Sistemas operativos empotrados

José María Gómez Cama

PID_00177263

Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción..... 5

Objetivos..... 7

1. Un caso básico introductorio..... 9
1.1. Sensor de temperatura 10
1.2. Definiciones 12
1.3. Etapa de inicialización 14
1.4. Bucle principal 16

2. Elementos de un sistema operativo..... 22

3. El gestor de procesos..... 23
3.1. Sistemas operativos Round-Robin 23
3.1.1. La implementación 25
3.1.2. Posibles mejoras: bucle síncrono 29
3.2. Sistemas operativos basados en eventos 35
3.2.1. Implementación 39
3.2.2. Planificación por prioridades 45
3.2.3. Planificación por envejecimiento 49
3.3. Sistemas operativos multitarea 50
3.3.1. Planificación colaborativa 50
3.3.2. Planificación anticipativa (*preemptive*) 51
3.3.3. Mútex 58
3.3.4. Acceso al núcleo 60
3.3.5. Un abrazo mortal 61
3.3.6. Prioridades 62

4. Sistemas en tiempo real..... 65

5. Controladores de periféricos..... 67
5.1. Información del hardware 68
5.2. Programación del controlador 75

6. Sistemas operativos y librerías de soporte..... 80
6.1. TinyOS 80
6.2. FreeRTOS 81
6.3. Contiki 81
6.4. QNX 82
6.5. RTEMS 83

Resumen..... 85

Bibliografía..... 87

Anexo..... 88

Introducción

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

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

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

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

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

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

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

Por otro lado, desde el punto de vista de la máquina virtual, podríamos tener los siguientes:

- La plataforma es tan sencilla que la maquina virtual que proporciona el SO es más compleja que la original.

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

En estos casos, es evidente que el SO pierde frente a una programación ad hoc, lo que no significa que no se utilicen elementos de un SO para ayudar en la programación. A modo de ejemplo –si la aplicación es sencilla y, por lo tanto, el número de tareas que hay que realizar no es alto; la arquitectura del sistema empotrado es reducida y los periféricos no tienen una gran complejidad (ADC, GPIO, USART, I2C, etc.)–, es muy común realizar una programación basada en un gestor de tareas en bucle.

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

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

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

Objetivos

El estudio de este módulo didáctico os permitirá alcanzar los objetivos siguientes:

1. Conocer los elementos básicos de un sistema operativo para sistemas empotrados.

2. Dominar las metodologías más habituales para la gestión de procesos en sistemas empotrados.

3. Entender el concepto de controlador de periféricos.

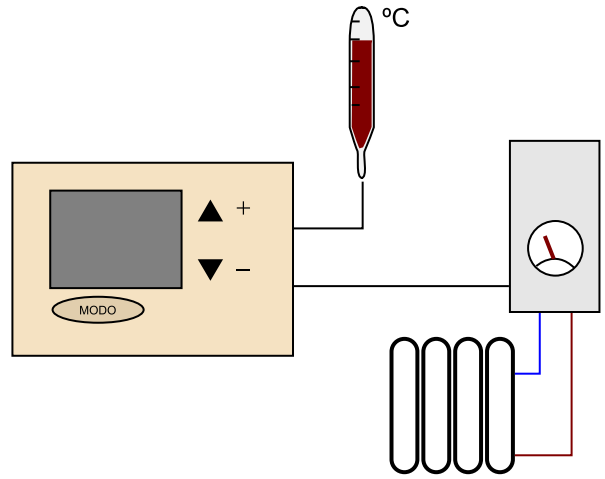
4. Saber desarrollar un controlador de periféricos a partir de su hoja de características.

5. Conocer algunos de sistemas operativos de propósito específico.

1. Un caso básico introductorio

Supongamos que nos piden diseñar un sistema empotrado que controle la temperatura de una habitación mediante una calefacción y que también proporcione información al usuario de dicha temperatura. En la figura siguiente se muestra un ejemplo:

Esquema de un sistema de calefacción



Las tareas que hay que realizar son:

- Tomar la temperatura de la habitación.
- Gestionar los botones de temperatura de consigna.
- Gestionar el botón de modo de visualización (temp. real o temp. consigna).
- Mostrar la temperatura en la pantalla.
- Actuar el relé para encender/apagar la caldera.

Partiendo de estas tareas, el sistema constará de los siguientes elementos:

- Sensor de temperatura con una resolución de una décima de grado.

- Pantalla para mostrar la temperatura actual y la de consigna.

- Botonera:
  - Dos botones para subir o bajar la temperatura de consigna.
  - Un botón para cambiar de visualización de temperatura actual a temperatura de consigna.

- Relé para controlar la puesta en marcha de la calefacción.

- Microcontrolador.

Teniendo en cuenta los tiempos de respuesta de un sistema de calefacción, que pueden ir de minutos a horas, es evidente que no se necesita un sistema especialmente rápido. Por este motivo, los procesos se pueden llevar a cabo de manera secuencial.

A partir de estos elementos, podemos determinar que, en lo que atañe a los periféricos de nuestro microcontrolador, deberemos utilizar:

- **Sensor de temperatura**: convertidor de analógico a digital conectado a sensor de temperatura.

- **Pantalla**: interfaz serie.

- **Botonera**: pines de entrada de propósito general.

- **Relé**: pin de salida de propósito general.

Con todo ello, podemos ver que la complejidad de la aplicación es reducida, lo que permite realizar una implementación ad hoc. En este caso, los pasos que deberíamos realizar serían:

- Definición de los tipos y las estructuras necesarias.

- Inicializar el sistema:
  - Configurar el reloj.
  - Configurar los puertos de entrada y salida.
  - Inicializar las variables principales.

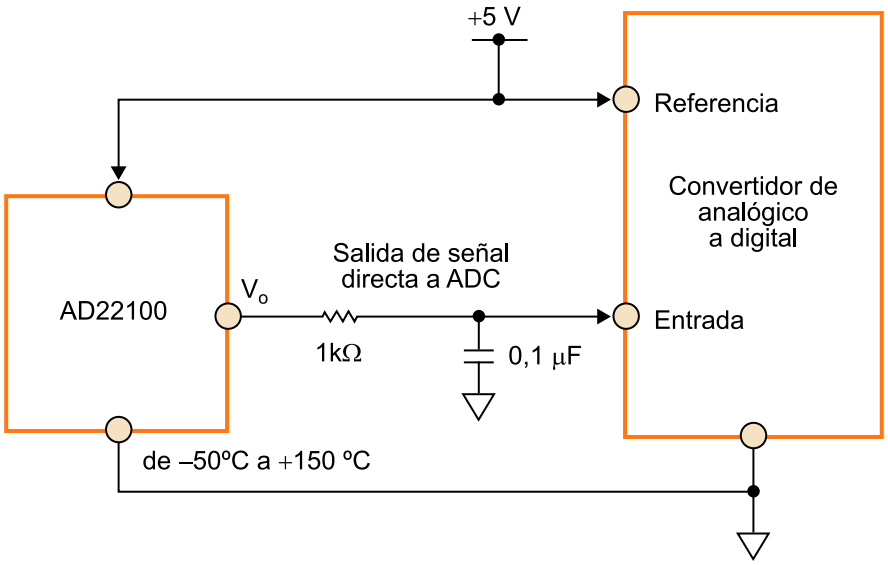
- Bucle principal:
  - Consultar las entradas.
  - Procesar la información.
  - Generar las salidas.

La implementación de los diferentes elementos se presenta a continuación, empezando por el sensor de temperatura, que requiere un procesado específico.

1.1. Sensor de temperatura

Para el caso del sensor de temperatura, podemos hacer uso del circuito de Analog Devices AD22100, que nos proporciona una variación de una tensión en función de la temperatura. El esquema se muestra en este esquema:

Esquema del sensor de temperatura



Con este circuito, se tiene una variación de 22,5 m V/°C. Suponiendo que el ADC disponible es de 12 bits, podemos obtener los valores para las diferentes temperaturas a partir de las ecuaciones siguientes:

$$\begin{cases} V[V] = \frac{22{,}5\ mV}{^\circ C} T[^\circ C] + 1{,}375\ V \\ V[ADC] = (2^{12\,\text{bits}} - 1) \cdot \frac{V[V]}{5\ V} \end{cases} \Rightarrow \text{Valor}[ADC] = \lfloor 18{,}43 \cdot T[^\circ C] + 1.126{,}125 \rfloor$$

Donde $[x]$ es el valor entero más próximo a 0 con respecto a x (equivalente a la función *floor* en C). Así pues, los voltajes que obtendremos para las diferentes temperaturas serán:

Temperatura (C)	Voltaje (V)	Valor (ADC)
-50	0,25	204
-25	0,8125	665
0	1,375	1.126
25	1,9375	1.586
50	2,5	2.047
100	3,625	2.968
150	4,75	3.890

Para hacer el paso inverso de manera sencilla en un microcontrolador, en principio, nos interesaría obtener una temperatura con precisión de décima de grados. Esto lo podríamos hacer utilizando variables en coma flotante, pero en un microcontrolador implican un mayor consumo de recursos, por lo que vamos a utilizar en su lugar el escalado de la variable de temperatura, de modo que podamos trabajar con variables enteras.

Teniendo en cuenta que la aplicación es un control de temperatura de una habitación, la resolución de una décima de grado se puede considerar más que suficiente. Por tanto, lo que podemos hacer es trabajar con unidades de décima de grado centígrado. Así pues, la temperatura de ebullición del agua es 100,0 °C o 1.000 d°C (decigrados Celsius).

Para ello, el proceso que seguiremos será el siguiente:

1) Pasar el valor del ADC a un entero con signo de 32 bits (Valor[ADC]$_{32}$).

2) Escalar el valor recibido por 32 o, lo que es equivalente, desplazar a la izquierda 5 posiciones Valor(ADC)$_{32}$.

3) Seguidamente, dividir el valor obtenido por 59. Este valor lo obtenemos de multiplicar por 32 los 18,43 de la fórmula del Valor(ADC), y dividirlo por 10 para que dé el resultado en d°C.

4) A este valor hay que restarle 610 para que los 0 grados queden en el valor entero 0.

En la tabla siguiente, podemos ver los valores obtenidos:

Conversión de la señal del sensor o temperatura

Temperatura(°C)	Valor (ADC)	Escalado	División	Temperatura (d°C)
–25	665	21.280	360	–250
0	1.126	36.032	610	0
25	1.586	50.752	860	250
50	2.047	65.504	1.110	500

El resultado final lo podemos guardar en un entero de 16 bits con signo, que nos permite ir de los –32.768 a los 32.767 d°C o, lo que es equivalente, de los –3.276,8 a los 3.276,7 °C.

1.2. Definiciones

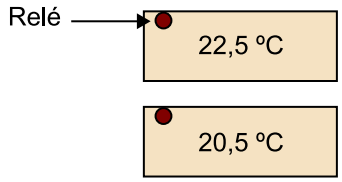
Para lleva a cabo nuestra aplicación, uno de los puntos importantes es establecer el modo como vamos a guardar los datos que no son enteros.

Por otro lado, para el estado de los botones haremos uso de un booleano que nos indicará si el botón esta apretado (*true*) o no (*false*). Los estados de los tres botones los agruparemos en una estructura para que sea más fácil de utilizar.

En el caso del relé, también haremos uso de un booleano, que nos indicará si este se encuentra cerrado o no. Para el tipo booleano, haremos uso del archivo de cabecera stdbool.h, que define el tipo *bool* y los valores *true* y *false*. La estructura sería:

```
struct buttons_t {
       bool up;
       bool down;
       bool mode;
};
```

Los estados de la pantalla se muestran en la figura siguiente:



Pantalla del controlador de temperatura

Para el estado de la pantalla, haremos uso de una enumeración:

- INIT: Iniciando.
- SENSOR: Temperatura del sensor.
- TARGET: Temperatura de consigna.

Para la temperatura, como hemos dicho, utilizaremos un entero de 16 bits con signo. En este caso, nos encontramos con la dificultad de que en lenguaje C no existe a priori el tipo 16 bits con signo. Cada compilador puede definir el tamaño del tipo entero. Por lo tanto, un entero (int) en una arquitectura es de 16 bits y en otra, de 32. Para solucionarlo, podemos hacer uso del archivo de cabecera stdint.h, que define los tipos enteros básicos en función de su signo y tamaño. Así, por ejemplo, tenemos un entero de 8 bits sin signo (uint8_t) o un entero de 16 con signo (int16_t).

Por conveniencia, integraremos el estado de la pantalla y los valores de la temperatura en una estructura:

```
struct state_t {
     enum {
            INIT, SENSOR, TARGET
     } screen;
     int16_t targetTemp;
     int16_t sensorTemp;
     bool relay;
};
```

A continuación, podemos pasar a inicializar los componentes.

1.3. Etapa de inicialización

Para llevar a cabo la inicialización, se puede hacer en un único método o de una manera más estructurada, separándola en función del tipo de dispositivo. En nuestro caso, elegiremos la segunda opción por ser la que permite una transferencia más fácil entre diferentes arquitecturas de microcontrolador.

Como se ha indicado en la sección anterior, el primer elemento que se debe inicializar es el reloj. A continuación, se indica el código de inicialización para el microcontrolador elegido:

```
void clockInit(void)
};
```

El siguiente paso es configurar los puertos. En general, los microcontroladores suelen inicializarse con todos los pines en modo entrada y para propósito general. Esta configuración minimiza los posibles problemas para el microcontrolador. Por contra, si hay elementos externos al microcontrolador, pueden tener un comportamiento indeterminado. Esto se puede solucionar en algunos casos incluyendo resistencias de *Pull-Up* o *Pull-Down*, lo cual a su vez incrementa el consumo dinámico. Pero es evidente que es importante inicializar desde el primer momento los puertos de salida siguiendo una secuencia que evite al máximo los posibles problemas.

