se realiza mediante una interrupción y una tarea de primer plano asociada, la cual se llama en cada iteración del bucle de *scan*. La implantación de este sistema con un sistema operativo en tiempo real es bastante más simple, ya que la ejecución de la tarea `ProcesaIntHorarios` se ejecutará automáticamente por el sistema operativo cuando la rutina de interrupción cambie la hora. Para ello, la tarea consta de un bucle sin fin en el cual se bloquea (mediante una

llamada al sistema) a la espera de que la rutina de interrupción la despierte cuando haya cambiado la hora y por tanto esta tarea tenga algo útil que hacer. Cuando despierte hará su trabajo y volverá a bloquearse de nuevo.

Para procesar el programa del PLC se usa un bucle de *scan* que continuamente estará leyendo las entradas, ejecutando el programa del PLC y actualizando las salidas. Esta tarea se ejecutará cuando no haya nada más que hacer, por lo que se le ha dado la menor prioridad. Ambas tareas se muestran a continuación:

```
void ProcesaIntHorarios() /* Alta prioridad */
{
  while(1){
    /* Bloquea hasta que la rutina de interrupción la
       desbloquee al cambiar la hora */
    /*Actualiza los interruptores horarios*/
  }
}

void ProcesaBuclePLC() /* Baja prioridad */
{
  while(1){
    /* Lee entradas */
    /* Ejecuta programa PLC */
    /* Actualiza Salidas */
  }
}
```

El resto del programa lo forman la rutina de atención a la interrupción y el programa principal:

```
__declspec(interrupt) IntPIT0(void)
{
  /* Interacciona con el HW */
  /* Actualiza la hora */
  /* Desbloquea ProcesaIntHorarios() */
}
void main(void)
{
  InitHW();   /* Inicializa el HW: Timers, etc. */
  InitRTOS(); /* Inicializa el S.O.T.R. */

  /* Informa al S.O.T.R. de la existencia de las tareas*/
  xTaskCreate(ProcesaIntHorarios, 2);
  xTaskCreate(ProcesaBuclePLC, 1);

  /* Arranca el planificador. */
```

```
  vTaskStartScheduler();
}
```

La rutina de interrupción se limitará a interactuar con el *hardware*, a actualizar la hora y a desbloquear a la tarea `ProcesaIntHorarios` (mediante una llamada al sistema operativo).

El programa principal ahora se encarga de inicializar el *hardware* y el sistema operativo en tiempo real,[3] de informar a éste de la existencia de las tareas y de su prioridad y de arrancar el planificador.[4] En cuanto se arranque el planificador, éste ejecutará la tarea más prioritaria, que en este caso es `ProcesaIntHorarios`. Ésta, nada más empezar a ejecutarse, se bloqueará en espera de que se ejecute la rutina de interrupción. Al bloquearse, el planificador empezará a ejecutar la siguiente tarea de menor prioridad, que en este caso es la que queda: `ProcesaBuclePLC`. La ejecución de esta tarea continuará hasta que se produzca la interrupción del temporizador 0. En este momento, la rutina de interrupción desbloqueará a la tarea `ProcesaIntHorarios`. Cuando finaliza la rutina de interrupción, el planificador empezará a ejecutar la tarea `ProcesaIntHorarios`, que actualizará los interruptores horarios y volverá a bloquearse, repitiéndose el ciclo hasta el infinito (bueno, en realidad hasta que se apague el dispositivo).

4.4. Tareas y datos

Cada tarea tiene asociado su **contexto**, el cual está formado por:

- Los registros internos del microprocesador.
- El contador de programa.
- La pila, que se usa para la llamada a funciones y para almacenar las variables locales.

El resto de datos (variables globales) pueden ser compartidos con el resto de tareas y no forman parte del contexto de la tarea.

Para que el cambio de una tarea a otra sea transparente para el programador, el sistema operativo en tiempo real se encarga de guardar el contexto de la tarea que deja el estado de "ejecución" y cargar el de la nueva tarea que empieza a ejecutarse.

---

[3] Algunos sistemas operativos, como por ejemplo FreeRTOS, no necesitan este paso.

[4] En un sistema operativo en tiempo real de verdad estas llamadas son un poco más complejas, pues hay que suministrar más información acerca de la tarea, tal como se mostrará más adelante. En el ejemplo los nombres de las llamadas al sistema operativo son los correspondientes a FreeRTOS.

4.4.1. Datos compartidos. Funciones reentrantes

En los sistemas *Foreground/Background* sólo hay que tomar precauciones para evitar problemas de coherencia de datos cuando éstos se comparten entre tareas de primer plano e interrupciones (segundo plano). Sin embargo, en un sistema operativo en tiempo real se pueden ejecutar varias tareas en paralelo, por lo que si éstas comparten datos, hay que tomar también precauciones. En las secciones 4.5 y 4.6 se estudian los distintos métodos disponibles para ello en un sistema operativo en tiempo real. No obstante, antes de ello se va a mostrar que los datos compartidos no siempre están a la vista, tal como se ilustra en el siguiente ejemplo:

```
void Tarea1(void)
{
  ...
  CuentaErrores(9);
  ...
}

void Tarea2(void)
{
  ...
  CuentaErrores(27);
  ...
}

static int num_errores = 0;
void CuentaErrores(int nuevos_errores)
{
  num_errores += nuevos_errores;
}
```

Para ilustrar el problema que aparece en la función `CuentaErrores` conviene ver cómo será el código en ensamblador generado por la instrucción `num_errores += nuevos_errores`:[5]

```
    move.l D0, num_errores
    add.l  D0, nuevos_errores
    move.l num_errores, D0
```

Supóngase que se está ejecutando la tarea 1 y ésta ha realizado la llamada: `CuentaErrores(9)`. Supóngase también que la variable `num_errores` vale todavía 0. Supóngase ahora que justo después de ejecutarse la primera

[5]El código mostrado en el ejemplo es para la familia de microcontroladores ColdFire de Freescale. No obstante, los demás microcontroladores dispondrán de códigos en ensamblador muy similares.

instrucción `move.l` el planificador decide pasar a ejecutar la tarea 2. Esta tarea incrementará la variable `num_errores`, que como aún vale 0 pues la tarea 1 aún no la ha modificado, pasará a valer 27. Cuando el planificador retome la ejecución de la tarea 1, ésta continuará ejecutando por donde se quedó, es decir, ejecutará la instrucción `add.l`, la cual sumará al registro `D0` el valor de `nuevos_errores`, que en esta llamada es 9. El problema es que el registro `D0` contiene el valor anterior de `num_errores`, que era 0, en lugar del actualizado por la tarea 2 (27). Por tanto `D0` pasará a valer ahora 9 y se guardará este valor en la variable `num_errores`. En definitiva, es como si no se hubiese llamado a la función `CuentaErrores` desde la tarea 2.

Este problema se origina porque la función `CuentaErrores` no es reentrante; lo cual quiere decir que esta función no puede ser llamada desde dos tareas distintas. Para que una función sea reentrante y por tanto pueda llamarse desde varias tareas, ha de cumplir lo siguiente:

- Sólo debe usar variables de forma atómica, salvo que dichas variables estén almacenadas en la pila de la tarea que la llama.
- Sólo debe llamar a funciones reentrantes.
- Debe usar el *hardware* de forma atómica.

4.4.2. Otro repaso de C

Cuando se realizan programas en C convencionales, el programador se despreocupa por completo del modo de gestionar las variables. El programador las define y el compilador se encarga de todo lo demás. En cambio, cuando se diseñan programas en tiempo real en el que las tareas comparten datos, aunque el programador sigue definiendo las variables y el compilador sigue asignando la memoria para almacenarlas; el programador ha de ser consciente de dónde se almacena cada tipo de variable. Para exponer cómo se almacenan las variables en C lo mejor es usar un ejemplo:

```
int global;
static int global_estatica;
char *pcadena = "¿Dónde está esta cadena?"

void funcion(int arg, int *parg)
{
  static int local_estatica;
  int local;
  ...
}
```

la variable `global` está almacenada en una posición fija de memoria y al alcance de todo el mundo.

La variable `global_estatica` también está almacenada en una posición fija de memoria y está al alcance de todas las funciones dentro del módulo.

Tanto el puntero `pcadena` como la cadena `"¿Dónde está esta cadena?"` están almacenados en posiciones fijas de memoria. Además `pcadena`, al igual que `global`, es accesible desde todo el programa.

Los argumentos de la función `arg` y `parg` se almacenan en los registros o en la pila,[6] por lo que no existirán problemas de datos compartidos. No obstante `parg` no se sabe si apunta a una variable local almacenada en la pila o a una variable global compartida con otras tareas. Por tanto, una función de este estilo será reentrante o no en función de cómo sea llamada desde el resto de tareas.

La variable `local_estatica` se almacena en una posición fija de memoria y por tanto se compartirá por todas las tareas que llamen a esta función.

Por último, la variable `local` se almacena en la pila de la tarea que llama a la función, por lo que formará parte de su contexto y no se compartirá con las demás tareas que llamen a esta función.

Ejercicio

¿Es reentrante la siguiente función?

```
static int num_errores = 0;
void ImprimeErrores()
{
  if(num_errores > 10){
    num_errores -= 10;
    printf("Se han producido otros 10 errores más\n");
  }
}
```

Esta función obviamente no es reentrante, pues viola dos de las reglas mencionadas anteriormente:

1. `num_errores` es una variable no local y no se usa de forma atómica.

2. La función `printf` no se sabe si es reentrante, ya ha sido escrita por otro programador. Para estar seguros habría que consultar la documentación de la librería del compilador y ver si se afirma que la función es reentrante. Si no se dice nada es mejor suponer que no lo es, que es lo que suele ocurrir. Si la documentación no dice nada claro y

[6] El lugar en donde se almacenan los argumentos y el valor devuelto por una función depende del compilador y del procesador. Normalmente se usan los registros siempre que los argumentos quepan en ellos, ya que es mucho más eficiente. Si existen problemas de espacio, bien por el número de argumentos o bien por el tamaño de éstos, entonces se recurre a la pila.

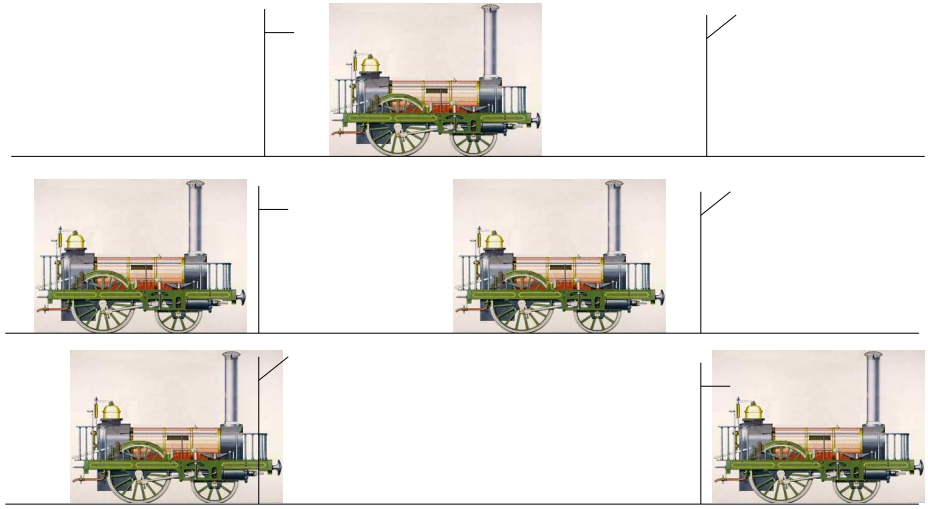


Figura 4.2: Semáforos.

se tiene acceso al código fuente, se tendría que leer dicho código para asegurarse de que no se viola ninguna de las reglas mencionadas anteriormente para decidir si es reentrante o no.

Realice el ejercicio 1

4.5. Semáforos

Desde los primeros tiempos del ferrocarril de descubrió que no era muy conveniente que un tren arrollase a otro. Por ello se inventaron unos dispositivos denominados semáforos, cuyo funcionamiento se ilustra en la figura 4.2. Cada vez que un tren entra en un tramo de vía, el semáforo que marca la entrada se baja. Si mientras el tren está dentro del tramo, llega otro tren a la entrada, el maquinista parará el tren. Cuando el primer tren salga del tramo de vía, el semáforo de la entrada volverá a subir, con lo que el segundo tren podrá entrar en el tramo de vía.

En un sistema en tiempo real, los semáforos se utilizan de la misma forma. Cada vez que una tarea tiene que usar un recurso compartido cuyo acceso está gobernado por un semáforo, lo primero que hace es comprobar que el semáforo esté levantado. Si lo está, el semáforo se bajará y la tarea podrá usar el recurso compartido, volviendo a levantar el semáforo cuando termine. Si por el contrario cuando se comprueba el semáforo éste está bajado, la tarea se bloquea hasta que el semáforo se suba.

A continuación se muestra un ejemplo para ilustrar el uso de un semáforo para proteger una zona crítica:

```
static int num_errores = 0;
void ImprimeErrores()
{
  xSemaphoreTake();
  if(num_errores > 10){
    num_errores -= 10;
    printf("Se han producido otros 10 errores más\n");
  }
  xSemaphoreGive();
}
```

En este ejemplo la llamada a `xSemaphoreTake` no retornará hasta que el semáforo esté libre. Cuando retorne se podrá usar el recurso compartido, que es la variable `num_errores` en este caso. Una vez que la función termina de usar el recurso compartido, llama a la función del sistema operativo `xSemaphoreGive` para levantar el semáforo e indicar así que el recurso compartido ya está libre.

Lamentablemente no existe un consenso entre los distintos autores de sistemas operativos en tiempo real en cuanto a la nomenclatura. Así, algunos usan la pareja Take–Give, otros Raise–Lower, otros Wait–Signal, otros Pend–Post, etc. En cualquier caso, lo único que varía es el nombre, ya que el funcionamiento es el mismo. En el ejemplo, se han usado las funciones de manejo de semáforos del sistema operativo FreeRTOS, `xSemaphoreTake()` y `xSemaphoreGive()`, aunque se han simplificado. En un sistema real las funciones de manejo de semáforos necesitan al menos un argumento que indique el semáforo usado, ya que en un sistema en tiempo real pueden existir varios semáforos: uno para cada zona crítica.

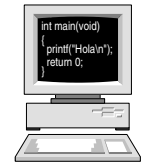
4.5.1. Ejemplo

Para ilustrar el funcionamiento de los semáforos, supóngase que en un sistema se necesita enviar la hora constantemente (en realidad sólo cuando cambie) por el puerto serie y el estado de 8 entradas digitales. Por tanto, el puerto serie se comparte ahora por dos tareas: `ImprimeHora()` para imprimir la hora y `EnviaEntradas()` para imprimir el estado de las entradas. Será por tanto necesario arbitrar el acceso al puerto serie por ambas tareas, por ejemplo mediante un semáforo. En primer lugar se muestra la tarea `ImprimeHora`:

```
void ImprimeHora(void)
{
  HORA copia_hora;
  char cadena[10];

  while(1){
```

```
    /* Se Bloquea hasta que llegue la interrupción
       de tiempo*/
    Disable(); /* Se copia la variable compartida
                  con la interrupción */
    copia_hora = hora_act;
    Enable();
    sprintf(cadena, "%02d:%02d:%02d\n", copia_hora.hora,
            copia_hora.min, copia_hora.seg);
    xSemaphoreTake();
    SeriePuts(cadena);
    xSemaphoreGive();
  }
}
```

Como se puede apreciar, la tarea consta de un bucle sin fin en el que se bloquea a la espera de que ocurra una interrupción de tiempo que se encargará de desbloquearla.[7] En ese momento, copiará la hora actual dentro de una zona crítica para evitar problemas de coherencia de datos. A continuación formateará la salida en una cadena para seguidamente enviarla por el puerto serie. Como se acaba de decir, el puerto serie está compartido por varias tareas, por lo que antes de usarlo hay que adquirir el semáforo. Por supuesto, cuando se termine de usar hay que liberar el semáforo para que el resto de tareas puedan usar el puerto serie en lugar de quedarse bloqueadas para siempre.



Realice los ejercicios 2 y 3

La tarea que imprime el estado de las entradas se muestra a continuación:

```
void EnviaEntradas(void)
{
  char cadena[100]; /* Guarda el mensaje a transmitir */
  uint8 entradas;
  static uint8 entradas_ant = 0;

  while(1){
    entradas = LeeEntradas();
    if(entradas_ant != entradas){ /* Sólo imprime si
                                     cambian las entradas */
      sprintf(cadena, "Entradas: %x\n", entradas);
      SemaphoreTake();
      SeriePuts(cadena);
      SemaphoreGive();
      entradas_ant = entradas;
    }
  }
```

[7] En la sección 4.5.3 se estudiará como hacerlo.

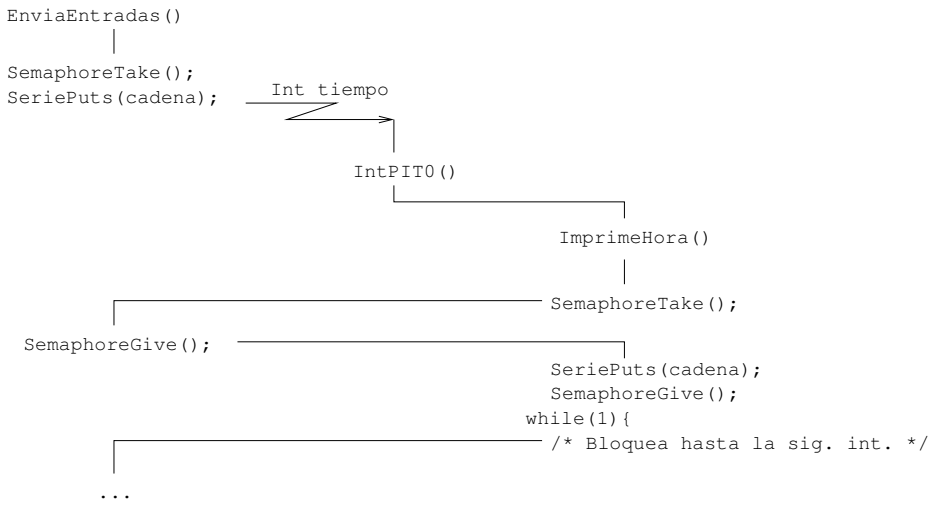


Figura 4.3: Ejemplo de ejecución con semáforos.

```
}
```

Esta tarea está continuamente leyendo el estado de las entradas mediante la función `LeeEntradas`. Esta función devuelve un byte con el estado de 8 entradas digitales. La tarea comprueba si ha cambiado alguna entrada y, en caso afirmativo, enviará su estado por el puerto serie. El acceso al puerto serie se realiza de la misma manera que en la tarea `ImprimeHora`: se formatea el mensaje en una cadena auxiliar y luego se envía la cadena por el puerto serie, usando el semáforo para arbitrar el acceso.

En la figura 4.3 se muestra cómo funciona la protección de una zona crítica con un semáforo. Se ha supuesto que mientras se está ejecutando `EnviaEntradas` y esta tarea está dentro de su zona crítica, se produce una interrupción de tiempo que despierta la tarea `ImprimeHora`. Cuando esta tarea intenta acceder al puerto serie para imprimir la hora, la llamada para pedir el semáforo la bloqueará y el planificador pasará a ejecutar la tarea `EnviaEntradas`. Cuando esta tarea libera el semáforo, el planificador le dará paso nuevamente a `ImprimeHora`, ya que al liberarse el semáforo esta tarea pasará al estado "lista" y tiene mayor prioridad que `EnviaEntradas`. La tarea `ImprimeHora` podrá ahora entrar en su zona crítica y enviar la hora por el puerto serie.

Ejemplo usando FreeRTOS

El ejemplo mostrado anteriormente se ha simplificado para no complicar la exposición. A continuación se muestra el mismo ejemplo pero usando el

sistema operativo en tiempo real FreeRTOS.[8] En primer lugar se muestra el programa principal con la inicialización del sistema:

```
#include "mcf5282.h"
#include "timer.h"
#include "serie.h"

/* Includes del Kernel. */
#include "FreeRTOS.h"
#include "semphr.h"
#include "task.h"

#define PRIO_IMP_HORA 2
#define PRIO_ENV_ENTR 1
#define TAM_PILA 1024

xSemaphoreHandle sem_serie;

void main(void)
{
  InitM5282Lite_ES(); /* Inicializa el Hardware del
                         microcontrolador */
  InitTimer();
  InitSerie();
  InitQueSeYo();

  /* Se inicializa el semáforo */
  vSemaphoreCreateBinary(sem_serie);

  /* Se crean las tareas */
  xTaskCreate(ImprimeHora,   "ImpHora", TAM_PILA, NULL,
              PRIO_IMP_HORA, NULL);
  xTaskCreate(EnviaEntradas, "EnvEntr", TAM_PILA, NULL,
              PRIO_ENV_ENTR, NULL);

  vTaskStartScheduler(); /* y por último se arranca el
                            planificador. */
}
```

Nótese que:

- Para que el compilador reconozca las funciones y las estructuras de datos del sistema operativo, es necesario incluir los archivos cabe-

[8] En el apéndice A se describen todas las llamadas al sistema operativo usadas en este texto.

cera `FreeRTOS.h` (núcleo), `semphr.h` (semáforos) y `task.h` (creación de tareas).

- El sistema operativo en tiempo real necesita una estructura para almacenar los datos de control de cada semáforo. Dicha estructura es del tipo `xSemaphoreHandle`. Como el semáforo se comparte por varias tareas se ha creado global.

- Cada tarea tiene una pila, creada automáticamente por el sistema operativo y cuyo tamaño viene dado por el tercer argumento de la llamada a `xTaskCreate`. Por simplicidad, en este ejemplo ambas tareas tienen pilas del mismo tamaño, pero pueden ser distintos si una tarea necesita un espacio mayor en la pila para almacenar variables locales.

- En primer lugar se inicializa el hardware, aunque no hay ninguna razón para hacerlo antes de crear las tareas, ya que éstas no se ejecutarán hasta que no se arranque el planificador del sistema operativo.

- En segundo lugar se ha inicializado el semáforo mediante una llamada a `vSemaphoreCreateBinary`. Es muy importante no olvidarse de este paso, pues si se intenta pedir un semáforo no inicializado el sistema no funcionará.

- En tercer lugar se crean las dos tareas. Para crear una tarea hay que darle al sistema operativo la dirección de la función (que en C se representa por su nombre), un nombre simbólico que se usa para depuración y el tamaño de la pila (en palabras). A continuación se puede pasar un puntero a los datos iniciales de la tarea, que como en este ejemplo no se usan se ha dejado a `NULL`.[9] El siguiente parámetro es la prioridad de la tarea y el último se usa para devolver una estructura de control de la tarea que sólo es útil si se quiere destruir la tarea durante el funcionamiento del sistema. Como en este ejemplo no es necesario se ha dejado también este parámetro a `NULL`.

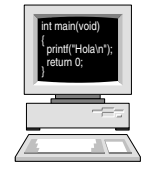
- El último paso consiste en arrancar el planificador mediante la llamada a la función `vTaskStartScheduler`. Esta función ya no devuelve el control, por lo que el programa no termina hasta que no se apague el dispositivo.

El código de la tarea `ImprimeHora` es similar al anterior:

[9] El segundo parámetro, inicializado a `NULL` en este ejemplo, es un puntero a los argumentos de entrada de la tarea, en caso de que ésta los necesite. Nótese que estos argumentos sólo se le pasarán cuando se arranque, no cada vez que se ejecute como consecuencia de un cambio de contexto. Estos argumentos pueden usarse para enviar a la tarea valores iniciales. Por ejemplo a la tarea `EnviaEntradas` se le podría enviar un argumento para indicarle qué puerto serie tiene que usar.

```
ImprimeHora()
{
  HORA copia_hora;
  char cadena[10];
  extern xSemaphoreHandle sem_serie;

  while(1){
    /* Se bloquea hasta que llegue la interrupción
       de tiempo*/
    DisableInt();
    copia_hora = hora_act;
    EnableInt();
    sprintf(cadena, "%02d:%02d:%02d\n", copia_hora.hora,
            copia_hora.min, copia_hora.seg);
    if(xSemaphoreTake(sem_serie, (portTickType) 1000 )
        == pdTRUE ){
      SeriePuts(cadena); /* Se tiene el semáforo: se puede
                            acceder al puerto serie */
      xSemaphoreGive(sem_serie); /*Se suelta el semáforo*/
    }else{
      /* Después de 1000 ticks no se ha obtenido el
         semáforo. Se podría dar un aviso o
         simplemente no hacer nada como en este caso */
    }
  }
}
```

Lo único que ha cambiado respecto a la versión anterior son las llamadas para gestionar el semáforo. Ahora para pedir dicho semáforo se usa la función `xSemaphoreTake`, la cual recibe la estructura encargada de almacenar la información relativa al semáforo usado para proteger el recurso. El segundo parámetro de `xSemaphoreTake` es un *timeout* que permite "saltarse" el semáforo si éste tarda más de un determinado tiempo en obtenerse. Este tiempo se mide en *ticks* de reloj.[10] Por ello es imprescindible verificar el valor devuelto por la función, que indicará si se ha obtenido el semáforo o no. Sólo si se ha obtenido el semáforo se podrá usar el recurso compartido. En caso contrario se puede generar un aviso, hacer algo alternativo o esperar otra vez hasta que el recurso quede libre.



Realice el ejercicio 5

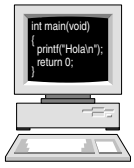
También se ha supuesto en este ejemplo que la tarea `ImprimeHora` está situada en un módulo distinto al de la función `main`. Por ello la estructura de control del semáforo, `sem_serie`, se ha declarado **`extern`** para indicar

[10] El tipo de este argumento es entero, aunque su tamaño depende del procesador. Por ello FreeRTOS define un tipo derivado denominado `portTickType` para mejorar la portabilidad. En la llamada a la función se ha usado un *cast* para evitar avisos del compilador.

que está definida en otro archivo.

Por último se muestra la tarea EnviaEntradas, la cual se ha modificado de una forma muy similar a ImprimeHora:

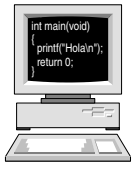
```
void EnviaEntradas(void)
{
  char cadena[100]; /* Guarda el mensaje a transmitir */
  uint8 entradas;
  static uint8 entradas_ant = 0;
  extern xSemaphoreHandle sem_serie;

  while(1){
    entradas = LeeEntradas();
    if(entradas_ant != entradas){ /* Sólo imprime si
                                     cambian las entradas */
      sprintf(cadena, "Entradas: %x\n", entradas);
      if(xSemaphoreTake(sem_serie, (portTickType) 1000)
          == pdTRUE ){
        /* Se tiene el semáforo: se puede acceder al
           puerto serie */
        SeriePuts(cadena);
         /* Se suelta el semáforo */
        xSemaphoreGive(sem_serie);
      }else{
        /* Después de 1000 ticks no se ha obtenido el
           semáforo. Se podría dar un aviso o
           simplemente no hacer nada como en este caso */
      }
      entradas_ant = entradas;
    }
  }
}
```



Realice el ejercicio 6

4.5.2. Múltiples semáforos

No hay nada que impida la existencia de varios semáforos en el sistema; es más, es lo normal en la mayoría de las aplicaciones, ya que cada semáforo puede proteger un sólo recurso compartido. En estos casos es tarea del programador el usar el semáforo correspondiente antes de usar un recurso compartido, ya que el sistema operativo no puede saber qué semáforos están protegiendo cada uno de los distintos recursos compartidos.



Una técnica para evitar problemas al usar distintos semáforos es usar una sola función para el manejo de cada recurso compartido. De esta forma se pedirá y se soltará el semáforo sólo en esa función. En el ejercicio 7 se discute esta técnica.

Realice el ejercicio 7

4.5.3. Semáforos usados para sincronizar tareas

Aunque la finalidad principal de los semáforos es proteger zonas críticas de código, también pueden usarse para sincronizar dos tareas entre sí, o una tarea con una rutina de atención a interrupción. Lo mejor para aclarar este concepto es mostrar un ejemplo para sincronizar la tarea `ImprimeHora` con la rutina de atención a la interrupción del temporizador PIT0. En el código de esta tarea mostrado anteriormente existía un comentario al principio de la tarea diciendo que ésta se bloqueaba a la espera de la interrupción de tiempo. Este bloqueo se va a realizar precisamente con un semáforo, al que se denominará `sem_hora`. el código de la tarea queda por tanto de la siguiente manera:

```
ImprimeHora()
{
  HORA copia_hora;
  char cadena[10];
  extern xSemaphoreHandle sem_serie;
  extern xSemaphoreHandle sem_hora;

  while(1){
    if(xSemaphoreTake(sem_hora, (portTickType) 2000 )
        == pdTRUE ){
      /* Ha saltado una nueva interrupción de tiempo */
      DisableInt();
      copia_hora = hora_act;
      EnableInt();
      sprintf(cadena, "%02d:%02d:%02d\n", copia_hora.hora,
              copia_hora.min, copia_hora.seg);
      if(xSemaphoreTake(sem_serie, (portTickType) 1000)
          == pdTRUE ){
              /* El puerto serie está libre */
        SeriePuts(cadena);
        xSemaphoreGive(sem_serie);
      }else{
        /* Después de 1000 ticks no se ha obtenido
           el semáforo. */
      }
    }
  }
}
```

Nótese que si no se obtiene el semáforo después del tiempo de espera de 2000 ticks del reloj del sistema operativo, no se ejecutará nada del bucle, volviendo a bloquearse de nuevo la tarea hasta que se libere el semáforo `sem_hora`.

La rutina de atención a la interrupción del temporizador queda ahora de la siguiente manera:

```
__declspec(interrupt) IntPIT0(void)
{
  extern xSemaphoreHandle sem_hora;

  /* Interacciona con el HW */
  /* Actualiza la hora */
  /* Desbloquea ImprimeHora() */
  xSemaphoreGiveFromISR(sem_hora, pdFALSE);
}
```

Como se puede observar, la rutina de atención a interrupción realiza la interacción con el hardware, actualiza la hora y, antes de terminar, libera el semáforo `sem_hora` para "desbloquear" a la tarea `ImprimeHora`. También habrá notado que la liberación del semáforo desde la rutina de atención a la interrupción se realiza de forma distinta. En la sección 4.8 se expone el porqué.

Las ventajas principales de este mecanismo frente al bucle de *scan* usando una bandera para activar la tarea, tal como se discutió en la sección 3.4.1, son dos:

1. La latencia de la tarea será mucho menor ahora, ya que en cuanto la interrupción libera el semáforo, la tarea de primer plano se ejecutará; salvo que haya alguna otra tarea más prioritaria.[11]

2. Además, no se pierden ciclos de procesador comprobando la bandera. Ahora, si una tarea tiene que esperar a que se ejecute una rutina de atención a interrupción, ésta se bloqueará esperando el semáforo y por tanto no ocupará la CPU hasta que la rutina de atención a interrupción libere el semáforo. Será en ese momento, y no antes, cuando la tarea se ejecute. Si se implanta el mismo sistema con un bucle de *scan* y una bandera para controlar la ejecución de la tarea, ésta se ejecutará en cada ciclo de *scan* sólo para comprobar que la bandera que tenía que activar la rutina de atención a interrupción aún vale cero, con lo cual no tiene nada que hacer. Obviamente, ese tiempo de CPU lo pueden usar las tareas que tengan trabajo útil que hacer. De todas formas, no hay que olvidar que el sistema operativo también presenta una pequeña sobrecarga, aunque normalmente ésta no es muy significativa.

Por último, se muestra el código del programa principal.

[11] En el código mostrado esto no es del todo cierto. En la sección 4.8 se muestra cómo realizarlo correctamente para que la latencia de la tarea se primer plano sea mínima.

```
#include "mcf5282.h"
#include "timer.h"
#include "serie.h"

/* Includes del Kernel. */
#include "FreeRTOS.h"
#include "semphr.h"
#include "task.h"

#define PRIO_IMP_HORA 2
#define PRIO_ENV_ENTR 1
#define TAM_PILA 1024

xSemaphoreHandle sem_serie;
xSemaphoreHandle sem_hora;

void main(void)
{
  InitM5282Lite_ES(); /* Inicializa el Hardware del
                         microcontrolador */
  InitTimer();
  InitSerie();
  InitQueSeYo();

  /* Se inicializan los semáforos */
  vSemaphoreCreateBinary(sem_serie);
  vSemaphoreCreateBinary(sem_hora);
  xSemaphoreTake(sem_hora, (portTickType) 1);
  /* Se pide el semáforo para marcarlo como ocupado
     hasta que salte la primera interrupción */

  /* Se crean las tareas */
  xTaskCreate(ImprimeHora,   "ImpHora", TAM_PILA, NULL,
              PRIO_IMP_HORA, NULL);
  xTaskCreate(EnviaEntradas, "EnvEntr", TAM_PILA, NULL,
              PRIO_ENV_ENTR, NULL);

  vTaskStartScheduler(); /* y por último se arranca el
                            planificador. */
}
```

Cabe destacar tan solo la definición del nuevo semáforo sem_hora y su inicialización. La diferencia de este semáforo respecto a sem_serie es que, al ser un semáforo usado para sincronización, se ha pedido con la llamada a xSemaphoreTake para marcarlo inicialmente como ocupado. Así,

cuando se ejecute la tarea `ImprimeHora`, ésta se bloqueará a la espera de la liberación del semáforo, de lo cual se encargará la rutina de interrupción del *timer* PIT0. Si no se pide el semáforo en la inicialización, la tarea `ImprimeHora` puede ejecutarse una vez sin necesidad de que haya ocurrido la primera interrupción. En este ejemplo no es grave, pues se imprimiría la hora 00:00:00. No obstante, en otras situaciones puede ser necesario que la tarea sólo se ejecute después de que ocurra la primera interrupción.

4.5.4. Problemas con los semáforos

Como se ha visto en esta sección, los semáforos son una herramienta muy potente para proteger recursos compartidos y sincronizar tareas. No obstante, al no ser algo automático, sino que el programador es responsable de coger y soltar el semáforo, se pueden producir errores en su uso. Los más típicos son:

- Se puede olvidar tomar el semáforo. En este caso se estará accediendo a un recurso compartido sin ninguna protección, por lo que si otra tarea está accediendo en ese momento se producirá un error.
- Se puede tomar el semáforo equivocado. Entonces también se usará un recurso compartido sin protección.
- Se puede olvidar soltar el semáforo. En este caso ya ninguna tarea podrá usar el recurso protegido por el semáforo.
- Se puede tener el semáforo demasiado tiempo. Esto hará aumentar la latencia del resto de tareas que usen el recurso protegido por el semáforo.
- Se puede producir una inversión de prioridad. En este caso una tarea de menor prioridad puede impedir la ejecución de una tarea de mayor prioridad, tal como se muestra en el siguiente apartado.

4.5.5. Inversión de prioridad

Como se acaba de decir, el uso de semáforos puede producir una inversión de prioridad, que consiste en que una tarea de menor prioridad impide la ejecución de una tarea de mayor prioridad.

Supóngase que en un sistema hay tres tareas a las que se denomina A, B y C. La tarea A tiene mayor prioridad que la B y ésta a su vez mayor que la C. La figura 4.4 ilustra la ejecución de las tareas. El eje x representa el tiempo y el eje y la tarea qué se está ejecutando en cada momento. Se ha supuesto que inicialmente las tareas A y B están bloqueadas y se está ejecutando la tarea C. Durante su ejecución, la tarea C toma un semáforo que protege un recurso compartido que también usa la tarea A. A continuación

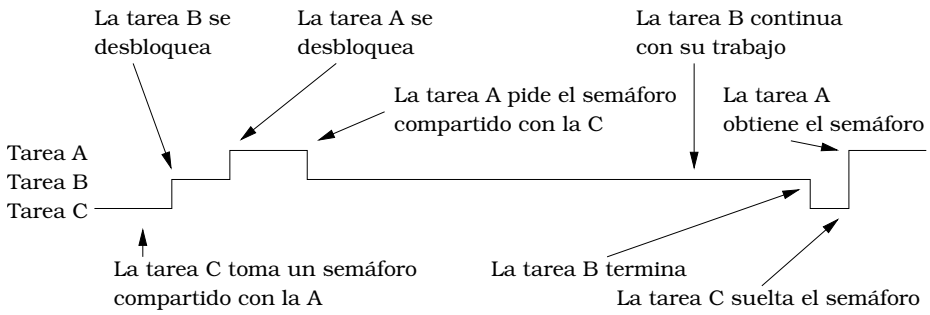


Figura 4.4: Inversión de prioridad

se ha supuesto que se desbloquea la tarea B y, como tiene mayor prioridad que la C, el planificador realizará un cambio de contexto para pasar a ejecutar la tarea B. A continuación la tarea A se desbloquea también y el planificador pasará a ejecutarla. El problema se produce al pedir la tarea A el semáforo que protege el recurso que comparte con la tarea C. Como el semáforo está ocupado, la tarea A ha de bloquearse hasta que éste se libere. Como puede observar en la figura, una vez que se bloquea la tarea A, el planificador busca la tarea más prioritaria que está en estado de "lista para ejecución", la cual es la B. Por tanto se ejecutará la tarea B y hasta que ésta no termine no se ejecutará la tarea C. Sólo cuando la tarea C se ejecute se liberará el semáforo que mantiene a la tarea A bloqueada. Por tanto, la tarea B en realidad está impidiendo que se ejecute la tarea A que tiene mayor prioridad que ella. Esto es lo que se conoce como inversión de prioridad: la tarea B se ha ejecutado aunque la tarea A tenía que hacerlo.