En el caso de la aplicación que nos ocupa, la salida que puede tener un efecto más indeseado es la del relé. Por este motivo, parece oportuno configurar esta primero. El siguiente elemento de salida sería la pantalla, aunque esta es menos preocupante, al incluir su propio sistema de control. Por último, tenemos las entradas de los botones y el ADC.

Así pues, el primer elemento que configuraremos será el relé como salida digital.

```
void relayInit(struct state_t *state) {
      state->relay = false;
      // Relay output initializing code
      ...
}
```

Seguidamente, configuraremos la pantalla. Para ello, llevaremos a cabo los siguientes pasos:

- Esta empezará a trabajar en estado INIT.
- Pondremos el valor de la temperatura de consigna al valor 25 °C (targetTemp = 250 d°C).

- Configuraremos el módulo SPI para poder comunicarnos con el LCD.
- Iniciaremos los *buffers* de comunicación con el LCD.
- Enviaremos los mensajes de configuración del LCD.
- Cambiamos el estado de la pantalla a TARGET.
- Presentaremos en la pantalla el estado consigna y la temperatura de consigna.

También deberemos configurar el estado actual de la pantalla, que será el de inicialización.

```
void lcdInit(struct state_t *state) {
      state->screen = INIT;
      // LCD hardware initializing code
      ...
      // End LCD hardware initializing code

      state->targetTemp = 250;
      state->screen = TARGET;

      setLCD(state);
}
```

A continuación, inicializaremos las tres entradas digitales de la botonera.

```
void buttonsInit(struct buttons_t *buttons) {
      buttons->down = false;
      buttons->mode = false;
      buttons->up = false;
      // Buttons input initializing code
      ...
}
```

Finalmente, haremos lo propio con el ADC, lo cual nos permitirá medir la temperatura. Para ello, configuraremos la entrada como analógica y la medida.

```
void tempInit(struct state_t *state) {
      state->sensorTemp = state->targetTemp;
      // Sensor ADC initializing code
...
}
```

Todos estos métodos los integraremos en uno para hacer el código más legible.

```
void initialize(struct buttons_t *buttons, struct state_t *state) {
       clockInit();
       relayInit(state);
       lcdInit(state);
```

```
        buttonsInit(buttons);
        tempInit(state);
}
```

Una vez finalizado, pasamos a la implementación del bucle principal.

1.4. Bucle principal

Una de las principales características de un sistema empotrado es que no suele parar nunca. Una vez arrancado, no para, a no ser que haya algún error en el código, o un evento imprevisto en el código que lleve a dicha situación.

Por este motivo, el programa de este tipo de sistemas suele basarse en la utilización de un bucle principal que nunca finaliza y que va realizando los diferentes pasos necesarios. El primero es la lectura de las entradas, en este caso de los botones y el sensor de temperatura. Para ello, ejecutaremos dos funciones. La primera será muy sencilla, ya que solo requiere consultar un registro:

```
void getButtons(struct buttons_t *buttons) {
        int data = IOPORT;

        if ((data & UP_BUTTON) != 0) buttons->up = true;
        else buttons->up = false;
        if ((data & DOWN_BUTTON) != 0) buttons->down = true;
        else buttons->down = false;
        if ((data & MODE_BUTTON) != 0) buttons->mode = true;
        else buttons->mode = false;
}
```

Donde IOPORT es el puerto al que están asignados todos los tres botones. En caso de que no fuera así, se debería modificar el código para tenerlo en cuenta. Por otro lado, tenemos las definiciones UP_BUTTON, DOWN_BUTTON y MODE_BUTTON, que son las máscaras para cada uno de los botones.

La segunda será algo más compleja, ya que la medida de temperatura requiere un pequeño cálculo para pasar del valor medido por el ADC a grados centígrados. En cualquier caso, suponemos que el valor lo leemos del registro correspondiente.

```
int16_t getTemp(void) {
        uint16_t adcValue16 = ADCVALUE;
        int32_t adcValue32 = adcValue16;
        int16_t result;
        adcValue32 <<= 5;
        adcValue32 /= 59;
        result = (int16_t) (adcValue32 - 610);
        return result;
```

```
}
```

Donde ADCVALUE es el registro que guarda el valor leído por el ADC.

En función de los valores recibidos, deberemos tomar las decisiones oportunas. El primer paso que haremos será ver si el usuario ha pulsado algún botón y actuar en consecuencia.

Lo primero que haremos es establecer la precedencia de los botones.

1) Si se pulsa un único botón, se lleva a cabo la acción para dicho botón.

- **Up**: se pasa al modo de pantalla temperatura de consigna y se sube la temperatura una décima de grado.

- **Down**: se pasa al modo de pantalla temperatura de consigna y se baja la temperatura una décima de grado.

- **Mode**: se cambia al siguiente modo de pantalla.

2) Si se pulsan dos botones, podemos tener tres casos más:

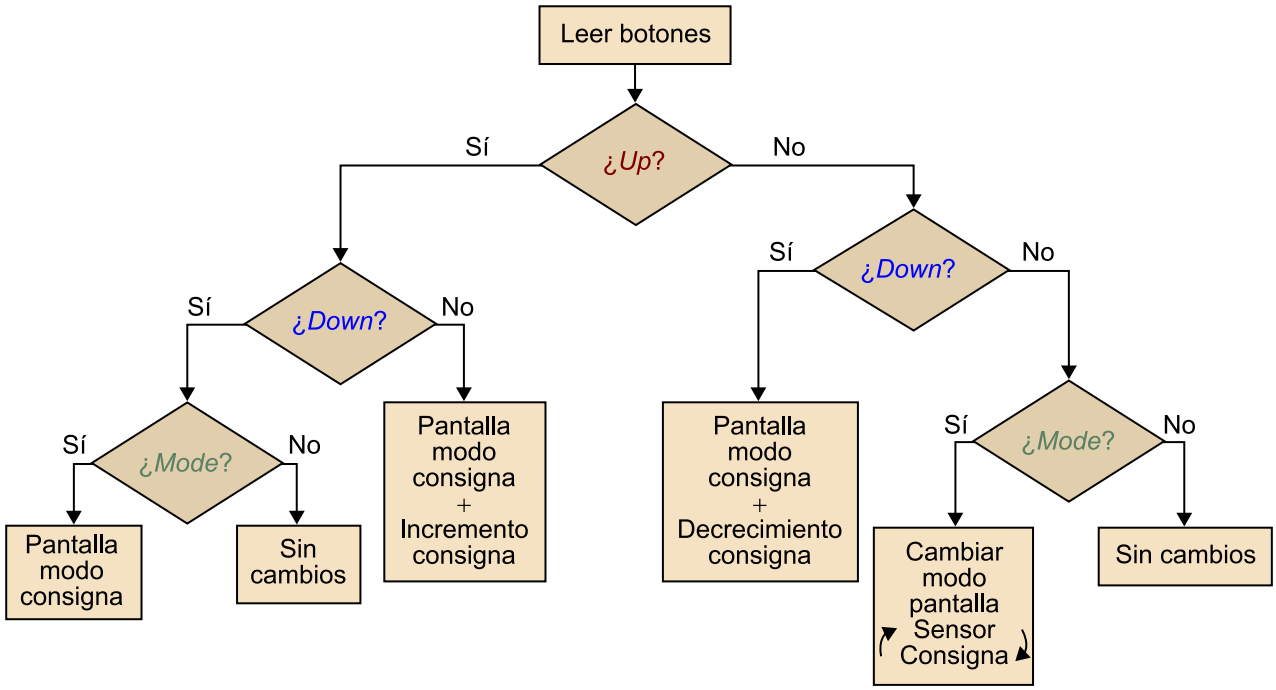
- **Up y Mode**: misma acción que Up solo.
- **Down y Mode**: misma acción que Down solo.
- **Up y Down**: no se realiza ninguna acción.

3) Si se pulsan los tres a la vez, se pasa a modo temperatura de consigna y ya está.

Finalmente, se saturará la temperatura máxima a 50 grados (500 d°C) y la mínima, a –25 grados (–250 d°C).

Este árbol de decisión se muestra en la siguiente figura:

Diagrama de flujo del control de los botones



Este se puede implementar en un método que tenga como entrada ambas estructuras, y modificará el estado y el valor de consigna.

```
void buttonsState(struct buttons_t *buttons, struct state_t *state) {
        if (buttons->up == true) {
                if (buttons->down == true) {
                        if (buttons->mode == true) {
                                state->screen = TARGET;
                        }
                } else {
                        state->screen = TARGET;
                        state->targetTemp += 1;
                }
        } else {
                if (buttons->down == true) {
                        state->screen = TARGET;
                        state->targetTemp -= 1;
                } else {
                        if (buttons->mode == true) {
                                if (state->screen == TARGET) {
                                        state->screen = SENSOR;
                                } else {
                                        state->screen = TARGET;
                                }
                        }
                }
        }
        if (state->targetTemp > 500)
```

```
            state->targetTemp = 500;
        else if (state->targetTemp < -250)
            state->targetTemp = -250;
}
```

En función del contenido de las temperaturas de la variable state, tomaremos la decisión referente al estado del relé. En principio, el relé estará en circuito abierto.

- Si la temperatura de consigna es mayor que la del sensor en más de cinco décimas de grado, cambiaremos el estado del relé a circuito cerrado.

- Si la temperatura de consigna es igual que la del sensor, cambiaremos el estado del relé a circuito abierto.

- En caso contrario, se mantendrá el estado anterior.

Con este funcionamiento, tenemos una cierta histéresis, que nos filtra el posible ruido proveniente de la entrada del ADC. Para ello, definimos previamente una constante global:

```
const uint16_t HISTERESYS_TEMP = 5;}
```

Y seguidamente, el método asociado:

```
void relayState(struct state_t *state) {
      if (state->targetTemp > (state->sensorTemp + HISTERESYS_TEMP)) {
            state->relay = true;
      } else {
            if (state->targetTemp <= state->sensorTemp) {
                  state->relay = false;
            }
      }
}
```

Con el resultado obtenido, se ejecuta la acción sobre el relé haciendo uso del método:

```
void setRelay(bool close) {
      if (close) IORELAY |= RELAY_PIN;
      else IORELAY &= ~RELAY_PIN;
}
```

Donde IORELAY es el registro del puerto que controla el relé, y RELAY_PIN es la máscara de dicho pin.

Finalmente, quedaría pendiente la presentación en pantalla en función del estado, la cual realizará las siguientes acciones:

- Mirar la temperatura que se debe mostrar en función del valor de la variable screen (TARGET, SENSOR).
- Convertir la temperatura de binario a decimal.
- Presentar la temperatura asociada.

Estos pasos los integraremos en el método:

```
void setLCD(struct state_t *state) {
      switch (state->screen) {
      case TARGET:
            printf("Target Temp: %.1f\n", state->targetTemp / 10.0);
            break;
      case SENSOR:
            printf("Sensor Temp: %.1f\n", state->sensorTemp / 10.0);
            break;
      default:
            fprintf(stderr, "Unknown lcd state: %d", state->screen);
      }
}
```

Con todo ello, tendremos una aplicación básica para poder controlar la temperatura de una habitación. El código completo se muestra a continuación:

```
int main(void) {
      struct buttons_t buttons;
      struct state_t state;

      initialize(&buttons, &state);

      for (;;) {
            // Main loop
            getButtons(&buttons);
            state.sensorTemp = getTemp();
            buttonsState(&buttons, &state);
            relayState(&state);

            setRelay(state.relay);

            setLCD(&state);
      }
      return EXIT_SUCCESS;
```

```
}
```

Aunque la aplicación en sí es muy sencilla, su análisis nos proporciona las bases para entender qué se requiere para un sistema operativo empotrado. Si no disponemos de un sistema de desarrollo basado en microcontrolador, podemos hacer uso de las funciones indicadas en el apéndice.

2. Elementos de un sistema operativo

A partir de la aplicación anterior, podemos establecer qué elementos necesitamos para poder llevar a cabo cualquier tarea en un sistema empotrado. Seguiremos una organización diferente a la utilizada en el apartado anterior, más semejante a la de los libros de dicha temática.

Recordemos que los elementos que forman parte del núcleo o *kernel* de un sistema operativo son:

- Planificador de tareas.
- Gestor de tareas.
- Controladores de periféricos.
- *Buffers*.

Para identificarlos, podemos analizar los diferentes métodos y funciones de la aplicación que están dentro del bucle principal:

- getButtons: lee los datos de los botones (periférico).
- getTemp: lee datos de temperatura (periférico).
- buttonsState: procesado de los botones (tarea).
- relayState: procesado del estado del relé (tarea).
- setRelay: modificar la salida del relé (periférico).

Teniendo en cuenta lo anterior, podemos ver que en el código anterior aparece un primitivo gestor de tareas, que sería el bucle principal. Por otro lado, tenemos los controladores de periféricos, que serían los métodos asociados a estos.

Vemos también que aparece la figura de los *buffers*, que permiten intercambiar la información entre los diferentes métodos (tareas y controladores). En este caso, son las estructuras state y buttons las que permiten dicho intercambio.

Sin embargo, el planificador de tareas es inexistente, y veremos que solo en el caso de SO de altas prestaciones será necesario.

Vamos a analizar cada uno de estos componentes a partir de ejemplos basados en soluciones actuales.

3. El gestor de procesos

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

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

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

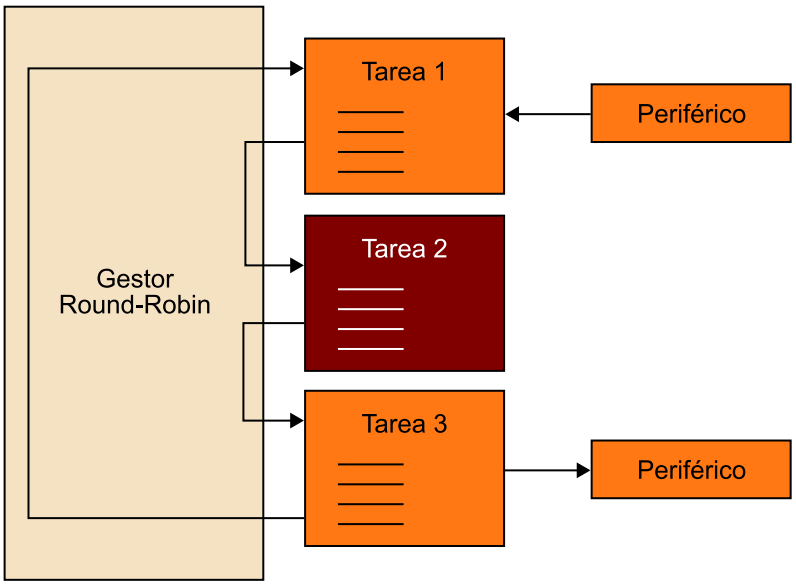
Para llevar a cabo dichas tareas, es necesario que se ejecuten las instrucciones necesarias en la CPU. Para ello, se requieren también recursos de memoria y posiblemente acceso a información de determinados periféricos. Dichos recursos deben ser proporcionados de alguna manera dentro del SO.

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

3.1. Sistemas operativos Round-Robin

Esta solución es la más básica de todas. Su funcionamiento es equivalente a la solución del control de caldera: repetición constante de un conjunto de tareas. Dichas tareas se encuentran organizadas en una lista circular. Se empieza por la primera y hasta que no finaliza no se pasa a la siguiente, tal como se muestra en la figura siguiente:

Diagrama de un sistema Round-Robin



Cada tarea se suele implementar en un método independiente; dichos métodos toman entradas de periféricos o de otras tareas, las procesan y generan salidas que destinan a periféricos o a otras tareas.

Para determinar el punto más crítico en cuanto a velocidad, hemos de analizar las tareas:

- **Tomar la temperatura de la habitación**. La tarea no requiere realizarse con mucha frecuencia, ya que, como hemos dicho anteriormente, el tiempo de variación de la temperatura de la habitación pueden ser minutos. Sin embargo, en muchas ocasiones, el usuario quiere comprobar si el sensor funciona acercando una fuente de calor. Por este motivo, es importante que el tiempo de respuesta no sea muy lento. Consideraremos que un segundo es un retraso razonable.

- **Gestión de botones**. Su cometido es ver el estado de los botones y, en función de ellos, modificar la temperatura de consigna o la visualización de la pantalla. Esta tarea está fundamentalmente dirigida a permitir la interacción con el usuario y, por lo tanto, el tiempo de respuesta debe ser especialmente rápido (nadie está dispuesto a tardar 30 segundos en conseguir que suba la temperatura unos grados). Por este motivo, el tiempo debe estar por debajo de la décima de segundo. Esta tarea también se encarga de comprobar el estado del botón de cambio de modo de visualización.

- **Actuar el relé de la caldera**. Su funcionamiento se basa en comparar la temperatura de la habitación y la de consigna. Por lo tanto, deberá estar atenta a los cambios que se produzcan en las dos tareas anteriores.

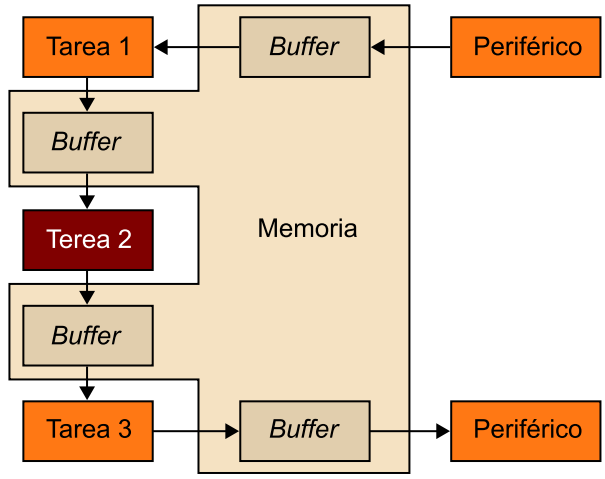
- **Mostrar información en pantalla**. Esta tarea se encarga de gestionar la pantalla. Su funcionamiento depende de las anteriores, ya que solo habrá

que realizar cambios cuando se realice alguna modificación de los parámetros o estados.

Podemos observar que la tarea que se debe realizar con mayor frecuencia es la de gestión de botones. Como la más rápida determina la frecuencia del bucle, todo el bucle ha de poder llevar a cabo las tareas en menos de una décima de segundo.

Es evidente que las tareas necesitan intercambiar información (temperaturas, estado del relé, estado de la pantalla). Para ello se crean unas variables en memoria que realizan dicha función, tal como se muestra en la siguiente figura:

Tareas en el bucle intercambiando información por medio de la memoria



Aunque en general el uso de variables está desaconsejado, en este caso es la única opción, ya que en C no es posible intercambiar datos entre funciones que no reciben parámetros.

3.1.1. La implementación

La implementación de un gestor Round-Robin es relativamente sencilla. Tan solo se requiere una cola anidada donde se guarda cada una de las referencias a las tareas que se deben realizar. El primer paso es definir el tipo puntero a tarea (en nuestro caso, task_t). Para ello, utilizamos la sentencia typedef.

Punteros a función

Un puntero a función es equivalente a un puntero a variable. Ambos permiten acceder de manera indirecta a su contenido. La segunda proporciona el valor contenido, mientras que la primera ejecuta el código asociado a dicha función.

La existencia de los punteros en cuestión, además, también permite crear listas como las utilizadas para el gestor.

```
typedef void (*task_t)(void);
```

Como podemos observar, las tareas no tienen ni entradas ni salidas. Por lo tanto, será necesario utilizar un *buffer* en memoria que permita intercambiar dicha información. Este será generalmente una variable o estructura global.

Por otro lado, tenemos la cola de tareas, que la definiremos como una estructura con un vector de tareas. Además, incluiremos una variable que indique el número de tareas y otra que será el índice de la tarea asociada.

```
struct {
        int number;
        int pointer;
        task_t queue[NUMBER_TASKS];
} tasks;
```

Seguidamente, hemos de iniciar la cola de tareas. Para ello, definimos el siguiente método, donde ponemos todas las entradas a su valor inicial.

```
void tasksInit(void) {
        int i;

        tasks.number = 0;
        tasks.pointer = 0;

        for (i = 0; i < NUMBER_TASKS; i++){
                tasks.queue[i] = NULL;
        }
}
```

Creamos también el método para añadir tareas nuevas a la cola, donde se incluye un control para no superar el tamaño máximo de dicha cola.

```
int tasksAdd(task_t task) {
        int pointer = tasks.number;

        if ((pointer+1) >= NUMBER_TASKS) {
                return -1;
        } else {
                tasks.queue[pointer] = task;
                tasks.number++;

                return pointer;
        }
}
```

Finalmente, tenemos el gestor de tareas, cuyo proceso es ir ejecutando las diferentes tareas de manera secuencial.

```
void tasksRun(void) {
        int i;
        task_t tp;

        for (;;){
                for (i = 0; i < tasks.number; i++){
                        tp = tasks.queue[i];
                        tp();
                }
        }
}
```

El intercambio de información se realiza mediante *buffers* en la memoria. Al funcionar únicamente una tarea cada vez, los *buffers* en este tipo de gestores pueden ser simples variables y estructuras. En nuestro caso, crearemos las estructuras globales state y buttons:

```
struct buttons_t {
        bool up;
        bool down;
        bool mode;
} buttons;

struct state_t {
        enum {
                INIT, SENSOR, TARGET
        } screen;
        int16_t targetTemp;
        int16_t sensorTemp;
        bool relay;
} state;
```

La definición es la misma que hacíamos en el caso anterior. La única diferencia es que las creamos fuera del método main, con lo que se convierten en variables globales accesibles desde todos los métodos.

Por el mismo motivo, modificamos todos los métodos para trabajar directamente sobre dichas estructuras y no hacer uso del paso por parámetros. A modo de ejemplo, mostramos el método de inicialización del LCD:

```
void lcdInit(void) {
        state.screen = INIT;
        // LCD hardware initializing code
        ...
```

```
        // End LCD hardware initializing code
        state.targetTemp = 250;
        state.screen = TARGET;

        setLCD();
}
```

Como se puede observar, en todo momento se supone que la variable state está accesible.

De modo equivalente, tenemos la lectura de la temperatura, que se guarda en la variable state:

```
void getTemp(void) {
       uint16_t adcValue16 = ADCVALUE;
       int32_t adcValue32 = adcValue16;
       int16_t result;
       adcValue32 <<= 5;
       adcValue32 /= 59;
       result = (int16_t) (adcValue32 - 610);

       state.sensorTemp = result;
}
```

Por otro lado, las tareas de procesado siguen una fórmula parecida:

```
void buttonsState(void) {
      if (buttons.up == true) {
            if (buttons.down == true) {
                  if (buttons.mode == true) {
                        state.screen = TARGET;
                  }
            } else {
                  state.screen = TARGET;
                  state.targetTemp += 1;

            }
      } else {
             if (buttons.down == true) {
                    state.screen = TARGET;
                    state.targetTemp -= 1;
             } else {
                    if (buttons.mode == true) {
                          if (state.screen == TARGET) {
                                state.screen = SENSOR;
                          } else {
                                state.screen = TARGET;
```

```
                        }
                    }
                }
        }
        if (state.targetTemp > 500)
                state.targetTemp = 500;
        else if (state.targetTemp < -250)
                state.targetTemp = -250;
}
```

En este caso, los datos de la estructura buttons son utilizados para modificar la estructura global state.

Una vez modificados todos los métodos para trabajar con variables globales, la estructura del programa nos quedará así:

```
int main(void) {
        initialize();

        tasksAdd(getButtons);
        tasksAdd(getTemp);
        tasksAdd(buttonsState);
        tasksAdd(relayState);
        tasksAdd(setRelay);
        tasksAdd(setLCD);

        tasksRun();

        return EXIT_SUCCESS;
}
```

Donde `initialize()` incluirá el método `tasksInit()`. Con este diseño, se observa que todo el proceso sigue un comportamiento muy previsible, ya que se conoce de antemano la secuencia que se seguirá en todo momento.

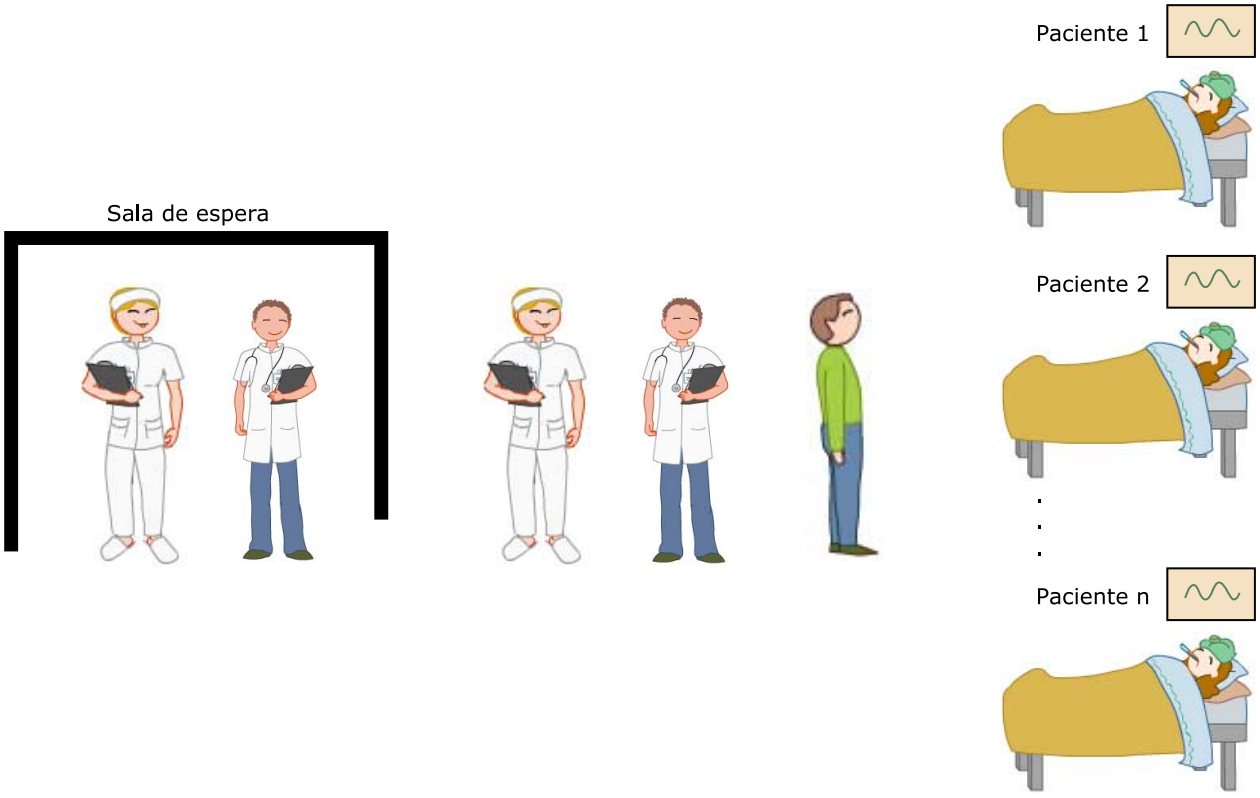
3.1.2. Posibles mejoras: bucle síncrono

El principal inconveniente de esta solución es el consumo innecesario de recursos, ya que buena parte del tiempo no habrá cambios y, por lo tanto, la CPU estará realizando un trabajo innecesario. Si a esto le sumamos que por lo general estos sistemas trabajan con baterías, podemos ver que vamos a gastar las baterías rápidamente. En este sentido, esta solución no es especialmente óptima. Para verlo, vamos a poner un ejemplo.