Algunos sistemas operativos en tiempo real arreglan este problema haciendo que la tarea de baja prioridad herede la prioridad de la de alta prioridad cuando coge un semáforo compartido con ésta. A este método se le conoce como herencia de prioridad. A los semáforos que incluyen este método para evitar la inversión de prioridad se les denomina ***mutexes***.[12]

4.5.6. Abrazo mortal

Otro problema que puede ocurrir cuando se usan semáforos se ilustra a continuación. Como este problema se resuelve usando semáforos con *timeout* y el sistema operativo FreeRTOS sólo dispone de este tipo de se-

[12]El término *mutex* es una abreviatura de *Mutual Exclusion*. El término tiene un doble significado. Por un lado designa los algoritmos que evitan el uso simultáneo por dos tareas de un recurso compartido. Por otro lado, a los semáforos con herencia de prioridad se les denomina también *mutexes*, ya que éstos son el método más usado para garantizar la exclusión mutua.

máforos, se va a usar en el ejemplo otro sistema operativo en tiempo real denominado μC/OS-II [Labrosse, 2002]. Así además se puede ver que aunque el sistema operativo es distinto, el uso de semáforos es muy similar.

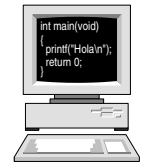
En el ejemplo se ha supuesto que dos tareas acceden a dos variables compartidas y que cada variable está protegida por un semáforo:

```
int a, b; /* Variables compartidas */

OS_EVENT *p_sem_a; /* semáforo para variable a */
OS_EVENT *p_sem_b; /* semáforo para variable b */

void Tarea1(void)
{
  OSSemPend(p_sem_a, WAIT_FOREVER);
  OSSemPend(p_sem_b, WAIT_FOREVER);
  a = b;
  OSSemPost(p_sem_b);
  OSSemPost(p_sem_a);
}

void Tarea2(void)
{
  OSSemPend(p_sem_b, WAIT_FOREVER);
  OSSemPend(p_sem_a, WAIT_FOREVER);
  b = a;
  OSSemPost(p_sem_a);
  OSSemPost(p_sem_b);
}
```

En el ejemplo no se ha mostrado el programa principal para simplificar la exposición, pero tendría una estructura similar a los mostrados para FreeRTOS. Dicho programa principal se limitará a inicializar el hardware, crear los semáforos y las tareas y, en último lugar, arrancar el planificador.

Como puede observar en el listado anterior, para pedir un semáforo en μC/OS-II se usa la llamada `OSSemPend`. El segundo argumento de esta llamada es un `timeout`, pero para simplificar el código, μC/OS-II permite que el `timeout` sea infinito. De esta forma no hace falta comprobar si realmente el semáforo está libre al retornar de la llamada a `OSSemPend`, tal como se ha realizado en los ejemplos con FreeRTOS. El problema de no usar `timeout` puede ser muy grave si se comete el error mostrado en el ejemplo. Supóngase que inicialmente ambos semáforos están libres. Supóngase también que se está ejecutando la `Tarea1` y que pide el semáforo `p_sem_a`. Como está libre, la llamada a `OSSemPend` retornará inmediatamente. Supóngase ahora que antes de que a la `Tarea1` le de tiempo a realizar la segunda llamada a `OSSemPend` se produce un cambio de contexto. Supóngase que este cam-

bio de contexto hace que comience a ejecutarse la Tarea2. Ésta pedirá en primer lugar el semáforo `p_sem_b`, el cual como está libre se le dará. A continuación pedirá el semáforo `p_sem_a`, pero como está cogido por la Tarea1, la Tarea2 se bloqueará a la espera de que dicho semáforo quede libre (la llamada a `OSSemPend` no retornará). Se producirá ahora un nuevo cambio de contexto, pues la Tarea2 no puede continuar. Cuando se ejecute la Tarea1, fruto de este cambio de contexto, ésta continuará la ejecución donde la dejó, y pedirá el semáforo `p_sem_b`. Como este semáforo está cogido por la Tarea2, la Tarea1 volverá a bloquearse. Como la Tarea1 está esperando el semáforo `p_sem_b` que sólo puede soltar la Tarea2 y la Tarea2 está esperando el semáforo `p_sem_a` que sólo puede soltar la Tarea1, ninguna de las tareas podrá ejecutarse más. Esto es lo que se conoce como un **abrazo mortal** (*Deadlock* en inglés).



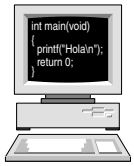
Realice los ejercicios 8 y 9

La forma de evitar que se produzca un abrazo mortal es pedir siempre los semáforos en el mismo orden. No obstante, en programas complejos es fácil equivocarse y pedirlos en distinto orden. Si se quiere minimizar el impacto de un abrazo mortal, se han de usar siempre semáforos con *timeout*. En este caso se produce un abrazo, pero éste no es mortal, pues la primera tarea en pedir el semáforo, cuando termine su *timeout*, lo soltará y permitirá a la otra realizar su tarea.

4.6. Métodos para proteger recursos compartidos

A modo de resumen, los métodos para proteger el acceso a recursos compartidos entre varias tareas son:

- Inhabilitar las interrupciones. Es el método más drástico, pues afecta a los tiempos de respuesta de todas las tareas e interrupciones. Sin embargo es el método más rápido (una o dos instrucciones de código máquina) y es la única manera de proteger datos compartidos con interrupciones. Sólo es válido cuando la zona crítica es muy pequeña para que las interrupciones no estén inhabilitadas mucho tiempo.
- Inhabilitar las conmutaciones de tareas. Es un método menos drástico que cortar las interrupciones, ya que sólo afectará a los tiempos de respuesta de las tareas de primer plano, que normalmente no son tan estrictos como los de las rutinas de interrupción. Para inhabilitar la conmutación de tareas basta con realizar una llamada al sistema operativo. Por ejemplo, en FreeRTOS basta con llamar a la función `vTaskSuspendAll`. Desde el momento en el que la tarea llama a esta función, ésta sólo será interrumpida por las rutinas de atención a interrupciones, pero no por otras tareas más prioritarias. Cuando la tarea termine de usar el recurso compartido ha de llamar a la función `xTaskResumeAll` para que el sistema operativo vuelva a conmutar tareas cuando sea necesario.



Realice el ejercicio 10

Este método requiere una sobrecarga un poco mayor que la inhabilitación de interrupciones, pero menor que el uso de semáforos. Dado que afecta a los tiempos de respuesta de todas las tareas, al igual que la inhabilitación de interrupciones sólo será recomendable si la zona crítica no es muy larga.

- Usar semáforos. La ventaja principal es que sólo afecta a las tareas que comparten el recurso. El inconveniente es que al ser un mecanismo más complejo, su gestión por parte del sistema operativo presenta una mayor carga al sistema. Por otro lado, como se ha mencionado en la sección anterior, si no se usan con cuidado pueden originar errores difíciles de detectar y corregir.

4.7. Colas para comunicar tareas

Los sistemas operativos en tiempo real también disponen de colas para permitir comunicar varias tareas entre sí. Normalmente se usan:

- Cuando se necesita un almacenamiento temporal para soportar ráfagas de datos.
- Cuando existen varios generadores de datos y un sólo consumidor y no se desea bloquear a los generadores a la espera de que el consumidor obtenga los datos.

El funcionamiento de las colas es similar al estudiado para los sistemas *Foreground/Background* en la sección 3.3.4; sólo que ahora la gestión la realiza el sistema operativo en lugar del programador. Las principales diferencias son:

- Las colas se crean mediante una la llamada al sistema. Esta llamada se encarga tanto de crear la memoria necesaria para albergar a todos los elementos de la cola, como de crear una estructura de control, a la que denominaremos “manejador”.
- El manejo de la cola, es decir, enviar y recibir datos, se realiza también exclusivamente mediante llamadas al sistema.
- En el sistema operativo FreeRTOS los elementos de la cola pueden ser de cualquier tipo, incluyendo estructuras de datos. En otros sistemas operativos, como por ejemplo en μC/OS-II, los elementos de la cola sólo pueden ser punteros. Con ello se consigue una mayor eficiencia, pero costa de complicarle la vida al programador y hacer el sistema más propenso a errores.
- Las funciones que envían y reciben datos de la cola se pueden bloquear cuando la cola esté llena o vacía, respectivamente. Esto es una

clara ventaja de los sistemas operativos en tiempo real, pues, tal como se ha visto en la sección 3.3.4, en los sistemas *Foreground/Background* es necesario ejecutar continuamente la tarea y verificar en ésta el estado de la cola para ver si hay datos nuevos o si hay hueco para enviar datos.

4.7.1. Gestión de colas en FreeRTOS

El sistema operativo FreeRTOS dispone de un manejo básico de colas. La interfaz se describe a continuación.

Creación de la cola en FreeRTOS

Al igual que en los semáforos, antes de usar una cola hay que crearla. De esta forma FreeRTOS asigna la memoria y las estructuras de datos necesarias para gestionar la cola. Para ello es necesario llamar a la función `xQueueCreate` antes de usar la cola. El lugar más apropiado para esta llamada es en la inicialización del sistema, al igual que con los semáforos.

El prototipo de `xQueueCreate` es:

```
xQueueHandle xQueueCreate(
                  unsigned portBASE_TYPE uxQueueLength,
                  unsigned portBASE_TYPE uxItemSize );
```

En donde:

- `uxQueueLength` es el número de elementos de la cola.
- `uxItemSize` es el tamaño de cada elemento.
- La función devuelve un "manejador" que ha de pasarse a las funciones que envían y reciben datos de la cola creada.

Conviene además añadir lo siguiente:

- Como la cola puede albergar cualquier tipo de datos y puede tener el tamaño que se desee (siempre que exista memoria suficiente); es necesario indicarle a FreeRTOS el número de elementos y el tamaño de cada uno de estos elementos al crear la cola.
- Al igual que con los semáforos, es imprescindible crear la cola antes de usarla. Por tanto, ésta ha de crearse antes de arrancar el planificador.
- Para usar las funciones de manejo de colas es necesario incluir previamente el archivo `queue.h`, donde se definen las estructuras de datos y los prototipos de las funciones.

Envío de datos a la cola

Para enviar un dato a la cola se usa la función `xQueueSend`, cuyo prototipo es:

```
portBASE_TYPE xQueueSend(xQueueHandle xQueue,
                         const void *pvItemToQueue,
                         portTickType xTicksToWait );
```

Los argumentos de la función y el valor devuelto por ella son:

- `xQueue` es el manejador de la cola, devuelto por `xQueueCreate`. Esto obliga a que la variable `xQueue` sea global tanto para la función desde donde se inicializa la cola como para todas las tareas que la usan.
- `pvItemToQueue` es un puntero al dato que se envía a la cola. Puede parecer extraño que se tenga que enviar un puntero que además es de tipo **`void`**. La razón de ello es que la función sea flexible al poder recibir cualquier tipo de dato. Recuerde que un puntero **`void`** puede apuntar a cualquier tipo de dato, por lo que a la función se le puede enviar la dirección de cualquier dato. Como al crear la cola se especifica el tamaño de cada dato de la cola, `xQueueSend` dispone de toda la información necesaria para copiar el dato que le pasamos a la cola.
- `xTicksToWait` es el timeout en ticks de reloj durante el cual la función ha de estar bloqueada si la cola está llena.
- La función devuelve `pdTRUE` si el dato se ha enviado o `errQUEUE_FULL` si no se ha podido enviar porque la cola sigue llena después de transcurrir el *timeout* especificado en el tercer argumento.

Conviene destacar que, tal como se acaba de decir, los datos se copian en la cola, por lo que hay que ser cuidadoso en no enviar datos muy "voluminosos", ya que esto haría el uso de la cola ineficiente al tener que copiarse un gran volumen de datos. En estos casos se puede optar por enviar un puntero a los datos, aunque ello es peligroso ya que se crean datos compartidos.

Recepción de datos de la cola

Para recibir un dato desde la cola se usa la función `xQueueReceive`, cuyo prototipo es:

```
portBASE_TYPE xQueueReceive(xQueueHandle xQueue,
                            void *pvBuffer,
                            portTickType xTicksToWait );
```

Los argumentos de la función y el valor devuelto por ella son:

- `xQueue` es el manejador de la cola, devuelto por `xQueueCreate`. Como se dijo antes, esto obliga a que la variable `xQueue` sea global tanto para la función desde donde se inicializa la cola como para todas las tareas que la usan.

- `pvBuffer` es un puntero al *buffer* donde se copiará el dato recibido de la cola. Obviamente este puntero ha de apuntar a una variable del mismo tipo que la que se ha enviado a la cola. Ojo, `pvBuffer` ha de apuntar a una variable, no a un puntero sin inicializar. En el ejemplo del apartado siguiente se muestra cómo hacerlo correctamente.

- `xTicksToWait` es el *timeout* en ticks de reloj durante el cual la función ha de estar bloqueada si la cola está vacía.

- La función devuelve `pdTRUE` si el dato se ha recibido o `pdFALSE` si no se ha recibido nada porque la cola sigue vacía después de transcurrir el *timeout* especificado en el tercer argumento. En este último caso no debe usarse el valor `*pvBuffer`, pues contendrá el valor anterior.

4.7.2. Ejemplo de manejo de colas usando FreeRTOS

En un sistema en tiempo real es necesario guardar un registro de los errores producidos para que en caso de que se produzca algún problema con el sistema, los ingenieros puedan al menos tener una idea de qué errores se han producido en el fallo. Las características del sistema serán:

- Por cada error se guardará una cadena de caracteres con el siguiente formato: `"Tx:Eyy\n"`. La `x` representa el número de la tarea en la que se ha producido el error y la `y` un código que indica el error producido.

- Existen 4 tareas en el sistema y cada una podrá enviar registros de error a la cola

- Existe una única tarea que se encarga de leer registros de error de la cola y escribirlos en una memoria EEPROM. Para escribir datos en la EEPROM existe la siguiente función:

  ```
  uint16 EscEEPROM(void *pbuff, uint16 tambuf);
  ```

  Que escribe en la memoria EEPROM `tambuf` bytes a partir de la dirección `pbuff`. La función devuelve el número de bytes que ha escrito en la memoria EEPROM.

En primer lugar se muestra el programa principal con la inicialización del sistema:

```
#include "mcf5282.h"
#include <string.h> /* Para strcpy */

/* Includes del Kernel. */
#include "FreeRTOS.h"
#include "queue.h"
#include "task.h"

#define PRIO_T_ERR 1
#define PRIO_T1    2
#define PRIO_T2    3
#define PRIO_T3    4
#define PRIO_T4    5
#define TAM_PILA 256

#define TAM_COLA 20 /* 20 mensajes */
#define TAM_MSG  8  /* cada mensaje: "Tx:Eyy\n\0"
                       ocupa 8 caracteres */
xQueueHandle cola_err;

void main(void)
{
  InitM5282Lite_ES(); /* Inicializa el Hardware del
                         microcontrolador */
  InitQueSeYo();

  /* Se crea la cola */
  cola_err = xQueueCreate(TAM_COLA, TAM_MSG);

  /* Se crean las tareas */
  xTaskCreate(TareaErr, "TareaE", TAM_PILA, NULL,
              PRIO_T_ERR, NULL);
  xTaskCreate(Tarea1, "Tarea1", TAM_PILA, NULL,
              PRIO_T1, NULL);
  xTaskCreate(Tarea2, "Tarea2", TAM_PILA, NULL,
              PRIO_T2, NULL);
  xTaskCreate(Tarea3, "Tarea3", TAM_PILA, NULL,
              PRIO_T3, NULL);
  xTaskCreate(Tarea4, "Tarea4", TAM_PILA, NULL,
              PRIO_T4, NULL);

  vTaskStartScheduler(); /* y por último se arranca el
                            planificador */
}
```

En primer lugar, destacar que al tener que usar colas, ha sido necesario incluir el archivo `queue.h`.

Las prioridades de las tareas se han elegido de forma que la tarea menos prioritaria sea la que se encarga de escribir en la memoria EEPROM los errores generados por el resto de tareas del sistema. Esto es así porque se supone que el resto de tareas están haciendo cosas críticas y por tanto no pueden estar esperando a que se escriba en una memoria EEPROM, lo cual es relativamente lento. Es por ello precisamente por lo que se usa una cola: para permitir que el resto de tareas puedan seguir haciendo su trabajo y sólo cuando no haya nada más que hacer se escriban los mensajes de error en la EEPROM. Obviamente habrá que dimensionar la cola de forma que en el caso más desfavorable no se pierdan mensajes. En este ejemplo se ha supuesto que con una cola con capacidad para 20 mensajes es más que suficiente, tal como se ha definido en `TAM_COLA`.

Como se ha visto antes, las colas constan de elementos de tamaño fijo. En este ejemplo, a la cola se envían cadenas de caracteres, pero todas ellas tienen el mismo formato: `"Tx:Eyy\n"`. Por tanto, cada cadena tendrá un tamaño de 8 bytes, tal como se ha definido en `TAM_MSG`.

Una vez definidos el número de elementos de la cola, se ha creado una variable denominada `cola_err` que será el "manejador" de la cola. Esto es similar a la variable de tipo `FILE` que devuelve `fopen` al abrir un archivo y que luego es necesario pasar a las funciones como `fprintf` que interaccionan con el archivo. De la misma forma, esta variable la devuelve la función `xQueueCreate` y hay que pasarla a las funciones `xQueueSend` y `xQueueReceive` cada vez que se quiera enviar o recibir datos hacia o desde esta cola. Por último, destacar que como el "manejador" de la cola se tiene que usar desde varias tareas se ha declarado como global.