Supongamos que tenemos un servicio de urgencias. Para hacerlo funcionar, montamos una sala donde recibimos a los pacientes. Estos pueden tener un gran número de síntomas pero, para simplificar, supondremos que por la sala

pasan tres enfermeras: dos que les toman la temperatura y la frecuencia cardiaca, y una tercera que les suministra la medicación. Además, pasan dos médicos especialistas que, en función de las medidas tomadas por las dos primeras enfermeras, proporcionan un diagnóstico e indican a la tercera la medicación que debe administrar. Solo una enfermera o especialista puede estar en la sala, por lo que el turno en el que pasan cada uno de ellos por la sala es controlado por un bedel. Una vez han acabado la enfermera o el médico, el bedel avisa al siguiente.

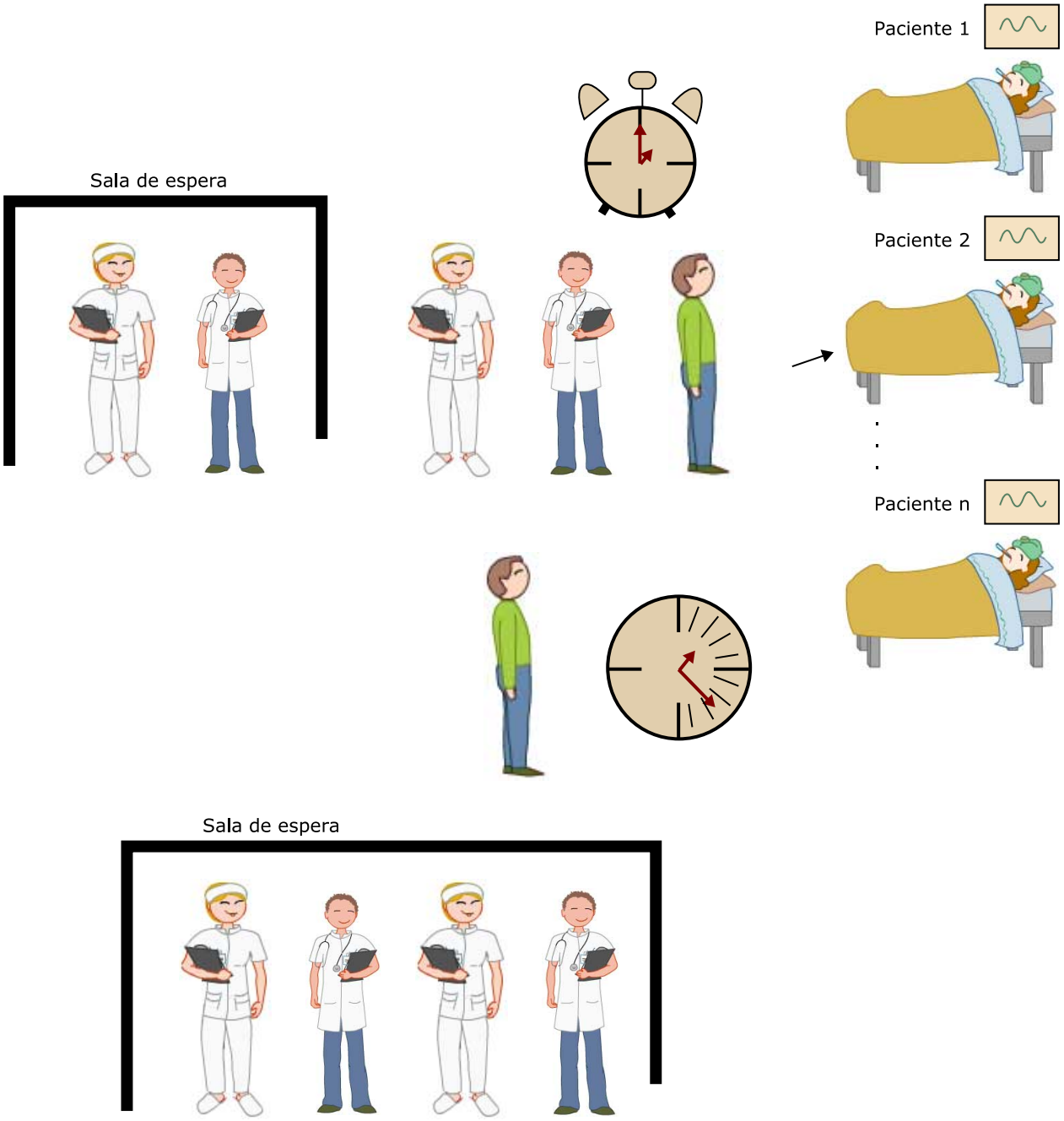
Diagrama de las enfermeras y especialistas pasando de sala en sala, avisados por el bedel



Como información de partida, sabemos que cada enfermera o especialista necesita como máximo cinco minutos para realizar su tarea. También sabemos que el tiempo mínimo entre pacientes es de una hora. Por lo tanto, tenemos tiempo de que el paciente sea reconocido por todos los implicados antes de que llegue el siguiente, con lo que conseguimos nuestro objetivo, como en el caso del Round-Robin.

Ahora supongamos que este proceso se realiza de noche y que, por lo tanto, los especialistas aprovechan para dormir mientras no están diagnosticando. Es evidente que sería más conveniente despertar a las enfermeras y especialistas cada hora, ya que no merece la pena que estén pasando por una sala vacía, cuando sabemos que como máximo habrá uno cada hora. Para lograrlo, habría que proporcionar un despertador al bedel, quien lo pondrá en funcionamiento cuando haya pasado por la sala la última enfermera o especialista; el despertador sonará al cabo de media hora.

Diagrama de las enfermeras y especialistas pasando del dormitorio a la sala y viceversa, con el despertador

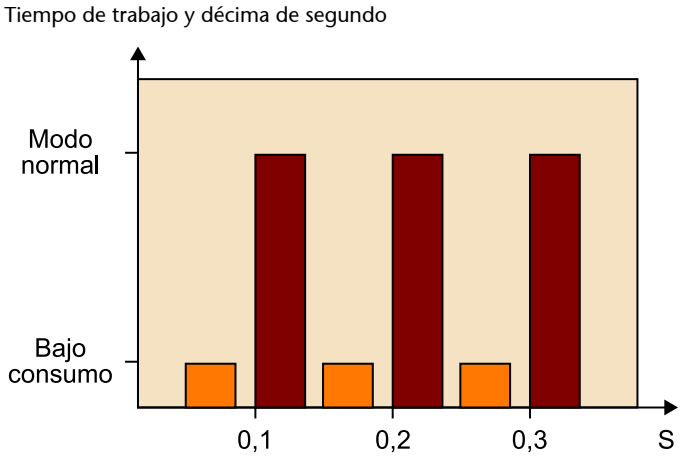


Esta misma solución la podemos utilizar en un microcontrolador trabajando a dos frecuencias de reloj o modos:

- **Modo normal**. En este, la CPU suele trabajar a la frecuencia de reloj más rápida posible para realizar el máximo de tareas también en el mínimo tiempo posible.

- **Modo bajo consumo**. Aquí la CPU y la mayoría de los periféricos están parados, y solo los periféricos imprescindibles trabajan.

Podemos considerar que las tareas del control de temperatura, botones y relé son las enfermeras, mientras que las tareas de decisión serían los especialistas. Por su parte, el bedel sería el gestor de tareas. Si dichas tareas se realizan suficientemente rápido (menos una décima de segundo), el microcontrolador puede dormirse el tiempo desde que se acaban las tareas hasta que llega la siguiente décima, de modo equivalente a como lo que hacen las enfermeras y los especialistas. Este proceso se muestra en el esquema siguiente:



Así pues, lo que podemos hacer es dejar en marcha un temporizador que cada décima de segundo dé una señal que despierte a la CPU (el despertador). Una vez despierta, la CPU realizará un bucle de control y, en cuanto acabe, se pondrá en estado de bajo consumo hasta que llegue la siguiente señal de reloj.

Como hipótesis, supongamos que, en promedio, se modifica el estado del sistema ocho veces cada hora. Estos cambios son equivalentes a 2 pulsaciones de botón y 2 modificaciones del estado del relé. Supongamos también que el tiempo que tarda en llevarse a cabo el bucle cuando no hay cambios es inferior a una milésima de segundo, frente a cuando sí los hay, que es una décima de segundo. Teniendo en cuenta que el número de bucles que se realizan en una hora son:

$$n_{\text{bucles}}\,[1\text{ hora}] = \frac{10\text{ bucles}}{1\text{ segundo}} \cdot \frac{3.600\text{ segundos}}{1\text{ hora}} = 36.000\text{ bucles}$$

Podemos ver que la potencia necesaria se reduce a:

$$t_{\downarrow}\text{equivalente} = (n_{\downarrow}\text{cambio} \cdot t_{\downarrow}\text{cambio} + n_{\downarrow}\text{estable} \cdot t_{\downarrow}\text{estable}) \;/\; (n_{\downarrow}\text{cambio} + n_{\downarrow}\text{estable}) =$$
$$(n_{\downarrow}\text{cambio} \cdot t_{\downarrow}\text{cambio}) + (n_{\downarrow}\text{total} - n_{\downarrow}\text{cambio}) \cdot t_{\downarrow}\text{estable} \;/\; n_{\downarrow}\text{total} =$$

Esto implica que reducimos el consumo prácticamente a una milésima parte del inicial, con un notable incremento de la duración de la batería. El coste es la simple generación de una interrupción de reloj.

Para lograrlo, son necesarias únicamente dos modificaciones:

1) Implementar una interrupción de reloj que cada cierto período de tiempo despierte al sistema.

2) Añadir en el método tasksRun un paso a modo bajo consumo que deje el micro parado hasta que se produzca la interrupción de reloj.

La implementación de interrupciones es siempre una tarea que se debe realizar con extremo cuidado, ya que esta para de manera asíncrona el trabajo que realiza la CPU, interrumpiendo la tarea que se está realizando. Este acceso asíncrono también implica la posibilidad de carreras críticas, al poderse modificar de modo "concurrente" el valor de una misma variable/posición de memoria.

En nuestro caso, la posibilidad de carreras críticas debe ser considerada imposible. El motivo es que la interrupción solo se habilita cuando el gestor de tareas ha realizado un ciclo completo (como sucedía con el bedel). De esta manera, siempre que salte una interrupción la CPU ya estará en estado de bajo consumo y, por lo tanto, la CPU no estará realizando ninguna tarea y no habrá carrera crítica.

En esta situación, la rutina de servicio de la interrupción será muy simple:

```
void timeInterrupt(void){
      disableTimeInterrupt();
      standardMode();
}
```

Donde se asume que el método standardMode() es el que vuelve a activar la CPU a la velocidad requerida. Este método deberá ser sustituido por el correspondiente a la arquitectura.

Por otro lado, el gestor de tareas también se ve modificado de la siguiente manera:

```
void tasksRun(void) {
      int i;
      task_t tp;

      for (;;) {
            for (i = 0; i < tasks.number; i++) {
                  tp = tasks.queue[i];
                  tp();
            }

            lowPowerMode();
      }
}
```

Donde el método lowPowerMode pone la CPU en bajo consumo. Por último, deberíamos modificar tanto la estructura tasks como el método de inicialización de tareas. A la primera añadiríamos un campo para establecer el período de la interrupción (usperiod) y un contador para saber el tiempo transcurrido (ustime) (us indica que trabajan en unidades de us, ya que generalmente los microcontroladores trabajan con períodos inferiores). La estructura quedaría así:

```
struct {
      int number;
      int pointer;
      uint32_t ustime;
      uint32_t usperiod;
      task_t queue[NUMBER_TASKS];
} tasks;
```

En el método de inicialización de tareas deberíamos añadir el método para llamar a las interrupciones, que incluiría como parámetros la tarea de interrupción y el período del reloj. Quedaría de la siguiente manera:

```
void tasksInit(const uint32_t usperiod) {
      int i;

      tasks.number = 0;
      tasks.pointer = 0;
      tasks.usperiod = usperiod;

      for (i = 0; i < NUMBER_TASKS; i++) {
          tasks.queue[i] = NULL;
      }

      startInterruptHandler(timeInterrupt, tasks.usperiod);
}
```

De donde asumimos que startInterruptHandler realiza las siguientes tareas:

1) Dar de alta el método timeInterrupt en la tabla de interrupciones.

2) Indicar a la interrupción del reloj el tiempo que debe esperar, valor recogido en la variable usperiod.

3) Habilitar la interrupción de reloj.

Como en el caso inicial, en el apéndice se han incluido los métodos que permiten emular en un sistema Posix el comportamiento del microcontrolador.

3.2. Sistemas operativos basados en eventos

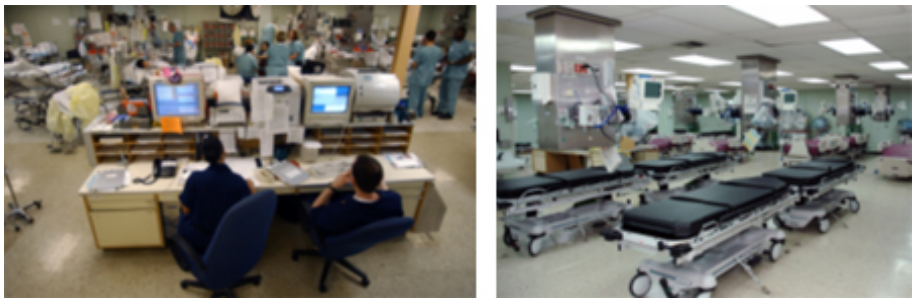
Los microcontroladores actuales proporcionan muchos recursos que, adecuadamente utilizados, nos pueden permitir una gran flexibilidad. Siguiendo con el símil de un hospital, supongamos que ahora organizamos una unidad de vigilancia intensiva con los últimos sistemas de monitorización.