Dentro del `main` está en primer lugar la inicialización del *hardware*, seguido de la creación de la cola con `TAM_COLA` elementos de un tamaño de `TAM_MSG` bytes. A continuación se crean todas las tareas necesarias y se arranca el planificador del sistema operativo.

Cada una de las tareas realizará su labor, la cual no interesa en este ejemplo, y si encuentra un error enviará un código a la cola de errores. El código de la primera tarea es:

```
void Tarea1(void *pvParameters)
{
  extern xQueueHandle cola_err;
  char cad_err[8];

  while(1){
    /* Proceso Tarea1 */
    if (error_1){
      strcpy(cad_err, "T1:E01\n");
      xQueueSend(cola_err, (void *)cad_err,
```

```
                (portTickType) 100);
    }

    /* Continuación proceso Tarea1 */
    if (error_2){
      strcpy(cad_err, "T1:E02\n");
      xQueueSend(cola_err, (void *)cad_err,
                (portTickType) 100);
    }

    /* Resto proceso Tarea1 */
  }
}
```

Para enviar el código de error a la cola, primero se copia el mensaje en la cadena `cad_err` y a continuación se envía dicha cadena a la cola. En la llamada a la función `xQueueSend`, el primer argumento es el "manejador" de la cola devuelto por `xQueueCreate` al crear la cola. El segundo argumento es la dirección de la cadena, pero convertida a puntero **void** para que el compilador no se queje. Por último, como `xQueueSend` se puede bloquear si la cola está llena, el tercer argumento indica el *timeout* que se está dispuesto a esperar si se da esta situación. Si la tarea no puede esperar nada porque esté haciendo algo crítico se podría poner este valor a cero y `xQueueSend` saldrá inmediatamente si la cola está llena.

El código de la tarea 2 será muy similar al de la tarea 1 en lo que respecta al envío de los mensajes de error. A continuación se muestra el código de esta tarea, en el que se puede apreciar que sólo se han cambiado los mensajes de error, que obviamente no pueden coincidir con el resto de tareas. También se ha puesto el *timeout* a cero para que `xQueueSend` no se bloquee si la cola está llena.

```
void Tarea2(void *pvParameters)
{
  extern xQueueHandle cola_err;
  char cad_err[8];

  while(1){
    /* Proceso Tarea2 */
    if (error_1){
      strcpy(cad_err, "T2:E01\n");
      xQueueSend(cola_err, (void *)cad_err,
                (portTickType) 0);
      /* El timeout es 0 para no bloquear la tarea
             si la cola está llena */
    }
```

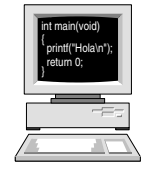
```
    /* Continuación proceso Tarea2 */
    if (error_27){
      strcpy(cad_err, "T2:E27\n");
      xQueueSend(cola_err, (void *)cad_err,
                 (portTickType) 0);
    }

    /* Resto proceso Tarea2 */
  }
}
```

Las tareas 3 y 4 no se muestran puesto que serán similares a las tareas 1 y 2 pero con otros tipos de errores y mensajes.

Por último, la tarea siguiente se encarga de sacar los mensajes de la cola y escribirlos en la memoria EEPROM:

```
void TareaErr(void *pvParameters)
{
  extern xQueueHandle cola_err;
  char cad_rec[8];

  while(1){
    if(xQueueReceive(cola_err, (void *)cad_rec,
                  (portTickType) 0xFFFFFFFF) == pdTRUE){
      /* Se ha recibido un dato. Se escribe en EEPROM */
      EscEEPROM((void *)cad_rec, 8);
    }
    /* si después de un timeout no se ha recibido nada
       la tarea se vuelve a bloquear a la espera de un
       nuevo dato */
  }
}
```

Como puede apreciar, la tarea consta de un bucle sin fin (como todas las tareas) que se bloquea a la espera de recibir un mensaje. La espera se ha puesto lo mayor posible. Se ha supuesto para ello que el tipo derivado `portTickType` es un entero de 32 bits. Si la función se desbloquea porque ha llegado un mensaje a la cola, ésta devolverá `pdTRUE` y por tanto se podrá procesar el mensaje, que `xQueueReceive` habrá copiado en la cadena `cad_rec`. Si se ha desbloqueado por un *timeout*, entonces no se podrá escribir nada en la EEPROM, pues no ha llegado ningún mensaje (`cad_rec` contendrá el mensaje anterior). En este caso simplemente se vuelve al principio del bucle para esperar que llegue un nuevo mensaje.



Realice el ejercicio 11.

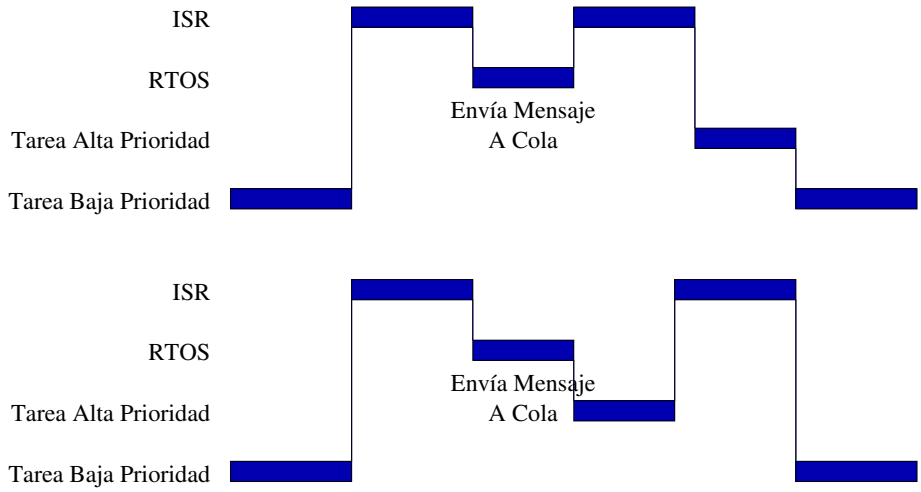


Figura 4.5: Interrupciones y sistemas operativos en tiempo real

4.8. Rutinas de atención a interrupción en los sistemas operativos en tiempo real

Cuando se usa un sistema operativo en tiempo real, es muy importante tener en cuenta dos precauciones a la hora de escribir rutinas de atención a las interrupciones:

- No se deben llamar funciones del sistema operativo en tiempo real que puedan bloquearse desde la rutina de atención a interrupción. Si se bloquea una interrupción, la latencia de ésta aumentará a límites intolerables y, lo que es peor, poco predecibles.

- No se deben llamar funciones del sistema operativo que puedan conmutar tareas, salvo que el sistema sepa que se está ejecutando una interrupción. Si el sistema operativo en tiempo real no sabe que se está ejecutando una interrupción (y si no se le avisa no tiene por qué enterarse), pensará que se está ejecutando la tarea que se ha interrumpido. Si la rutina de interrupción realiza una llamada que hace que el planificador pueda conmutar a una tarea de mayor prioridad (como por ejemplo escribir un mensaje en una cola), el sistema operativo realizará la conmutación, con lo que la rutina de atención a la interrupción no terminará hasta que terminen las tareas de mayor prioridad. En la figura 4.5 se ilustra el proceso gráficamente.

  La parte superior de la figura 4.5 muestra el comportamiento desea-

do: la rutina de atención a la interrupción (ISR)[13] llama al sistema operativo en tiempo real y esta llamada despierta a la tarea de alta prioridad. No obstante, el sistema operativo espera a que termine de ejecutarse la rutina de interrupción y al finalizar ésta, en lugar de volver a la tarea de baja prioridad que estaba ejecutándose antes de producirse la interrupción, realiza un cambio de contexto y pasa a ejecutar la tarea de alta prioridad.

La parte inferior de la figura 4.5 muestra lo que ocurre si el sistema operativo no se entera de cuándo se está ejecutando una tarea y cuándo una rutina de interrupción. En este caso, al realizar la llamada al sistema desde la rutina de interrupción y despertarse la tarea de alta prioridad, el sistema operativo realiza el cambio de contexto y pasa a ejecutar la tarea de alta prioridad. Cuando ésta termina, vuelve a conmutar al contexto de la tarea de baja prioridad, con lo que continúa ejecutándose la rutina de interrupción y, cuando ésta termine, la tarea de baja prioridad. Obviamente, en este caso la rutina de interrupción tardará demasiado en ejecutarse. Este retraso puede originar que se pierdan interrupciones, lo cual es intolerable en un sistema en tiempo real.

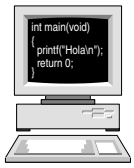
Existen tres métodos para tratar este problema:

- Avisar al sistema operativo en tiempo real de la entrada y salida de la rutina de atención a interrupción. Este método añade a la API del sistema operativo dos funciones: una para avisar al sistema de la entrada en la rutina de interrupción y otra para avisar de la salida. Así el sistema operativo sabe que no debe conmutar a otra tarea de mayor prioridad aunque ésta haya pasado al estado de "lista" hasta que no se salga de la rutina de interrupción. Este es el método usado por μC/OS-II.

- El sistema operativo en tiempo real intercepta todas las interrupciones y luego llama a la rutina de atención a interrupción proporcionada por la aplicación. En este método, usado por RTAI, la gestión de las interrupciones corre a cargo del sistema operativo, es decir, cuando se produce una interrupción se ejecuta una rutina del sistema que ejecuta una rutina de atención proporcionada por la aplicación. Obviamente en este caso no hace falta avisar al sistema operativo de la entrada en la rutina de interrupción pues él ya toma nota antes de llamar a la rutina de atención proporcionada por la aplicación. Lo que sí es necesario es indicar al sistema operativo en la inicialización qué rutinas tratan qué interrupciones, normalmente mediante una función de la API [Simon, 1999].

[13] ISR son las siglas en inglés de *Interrupt Service Routine*.

- Existen funciones especiales para su llamada desde las ISR. Este método, que es el usado por FreeRTOS, añade a la API funciones especiales para ser llamadas desde dentro de la rutina de interrupción que se diferencian de las normales en que no conmutan las tareas. Por ejemplo, existe una función para enviar un mensaje a una cola y otra para mandarlo desde la rutina de interrupción. En estos casos sigue siendo necesario llamar a una función para indicar la salida de la interrupción de forma que el sistema operativo ejecute el planificador y conmute a otra tarea de mayor prioridad que haya cambiado al estado "lista" como consecuencia de la ejecución de la rutina de interrupción.



Realice el ejercicio 12.

4.8.1. Rutinas de atención a interrupción en FreeRTOS

Como se acaba de decir, en FreeRTOS existen dos tipos de llamadas al sistema operativo: las normales y las diseñadas para ser llamadas desde una rutina de atención a interrupción. Por ejemplo, en la sección 4.5.3 se usa la función `xSemaphoreGiveFromISR` para liberar un semáforo desde una rutina de atención a interrupción. También en el manejo de colas existen funciones para ser llamadas desde las rutinas de atención a interrupción. Así, mientras que para enviar un dato a una cola en una tarea se usa la función `xQueueSend`, para enviarlo desde una rutina de atención a interrupción se usará la función `xQueueSendFromISR`. De la misma forma, para recibir un dato de la cola, en lugar de llamar a `xQueueReceive`, se llamará a `xQueueReceiveFromISR`.

Las funciones diseñadas para ser llamadas desde las rutinas de atención a interrupción tienen dos diferencias con respecto a las normales:

- No pueden bloquearse, al estar pensadas para ser llamadas desde una interrupción.

- No producen cambios de contexto, aunque la escritura o recepción de datos de la cola despierten a una tarea más prioritaria que la tarea que estaba ejecutándose antes de que se produjese la interrupción. En este caso, el cambio de contexto es necesario hacerlo "a mano" al finalizar la ejecución de la rutina de interrupción, de forma que desde la interrupción se vuelva a la rutina recién despertada (que tiene más prioridad) en lugar de a la que se estaba ejecutando cuando se produjo la interrupción.

Funciones para manejo de colas desde rutinas de atención a interrupción en FreeRTOS

Aunque en FreeRTOS existen varias funciones diseñadas para ser llamadas desde una rutina de atención a interrupción, todas funcionan de la

misma forma. Por ello se van a estudiar en detalle solamente las funciones de manejo de colas.

Para enviar un dato a la cola desde una rutina de interrupción (ISR) se usa la función xQueueSendFromISR:

```
portBASE_TYPE xQueueSendFromISR(xQueueHandle xQueue,
                    const void *pvItemToQueue,
                    portBASE_TYPE xTaskPreviouslyWoken );
```

Los argumentos de la función y el valor devuelto por ella son:

- Lo dicho para xQueue y pvItemToQueue en la función xQueueSend es igual de válido ahora, ya que ambos argumentos funcionan de la misma forma en este caso.

- xTaskPreviouslyWoken permite que desde la ISR se puedan hacer varias llamadas a xQueueSendFromISR (o a otras funciones similares como xSemaphoreGiveFromISR). En la primera llamada este argumento ha de valer siempre pdFALSE y en las siguientes se pasará el valor devuelto por xQueueSendFromISR en la llamada anterior. No obstante, en los ejemplos de este libro sólo se va a llamar a estas funciones una sola vez desde la rutina de interrupción, por lo que no usaremos esta característica de FreeRTOS.

- La función devuelve pdTRUE si se ha despertado alguna tarea o pdFALSE si no.

Para recibir un dato desde la cola dentro de una rutina de interrupción se usa la función xQueueReceiveFromISR:

```
portBASE_TYPE xQueueReceiveFromISR(xQueueHandle xQueue,
                           void *pvBuffer,
                           portBASE_TYPE *pxTaskWoken);
```

Los argumentos de la función y el valor devuelto son:

- Lo dicho para xQueue y pvBuffer en la función xQueueReceive es igual de válido ahora, ya que ambos argumentos funcionan de la misma forma en este caso.

- *pxTaskWoken permite que desde la ISR se puedan hacer varias llamadas a xQueueReceiveFromISR o similares. Este puntero ha de contener la dirección de una variable que en un principio se inicializará a pdFALSE. Si alguna de las llamadas a xQueueReceiveFromISR hace que alguna tarea tenga que despertar por estar esperando a que se liberara sitio en la cola, la variable *pxTaskWoken pasará a valer pdTRUE, pero si no mantendrá su valor.

- La función devuelve `pdTRUE` si se ha recibido un dato de la cola y por tanto se puede usar o `pdFALSE` si no.

Tanto en la función de envío como en esta de recepción, si se detecta que hay que despertar a una tarea, es necesario forzar el cambio de contexto al finalizar la rutina de interrupción mediante la llamada a la función `taskYIELD`. Esto se ilustrará en el ejemplo siguiente.

Ejemplo de manejo de colas desde una rutina de atención a interrupción usando FreeRTOS

Para ilustrar el uso de colas en FreeRTOS desde una rutina de atención a interrupción, se va a volver a implantar el ejemplo mostrado en la sección 3.3.4 (página 70). En dicho ejemplo se usa una cola para comunicar la rutina de interrupción del puerto serie con la tarea de primer plano. La rutina de atención a la interrupción se limita a copiar el carácter recibido de la UART en la cola. La tarea de primer plano se encarga de verificar si hay caracteres nuevos en la cola en cada iteración del bucle de *scan*. Si hay un carácter nuevo, lo saca de la cola y lo copia en una cadena denominada `mensaje`. Cuando recibe un mensaje completo, indicado por la recepción del carácter de retorno de carro, se procesa dicho mensaje.

El sistema ahora será el mismo, salvo que se usarán los recursos del sistema operativo para comunicar mediante una cola la rutina de interrupción y la tarea. Al igual que en el ejemplo de la sección 3.3.4, el sistema está realizado para el microcontrolador ColdFire MCF5282.

En primer lugar se muestra la inicialización del sistema.

```
#include <stdio.h>
#include "mcf5282.h"
#include "interrupciones.h"
/* Kernel includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

#define TAM_COLA 100
static xQueueHandle cola_rec; /* Cola para recibir */

void InitSerie(void)
{
  /* Primero se crea la cola */
  cola_rec = xQueueCreate(TAM_COLA, sizeof(char));
  if(cola_rec == NULL){
    /* Error fatal: Indicar el error y abortar */
  }
```

```
  InitUART0(19200);/* Inicialización del puerto serie 0 */
}
```

En este ejemplo se ha optado por estructurar un poco mejor el código en módulos. Así, todo el tratamiento del puerto serie se va a realizar en un sólo archivo, al que se llamará por ejemplo `serie.c`.

Lo primero que hay que destacar es que para evitar un trasiego innecesario de variables globales entre módulos, el "manejador" de la cola se ha declarado global estático para que sólo sea visible dentro de `serie.c`. Dentro del `main` será necesario llamar a la función `InitSerie`, la cual, como puede apreciarse en la figura, se encarga de crear la cola de recepción y de inicializar el puerto serie.

Nótese que se ha verificado que la función `xQueueCreate` devuelva un puntero válido. Si ésta devuelve `NULL` es señal de que algo ha ido mal en la creación de la cola (por ejemplo que no hay memoria suficiente) y por tanto no podemos usarla. El qué hacer en este caso depende del sistema real en el que se implante. Por ejemplo en un sistema empotrado podría encenderse un LED que indicase un fallo general y dejar bloqueado el sistema, por ejemplo mediante un bucle sin fin, hasta que venga algún responsable de mantenimiento:[14]

```
  cola_rec = xQueueCreate(TAM_COLA, sizeof(char));
  if(cola_rec == NULL){
     EscribePuertoB(0x01); /* LED 0 indica error Fatal */
     while(1); /* Se queda bloqueado el sistema hasta que
                  venga el técnico de mantenimiento */
  }
```

La rutina de atención a la interrupción se muestra a continuación:

```
__declspec(interrupt) void InterruptUART0(void)
{
  portBASE_TYPE xTaskWokenByPost = pdFALSE;
  char car_recibido;

  if(MCF_UART0_UISR & MCF_UART_UISR_FFULL_RXRDY){
    /* Llegó un carácter. Se lee del puerto serie */
    car_recibido = MCF_UART0_URB;
    /* Y se envía a la cola de recepción */
    xTaskWokenByPost = xQueueSendFromISR(cola_rec,
                       &car_recibido, xTaskWokenByPost);
    if(xTaskWokenByPost == pdTRUE ){
      taskYIELD(); /* Si el envío a la cola ha despertado
                      una tarea, se fuerza un cambio de
                      contexto */
```

[14] O hasta que salte el *watchdog* que reinicializará el sistema.

```
    }
  }
}
```

En primer lugar se ve si quien ha interrumpido ha sido el receptor de la UART. Si es así se copia el carácter del buffer de recepción a la variable `car_recibido` y éste se envía a la cola. Es necesario hacerlo así porque el segundo argumento de la función `xQueueSendFromISR` ha de ser la dirección del dato a enviar a la cola, y es más fácil escribir la dirección de una variable local que la dirección del registro *hardware* de la UART.

La función `xQueueSendFromISR` devolverá `pdTRUE` si como consecuencia del envío a la cola alguna tarea se ha despertado (ha pasado del estado bloqueada a lista). Sólo en este caso habrá que llamar al planificador mediante la función `taskYIELD` para que éste evalúe si la tarea que acaba de despertarse es más prioritaria que la que se estaba ejecutando cuando saltó la interrupción y por tanto es necesario realizar un cambio de contexto. Si no se ha despertado ninguna tarea como consecuencia del envío a la cola, se sale de la interrupción sin hacer nada más.

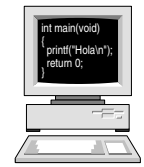
Por último, se muestra la tarea que se encarga de procesar los caracteres que llegan por el puerto serie:

```
void ProcesaRecSerie(void *pvParameters)
{
  static char mensaje[100];
  static BYTE8 indice=0;
  char car_rec;

  while(1){
    if(xQueueReceive(cola_rec, &car_rec,
                  (portTickType) 0xFFFFFFFF) == pdTRUE ){
      /* Se ha recibido un carácter de la cola.
         Se almacena */
      mensaje[indice] = car_rec;
      if(mensaje[indice] == '\n'){
        /* El \n indica el final del mensaje */
        mensaje[indice+1] = '\0';
        ProcesaMensaje(mensaje);
        indice = 0;
      }else{
        indice++;
      }
    }
  }
}
```

Como puede observar, la tarea se bloquea nada más empezar su bucle

sin fin a la espera de que llegue algún carácter a la cola. Cuando la función `xQueueReceive` retorne debido a la llegada de un carácter a la cola, éste se añadirá a la cadena usada para guardar el mensaje (`mensaje`) y si éste es el avance de línea se añadirá el terminador nulo a la cadena del mensaje y se llamará a la función `ProcesaMensaje` para procesarlo.

Tenga en cuenta que es imprescindible verificar que la llamada a la función `xQueueReceive` ha devuelto `pdTRUE` por la llegada de un carácter antes de procesarlo. Si se ha desbloqueado por un *timeout*, la función devolverá `pdFALSE` y no se hace nada, volviendo a esperar la llegada de un carácter a la cola.



Realice el ejercicio 13.

4.9. Gestión de tiempo

En un sistema en tiempo real, son numerosas las situaciones en las que es necesario garantizar la ejecución periódica de una tarea o suspender la ejecución de una tarea durante un determinado periodo de tiempo.

Por ejemplo, en un sistema de control es necesario ejecutar el algoritmo de control del sistema cada periodo de muestreo. En otros sistemas puede ser conveniente ejecutar ciertas tareas no críticas cada cierto tiempo para ahorrar energía. Por ejemplo, en una estación meteorológica puede ser conveniente ejecutar las rutina de medida cada varios segundos.

También es frecuente que en el diálogo con el *hardware* sea necesario esperar un tiempo determinado para que éste realice su tarea antes de continuar dialogando con él. Por ejemplo, en un sistema de bombeo será necesario esperar unos segundos desde que se conecta la bomba hasta que se comienza a monitorizar la presión generada por ésta.

Por último, también es necesario una gestión interna del tiempo para poder ofrecer *timeouts* en las funciones que se bloquean esperando mensajes o semáforos.

La gestión de tiempo en la mayoría de los sistemas operativos se basa en usar una interrupción periódica para incrementar un contador en el que se lleva la cuenta del tiempo transcurrido desde que se arrancó el sistema. Cada incremento del contador se denomina *tick* de reloj. Normalmente este periodo de tiempo (*tick*) es configurable al compilar el núcleo. En la versión de FreeRTOS para ColdFire usada en este texto, el *tick* de reloj es de 5 ms por defecto y se usa el temporizador PIT3 para generar la interrupción periódica.

4.9.1. Gestión de tiempo en FreeRTOS

Existen dos funciones para generar retrasos en FreeRTOS:

- **void** `vTaskDelay(portTickType xTicksToDelay)`. Suspende la tarea durante el número de `ticks` especificado.

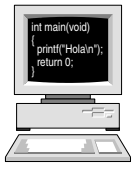
- **void** vTaskDelayUntil(portTickType *pxPreviousWakeTime, portTickType xTimeIncrement).

  Suspende la tarea hasta el instante especificado por la expresión: *pxPreviousWakeTime + xTimeIncrement.

Si lo que se desea es generar un retardo durante la ejecución de una tarea, lo más conveniente es usar la primera función, especificando el número de ticks de reloj que se desea esperar. Hay que tener en cuenta que el sistema operativo lo único que hace es esperar a que se produzcan el número de ticks especificados en la llamada, por lo que el retardo dependerá del instante en el que se produzca la llamada en relación con el instante en el que se produce el siguiente *tick*. Si por ejemplo la llamada se produce un "pelín" antes del siguiente *tick*, el retardo será prácticamente *ticks* - 1 (más el "pelín"). Si la llamada se produce justo un "pelín" después de un *tick*, el retardo será prácticamente igual al número de *ticks* especificado (menos el "pelín"). Por tanto si se desea que una tarea se retrase al menos 27 *ticks* habrá que realizar la llamada vTaskDelay(28).

Si lo que se desea es que una tarea se ejecute periódicamente, la función anterior no sirve, ya que el tiempo que tarda en ejecutarse la función desde que se desbloquea es variable. Para conseguir una ejecución periódica es mejor usar la segunda función que despierta a la tarea en un instante determinado. Su uso quedará claro en el segundo ejemplo mostrado a continuación.

Si se desea especificar el tiempo en milisegundos en lugar de en ticks de reloj, será necesario dividir el tiempo deseado por la duración del tick de reloj. Para mejorar la legibilidad y portabilidad del código, FreeRTOS define una constante denominada configTICK_RATE_MS con el número de milisegundos que dura un tick.



Por último, hay que destacar que los retardos reales pueden ser mayores que los especificados si cuando una vez terminado el retardo y la tarea pasa a estar lista para su ejecución, existe otra tarea de mayor prioridad también lista para ejecutarse.

Realice los ejercicios 14 y 15.

4.9.2. Ejemplo: arranque de una bomba

A continuación se muestra un primer ejemplo para ilustrar el uso de las funciones de retardo en FreeRTOS. La función ArrancaBomba arranca una bomba mediante un arrancador estrella/triángulo y después de un tiempo monitoriza la presión de salida de la bomba para verificar que ésta está funcionando correctamente. El tiempo de retardo desde que se conecta el motor en triángulo hasta que se conecta en estrella es de 500 ms, mientras que el tiempo que se espera antes de verificar si se está bombeando es igual a 1 minuto, por si acaso la bomba estaba descargada.

```
void ArrancaBomba(void)
```

```
{
  ConectaTensiónEstrella();
  vTaskDelay(500/configTICK_RATE_MS);
  ConectaTensiónTriangulo();
  vTaskDelay(60000/configTICK_RATE_MS);
  if(PresionOK()==0){ /* No hay presión. Por tanto la
                         bomba no está funcionando */
    DesconectaTension();
    AlarmaFalloArranque();
  }
}
```

4.9.3. *Ejemplo: tarea periódica*

Existen diversas situaciones en las que una tarea ha de ejecutarse periódicamente. Un primer ejemplo es un sistema de control, en el que el algoritmo de control ha de ejecutarse cada periodo de muestreo. Otro ejemplo típico es el almacenamiento periódico de una serie de variables del sistema en memoria para su posterior análisis.

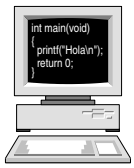
En FreeRTOS se usa la función `vTaskDelayUntil` para que una tarea se ejecute periódicamente, tal como se muestra a continuación:

```
void TareaPeriodica(void *pvParameters)
{
  portTickType xLastWakeTime;
  portTickType xPeriodo;

  xPeriodo = 20/configTICK_RATE_MS; /* Periodo 20 ms */

  /* Inicializa xLastWakeTime con el tiempo actual */
  xLastWakeTime = xTaskGetTickCount();
  while(1){
    vTaskDelayUntil(&xLastWakeTime, xPeriodo ); /* Espera
                                   el siguiente periodo */
    /* Realiza su proceso */
  }
}
```

En primer lugar, cabe destacar que se han definido dos variables del tipo `portTickType`, una para almacenar el tiempo (obviamente expresado en el número de ticks desde el arranque) de la última vez que nos han despertado, y otra para almacenar el periodo de la tarea.

La función `vTaskDelayUntil` recibe como primer parámetro la dirección de una variable en la que guardará automáticamente el valor de tiempo en el que se ha despertado a la tarea. Por tanto la tarea se limitará a pasarle

esta variable, pero bajo ningún concepto debe modificarla. El tiempo en el que ha de despertarse la tarea se calcula sumándole a esta variable el valor del segundo parámetro de la función, en el que se ha almacenado el periodo de ejecución de la tarea expresado en ticks de reloj.



Realice el ejercicio 16.

Por último, destacar que antes de entrar en el bucle sin fin se ha inicializado `xLastWakeTime` con el tiempo actual mediante la llamada a la función `xTaskGetTickCount`.

Conviene también tener en cuenta que una tarea periódica sólo tendrá un periodo exacto si es la más prioritaria del sistema. Si no lo es, su ejecución puede verse retrasada por otras tareas con mayor prioridad.

4.9.4. Preguntas típicas

Algunas preguntas típicas sobre la gestión de tiempo son:

- ¿Cuánto debe durar un *tick* de reloj? La duración del *tick* de reloj suele ser configurable fácilmente.[15] Valores típicos son 5 ó 10 ms. Valores menores permiten temporizaciones más exactas pero a cambio el rendimiento del sistema se resiente, ya que se emplea un porcentaje considerable del tiempo de la CPU ejecutando la rutina de gestión de tiempo. Valores mayores mejoran el rendimiento a costa de una pérdida de precisión en las temporizaciones.

- ¿Y si se necesita una temporización precisa? Si se necesita una temporización muy precisa y no es posible disminuir el *tick* del reloj, la solución es usar un temporizador *hardware* que dispare una interrupción que realice el trabajo o que a su vez despierte la tarea a temporizar. Obviamente esta tarea tendrá que tener una alta prioridad si se quiere que se ejecute en cuanto pase del estado de bloqueada al estado de lista.

4.10. Ejercicios

1. Modifique la función `ImprimeErrores` mostrada en la página 92 para que sea reentrante. Razone si la solución adoptada presenta algún inconveniente.

2. En la tarea `ImprimeHora` mostrada en la página 95, ¿se podría solucionar el arbitraje del acceso al puerto serie cortando las interrupciones mientras se llama a `SeriePuts()`? En caso afirmativo ¿existe alguna ventaja en el uso de semáforos?

[15] Basta para ello modificar la programación del temporizador que dispara la rutina de cuenta de tiempo.

3. En la tarea `ImprimeHora` mostrada en la página 95, ¿por qué no se incluye la llamada a `sprintf()` dentro de la zona protegida por el semáforo? ¿Qué condiciones han de darse para que sea seguro el no incluirla?

4. Escriba una función para convertir un byte a binario. El prototipo de la función será:

   ```
   char *itoaBin(uint8 numero, char *cadena);
   ```

   La función recibe una cadena que ha de tener al menos 9 caracteres de capacidad y escribe en esa cadena el valor del argumento `numero` en binario. Devuelve la dirección de la cadena que se le pasa para poder ser usada directamente en una función que maneje cadenas, como por ejemplo en `printf` si se desea imprimir.

5. Modifique la tarea `ImprimeHora` mostrada en la página 98 para que si no se obtiene el semáforo vuelva a intentarlo hasta que lo consiga. Sólo entonces se volverá al principio de la función, justo después de `while(1)`.

6. Repita el ejercicio anterior pero para la tarea `EnviaEntradas` mostrada en la página 100.

7. Modifique las tareas `ImprimeHora` y `EnviaEntradas`, mostradas en las páginas 98 y 100 respectivamente, para que el envío de la cadena al puerto serie se realice mediante una función. La función tendrá como prototipo:

   ```
   int ImprimeSerie(char *pcadena);
   ```

   No olvide que como esta función se llama desde dos tareas, ha de ser reentrante. Por lo tanto, ha de usar un semáforo (`sem_serie`) para arbitrar el acceso al puerto serie.

   La función devolverá un 1 si se ha enviado la cadena o un 0 si ha transcurrido un *timeout* sin que el semáforo haya quedado libre.

8. Reescriba el código mostrado en la sección 4.5.6 (página 105) usando el sistema operativo FreeRTOS. Use para todos los semáforos un *timeout* de 1000. Describa la ejecución de ambas tareas suponiendo que inicialmente ambos semáforos están libres, que inicialmente se ejecuta la `Tarea1` y que se produce un cambio de contexto justo antes de que la `Tarea1` pida el semáforo B. Ilustre esta descripción mediante un diagrama similar al mostrado en la figura 4.4

9. En FreeRTOS también es posible crear semáforos sin *timeout*, al igual que en μC/OS-II. Para ello basta con pedir el semáforo de la siguiente manera:

```
while(xSemaphoreTake(semaforo_a, (portTickType) 1000)
      != pdTRUE);
```

Modifique el ejemplo mostrado en la sección 4.5.6 (página 105) usando esta técnica.

10. En el ejemplo de la sección 4.5.1 (página 96) se arbitra el acceso al puerto serie por dos tareas mediante el uso un semáforo. Modifique el código para proteger dicho acceso mediante la inhabilitación de la conmutación de tareas.

11. En el ejemplo sobre el uso de colas en FreeRTOS, mostrado en la sección 4.7.2, se pierde un poco de tiempo en las tareas copiando el mensaje de error en la cadena `cad_err`. Modifique el código de las tareas para evitar esta copia.

12. En la sección 4.8 se discuten tres alternativas para realizar llamadas al sistema operativo desde las rutinas de atención a interrupción. Discuta las ventajas e inconvenientes de cada una de estas alternativas en cuanto a la latencia y la facilidad de programación.

13. Escriba el programa principal para implantar el sistema expuesto en la sección 4.8.1 (página 120). Tenga en cuenta que es necesario inicializar el puerto serie, crear la tarea y arrancar el planificador.

14. Use una llamada a `vTaskDelay` para generar un retardo de 500 ms.

15. Si el tipo `portTickType` fuese un entero de 16 bits, ¿cuál será el retardo máximo que se puede conseguir si el *tick* de reloj es de 5 ms? ¿y si `portTickType` fuese un entero de 32 bits?

16. A la vista del ejemplo mostrado en la sección 4.9.3, si la tarea empieza a ejecutarse en el tick 27, ¿Cuando será la primera vez que se ejecute el cuerpo de la tarea, indicado mediante el comentario `/* Realiza sus tareas */`? Suponga que no hay tareas de mayor prioridad listas para ejecutar y que el tiempo de ejecución de las funciones `xTaskGetTickCount` y `vTaskDelayUntil` es despreciable

APÉNDICE A

API de FreeRTOS

En este apéndice se incluye una breve descripción de las funciones de la API[1] de FreeRTOS usadas en este texto. Esta información está extraída de la página web del sistema operativo en donde se puede obtener una información más actualizada y completa, así como ejemplos de uso de cada una de las funciones.

El sistema operativo FreeRTOS se encuentra en www.freertos.org

Algunos parámetros como el *tick* de reloj son configurables y dependen de la versión de FreeRTOS. Los valores de los parámetros mostrados en este apéndice son los parámetros por defecto de la versión para ColdFire de FreeRTOS.

A.1. Nomenclatura

En la escritura del código fuente de FreeRTOS se han seguido una serie de convenciones a la hora de dar nombres a las variables y funciones. Antes de estudiar la interfaz de este sistema operativo, conviene familiarizarse con ellas. Las convenciones seguidas son las siguientes:

- Variables:
  - Las variables de tipo **`char`** se preceden con una `c`.
  - Las variables de tipo **`short`** se preceden con una `s`.
  - Las variables de tipo **`long`** se preceden con una `l`.
  - Las variables de tipo **`float`** se preceden con una `f`.
  - Las variables de tipo **`double`** se preceden con una `d`.
  - Las variables de tipo **`void`** se preceden con una `v`.
  - Otros tipos de variables, como por ejemplo las estructuras o los tipos definidos con **`typedef`**, se preceden por una `x`.

[1] API son las siglas del inglés *Application Programmer Interface*: interfaz del programador de aplicaciones.

  - Los punteros se preceden con una p adicional. Por ejemplo, un puntero a carácter se precederá por pc.
  - Las variables sin signo (**unsigned**) se preceden con una u adicional. Por ejemplo, una variable de tipo **unsigned short** se precederá por us.

- Funciones:
  - Los nombres de las funciones de la API tienen un prefijo que indica el tipo de variable que devuelven. Por ejemplo, el nombre de la función xTaskCreate se ha precedido por una x porque devuelve un tipo derivado.
  - Los nombres de las funciones, después del prefijo que indica el tipo de dato devuelto, contienen el nombre del fichero en el que están definidas. Por ejemplo, la función xTaskCreate está definida en el archivo Task.c

En las secciones siguientes, para cada función expuesta, se ha escrito antes de su prototipo el archivo cabecera (.h) en el que se declara dicha función.

A.2. Inicialización del sistema

- Creación de tareas:

```
#include "task.h"

portBASE_TYPE xTaskCreate(
                    pdTASK_CODE pvTaskCode,
                    const portCHAR * const pcName,
                    unsigned portSHORT usStackDepth,
                    void *pvParameters,
                    unsigned portBASE_TYPE uxPriority,
                    xTaskHandle *pvCreatedTask
                    );
```

  La función devuelve pdPASS si se ha creado la tarea o un código de error si no ha sido posible.