En este caso, tenemos un conjunto de pacientes con monitores que registran diferentes bioseñales, proporcionando información de estas. Dichos monitores son capaces de generar una alarma cuando se superan ciertos umbrales e indicar el médico que está de guardia para ese paciente.

Además, la sala dispone de una librería donde se recogen los historiales de cada paciente en la UVI, con la información de los diferentes monitores, así como los diagnósticos llevados a cabo por el médico y los tratamientos que deben seguir los enfermos.

Añadamos dos últimos condicionantes: siempre hay una enfermera de guardia y solo puede estar un médico en la UVI. Si nos piden que organicemos esta UVI de una manera óptima y segura, ¿qué opciones tenemos?

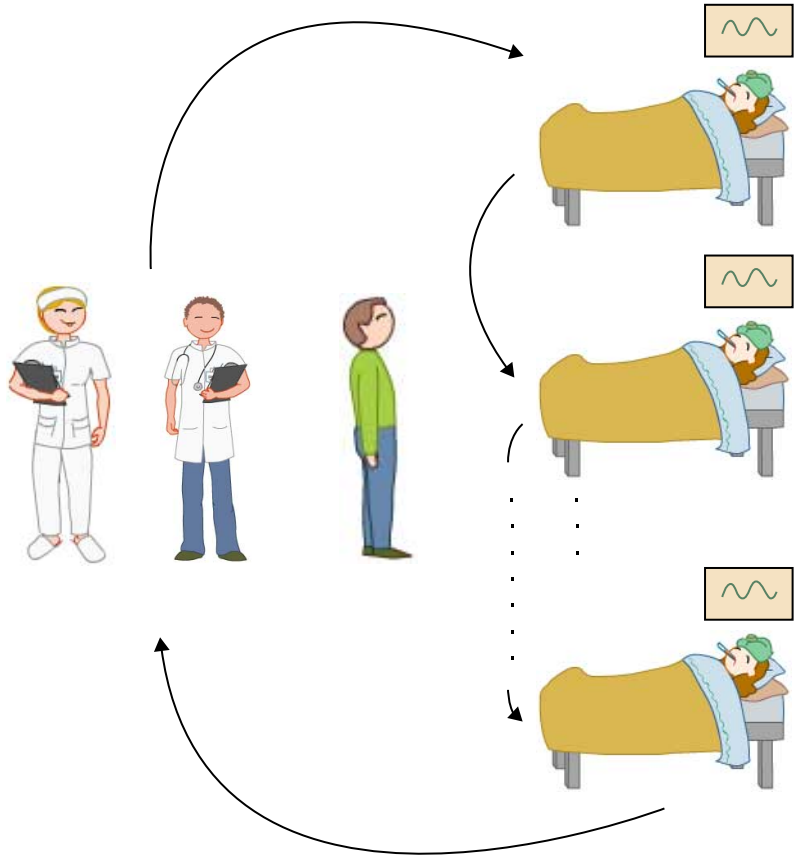
Imágenes de dos UVI



La de la izquierda muestra la sala con los pacientes y los monitores centralizados en un ordenador. La de la derecha muestra la sala vacía, con los monitores distribuidos por cama. Fuente: imágenes de dominio público, US Navy.

Una primera sería que la enfermera fuera avisando a cada médico para que pasara de manera cíclica por los pacientes que le tocan. El médico miraría si se han superado los umbrales y, en ese caso, indicaría si hay que llevar a cabo alguna acción. Esta solución sería equivalente a la que propusimos inicialmente para urgencias y la podemos considerar poco adecuada, teniendo en cuenta que los monitores incluyen la posibilidad de generar alarmas.

Médicos visitando de manera cíclica a los diferentes pacientes



Una segunda opción sería hacer uso de las alarmas de los monitores. Cada vez que salta una alarma, la enfermera toma nota del motivo de esta en el historial del paciente y avisa al médico indicado para darle paso a la sala. Mientras está en la sala, el médico consulta el historial del paciente, analiza los nuevos datos y escribe en el historial si hay que realizar alguna acción. Cuando termina, sale de la sala y la enfermera descansa hasta que llega la siguiente alarma.

En esta situación, es posible que algún médico no tenga los recursos necesarios para diagnosticar al paciente, en cuyo caso puede solicitar que se avise a un nuevo médico. Este contará en principio con más información y podrá dar un diagnóstico y tratamiento para situaciones menos comunes.

Esta solución funciona bien cuando la posibilidad de que se dé una alarma mientras haya un médico en la sala es nula o, dicho de otro modo, cuando el tiempo entre alarmas es muy superior al tiempo que tarda el médico en realizar una actuación. Sin embargo, esto ocurre cuando el tiempo entre alarmas es indeterminado, aunque la media es siempre superior al tiempo de actuación. En este caso, nos encontramos con dos alternativas.

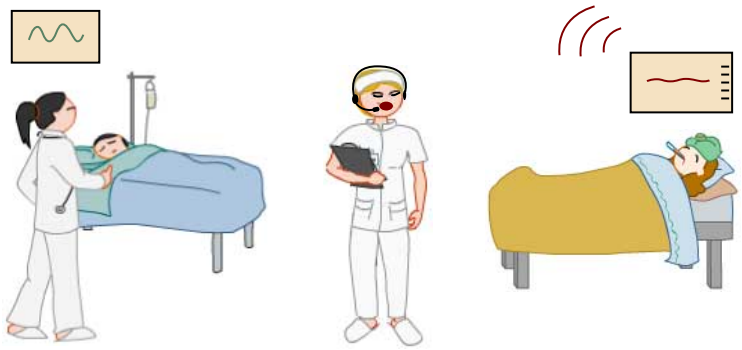
El escenario sería el siguiente: ha saltado una primera alarma y un médico está en la sala diagnosticando al paciente. Durante este tiempo, salta una segunda alarma. La enfermera puede hacer varias cosas: la primera es obviar la alarma

y esperar a que el médico acabe su trabajo y mirar entonces la alarma. Esta solución, sin embargo, plantea que la alarma estará constantemente sonando y molestando al resto de los pacientes, por lo que la descartamos.

Una segunda opción es que la enfermera mire la alarma y tome nota del médico. Con esta información, puede hacer también dos cosas, echar al médico que está en la sala, avisar al segundo médico y hacer que pase a la sala. En cuanto termine este segundo, da paso de nuevo al primero para que termine su trabajo. Esta opción tiene la ventaja de que la alarma deja de molestar, pero es probable que el médico que debe quedarse esperando se moleste un tanto. Además, añade la incertidumbre de no saber nunca cuánto tiempo va a tardar un médico al poder ser interrumpido en multitud de ocasiones.

Una tercera alternativa es que la enfermera, en lugar de echar al médico que está en esos momentos en la sala, avise al segundo médico para que espere en la sala. Cuando termine el primer médico, el segundo puede entrar y llevar a cabo su trabajo. Esta opción tiene la ventaja de que hasta que no se termina con un paciente no se empieza con el siguiente, lo cual provoca que el tiempo que tarda cada médico sea más fácil de predecir.

Enfermera avisando al médico según lo indicado por el monitor del paciente, mientras el médico lo trata.



Aunque la introducción de la utilización del reloj reduce el consumo del sistema, es evidente que en el caso de tener eventos asíncronos, como las alarmas de una UVI, la solución del reloj no es especialmente adecuada.

En el contexto de un microcontrolador, entenderemos por evento cualquier modificación de una entrada o periférico, independientemente de su origen, que sea notificada de manera asíncrona a la CPU por medio de una interrupción.

De modo equivalente a las alarmas de la UVI, el evento suele requerir un mínimo procesado que es llevado a cabo por la rutina de servicio de la interrupción, por ejemplo la comparación entre una señal y un umbral. Si detecta la necesidad, añade una nueva tarea (médico) al gestor de tareas (enfermera). Estas tareas se van ejecutando siguiendo un esquema: la primera que llega es la

primera que se ejecuta (*first in, first executed*), y así hasta que se llevan a cabo todas las tareas. En caso necesario, una tarea puede añadir al gestor de tareas una nueva, de modo equivalente a como lo hace la ISR.

Un aspecto importante en este tipo de SO es el intercambio de información entre tareas. Para entenderlo, seguiremos con el ejemplo de la UVI.

Supongamos que un médico y la enfermera están con un determinado paciente. El médico está escribiendo el tratamiento en el historial. Justo en ese momento, salta una alarma en el monitor de dicho paciente. En este caso, ¿qué hacemos con el historial? En principio, la enfermera se lo debe tomar al médico, añadir los nuevos datos de la alarma, avisarle de que debe volver a entrar cuando hayan pasado el resto de los médicos y devolverle el historial para que siga escribiendo el tratamiento.

La pregunta es ¿cómo añade los datos la enfermera? Si escribe justo después de donde lo está haciendo el médico, podemos tener un problema grave. A modo de ejemplo, supongamos que el médico está escribiendo la dosis de un medicamento que son 100 mg, y justo después de escribir el 1, salta la alarma. La enfermera toma el historial y escribe detrás el código de la alarma, que es un tres. Una vez hecho, le devuelve el historial al médico, quien simplemente escribe los dos ceros y las unidades que faltan. Resultado: en lugar de poner 100 mg en el historial, pone 1.300 mg, lo cual puede tener un resultado fatal.

En este caso, es evidente que se debe conseguir que cuando el médico escriba algo, se realice toda la escritura completa. Un proceso que garantiza que se escribe o lee algo de manera completa se denomina atómico. En este caso, se puede asegurar que no aparecerán modificaciones que pudieran producir errores en el sistema.

Para ello, se debe tomar alguna precaución. Ésta puede ser simplemente limitar las posibles alarmas mientras el médico escribe en el historial. Una vez acabado, se vuelven a activar, y si ha saltado alguna alarma, la enfermera realiza las acciones pertinentes sin afectar al médico. En caso de que necesite otra vez que visite al paciente, lo vuelve a poner en la cola.

Esta solución es la que se suelen utilizar en los microcontroladores. Para evitar que una tarea y una interrupción modifiquen a la vez un *buffer*, generando lo que se conoce como carreras críticas, se deshabilitan las interrupciones. Una vez modificado el *buffer*, se vuelven a habilitar. Teniendo en cuenta que las interrupciones pueden requerir una respuesta inmediata, es importante que su deshabilitación sea lo más corta posible.

Para lograrlo, se sigue un proceso en dos pasos:

1) Se prepara toda la información que se debe añadir en el *buffer* en variables temporales internas de la función.

2) Una vez se tiene toda la información, se llama a una función que realiza los cambios oportunos.

De esta manera se reduce tiempo durante el cual se deben bloquear las interrupciones.

3.2.1. Implementación

La implementación en este caso es muy semejante a la realizada para el caso del Round-Robin síncrono. La principal diferencia es que la cola de tareas es dinámica. Por lo tanto, pasa a ser un *buffer* de tareas, con lo que deberá recibir el mismo tratamiento que cualquier otro *buffer*. En este sentido, será necesario que los accesos sean atómicos.

Siguiendo con el ejemplo del control de calefacción, debemos realizar dos cambios en la manera de trabajar. La primera es determinar el orden en el que se realizan las tareas y la segunda, el modo como vamos a pasar la información entre ellas.

Para ello, el primer paso es determinar qué interrupciones vamos a tener. En nuestro caso, vemos que tendremos dos principales:

- **temperatureInterrupt**: nos la generará el ADC cuando haya acabado la medida de temperatura;

- **buttonsInterrupt**: la generará el bloque de entradas/salidas cuando reciba el cambio de estado del pulsador.

Una vez recibidas dichas interrupciones, estas deberán llamar a alguna de las tareas que teníamos definidas:

- **getButtons**: lee los datos de los botones;
- **getTemp**: lee datos de temperatura;
- **buttonsState**: procesado de los botones;
- **relayState**: procesado del estado del relé;
- **setRelay**: modificar la salida del relé.

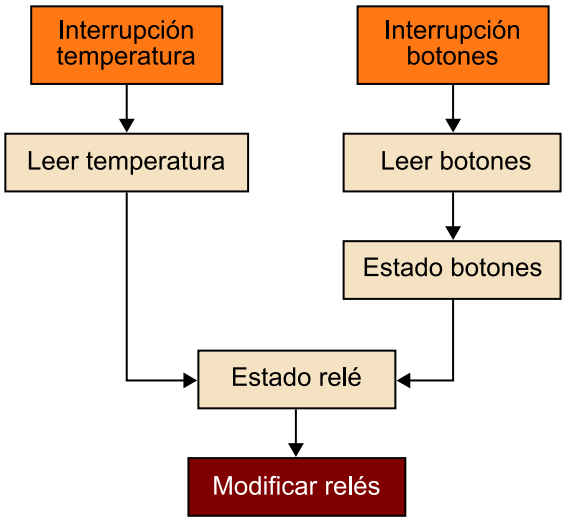
Vemos que tenemos dos caminos que seguir:

1) De temperatureInterrupt pasamos a getTemp. De esta pasamos a relayState y, finalmente, a setRelay.

2) De buttonsInterrupt hemos de pasar a getButtons, que sería la tarea compañera. De esta a buttonsState, que pasaría a relayState y a setRelay.

Como era de esperar, hay un punto de unión que es relayState, ya que es la que define la siguiente salida. Podemos ver el comportamiento en el diagrama de flujo de la figura siguiente:

Diagrama de tareas del controlador de calefacción



A partir de este diagrama, se puede determinar qué tarea debe llamar cada una de las tareas. Cada interrupción añade una tarea de gestión de los datos al gestor de tareas. De esta manera, se minimiza el tiempo que el microcontrolador está en una interrupción. Cada tarea, a su vez, añade la tarea o tareas siguientes al gestor, y así hasta que se llega a la salida.