  Sus argumentos son:

  - pvTaskCode es el nombre de la función que implanta la tarea. Recuerde que esta función nunca debe retornar, es decir, ha de consistir en un bucle sin fin.
  - pcName es un nombre descriptivo para la tarea. Sólo es necesario para depuración y su longitud máxima es de 10 caracteres en esta versión de FreeRTOS.

- `usStackDepth` es el tamaño de la pila asociada a la tarea. Esta pila la crea el sistema operativo. El tamaño se especifica en palabras (4 Bytes). Si la función no usa funciones de entrada/salida (por ejemplo `sprintf`) es suficiente con una pila de 100 palabras. Si se usan será necesario aumentarla a 1000 palabras.
- `pvParameters` es un puntero a los parámetros iniciales. Si la tarea no necesita parámetros se puede poner a `NULL`
- `uxPriority` es la prioridad de la tarea.
- `pvCreatedTask` se usa para que la función devuelva un manejador de la tarea creada en caso de que se desee borrarla. Si el sistema es estático, es decir, una vez inicializado no se crean ni se destruyen tareas, este argumento se dejará a `NULL`.

- Arrancar el planificador:

  ```
  #include "task.h"

  void vTaskStartScheduler( void );
  ```

  Esta función arranca el planificador. A partir de este momento el sistema operativo toma el control y decidirá en cada momento qué tarea se ejecuta en la CPU. Si no hay ninguna tarea lista, ejecutará la tarea inactiva, la cual se crea automáticamente en este momento.

  Si todo va bien, esta función no retornará nunca,[2] pero si no puede ejecutarse el sistema operativo por falta de RAM, retornará inmediatamente.

A.3. Gestión de tiempo

Las siguientes funciones permiten bloquear una función durante un tiempo o hasta un determinado instante. En la versión de FreeRTOS para ColdFire el tick de reloj por defecto es de 5 ms.

- Bloquea la tarea durante `xTicksToDelay` *ticks*. Este tiempo se empieza a contar desde la llamada a la función.

  ```
  #include "task.h"

  void vTaskDelay(portTickType xTicksToDelay);
  ```

- Bloquea la tarea hasta `*pxPreviousWakeTime + xTimeIncrement` y guarda este valor en la variable `*pxPreviousWakeTime`.

[2] Es posible parar el planificador desde una tarea llamando a la función `vTaskEndScheduler`, aunque en los ejemplos de este texto no se ha usado esta funcionalidad.

```
#include "task.h"

void vTaskDelayUntil(portTickType *pxPreviousWakeTime,
                     portTickType xTimeIncrement);
```

- Devuelve el número de *ticks* transcurridos desde que se arrancó el planificador.

  ```
  #include "task.h"

  volatile portTickType xTaskGetTickCount(void);
  ```

A.4. Funciones de manejo de semáforos

En FreeRTOS los semáforos se implantan usando el mecanismo de colas. Es decir, se define una cola de un elemento de tamaño cero, ya que sólo se necesita el mecanismo de sincronización de la cola, pero no su almacenamiento de datos. Las funciones expuestas a continuación son en realidad macros que son sustituidas por el preprocesador de C por llamadas a las funciones de manejo de colas. Su definición puede consultarse en el archivo `semphr.h`.

- La inicialización de un semáforo se realiza con:

  ```
  #include "semphr.h"

  vSemaphoreCreateBinary(xSemaphoreHandle xSemaphore);
  ```

  En donde la variable `xSemaphore`, de tipo `xSemaphoreHandle`, debe ser creada por la aplicación y ser accesible por todas aquellas tareas que necesiten usar el semáforo.

- Para coger un semáforo se usa:

  ```
  #include "semphr.h"

  portBASE_TYPE xSemaphoreTake(
                          xSemaphoreHandle xSemaphore,
                          portTickType xBlockTime);
  ```

  Sus argumentos son:

  - `xSemaphore` es el manejador del semáforo que se desea coger.
  - `xBlockTime` el el *timeout* tras el cual la macro retornará aunque no se haya podido coger el semáforo.

La macro devuelve pdTRUE si se ha obtenido el semáforo o pdFALSE si ha transcurrido el *timeout* xBlockTime sin obtenerlo.

- Para soltar un semáforo se usa:

```
#include "semphr.h"

xSemaphoreGive(xSemaphoreHandle xSemaphore);
```

Su argumento indica el semáforo que debe soltarse.

- Para soltarlo desde una interrupción:

```
#include "semphr.h"

xSemaphoreGiveFromISR(
              xSemaphoreHandle xSemaphore,
              portBASE_TYPE xTaskPreviouslyWoken);
```

Sus argumentos son:

  - xSemaphore es el manejador del semáforo que se desea soltar.
  - xTaskPreviouslyWoken se usa cuando es necesario realizar varias llamadas a esta macro desde una misma rutina de interrupción. Si sólo se llama a la macro una vez ha de dejarse a pdFALSE.

La macro devuelve pdTRUE si como consecuencia de soltar el semáforo se ha despertado a una tarea. Este valor permite realizar un cambio de contexto en este caso al terminar la ejecución de la rutina de atención a la interrupción, tal como se ha mostrado en la sección 4.8.1.[3]

A.5. Funciones de manejo de colas

- Creación de la cola:

```
#include "queue.h"

xQueueHandle xQueueCreate(
              unsigned portBASE_TYPE uxQueueLength,
              unsigned portBASE_TYPE uxItemSize);
```

Esta función crea una cola de longitud uxQueueLength. Cada elemento tendrá el tamaño uxItemSize. La función devuelve una estructura para manejar la cola o 0 si no puede crearla. Dicha estructura ha de

[3]En esta sección se ha mostrado esta funcionalidad para el manejo de colas, pero en el caso de los semáforos se realiza de la misma manera.

ser creada previamente y ha de ser accesible para todas las tareas que necesiten usar la cola.

En la sección 4.7.2 (página 111) se muestra un ejemplo de uso de esta función.

- Para enviar datos a la cola:

```
#include "queue.h"

portBASE_TYPE xQueueSend(xQueueHandle xQueue,
                         const void * pvItemToQueue,
                         portTickType xTicksToWait);
```

Esta función envía el dato al que apunta `pvItemToQueue` a la cola `xQueue`. Si la cola está llena la tarea se bloquea. Si después de pasar `xTicksToWait` *ticks* sigue sin haber sitio en la cola, la función retornará.

La función devuelve `pdTRUE` si el dato se ha enviado o `errQUEUE_FULL` si transcurrido el *timeout* la cola sigue llena.

En la sección 4.7.2 (página 111) se muestra un ejemplo de uso de esta función.

- Para recibir datos de la cola:

```
#include "queue.h"

portBASE_TYPE xQueueReceive(
                    xQueueHandle xQueue,
                    void *pvBuffer,
                    portTickType xTicksToWait);
```

Esta función recibe un dato de la cola `xQueue`. El dato se almacena en la dirección `pvBuffer`, por lo que dicho puntero ha de apuntar a una variable del mismo tipo que las albergadas en la cola. Si la cola está vacía, la función se bloquea hasta que llegue un dato a la cola o hasta que transcurran `xTicksToWait` *ticks* de reloj.

La función devuelve `pdTRUE` si el dato se ha recibido o `pdFALSE` s en caso contrario.

En la sección 4.7.2 (página 111) se muestra un ejemplo de uso de esta función.

- Para enviar datos a la cola desde una interrupción:

```
#include "queue.h"
```

```
portBASE_TYPE xQueueSendFromISR(
                  xQueueHandle pxQueue,
                  const void *pvItemToQueue,
                  portBASE_TYPE xTaskPreviouslyWoken);
```

Esta función es igual a xQueueSend salvo que no se bloquea cuando la cola está llena y por tanto, como su propio nombre indica, puede usarse desde una rutina de interrupción.

El argumento xTaskPreviouslyWoken se usa cuando es necesario realizar varias llamadas a esta función desde una misma rutina de interrupción. Si sólo se llama a la función una vez ha de dejarse a pdFALSE.

La función devuelve pdTRUE si como consecuencia del envío del dato a la cola se ha despertado alguna tarea. Este valor permite realizar un cambio de contexto en este caso al terminar la ejecución de la rutina de atención a la interrupción, tal como se ha mostrado en la sección 4.8.1 (página 120).

- Para recibir datos de la cola en una interrupción:

```
#include "queue.h"

portBASE_TYPE xQueueReceiveFromISR(
                  xQueueHandle pxQueue,
                  void *pvBuffer,
                  portBASE_TYPE *pxTaskWoken);
```

Esta función es igual a xQueueReceive salvo que no se bloquea cuando la cola está vacía y por tanto, como su propio nombre indica, puede usarse dentro de una rutina de interrupción.

El argumento pxTaskWoken sirve para la función indique si como consecuencia del envío del dato a la cola se ha despertado alguna tarea. Para ello se le pasará un puntero a una variable de tipo portBASE_TYPE, la cual se pondrá a pdTRUE si se ha despertado una tarea o pdFALSE en caso contrario.

La función devuelve pdTRUE si se ha recibido un dato o pdFALSE en caso contrario.

A continuación se muestra un ejemplo de uso de esta función. En el ejemplo se ha supuesto que existe una cola para enviar caracteres al puerto serie y una rutina de atención a la interrupción del puerto serie que indica que dicho puerto serie está listo para aceptar el siguiente carácter.

```
#include "queue.h"
```

```
xQueueHandle xcola_env;

/* La cola xcola_env se inicializará en main() y desde
   alguna tarea se enviarán datos a esta cola */

/* Rutina de atención a la interrupción de recepción
   del puerto serie */

__declspec(interrupt) void InterruptUART0(void)
{
  portBASE_TYPE xdato_recibido;
  portBASE_TYPE xTaskWokenByReceive = pdFALSE;
  char ccaracter;

  /* Se obtiene el dato de la cola */
  xdato_recibido = xQueueReceiveFronISR(cola_env,
                       &ccaracter,
                       &xTaskWokenByReceive);

  if(xdato_recibido == pdTRUE){
    EnviaCar(ccaracter);
  }
  if(xTaskWokenByReceive == pdTRUE ){
      taskYIELD(); /* Si el envío a la cola ha
                      despertado una tarea, se
                      fuerza un cambio de
                      contexto */
    }
  }
}
```

En el ejemplo se ha supuesto que existe una función denominada EnviaCar que envía el carácter que se le pasa como argumento al buffer de transmisión del puerto serie.

APÉNDICE B

Un ejemplo real: autómata programable

En este apéndice se va a mostrar el diseño de un sistema empotrado completo usando las técnicas de programación en tiempo real discutidas en este libro. Se empezará por una versión básica que se implantará mediante un simple bucle de *scan* y se terminará con la versión competa usando un sistema operativo en tiempo real. Todo el código está escrito para la plataforma MCF5282Lite-ES, aunque sería fácil adaptarlo a otras plataformas, ya que el código se ha escrito buscando siempre la mayor portabilidad posible.

B.1. Introducción

Un PLC (*Programmable Logic Controller*) o autómata programable, es un sistema basado en microprocesador orientado al control industrial. Como los primeros sistemas de automatización industrial se realizaban con lógica de relés, los PLC se diseñaron para emular este funcionamiento. Para ello, el PLC está continuamente leyendo el estado de sus entradas, ejecutando un programa para calcular el valor que deben de tener sus salidas a partir de los valores de las entradas y actualizando dichas salidas. Tanto las entradas como las salidas son valores digitales de un bit (On/Off). Las entradas se conectan a pulsadores, interruptores, fines de carrera, etc. Las salidas se conectan a contactores, lámparas, etc. De esta forma el PLC es capaz de automatizar cualquier proceso industrial.[1]

Como los primeros sistemas de automatización industrial basados en relés eran realizados por electricistas, al ser el PLC un sustituto de estos sistemas, se diseñaron lenguajes de programación que emularan a este tipo de circuitos basados en lógica de relés. Los sistemas modernos tienen interfaces gráficas que permiten dibujar estos esquemas en la pantalla del ordenador, pero los antiguos disponían de un lenguaje de programación

[1] Los PLC que se pueden encontrar en el mercado disponen de funcionalidades adicionales a las del PLC que se va a implantar en este ejemplo. Además de trabajar con entradas y salidas digitales, pueden tener entradas y salidas analógicas, sistemas de comunicación para conectarse a otros dispositivos, etc.

textual para describir estos circuitos. Este lenguaje es similar al lenguaje ensamblador de los microprocesadores. La principal diferencia radica en el número de operandos de cada instrucción. En el caso de un microcontrolador como el ColdFire, las instrucciones son de dos direcciones, es decir, para realizar una operación se especifican dos operandos, de forma que uno de ellos es a la vez fuente y destino. Por ejemplo, si en el ColdFire se ejecuta la instrucción:

```
add.l D0, D1
```

La operación realizada por el microcontrolador es: `D0 = D0 + D1`.

El lenguaje de los PLC usa instrucciones de una sola dirección. En este caso el segundo operando y el resultado está implícito y es siempre un registro denominado **acumulador**. Por tanto, una instrucción del PLC podría ser:

```
A E0
```

Esta instrucción realizará un AND entre el valor de la entrada 0 (que es una entrada digital de un bit) y el valor almacenado en el acumulador, guardándose el resultado de la operación en el acumulador.

Las instrucciones soportadas por el PLC que se diseñará en esta sección serán las siguientes:

Instrucción	**Argumento**	**Funcionamiento**
`L ARG`	E0-E7, S0-S7	Carga un bit en el acumulador
`LN ARG`	E0-E7, S0-S7	Carga un bit **negándolo** en el acumulador
`A ARG`	E0-E7, S0-S7	Efectúa un AND entre el acumulador y el argumento
`O ARG`	E0-E7, S0-S7	Efectúa un OR entre el acumulador y el argumento
`= ARG`	S0-S7	Escribe el acumulador en la salida indicada en el argumento

El PLC dispondrá de 8 entradas, denominadas E0, E1, ... , E7 y 8 salidas denominadas S0, S1, ... , S7.

Nótese que las salidas también pueden leerse como argumento, lo cual es muy útil en el diseño de automatismos para realizar enclavamientos.

B.1.1. Programación del PLC

En la figura B.1 se muestra un ejemplo sencillo de programación del PLC. Por un lado se muestra el esquema de contactos y por otro el programa en lista de instrucciones que se introducirá en el PLC. Como se puede observar en la figura, el programa activará la salida S0 cuando estén activas a la vez las dos entradas E0 y E1. Por otro lado activará la salida S1

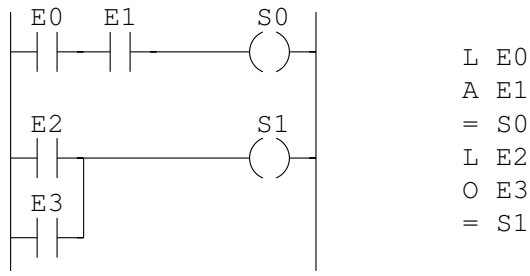


Figura B.1: Ejemplo de programa del PLC.

cuando alguna de las dos entradas E2 o E3 estén activas (o las dos a la vez).

B.2. Diseño con bucle de *scan*

El programa constará de un bucle de *scan* en el que se leerán las entradas, se ejecutará el programa del PLC para obtener el valor de las salidas y por último, cuando se haya terminado de ejecutar el programa del PLC, se actualizarán las salidas.

Para almacenar el programa en la memoria del PLC de la forma más compacta posible, en lugar de almacenar el código fuente, se almacenará un código máquina en el que se codificará cada instrucción mediante dos bytes, un primer byte para el código de operación y un segundo para el argumento.

El código de operación codificará la instrucción a ejecutar mediante un número de 8 bits. La elección de este número es totalmente arbitraria. Por ejemplo, siguiendo la tabla del apartado anterior, se asignará un 1 para la instrucción `L`, un 2 par la instrucción `LN` y así sucesivamente. El código de operación 0 se ha reservado para indicar el final del programa.

En el argumento, también de 8 bits, se codificará el número de la entrada o de la salida. Como el PLC dispondrá de 8 entradas y 8 salidas, una codificación cómoda puede consistir en asignar los números 0 al 7 a las salidas S0 a S7, y los números del 8 al 15 a las entradas E0 a E7. Siguiendo este esquema de codificación, la instrucción `L E0` se codificará con el número `0x01` para el código de operación y `0x08` para el argumento.

B.2.1. Consideraciones sobre el diseño del programa

La construcción del programa ha de dividirse en tareas. Una división clara es la siguiente:

- Lectura de las entradas.
- Ejecución del programa del PLC.
- Actualización de las salidas.

Las primera y la última tarea son muy sencillas: sólo dialogan con el *hardware* para obtener o enviar un byte con las señales digitales.

La segunda es un poco más compleja, ya que ha de ejecutar el programa del PLC. Como se ha dicho en la sección anterior, cada instrucción del PLC necesita dos bytes. Obviamente, lo más elegante es almacenar estos dos bytes en una estructura de datos:

```
typedef struct{
  uint8 codop; /* Operación */
  uint8 arg;   /* Argumento */
}INSTRUCCION;
```

Para almacenar el programa completo, se usará un vector de estructuras, el cual se inicializará con el programa codificado en código máquina:

```
  INSTRUCCION programa[] = {0x1, 0x8, 0x3, 0x9, 0x5, 0x0,
                           0x1, 0xa, 0x4, 0xb, 0x5, 0x1,
                           0x0, 0x0};
```

En el código anterior se ha codificado el programa mostrado en la figura B.1.

Para ejecutar el programa del PLC se recorre el vector y se obtendrá el código de operación y el argumento de cada instrucción. A continuación se analizará el código de operación y se realizarán las acciones pertinentes para ejecutar la operación requerida. Estas operaciones necesitan manipular bits dentro de una palabra, para lo que se usan las técnicas de programación en C de bajo nivel expuestas en el capítulo 2. El final del programa del PLC se indica mediante un código de operación especial, que en este caso es el 0.

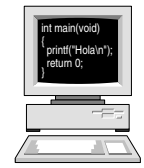
A continuación se muestra el ejemplo completo:

```
#include <stdio.h>

#include "mcf5282.h"      /* Defs. del HW del MCF5282 */
#include "M5282Lite-ES.h" /* Defs. de la tarjeta ES  */


typedef struct{
  uint8 codop; /* Operación */
  uint8 arg;   /* Argumento */
}INSTRUCCION;
```

```
uint8 LeeEntradas(void);
uint8 procesa(uint8 entradas, INSTRUCCION *pprog);
void  EscribeSalidas(uint8 salidas);

void main (void)  {
  uint8 entradas, salidas;
  INSTRUCCION programa[] = {0x1, 0x8, 0x3, 0x9, 0x5, 0x0,
                           0x1, 0xa, 0x4, 0xb, 0x5, 0x1,
                           0x0, 0x0};

  /* Se Inicializa la tarjeta de E/S y el display */
  InitM5282Lite_ES();
  InitDisplay();

  for(;;){
    /* Lectura de entradas */
    entradas = LeeEntradas();

    /* Se interpreta el programa del PLC */
    salidas = procesa(entradas, programa);

    /* Y se actualizan las salidas */
    EscribeSalidas(salidas);
  }
}

uint8 LeeEntradas(void)
{
  return LeePuertoA();
}

void EscribeSalidas(uint8 salidas)
{
  /* En el hardware de la tarjeta de ES las salidas son
     activas en nivel bajo. Por tanto antes de enviar
     las salidas hay que invertirlas con el operador ~ */
  EscribePuertoA(~salidas);
}

uint8 procesa(uint8 entradas, INSTRUCCION *pprog)
{
  uint8 codop, arg, valor_arg, valor_salida, acc=0;
  static uint8 salidas;
```

```
while( pprog->codop != 0x0){
  codop = pprog->codop;
  arg   = pprog->arg;
  if(arg < 8){ /* salida. */
    valor_arg = (salidas >> arg) & 0x01;
  }else if(arg < 16){ /* Si el nº de bit (arg) está
                         entre 8 y 15 es una entrada */
    valor_arg = (entradas >> (arg-8)) & 0x01;
  }else{
    /* Argumento inválido. Se aborta el programa del
       PLC */
    break;
  }

  switch(codop){
    /* Case 0 no hace falta, pues si se lee un codop = 0
       no se entra en el bucle */
  case 1: /* load */
    acc = valor_arg;
    break;
  case 2: /* load Negado*/
    acc = (~valor_arg) & 0x01;
    break;
  case 3: /* AND */
    acc &= valor_arg;
    break;
  case 4: /* OR */
    acc |= valor_arg;
    break;
  case 5: /* = */
    if(acc){
      /* Se coloca un 1 en el bit correspondiente a
         la salida */
      valor_salida = 1 << arg;
      salidas |= valor_salida; /* Puesta a uno */
    }else{
      /* Se coloca un 0 en el bit correspondiente a
         la salida */
      valor_salida = ~(1 << arg);
      salidas &= valor_salida; /* Puesta a cero */
    }
    break;
  default: /* Código de operación no válido */
    break; /* de momento no se hace ná de ná */
```

```
    }
    pprog++; /* Se pasa a la siguiente instrucción */
  }
  return salidas;
}
```



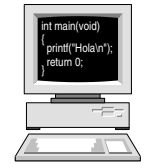
Realice los ejercicios 1 y 2.

Después de estudiar el programa quizás se esté preguntando el por qué usar dos tareas para realizar la entrada y salida de datos. Al fin y al cabo ambas tareas son tan simples que sólo tienen una línea. Ahora bien, aunque es cierto que podría incluirse el código de ambas tareas dentro del bucle de *scan*, el uso de estas dos tareas presenta varias ventajas:

- El código es mucho más claro. El bucle de *scan* sólo tiene llamadas a las distintas tareas, tal como se expuso en la sección 1.5.

- Se mejora la portabilidad, ya que todo el código que depende del *hardware* está dentro de esas dos tareas. Si se cambia de microcontrolador sólo será necesario modificar esas dos tareas.

- Se puede probar y depurar el programa sin necesidad de que esté el *hardware* definitivo disponible. Para ello basta con cambiar las funciones de entrada/salida por otras que simulen el funcionamiento de dicha entrada/salida y ejecutar el programa en otro *hardware*, como por ejemplo en un ordenador personal.



Realice el ejercicio 3.

B.3. Diseño con sistema *Foreground/Background*

El PLC diseñado en la sección anterior es demasiado simple, sobre todo si se compara con los modelos existentes en el mercado. En esta y en las siguientes secciones se van a añadir al PLC algunas funcionalidades de las existentes en los PLC reales. La primera de ellas va a consistir en añadir **interruptores horarios**.

Un interruptor horario no es más que un automatismo que permite conectar y desconectar un aparato a una determinada hora, lo cual es muy útil por ejemplo en aplicaciones de domótica para controlar el alumbrado, la calefacción, los aparatos que funcionan con tarifa nocturna, etc.

Por tanto, en esta sección se va a ampliar el PLC diseñado en la sección anterior para que disponga de 8 interruptores horarios, además de las 8 entradas que ya estaban disponibles. A los interruptores horarios se les denominará I0–I7 en el lenguaje de programación del PLC. Por tanto, las instrucciones del PLC serán ahora:

Instrucción	Argumento	Funcionamiento
`L ARG`	I0-I7, E0-E7, S0-S7	Carga un bit en el acumulador
`LN ARG`	I0-I7, E0-E7, S0-S7	Carga un bit **negándolo** en el acumulador
`A ARG`	I0-I7, E0-E7, S0-S7	Efectúa un AND entre el acumulador y el argumento
`O ARG`	I0-I7, E0-E7, S0-S7	Efectúa un OR entre el acumulador y el argumento
`= ARG`	S0-S7	Escribe el acumulador en la salida indicada en el argumento

Además, puesto que será necesario implantar un reloj en el PLC para implantar los interruptores horarios, se aprovechará para visualizar la hora del sistema en el *display* de la tarjeta ES.

B.3.1. Programación del PLC

En la figura B.2 se muestra un ejemplo de programación del PLC incluyendo un interruptor horario. Al igual que en el ejemplo de la sección anterior, se muestran el esquema de contactos y el programa en lista de instrucciones que se introducirá en el PLC. Como se puede observar, los interruptores horarios se tratan igual que las entradas normales. La única diferencia radica en que éstos se activan y desactivan automáticamente a una hora determinada, en lugar de ser activados por un interruptor externo al PLC.

Además del programa anterior, será necesario especificar la hora de conexión y de desconexión del interruptor horario. No obstante, esta información no se incluye en el programa, sino en los datos de configuración del autómata. Es decir, para cada interruptor horario se almacenarán una hora de conexión y otra de desconexión. En el ejemplo de la figura, la salida S2 se activará a las 18:00 horas y se desconectará a las 23:30 horas.

Las instrucciones del PLC serán iguales a la de la versión anterior, salvo que el argumento ahora ha de poder codificar los interruptores horarios. Para mantener la compatibilidad con los programas anteriores, se mantendrá la codificación anterior ampliada para los interruptores horarios. Así, los números 0x00 al 0x07 se corresponderán con las salidas S0 a S7, los números del 0x08 al 0x0F con las entradas E0 a E7 y los números 0x10 a 0x17 con los 8 interruptores horarios I0 a I7. Siguiendo este esquema de codificación, la instrucción `L I7` se codificará con los números `0x01` y `0x17`

B.3.2. Consideraciones sobre el diseño del programa

El programa mostrado a continuación se ha realizado a partir del mostrado en la sección anterior. No obstante, al ser algo más complejo se ha

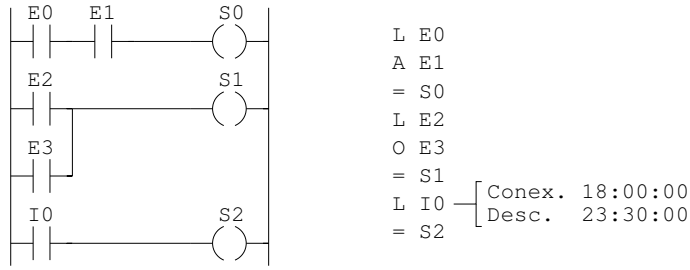


Figura B.2: Ejemplo de programa del PLC con interruptores horarios.

optado por dividirlo en dos módulos:

- Bucle de *scan* e intérprete del programa del PLC.
- Gestión de tiempo.

El primer módulo será prácticamente igual al desarrollado en la sección anterior. Tan solo habrá que añadir la lectura del estado de los interruptores horarios y modificar la decodificación de las instrucciones para tener en cuenta el nuevo formato y los nuevos argumentos.

El segundo módulo ha de gestionar la hora actual mediante el temporizador PIT0.[2] Para ello se configura el temporizador para generar una interrupción cada segundo y se asocia una rutina de atención a la interrupción a este temporizador, que será la tarea de segundo plano. Además de esta tarea, este módulo incluye dos tareas de primer plano: una para imprimir la hora en el *display* y otra para actualizar los interruptores horarios. Para conseguir una mayor eficiencia estas tareas sólo se ejecutan cuando cambia la hora actual, para lo cual se usan unas banderas que sincronizan estas tareas con la rutina de atención a la interrupción.

Para conseguir una **encapsulación** de los datos, el resto de módulos no acceden directamente a las estructuras de datos de este módulo. Se han creado para ello funciones específicas para realizar el interfaz con el módulo. Así por ejemplo, para leer el estado de los interruptores horarios desde el intérprete del programa del PLC se ha creado la función:

```
BYTE8 EstadoIntHor();
```

La cual lo único que hace es devolver el byte de estado de los interruptores horarios.

[2] La documentación completa de este temporizador está en el capítulo 19 del manual [FreeScale, 2005].

Estructuras de datos

Para gestionar los interruptores horarios, hacen falta varios datos. En primer lugar será necesario almacenar la hora actual como horas, minutos y segundos. Lo más cómodo en este caso es usar una estructura con tres bytes, uno para cada dato:

```
typedef struct{
        uint8 hor;
        uint8 min;
        uint8 seg;
}HORA;
```

Además, para cada interruptor es necesario almacenar dos horas, la de conexión y la de desconexión. Nuevamente lo más cómodo es definir una estructura para cada interruptor horario que contenga ambas horas:

```
typedef struct{
        HORA arranque;
        HORA paro;
}INT_HOR;
```

Para almacenar las horas de conexión y desconexión de los ocho interruptores horarios se usa un vector de estas estructuras. La inicialización de este vector definirá el comportamiento de los interruptores horarios. Por último, para que la integración de los interruptores horarios con el resto de entradas en el intérprete del programa del PLC sea lo más fácil posible, se genera un byte para almacenar el estado de los ocho interruptores horarios. Así, en el bit 0 se almacenará el estado del interruptor cero, el bit 1 el del 1 y así sucesivamente.

En primer lugar se muestra el módulo que contiene el bucle de *scan* y las tareas encargadas de leer las entradas, actualizar las salidas y de interpretar el programa del PLC:

```
#include <stdio.h>

#include "mcf5282.h"      /* Defs. del HW del MCF5282 */
#include "M5282Lite-ES.h" /* Defs. de la tarjeta ES  */


typedef struct{
  uint8 codop; /* Operación */
  uint8 arg;   /* Argumento */
}INSTRUCCION;

uint8 LeeEntradas(void);
uint8 procesa(uint8 entradas, INSTRUCCION *pprog);
```

```
void  EscribeSalidas(uint8 salidas);

void main (void)  {
  uint8 entradas, salidas;
  INSTRUCCION programa[] = {0x1, 0x8, 0x3, 0x9, 0x5, 0x0,
                           0x1, 0xa, 0x4, 0xb, 0x5, 0x1,
                           0x1, 0x10,0x5, 0x2, 0x0, 0x0};

  /* Se Inicializa la tarjeta de E/S y el display */
  InitM5282Lite_ES();
  InitDisplay();

  InitTemporizador(); /* Se inicializa en temporizador */
  Enable();     /* Se habilitan las interrupciones */

  for(;;){
    /* Lectura de entradas */
    entradas = LeeEntradas();
    /* Se actualizan los interruptores horarios */
    ActualizaIntHorarios();
    /* Se interpreta el programa del PLC */
    salidas = procesa(entradas, programa);
    /* Se actualizan las salidas */
    EscribeSalidas(salidas);
    /* Se imprime ha hora actual en el display */
    ImpHora();
  }
}

uint8 LeeEntradas(void)
{
  return LeePuertoA();
}

void EscribeSalidas(uint8 salidas)
{
  /* En el hardware de la tarjeta de ES las salidas son
     activas en nivel bajo. Por tanto antes de enviar
     las salidas hay que invertirlas con el operador ~ */
  EscribePuertoA(~salidas);
}

uint8 procesa(uint8 entradas, INSTRUCCION *pprog)
{
```

```
uint8 codop, arg, valor_arg, valor_salida, acc=0;
static uint8 salidas;
uint8 int_hor;

int_hor = IntHorarios(); /* Se lee el estado de los
                            interruptores horarios */
while( pprog->codop != 0x0){
  codop = pprog->codop;
  arg   = pprog->arg;
  if(arg < 8){ /* salida. */
    valor_arg = (salidas >> arg) & 0x01;
  }else if(arg < 16){ /* Si el nº de bit (arg) está
                         entre 8 y 15 es una entrada */
    valor_arg = (entradas >> (arg-8)) & 0x01;
  }else if(arg < 0x18){ /* Es un interruptor horario */
    valor_arg = (int_hor >> (arg & 0x7)) & 0x01;
  }else{
    /* Argumento inválido. Se aborta el programa */
    break;
  }

  switch(codop){
    /* Case 0 no hace falta, pues si se lee un codop = 0
       no se entra en el bucle */
  case 1: /* load */
    acc = valor_arg;
    break;
  case 2: /* load Negado*/
    acc = (~valor_arg) & 0x01;
    break;
  case 3: /* AND */
    acc &= valor_arg;
    break;
  case 4: /* OR */
    acc |= valor_arg;
    break;
  case 5: /* = */
    if(acc){
      /* Se coloca un 1 en el bit correspondiente a
         la salida */
      valor_salida = 1 << arg;
      salidas |= valor_salida; /* Puesta a uno */
    }else{
      /* Se coloca un 0 en el bit correspondiente a
```

```
            la salida */
          valor_salida = ~(1 << arg);
          salidas &= valor_salida; /* Puesta a cero */
        }
        break;
      default: /* Código de operación no válido */
        break; /* de momento no se hace ná de ná */
      }
      pprog++; /* Se pasa a la siguiente instrucción */
    }
    return salidas;
}
```

Como se puede apreciar la única tarea que ha cambiado ha sido procesa, en donde se ha incluido la llamada a la función IntHorarios para obtener el estado de los interruptores horarios y la decodificación del argumento de entrada para tener en cuenta si dicho argumento indica un interruptor horario.

El otro cambio respecto a la versión anterior ha sido en la inicialización del sistema, realizada como siempre antes de entrar en el bucle de *scan*. En esta inicialización se llama a la función InitTemporizador para inicializar el temporizador encargado de gestionar la hora y a la función Enable para habilitar las interrupciones.

El módulo que gestiona el temporizador y los interruptores horarios se muestra a continuación. En primer lugar se muestra la cabecera:

```
#ifndef TEMPORIZADOR_H
#define TEMPORIZADOR_H

/* Inicializa el timer 0 para que funcione por
 * interrupciones, generando una interrupción cada
 * segundo.
 */
void InitTemporizador(void);

/* Tarea para imprimir la hora en el display */
void ImpHora(void);

/* Tarea para actualizar el estado de los interruptores
 * horarios. Ha de llamarse dentro del bucle de scan.
 */
void ActualizaIntHorarios(void);

/* Esta función devuelve el estado de los interruptores
 * horarios.
```

```
 * Se devuelve un byte, en el cual el bit menos
 * significativo es el estado del interruptor I0. Si
 * dicho bit está a 0, el interruptor I0 está apagado.
 */
uint8 IntHorarios();

#endif
```

Y a continuación se muestra el código:

```
#include <stdio.h>

#include "mcf5282.h"      /* Defs. del HW del MCF5282 */
#include "M5282Lite-ES.h" /* Defs. de la tarjeta ES  */

#include "interrupciones.h"
#include "temporizador.h"

typedef struct{
    uint8 hor;
    uint8 min;
    uint8 seg;
}HORA;

typedef struct{
    HORA arranque;
    HORA paro;
}INT_HOR;

/* Prototipos de las funciones privadas */
int  ComparaHora(HORA a, HORA b);
void InterruptTimer0(void);

/* bits de salida de los interruptores horarios */
static uint8 sal_int_hor = 0;

/* Almacena la hora actual */
static HORA hora_act={0,0,0};

/* Banderas para sincronizar las tareas de primer plano */
static uint8 band_ActualizaIntHorarios = 0;
stacit uint8 band_ImpHora = 1; /* Se pone a 1 para que
                           se imprima la hora inicial */

/* Inicializa el PIT0 para generar una interrupción cada
```

```
 * segundo. Para ello, teniendo en cuenta que el reloj
 * del sistema es de 32MHz, se usa como preescaler el
 * valor 2048, ya que 32E6/2048 = 15625 (0x3d09). Con
 * otros valores de preescaler no se obtiene un resultado
 * exacto por lo que el reloj adelantaría o atrasaría.
 */

void InitTemporizador(void)
{
  /* Se copia el vector de interrupción */
  *(void**)(0x20000000+(64+55)*4)=(void*)InterruptTimer0;

  MCF_PIT0_PMR  =  15624; /* Módulo de cuenta */
  MCF_PIT0_PCSR = 0x0a0f; /* Configura el preescaler
      y habilita el timer y las interrupciones */
  MCF_INTC0_ICR55 = 0x08; /* Level 1, prioridad 0 */
  /* Se desenmascara la INT55 */
  MCF_INTC0_IMRH &= ~MCF_INTC_IMRH_INT55;
}

__declspec(interrupt) void InterruptTimer0(void)
{
  hora_act.seg++;
  if(hora_act.seg == 60){
    hora_act.seg = 0;
    hora_act.min++;
    if(hora_act.min == 60){
      hora_act.min = 0;
      hora_act.hor++;
      if(hora_act.hor == 24){
        hora_act.hor = 0;
      }
    }
  }

  /* Lanza las tareas de primer plano */
  band_ImpHora = 1;
  band_ActualizaIntHorarios = 1;

  /* Borra el flag de interrupción reescribiendo el
     módulo de cuenta */
  MCF_PIT0_PMR =   15624;
}
```

```
void ImpHora(void)
{
  char buf[17];
  HORA copia_hora;
  uint8 est_int;

  if(band_ImpHora == 1){ /* Ha cambiado la hora*/
    /* Se obtiene una copia de la hora actual, ya que
       ésta se comparte con la interrupción. */
    est_int = Disable();
    copia_hora = hora_act;
    if(est_int){
      Enable();
    }
    /* Se imprime la hora en una cadena */
    sprintf(buf, "   %02d:%02d:%02d", copia_hora.hor,
            copia_hora.min, copia_hora.seg);
    /* Y se envía al display */
    DisplayGotoLinea0();
    PutsDisplay(buf);

    /* Por último se pone la bandera a cero para no
       volver a ejecutar la tarea hasta la siguiente
       interrupción */
    band_ImpHora = 0;
  }
}

void ActualizaIntHorarios()
{
  /* Vector con las horas de conexión y desconexión
     de los 8 interruptores horarios */
  static INT_HOR int_hor[8] = {18,00,00, 23,59,59,
         00,00,00, 08,00,00,   19,00,00, 01,00,01,
         01,00,00, 03,00,00,   18,00,00, 23,59,59,
         00,00,00, 08,00,00,   19,00,00, 01,00,01,
         00,00,00, 00,00,00};
  int i;
  HORA copia_hora;

  if(band_ActualizaIntHorarios == 1){
    /* Se obtiene una copia de la hora actual, ya que
       ésta se comparte con la interrupción. */
    est_int = Disable();
```

```
    copia_hora = hora_act;
    if(est_int){
      Enable();
    }
    for(i=0; i<8; i++){
      if(ComparaHora(hora_act, int_hor[i].arranque)==0){
        /* encender */
        sal_int_hor |= 1<<i;
      }
      if(ComparaHora(hora_act, int_hor[i].paro)==0){
        /* apagar */
        sal_int_hor &= ~(1<<i);
      }
    }
    band_ActualizaIntHorarios = 0;
  }
}

/* Compara dos horas a y b. Si a es mayor que b, devuelve
 * un 1, si es menor un -1 y si son iguales un 0
 */
int ComparaHora(HORA a, HORA b)
{
  if(a.hor > b.hor){
    return 1;
  }else if(a.hor < b.hor){
    return -1;
  }else if(a.min > b.min){
    return 1;
  }else if(a.min < b.min){
    return -1;
  }else if(a.seg > b.seg){
    return 1;
  }else if(a.seg < b.seg){
    return -1;
  }else{                    // iguales
    return 0;
  }
}

uint8 IntHorarios()
{
  return sal_int_hor;
}
```