La principal modificación se encuentra en el gestor de tareas, que debe tener en cuenta la posibilidad de que una interrupción solicite añadir una tarea mientras está modificando el contenido de la lista. Esta pasa a ser más sencilla, ya que desaparece la interrupción de reloj y los campos asociados, siendo muy semejante al caso Round-Robin. Sin embargo, se ha modificado el nombre del campo number por last para indicar que la lista puede crecer de manera arbitraria.

```
struct {
      int last;
      int pointer;
      task_t queue[NUMBER_TASKS];
} tasks;
```

El método de inicialización de la lista es también semejante al Round-Robin, con la salvedad comentada anteriormente.

```
void tasksInit(void) {
      int i;
```

```
        tasks.last = 0;
        tasks.pointer = 0;

        for (i = 0; i < NUMBER_TASKS; i++) {
            tasks.queue[i] = NULL;
        }
    }
```

No obstante, los métodos de gestión de la lista sí que se ven modificados de modo notable. En primer lugar, aparecen los métodos startCriticalSection y endCriticalSection, que se encargan de evitar que ninguna otra tarea o interrupción modifique el estado de la lista de tareas mientras se realiza esta operación. Se ha buscado que el número de operaciones que se realiza dentro de dicha sección sea el mínimo necesario para asegurar que la lista de tareas siga funcionando una vez se esté fuera de la sección crítica. En concreto, el campo last de la lista y la asignación a dicha lista se deben realizar de manera atómica para asegurar la consistencia de los datos en la lista.

```
int tasksAdd(task_t task) {
        int pointer;

        startCriticalSection();

        pointer = tasks.last;

        tasks.queue[pointer] = task;

        pointer++;

        if (pointer >= NUMBER_TASKS) {
            pointer = 0;
        }

        tasks.last = pointer;

        endCriticalSection();

        return pointer;
}
```

De la misma manera se lleva a cabo la gestión de las tareas que hay que ejecutar. En este caso, se comprueba en primer lugar que la lista no esté vacía (pointer igual a last). Si no lo está, se determina la siguiente tarea que hay que realizar. Una vez hecho, se sale de la zona crítica. En función de si hay tarea que realizar o no, se ejecuta la tarea o se pasa al estado de bajo consumo.

Es importante destacar que solo las tareas relacionadas con la gestión de la lista quedan dentro de la zona crítica. Los motivos son dos: el primero es que la llamada al método no se ve afectada por la entrada de una nueva interrupción, más allá de retrasar su puesta en marcha, por lo que se puede sacar fuera; el segundo es que si se hiciera correr la tarea dentro de la zona crítica, las interrupciones seguirían deshabilitadas, por lo que se dejaría de recibir nuevos datos.

```
void tasksRun(void) {
      task_t tp;

      for (;;) {
            startCriticalSection();

            if (tasks.pointer != tasks.last) {
                  tp = tasks.queue[tasks.pointer];
                  tasks.queue[tasks.pointer] = NULL;

                  tasks.pointer++;

                  if (tasks.pointer >= NUMBER_TASKS) {
                        tasks.pointer = 0;
                  }
            } else {
                  tp = NULL;
            }

            endCriticalSection();

            if (tp != NULL) {
                  tp();
            } else {
                  lowPowerMode();
            }
      }
}
```

A estos cambios hay que añadir la aparición de las interrupciones de la lectura de temperatura y del estado de los botones. Para estos cambios, hemos modificado ligeramente las estructuras buttons y state, incluyendo dos campos donde las interrupciones copiarán los valores de las entradas.

```
volatile struct buttons_t {
      uint16_t portValue;
      bool up;
      bool down;
      bool mode;
```

```
} buttons;


volatile struct state_t {
      enum {
            INIT, SENSOR, TARGET
      } screen;
      uint16_t adcValue;
      int16_t targetTemp;
      int16_t sensorTemp;
      bool relay;
} state;
```

En el caso de buttons, se llama portValue, y en el de state, adcValue. Su función se deduce fácilmente a partir del comportamiento de las dos interrupciones.

Variables de tipo volatile

Una variable de tipo volatile es aquella cuyo valor se puede modificar de manera asíncrona (por ejemplo, una interrupción), por lo que no queremos que el compilador optimice el código para su contenido.

```
void buttonsInterrupt(void) {
      standardMode();


      buttons.portValue = getButtonsValue();
      tasksAdd(getButtons);
}
```

Vemos que la interrupción empieza por volver el microcontrolador a estado normal y que, seguidamente, copia el estado de los botones a portValue. Como en casos anteriores, getButtonsValue es un método *inline* que devuelve el contenido del registro de puerto asociado a los botones. Se mantiene como método para evitar la modificación de dicho registro por error. Una vez copiado el valor, se lanza la tarea getButtons, que es la encargada de traducir del formato del puerto de interrupción a la estructura que utilizamos.

Este mismo proceso sigue la interrupción del ADC, tal como se muestra a continuación.

```
void adcInterrupt(void) {
      standardMode();


      state.adcValue = getADCValue();
      tasksAdd(getTemp);
}
```

Las tareas asociadas son básicamente iguales que en el caso del Round-Robin, con dos salvedades:

1) la utilización de buttons o state como entrada y

2) que al final incluyen el método taskAdd, con la tarea que le sigue a continuación, en este caso buttonsState.

```
void getButtons(void) {
      if ((buttons.portValue & (1<<UP_BUTTON)) != 0)
            buttons.up = true;
      else
            buttons.up = false;
      if ((buttons.portValue & (1<<DOWN_BUTTON)) != 0)
            buttons.down = true;
      else
            buttons.down = false;
      if ((buttons.portValue & (1<<MODE_BUTTON)) != 0)
            buttons.mode = true;
      else
            buttons.mode = false;

      tasksAdd(buttonsState);
}
```

Esta misma situación se sigue para getTemp.

```
void getTemp(void) {
      uint16_t adcValue16 = state.adcValue;
      int32_t adcValue32 = adcValue16;
      int16_t result;

      adcValue32 <<= 5;
      adcValue32 /= 59;
      result = (int16_t) (adcValue32 - 610);

      state.sensorTemp = result;

      tasksAdd(relayState);
}
```

El resto de las tareas incluyen la modificación de añadir taskAdd.

Esta misma filosofía es la que siguen SO como TinyOS para gestionar las diferentes tareas e interrupciones. Se puede analizar el código resultante del programa Nesc y compararlo con el presentado anteriormente.

3.2.2. Planificación por prioridades

En el ejemplo de la UVI, hemos tratado de la misma manera todas las alarmas. Sin embargo, es evidente que una señal de parada cardiaca debe ser tratada más rápido que una subida de temperatura. Por este motivo, en muchos casos se establece una prioridad en función de la gravedad de la alarma.

Este mismo proceso se puede llevar a cabo en un SO. En este caso, podemos establecer la prioridad de las interrupciones y tareas. Cuando termina una tarea, se mira cuál es la primera que tiene mayor prioridad y es la que empieza a correr. Y así sucesivamente hasta que terminan todas las tareas de la lista.

En este caso, la tarea ya no puede ser definida como un puntero de función, sino como una estructura, que incluirá el campo puntero a la tarea y un segundo puntero para especificar la prioridad. Además, se añade un puntero a la siguiente tarea de la misma prioridad.

```
struct task_t {
      taskp_t pointer;
      uint8_t prioriy;
      struct task_t* next;
};
```

La estructura tasks se modifica teniendo dos vectores de punteros a tareas. El primero indica la primera tarea de la cola. El segundo indica la última tarea de cada cola. De esta manera, no es necesario recorrer todas las tareas para localizar la última.

```
struct {
      struct task_t* queue[MAX_PRIORITY];
      struct task_t* last[MAX_PRIORITY];
} tasks;
```

Así, se ha modificado el método de inicialización de la estructura de tareas.

```
void tasksInit(void) {
      int i;

      for (i = 0; i < MAX_PRIORITY; i++) {
            tasks.queue[i] = NULL;
            tasks.last[i] = NULL;
      }
}
```

También se ha modificado el modo de añadir tareas. Ahora debe incluir el parámetro de prioridad, que no deberá superar el máximo permitido. Es importante destacar que se requiere el uso de memoria dinámica. Por ello, es importante controlar el modo como se crea la estructura de tareas y, sobre todo, la manera de liberar la memoria cuando finaliza su uso.

Como se observa, la creación de la estructura de la tarea se realiza fuera de la sección crítica, ya que no se ve afectada por la entrada de una nueva interrupción. Para evitar posibles problemas, se limita la prioridad de la tarea a la prioridad máxima permitida, ya que un sistema empotrado no puede pararse por este motivo.

Un vez creada e inicializada la estructura de una tarea, se accede a la estructura tasks. En este momento, se considera que ya se entra en zona crítica, por lo que se inhabilitan las interrupciones.

```
int tasksAdd(taskp_t task, uint8_t priority) {
      struct task_t* stask;
      struct task_t* pointer;

      if (priority >= MAX_PRIORITY) {
            priority = MAX_PRIORITY-1;
      }

      stask = malloc(sizeof(struct task_t));
      stask->pointer = task;
      stask->prioriy = priority;
      stask->next = NULL;

      startCriticalSection();

      pointer = tasks.last[priority];

      if (pointer != NULL) {
            pointer->next = stask;

            tasks.last[priority] = stask;
      } else {
            tasks.queue[priority] = stask;
            tasks.last[priority] = stask;
      }
      endCriticalSection();

      return 1;
}
```

El gestor de tareas también se debe modificar de manera adecuada. En este caso, el primer paso es mirar las diferentes colas, empezando por la de más prioridad, y se observar si hay alguna pendiente. Si es así, se lleva a cabo el método asociado. Como se está accediendo a tasks, se considera de zona crítica.

```
void tasksRun(void) {
      int i;
      struct task_t* stask;
      struct task_t* pointer = NULL;
      taskp_t tp;

      for (;;) {
            startCriticalSection();

            for (i = 0; i < MAX_PRIORITY; i++) {
                  pointer = tasks.queue[i];
                  if (pointer != NULL)
                        break;
            }

            if (pointer != NULL) {
                  tp = pointer->pointer;

                  stask = pointer->next;

                  if (stask == NULL) {
                      tasks.queue[i] = NULL;
                      tasks.last[i] = NULL;
                  } else {
                      tasks.queue[i] = stask;
                  }
            } else {
                tp = NULL;
            }

            endCriticalSection();

            if (pointer != NULL)
                  free(pointer);

            if (tp != NULL) {
                  tp();
            } else {
                  lowPowerMode();
            }
      }
```

```
}
```

Una vez se ha determinado el método asociado a la tarea, y se han reorganizado las tareas dentro de la cola, se sale de la zona crítica. Una vez hecho, se libera la memoria que tiene ocupada la estructura de la tarea.

Después de realizadas estas modificaciones, es necesario actualizar las llamadas al método addTasks, dándole las prioridades adecuadas a las diferentes tareas. En concreto, se darían las prioridades más altas a las tareas requeridas para controlar los periféricos temporizados (getTemp, setRelay). Por otro lado, aquellas relacionadas con el usuario se pueden reducir de prioridad, puesto que el tiempo de respuesta requerido suele ser menor (getButtons, buttonsState, setLCD, etc.). Finalmente, las tareas prioritarias no relacionadas con los periféricos reciben un prioridad intermedia (relayState).

Actividad

Modificad las llamadas al método addTask en las diferentes tareas para tener en cuenta las prioridades indicadas anteriormente.

Sin embargo, la solución basada en prioridades tampoco es la panacea. Siguiendo con el ejemplo de la UVI, supongamos que de repente salta un gran número de alarmas, unas de gran prioridad, como paros cardiacos, paros respiratorios, etc., y otras más leves, como un inicio de fiebre. Siguiendo nuestro esquema inicial, se van tratando los casos más agudos (mayor prioridad) y se espera a finalizar con ellos para empezar a tratar los menos prioritarios. No obstante, pasa el tiempo y un paciente con fiebre no es tratado por su médico, por lo que supera los 42 °C y entra en coma.

Evidentemente, esta es una situación patológica, pero puede darse en un SO. Si muchas tareas prioritarias ocupan los recursos del microcontrolador, las menos prioritarias no se pueden ejecutar, con lo que mueren de inanición. Si esto se lleva al límite, se puede dar el caso que se muestra en el recuadro adyacente.

Ejemplo

Se dice que cuando se quiso parar un gran ordenador IBM 7094 del MIT en 1973, se encontró un proceso (el equivalente de una tarea en grandes equipos) que llevaba desde 1967 sin ser ejecutado. En la imagen, un IBM 7094 de la NASA para el proyecto Mercuri.

Para solucionar este problema, se han buscado alternativas, como el envejecimiento de las tareas.

3.2.3. Planificación por envejecimiento

Para solventar el problema de que un paciente con una ligera subida de temperatura acabe con un coma, lo que podemos hacer es incrementar la prioridad en función del tiempo que lleva sin ser tratado. Así, cada cierto tiempo se revisa la prioridad de las alarmas y las que llevan más de un determinado tiempo sin ser tratadas son incrementadas de prioridad. Pasado un tiempo preestablecido, unas tareas de baja prioridad (subida de fiebre) pasan a ser de alta prioridad. Este tiempo de subida debe ser lo suficientemente corto para que ningún paciente acabe con una patología grave debido a un retraso, y lo suficientemente largo para que la priorización no pierda su razón de ser.

En el caso de un microcontrolador, dicha modificación de la prioridad se puede llevar a cabo por el gestor de tareas cada vez que finaliza una. Para ello, deberá modificar la estructura de la tarea, de modo que incluya el momento en el que fue creada la tarea para poder determinar su edad. Además, será necesario disponer de un contador del sistema que determine el tiempo actual desde el punto de vista del microcontrolador.

Actividad

Modificad la estructura task_t y los diferentes métodos para tener en cuenta el envejecimiento.

Estas soluciones dan respuesta a los problemas cuando son relativamente sencillos, pero en muchos casos interesa dar respuesta a varios problemas de manera concurrente. Para ello, es necesario saltar a un sistema multitarea.

3.3. Sistemas operativos multitarea

Hasta el momento, hemos trabajado con tareas que ejecutaban un conjunto de procesos de principio a fin; una vez terminaban, se pasaba a la siguiente, y así de manera consecutiva. Cada tarea debía dar paso a la siguiente, y si no lo hacía, el resto de las tareas simplemente no se ejecutaban. Este tipo de funcionamiento se conoce también como planificación cooperativa. Es decir, es la "buena voluntad" de cada tarea la que permite que el sistema funciones. Si una de ellas es "egoísta", el resto se quedan sin recursos.

La gran ventaja de esta planificación es que las tareas no son interrumpidas, con la única excepción de las interrupciones que podrán tomar el control de manera temporal. Este modo de trabajo simplifica mucho la gestión de recursos, ya que es únicamente una tarea la que accede en cada momento a ellos. Sin embargo, es evidente que en aplicaciones complejas el depender de que una tarea permita o no pasar a la siguiente puede dificultar su diseño. Por este motivo aparecen los sistemas multitarea.

Retomando el ejemplo de la UVI, si un médico se dedicara en exclusiva a cada paciente, estando con él desde que entra hasta que se le da el alta, el resto de los pacientes no podrían ser visitados. Por ello, se establece un turno de visitas, de modo que va pasando por cada uno de los pacientes. Este mismo principio sería el seguido por este tipo de sistemas operativos.

3.3.1. Planificación colaborativa

El funcionamiento se basa en un principio semejante al Round-Robin síncrono. Recordemos que en esta solución el gestor de tareas se ponía en funcionamiento cada cierto tiempo y ejecutaba todas las tareas. Una vez finalizadas, pasaba a modo bajo consumo. En este caso, en lugar de esperar a que se finalice una tarea, es la propia tarea la que, cuando llega a un punto que no puede continuar, solicita un cambio de contexto.

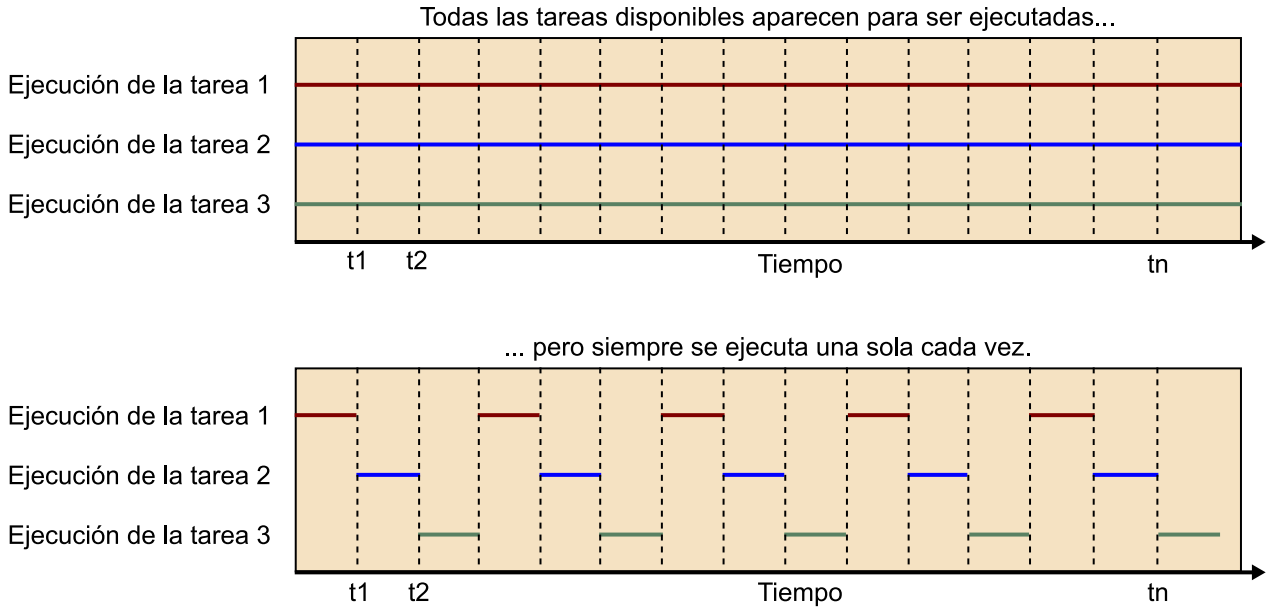
Supongamos, por ejemplo, que la tarea 1 ha llenado el *buffer* de envío del RS-232 y que una segunda quiere enviar. Si no ha dado tiempo a vaciar el *buffer*, la segunda se quedaría esperando. Lo que puede hacer es solicitar un cambio de contexto para que sigan el resto de las tareas y cuando finalicen volverla a llamar. De esta manera se minimiza el impacto.

Este modelo tiene el inconveniente de que las tareas deben estar bien programadas, lo cual no siempre es así. Por este motivo, es mejor que sea el propio gestor el que establezca el tiempo de trabajo para cada tarea y el momento de llevar a cabo un cambio de contexto.

3.3.2. Planificación anticipativa (*preemptive*)

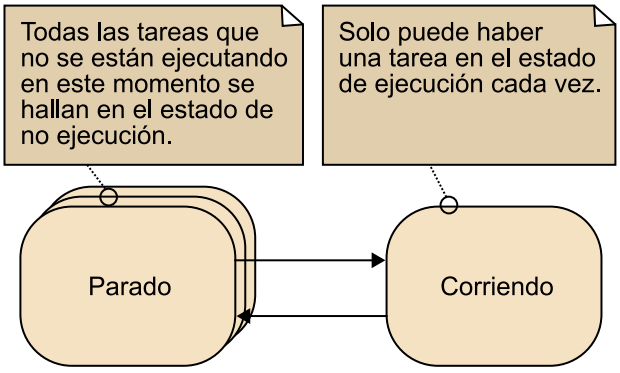
En este caso, el gestor de tareas es interrumpido de periódicamente; en cada interrupción, este detiene la tarea que está corriendo y pasa a ejecutar la siguiente. Así hasta que una de las tareas finaliza, en cuyo caso se quita de la lista. Con este método, da la sensación de que realizan múltiples tareas a la vez, aunque lo que hacen en realidad es ir ejecutando partes de una tarea cada vez.

Diagrama de tiempos de un sistema multitarea



Las diferentes tareas van pasando del estado parado al estado corriendo continuamente.

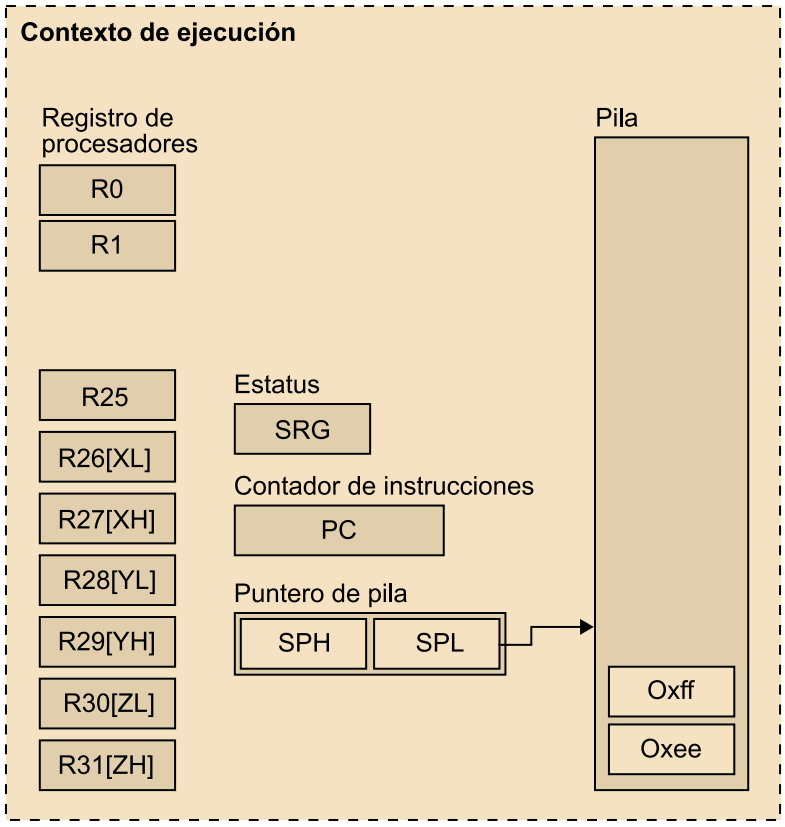
Paso del estado parado a corriendo de una tarea



Sin embargo, al estar realizando diferentes tareas a la vez, aparecen nuevos retos que hay que solucionar. Si volvemos al ejemplo de la UVI, vemos que si el médico va cambiando de paciente, también debe ir cambiando de historial clínico, lo que implica que vuelva a leerlo para recordar la información allí recogida. En caso contrario, podría tratar de manera incorrecta a los diferentes pacientes, al basarse en información errónea.

Esta misma situación se da en una CPU. Si una tarea es interrumpida para llevar a cabo otra, es necesario que guarde la información recogida en los registros de la CPU. En caso contrario, al volverse a ejecutar, los registros podrían haber sido modificados, lo que daría lugar a resultados erróneos. Este proceso se denomina cambio de contexto. En general, se deben guardar los registros y la pila, que son los que recogen el estado de la CPU en ese momento. Ello supone un incremento de los recursos necesarios para cada tarea, así como un incremento del tiempo para pasar de una tarea a otra, al ser necesaria la copia de los diferentes registros. A modo de ejemplo, para un ATMega se deben copiar 32 registros. Para simplificar este proceso, todas estas copias se suelen realizar sobre la pila, ya que la mayoría de los microcontroladores incluyen un conjunto de instrucciones que facilitan dicho proceso.

Contexto de ejecución de los registros del AVR



Este es el modo de trabajar del FreeRTOS. En otros casos, se realiza una copia directa, como sería el caso de los TOSThreads de TinyOS.

Una vez realizada la copia, se recupera la información de los registros de la siguiente tarea que hay que ejecutar y se le da paso. Así, cada tarea que se ejecuta ve el microcontrolador en el mismo estado en el que estaba cuando fue parada y puede continuar trabajando sin posibilidad de que se produzcan errores.

Para realizar un ejemplo, nos basaremos en los threads de Posix, aunque se pueden realizar soluciones semejantes con FreeRTOS. Vamos a ver qué sucedería con dos tareas que funcionan de manera continuada y que no están muy optimizadas, como las que se muestra a continuación.

```
void* vPrintTask( void* arg ) {
      const char *pcTaskName = (char *) arg;
      volatile unsigned long ul;

      /* As per most tasks, this task is implemented in an infinite loop. */
      for (;;) {
            /* Print out the name of this task. */

            vPrintString(pcTaskName);

            /* Delay for a period. */
            for (ul = 0; ul < mainDELAY_LOOP_COUNT; ul++) {
                  /* This loop is just a very crude delay implementation. There is
                  nothing to do in here. Later exercises will replace this crude
                  loop with a proper delay/sleep function. */
            }
      }

      return NULL;
}
```

Esta tarea se ejecuta continuamente, mostrando un mensaje recogido en pcTaskName y esperando un cierto tiempo. El tiempo se define por un bucle, que ha de contar hasta alcanzar el valor mainDELAY_LOOP_COUNT (se puede empezar con 1000000). Evidentemente, esta no es la mejor manera de incluir un bucle de espera.

Para imprimir hacemos uso de la función vPrintString, que hemos escrito de manera semejante a como lo haríamos si tuviéramos que enviar los datos a una pantalla LCD. Por este motivo, hacemos uso de un putc, en lugar de printf.

```
void vPrintString(const char *pcString) {
      volatile clock_t now;
      char message[80];
      int i;
      int length;

      // Get the current time
      now = clock();

      length = snprintf(message, sizeof(message), "%lu:\t", (unsigned long) now);
```

```
        for( i = 0; i < length; i++ ){
              putc(message[i], stdout);
        }

        length = strlen(pcString);

        for( i = 0; i < length; i++ ){
              putc(pcString[i], stdout);
        }
  }
```

Podemos crear una segunda tarea igual modificando únicamente el nombre por vTask2 y sustituyendo el 1 por el 2 en la cadena del mensaje. Así podríamos escribir la función main como:

```
  int main(void) {
        pthread_t threadID1;
        pthread_t threadID2;
        void *exit_status;
        const char* message1 = "Sensor 1: \n";
        const char* message2 = "Sensor 2: \n";

        pthread_create(&threadID1, NULL, vPrintTask, (void *) message1);
        pthread_create(&threadID2, NULL, vPrintTask, (void *) message2);

        pthread_join(threadID1, &exit_status);

        return EXIT_SUCCESS;
  }
```

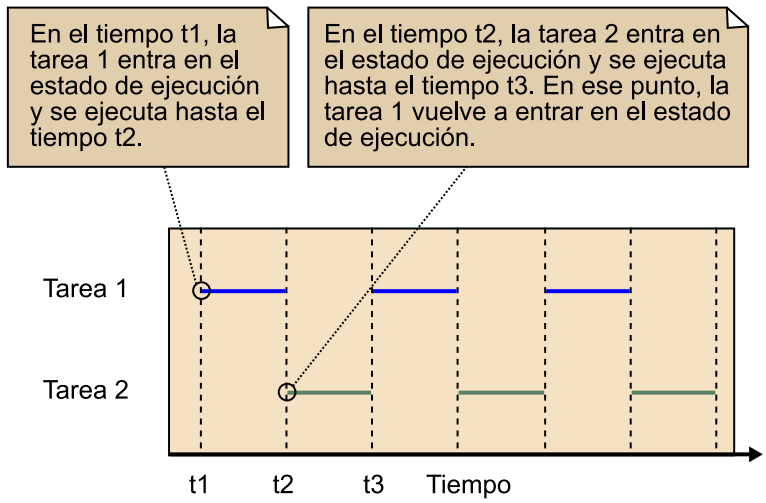
Si ejecutamos el programa, vemos que, aunque tenemos un bucle infinito en ambas tareas, estas se ejecutan de manera intercalada.