Por último, en este ejemplo es necesario incluir dos funciones para habilitar e inhabilitar las interrupciones. Ambas funciones se han expuesto en detalle en la sección 3.3.3, aunque se incluyen aquí para mayor comodidad:

```
#ifndef INTERRUPCIONES_H
#define INTERRUPCIONES_H
#include "mcf5282.h"

inline void Enable(void)
{
  asm
  {
    move.w  #0x2000,SR
  }
}

inline uint8 Disable(void)
{
  uint8 ret;
  asm
  {
    move.l d3, -(a7) // Guarda d3 en la pila
    move.w SR, d3    // Lee el SR
    asr    #8, d3    // Pasamos el campo I al bit 0
    andi   #7, d3    // y lo dejamos a él solito
    move.w d3, ret   // Lo copiamos a la variable de retorno
    move.l (a7)+, d3
    move.w  #0x2700,SR // Inhabilitamos Interrupciones
  }

  return !ret; // Si vale 0 es que estaban habilitadas
               // pero la función ha de devolver 1 en
               // este caso
}
#endif
```

B.4. Diseño basado en el sistema operativo en tiempo real FreeRTOS

En esta sección se va a volver a realizar el mismo sistema de la sección anterior pero usando un sistema operativo en tiempo real. Por tanto las tareas serán las mismas que en el ejemplo anterior, salvo que ahora en lugar de estar controladas por un bucle de *scan*, estarán controladas por el sistema operativo en tiempo real.

La tarea encargada de ejecutar el programa del PLC se ejecutará periódicamente con un periodo igual a 10 ms (2 ticks de reloj). Por otro lado, para conseguir una temporización más precisa, se programará un temporizador con un periodo de 1 segundo y las tareas encargadas de imprimir la hora y de actualizar los interruptores horarios se sincronizarán con esta interrupción de tiempo

B.4.1. Consideraciones sobre el diseño del programa

La sincronización entre la interrupción del temporizador de 1 segundo y las tareas encargadas de imprimir la hora y actualizar los interruptores horarios se realizará mediante dos semáforos. Dichos semáforos serán privados al módulo de gestión de tiempo, por lo que se inicializarán en la función de inicialización del módulo `InitTemporizador`. El código de este módulo se muestra a continuación. La cabecera es igual que la mostrada en el apartado anterior:

```
#include <stdio.h>

#include "mcf5282.h"      /* Defs. del HW del MCF5282 */
#include "M5282Lite-ES.h" /* Defs. de la tarjeta ES  */

#include "interrupciones.h"
#include "temporizador.h"

#include "FreeRTOS.h" /* Defs. del S.O.T.R. */
#include "task.h"
#include "semphr.h"

typedef struct{
    uint8 hor;
    uint8 min;
    uint8 seg;
}HORA;

typedef struct{
    HORA arranque;
    HORA paro;
}INT_HOR;

/* Prototipos de las funciones privadas */
int  ComparaHora(HORA a, HORA b);
void InterruptTimer0(void);

/* bits de salida de los interruptores horarios */
```

```
static uint8 sal_int_hor = 0;

/* Almacena la hora actual */
static HORA hora_act={0,0,0};

/* Semáforos para sincronizar las tareas */
static xSemaphoreHandle sem_ActualizaIntHorarios;
static xSemaphoreHandle sem_ImpHora;

/* Inicializa el PIT0 para generar una interrupción cada
 * segundo. Para ello, teniendo en cuenta que el reloj
 * del sistema es de 32MHz, se usa como preescaler el
 * valor 2048, ya que 32E6/2048 = 15625 (0x3d09). Con
 * otros valores de preescaler no se obtiene un resultado
 * exacto por lo que el reloj adelantaría o atrasaría.
 */

void InitTemporizador(void)
{
  /* Se inicializan los semáforos */
  vSemaphoreCreateBinary(sem_ActualizaIntHorarios);
  vsemaphoreCreateBinary(sem_ImpHora);

  /* Se copia el vector de interrupción */
  *(void**)(0x20000000+(64+55)*4)=(void*)InterruptTimer0;

  MCF_PIT0_PMR  =  15624; /* Módulo de cuenta */
  MCF_PIT0_PCSR = 0x0a0f; /* Configura el preescaler
      y habilita el timer y las interrupciones */
  MCF_INTC0_ICR55 = 0x08; /* Level 1, prioridad 0 */
  /* Se desenmascara la INT55 */
  MCF_INTC0_IMRH &= ~MCF_INTC_IMRH_INT55;
}

__declspec(interrupt) void InterruptTimer0(void)
{
  portBASE_TYPE xTaskWoken = pdFALSE;

  hora_act.seg++;
  if(hora_act.seg == 60){
    hora_act.seg = 0;
    hora_act.min++;
    if(hora_act.min == 60){
      hora_act.min = 0;
```

```
      hora_act.hor++;
      if(hora_act.hor == 24){
        hora_act.hor = 0;
      }
    }
  }

  /* Lanza las tareas */
  xTaskWoken = xSemaphoreGiveFromISR(sem_ImpHora,
                                     xTaskWoken);
  xTaskWoken = xSemaphoreGiveFromISR(sem_ActualizaIntHorarios,
                                     xTaskWoken);

  /* Borra el flag de interrupción reescribiendo el
     módulo de cuenta */
  MCF_PIT0_PMR =   15624;

  if(xTaskWoken == pdTRUE){
    taskYIELD (); /* Si el semáforo ha despertado
                     una tarea, se fuerza un cambio
                     de contexto */
  }
}

void ImpHora(void)
{
  char buf[17];
  HORA copia_hora;
  uint8 est_int;

  while(1){
    if(xSemaphoreTake(sem_ImpHora,
                      (portTickType)0xFFFF)== pdTRUE){
      /* Se liberó el semáforo => ha cambiado la hora */
      /* Se obtiene una copia de la hora actual, ya que
         ésta se comparte con la interrupción. */
      est_int = Disable();
      copia_hora = hora_act;
      if(est_int){
        Enable();
      }
      /* Se imprime la hora en una cadena */
      sprintf(buf, "   %02d:%02d:%02d", copia_hora.hor,
              copia_hora.min, copia_hora.seg);
```

```
      /* Y se envía al display */
      DisplayGotoLinea0();
      PutsDisplay(buf);
  }
}

void ActualizaIntHorarios()
{
  /* Vector con las horas de conexión y desconexión
     de los 8 interruptores horarios */
  static INT_HOR int_hor[8] = {18,00,00, 23,59,59,
         00,00,00, 08,00,00,   19,00,00, 01,00,01,
         01,00,00, 03,00,00,   18,00,00, 23,59,59,
         00,00,00, 08,00,00,   19,00,00, 01,00,01,
         00,00,00, 00,00,00};
  int i;
  HORA copia_hora;

  if(xSemaphoreTake(sem__ActualizaIntHorarios,
                    (portTickType)0xFFFF)== pdTRUE){
    /* Se liberó el semáforo => ha cambiado la hora */
    /* Se obtiene una copia de la hora actual, ya que
       ésta se comparte con la interrupción. */
    est_int = Disable();
    copia_hora = hora_act;
    if(est_int){
      Enable();
    }
    for(i=0; i<8; i++){
      if(ComparaHora(hora_act, int_hor[i].arranque)==0){
        /* encender */
        sal_int_hor |= 1<<i;
      }
      if(ComparaHora(hora_act, int_hor[i].paro)==0){
        /* apagar */
        sal_int_hor &= ~(1<<i);
      }
    }
  }
}

/* Compara dos horas a y b. Si a es mayor que b, devuelve
 * un 1, si es menor un -1 y si son iguales un 0
 */
```

```
int ComparaHora(HORA a, HORA b)
{
  if(a.hor > b.hor){
    return 1;
  }else if(a.hor < b.hor){
    return -1;
  }else if(a.min > b.min){
    return 1;
  }else if(a.min < b.min){
    return -1;
  }else if(a.seg > b.seg){
    return 1;
  }else if(a.seg < b.seg){
    return -1;
  }else{                    // iguales
    return 0;
  }
}

uint8 IntHorarios()
{
  return sal_int_hor;
}
```

Como se puede apreciar, la rutina de interrupción libera los semáforos `sem_ActualizaIntHorarios` y `sem_ImpHora` y las tareas se limitan a esperar la liberación de dichos semáforos, realizando su cometido cuando esto ocurra y volviéndose a bloquear a la espera de la liberación de su semáforo. Esto permite que dichas tareas sólo usen la CPU cuando sea necesario, haciendo que el sistema sea mucho más eficiente que el basado en banderas.

A continuación se muestra el módulo principal. La función `main` se limita ahora a inicializar el sistema, crear las tareas y arrancar el planificador. El bucle de *scan* se ha implantado en una tarea que se ejecuta periódicamente con un periodo de 10 ms. Nótese que ahora en el bucle de *scan* no se llama a las tareas encargadas de imprimir la hora y de actualizar los interruptores horarios. Éstas serán llamadas automáticamente por el planificador del sistema operativo cada vez que la rutina de interrupción libere los semáforos que las mantienen bloqueadas.

```
#include <stdio.h>

#include "mcf5282.h"       /* Defs. del HW del MCF5282 */
#include "M5282Lite-ES.h" /* Defs. de la tarjeta ES  */

#include "FreeRTOS.h" /* Defs. del S.O.T.R. */
```

```
#include "task.h"

/* Definiciones del Kernel*/
#define BUCLE_SCAN_PRIO  (tskIDLE_PRIORITY + 1)
#define IMP_HORA_PRIO    (tskIDLE_PRIORITY + 2)
#define ACT_INT_HOR_PRIO (tskIDLE_PRIORITY + 3)

/* Las pilas necesitan 1 kb (256 kWords) si se usa sprintf */
#define BUCLE_SCAN_STACK_SIZE  256
#define IMP_HORA_STACK_SIZE    256
#define ACT_INT_HOR_STACK_SIZE 256

typedef struct{
  uint8 codop; /* Operación */
  uint8 arg;   /* Argumento */
}INSTRUCCION;

uint8 LeeEntradas(void);
uint8 procesa(uint8 entradas, INSTRUCCION *pprog);
void  EscribeSalidas(uint8 salidas);

void main (void)
{
  /* Se Inicializa la tarjeta de E/S y el display */
  InitM5282Lite_ES();
  InitDisplay();

  InitTemporizador(); /* Se inicializa en temporizador */
  Enable();           /* Se habilitan las interrupciones */

  /* Se crean las tareas */
  xTaskCreate(BucleScan,
              (const signed portCHAR * const) "BucScan",
              BUCLE_SCAN_STACK_SIZE, NULL, BUCLE_SCAN_PRIO,
              (xTaskHandle * ) NULL );
  xTaskCreate(ImpHora,
              (const signed portCHAR * const) "ImpHora",
              IMP_HORA_STACK_SIZE, NULL, IMP_HORA_PRIO,
              (xTaskHandle * ) NULL );
  xTaskCreate(ActualizaIntHorarios,
              (const signed portCHAR * const) "ActHor",
              ACT_INT_HOR_STACK_SIZE, NULL,  ACT_INT_HOR_PRIO,
              (xTaskHandle * ) NULL );
```

```
  /* Y por último se arranca el planificador */
  vTaskStartScheduler();

  /* Se supone que aquí no se llegará nunca */
}

void BucleScan(void)
{
  uint8 entradas, salidas;
  INSTRUCCION programa[] = {0x1, 0x8, 0x3, 0x9, 0x5, 0x0,
                           0x1, 0xa, 0x4, 0xb, 0x5, 0x1,
                           0x1, 0x10,0x5, 0x2, 0x0, 0x0};
  portTickType xLastWakeTime ;
  portTickType xPeriodo ;

  xPeriodo = 10/ configTICK_RATE_MS ; /* Periodo 10 ms */

  /* Inicializa xLastWakeTime con el tiempo actual */
  xLastWakeTime = xTaskGetTickCount ();
  while (1){
    /* Espera el siguiente periodo */
    vTaskDelayUntil (&xLastWakeTime , xPeriodo );

    /* Lectura de entradas */
    entradas = LeeEntradas();
    /* Se interpreta el programa del PLC */
    salidas = procesa(entradas, programa);
    /* Se actualizan las salidas */
    EscribeSalidas(salidas);
  }
}

uint8 LeeEntradas(void)
{
  return LeePuertoA();
}

void EscribeSalidas(uint8 salidas)
{
  /* En el hardware de la tarjeta de ES las salidas son
     activas en nivel bajo. Por tanto antes de enviar
     las salidas hay que invertirlas con el operador ~ */
  EscribePuertoA(~salidas);
}
```

```
uint8 procesa(uint8 entradas, INSTRUCCION *pprog)
{
  uint8 codop, arg, valor_arg, valor_salida, acc=0;
  static uint8 salidas;
  uint8 int_hor;

  int_hor = IntHorarios(); /* Se lee el estado de los
                              interruptores horarios */
  while( pprog->codop != 0x0){
    codop = pprog->codop;
    arg   = pprog->arg;
    if(arg < 8){ /* salida. */
      valor_arg = (salidas >> arg) & 0x01;
    }else if(arg < 16){ /* Si el nº de bit (arg) está
                           entre 8 y 15 es una entrada */
      valor_arg = (entradas >> (arg-8)) & 0x01;
    }else if(arg < 0x18){ /* Es un interruptor horario */
      valor_arg = (int_hor >> (arg & 0x7)) & 0x01;
    }else{
      /* Argumento inválido. Se aborta el programa */
      break;
    }

    switch(codop){
      /* Case 0 no hace falta, pues si se lee un codop = 0
         no se entra en el bucle */
    case 1: /* load */
      acc = valor_arg;
      break;
    case 2: /* load Negado*/
      acc = (~valor_arg) & 0x01;
      break;
    case 3: /* AND */
      acc &= valor_arg;
      break;
    case 4: /* OR */
      acc |= valor_arg;
      break;
    case 5: /* = */
      if(acc){
        /* Se coloca un 1 en el bit correspondiente a
           la salida */
        valor_salida = 1 << arg;
```

```
        salidas |= valor_salida; /* Puesta a uno */
      }else{
        /* Se coloca un 0 en el bit correspondiente a
           la salida */
        valor_salida = ~(1 << arg);
        salidas &= valor_salida; /* Puesta a cero */
      }
      break;
    default: /* Código de operación no válido */
      break; /* de momento no se hace ná de ná */
    }
    pprog++; /* Se pasa a la siguiente instrucción */
  }
  return salidas;
}
```

B.5. Ejercicios

1. El PLC diseñado tiene pocas instrucciones. Se podría ampliar con las siguientes:

Mnemónico	**Argumento**	**Funcionamiento**
`AN ARG`	E0-E7, S0-S7	Efectúa un AND entre el acumulador y el argumento **negado**
`ON ARG`	E0-E7, S0-S7	Efectúa un OR entre el acumulador y el argumento **negado**
`=N ARG`	S0-S7	Escribe el acumulador **negado** en la salida indicada en el argumento

2. El programa del PLC mostrado en la figura B.1 es muy simple. Modifíquelo para implantar un circuito marcha-paro.

3. Modifique el programa anterior para poder simular el PLC en un ordenador personal. Para ello la tarea `LeeEntradas` pedirá al usuario el valor de las entradas digitales (con un `scanf` por ejemplo) y la tarea `EscribeSalidas` mostrará por pantalla el valor de las salidas.

Bibliografía

[Auslander et al., 1996] Auslander, D. M., Ridgely, J. R., and Jones, J. C. (1996). Real-time software for implementation of feedback control. In Levine, W. S., editor, *The Control Handbook*, chapter 17, pages 323–343. CRC Press and IEEE Press.

[Barry, 2007] Barry, R. (2007). FreeRTOS manual. Disponible Online en: www.freertos.org.

[FreeScale, 2005] FreeScale (2005). *MCF5282 ColdFire Microcontroller User's Manual*. FreeScale, 3 edition.

[Kernighan and Ritchie, 1991] Kernighan, B. W. and Ritchie, D. M. (1991). *El lenguaje de programación C*. Prentice Hall, 2 edition.

[Labrosse, 2002] Labrosse, J. J. (2002). *MicroC/OS-II. The Real-Time Kernel*. CMP books, 2 edition.

[Simon, 1999] Simon, D. E. (1999). *An Embedded Software Primer*. Addison Wesley.

Índice alfabético