Consola programa con dos tareas FreeRTOS

```
Terminal — bash — 80×24
Last login: Mon May 30 23:01:37 on ttys000
zagal:~ chema$ /Users/chema/Documents/Workspace.eclipse/PthreadHeaterController/
Debug/PthreadHeaterController
11995704::              Sensor 1:
Sen2s3o7r9 :2   :S e
n2s5o4r3 :1     :S e
nsor 2:
2687:   Sensor 1:
2707:   Sensor 2:
2722:   Sensor 1:
2736:   Sensor 2:
2749:   Sensor 1:
2767:   Sensor 2:
22788062::              SeSnesnosro r2 :1 :

2990:   Sensor 2: 3
010:    3042:   SensSoern s1o:r
2:
3132124:1       :       SeSnesnosro r2 :1 :

3270:   Sensor 2:
3291:   3316:   SensorS e2n:s o
r 1:
3386:   Sensor 2:
```

De este modo, las tareas van saltando de un estado a otro de manera continuada, tal como se muestra en la figura siguiente:

Paso de una tarea a otra en un gestor anticipativo



Por lo tanto, pese a que el programa está escrito incorrectamente, ambas tareas se ejecutan de manera correcta. Podemos mejorar el estilo modificando la tarea.

```
void* vPrintTask( void* arg ) {
      const char *pcTaskName = (char *) arg;
      struct timespec tim, tim2;

      /* As per most tasks, this task is implemented in an infinite loop. */
      for (;;) {
            /* Print out the name of this task. */
```

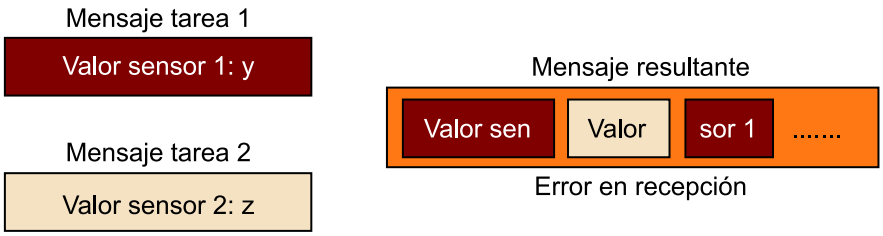
```
            vPrintString(pcTaskName);

            tim.tv_sec = 1;
            tim.tv_nsec = 0;

            nanosleep(&tim , &tim2);
        }

        return NULL;
    }
```

En este caso, la espera es controlada por el gestor, con lo que el microcontrolador queda liberado para llamar a otra tarea si queda alguna en cola.

Otra dificultad es el uso de los periféricos. Si una única tarea debe acceder a un periférico, no hay problema. Sin embargo, si son varias, como es el caso que estamos mostrando, pueden aparecer problemas. A modo de ejemplo, supongamos que leemos datos de dos sensores mediante dos tareas independientes y que deseamos presentar los datos en una pantalla LCD.

Si la tarea del sensor 1 empieza a enviar datos en la pantalla y es interrumpida por el gestor, la segunda tarea puede empezar a enviar datos sin que haya finalizado la primera. En este caso, lo que recibiremos en el PC será una mezcla de los dos mensajes.

Ejemplo de envío de mensaje sin tener en cuenta concurrencia



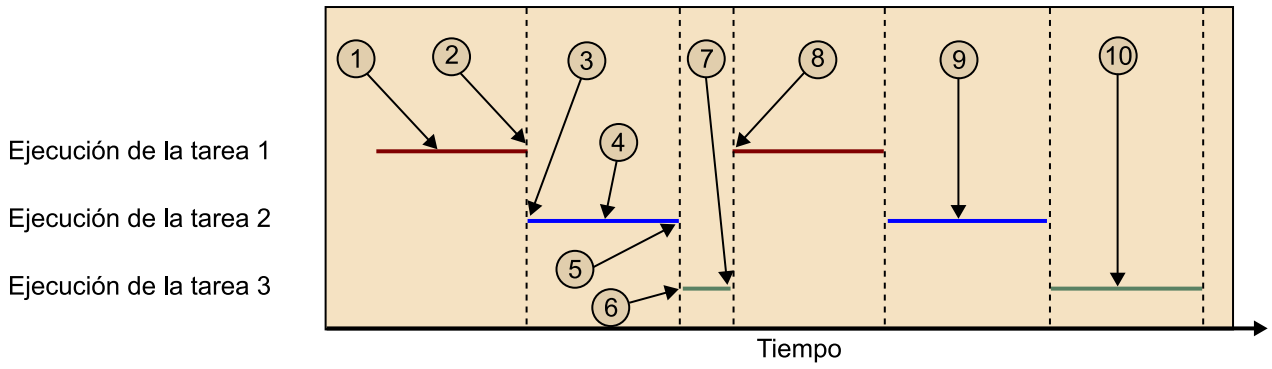
Sería equivalente a que dos médicos fueran escribiendo en un único historial: el lío sería monumental. En nuestro caso, lo podemos reproducir poniendo el valor de tim.tv_sec a 0 y tim.tv_nsec a 500. Al cabo de un rato, podemos encontrarnos una línea como la siguiente:

Resultado en la consola de la mezcla de los mensajes

```
Terminal — bash — 80×24
Last login: Mon May 30 23:01:37 on ttys000
zagal:~ chema$ /Users/chema/Documents/Workspace.eclipse/PthreadHeaterController/
Debug/PthreadHeaterController
11995704::              Sensor 1:
Sen2s3o7r9 :2   :S e
n2s5o4r3 :1     :S e
nsor 2:
2687:   Sensor 1:
2707:   Sensor 2:
2722:   Sensor 1:
2736:   Sensor 2:
2749:   Sensor 1:
2767:   Sensor 2:
22788062::              SeSnesnosro r2 :1 :

2990:   Sensor 2: 3
010:    3042:   SensSoern s1o:r
2:
3132124:1       :       SeSnesnosro r2 :1 :

3270:   Sensor 2:
3291:   3316:   SensorS e2n:s o
r 1:
3386:   Sensor 2:
```

Por lo tanto, hay que encontrar alguna solución. La primera alternativa es que la tarea 1 bloquee la pantalla hasta que haya finalizado de presentar los datos. Si durante esta escritura, el gestor interrumpe la tarea 1, la tarea 2 se encontrará el periférico bloqueado, por lo que no podrá acceder a él. El diagrama se muestra en la figura siguiente:

Diagrama de tiempos de un gestor de tareas



Los diferentes puntos dentro del diagrama de tiempo se describen a continuación.

1) Se está ejecutando la tarea 1.

2) El gestor suspende la tarea 1...

3) ... y continúa la tarea 2.

4) Mientras la tarea 2 se está ejecutando, bloquea un periférico para poder acceder a él de manera exclusiva.

5) El gestor suspende la tarea 2...

6) ... y continúa la tarea 3.

7) La tarea 3 intenta acceder al mismo periférico y se encuentra con que está bloqueado, por lo que se suspende a sí misma.

8) El gestor continúa con la tarea 1, etc.

9) En la siguiente ocasión en la que la tarea 2 se está ejecutando, finaliza con el periférico y lo libera.

10) En la siguiente ocasión en la que la tarea 3 se está ejecutando, se encuentra con el periférico liberado y en este caso hace uso de él hasta que es suspendida por el gestor.

Para llevar a cabo este bloqueo, la primera opción es bloquear directamente el gestor de tareas, o lo que es equivalente: parar las interrupciones. De este modo, se asegura que no habrá problemas. El principal problema de esta solución es que se paran todas las tareas, incluidas aquellas que no están relacionadas con el periférico. Para solventarlo, lo mejor es hacer uso de un mútex.

3.3.3. Mútex

Mútex es la abreviatura de exclusión mutua. En este caso, mientras una tarea bloquea un mútex, el resto de las tareas no puede acceder al periférico.

a) La solución más simple es crear una variable en la memoria que indique el bloqueo. Para modificarla, se desactivan temporalmente las interrupciones, se comprueba el estado de la variable y, en caso adecuado, se modifica. Finalmente, se vuelven a activar las interrupciones.

b) Una segunda opción, que depende de la arquitectura y las instrucciones disponibles por la CPU, es utilizar un flag que puede ser leído, comparado y modificado en una sola instrucción. Este tipo de flags se conoce como **mutuamente exclusivo**. Cuando una tarea quiere entrar en un código crítico, debe haber un comando atómico que permita leer, comparar y modificar un valor en una misma instrucción. Así, no es necesario parar las interrupciones.

c) Una tercera es un intercambio con un valor de la memoria: si el valor es distinto, el mútex está libre y si es igual, está ocupado. La instrucción XCH de AVR recién introducida permite hacerlo, pero no está disponible en los microcontroladores de Atmel actuales. Se puede modificar el programa anterior para hacer uso del mútex.

En primer lugar, se debe modificar el funcionamiento de vPrintString para incluir el mútex, en este caso lock. Lo creamos como variable global, ya que tiene que poder ser accedido por todos los threads.

```
pthread_mutex_t lock;

void vPrintString(const char *pcString) {
      volatile clock_t now;
      char message[80];
      int i;
      int length;

      pthread_mutex_lock(&lock);
      // Get the current time
      now = clock();

      engthh = snprintf(message, sizeof(message), "%lu:\t", (unsigned long) now);

      for( i = 0; i < length; i++ ){
            putc(message[i], stdout);
      }

      length = strlen(pcString);

      for( i = 0; i < length; i++ ){
            putc(pcString[i], stdout);
      }
      pthread_mutex_unlock(&lock);
}
```

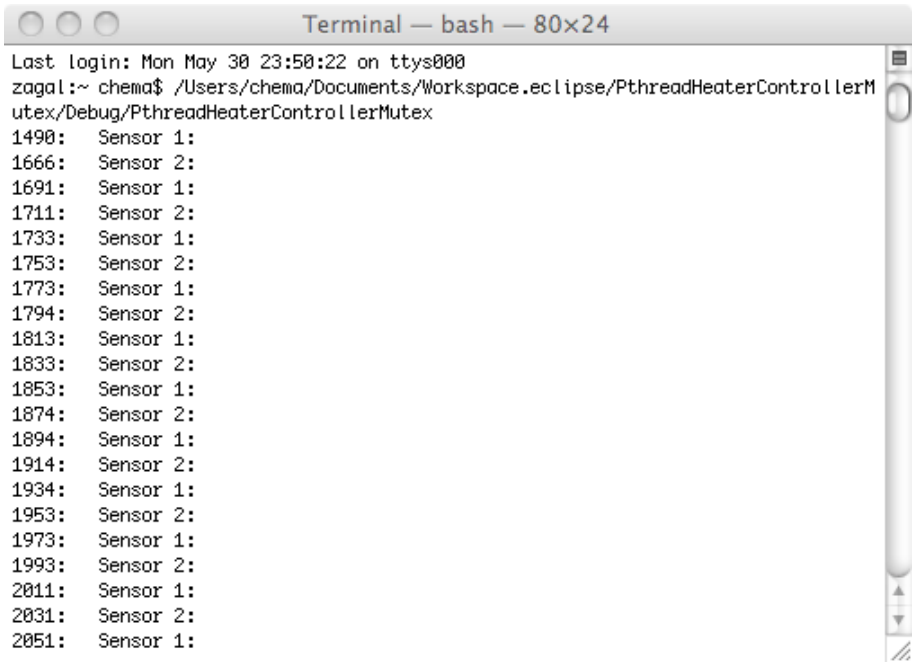
También se debe modificar la función main para iniciar el mútex.

```
int main(void) {
      pthread_t threadID1;
      pthread_t threadID2;
      void *exit_status;
      const char* message1 = "Sensor 1: \n";
      const char* message2 = "Sensor 2: \n";

      pthread_mutex_init(&lock, NULL);

      pthread_create(&threadID1, NULL, vPrintTask, (void *) message1);
      pthread_create(&threadID2, NULL, vPrintTask, (void *) message2);

      pthread_join(threadID1, &exit_status);

      return EXIT_SUCCESS;
```

```
}
```

De este modo, se consigue que la presentación de los datos se realice correctamente.

Consola con el mútex en funcionamiento



Por sus características, la implementación de los mútex se suele dejar a cargo del SO. De esta manera, si al intentar acceder a un mútex, se ve que está bloqueado en ese momento, el gestor de tareas puede dar paso a otra tarea mientras se espera a que finalice el bloqueo y así se minimiza el impacto temporal.

Sin embargo, es importante indicar que los *buffers* que intercambian datos con interrupciones (núcleo del SO) no pueden utilizar mútex, ya que si una interrupción se encuentra un mútex bloqueado, no podría finalizar su tarea y se perdería el dato.

3.3.4. Acceso al núcleo

Entenderemos por acceso al núcleo, o llamadas al sistema, una solicitud para que este realice una acción:

- Añadir/quitar tareas.
- Acceso a periféricos.
- Acceso a *buffers*.
- Bloquear/desbloquear mútex.

En este caso, existen dos opciones:

1) Bloquear el Kernel.

2) Trabajar con dos colas.

El bloqueo del Kernel sería semejante a un bloqueo de interrupciones, pero por software. En este caso, se crea un mútex que, en el momento en el que una tarea realiza una llamada al sistema, se bloquea. Cuando otra tarea intenta hacer una llamada, se comprueba el estado del mútex y si está bloqueado, la tarea se queda en espera. Cuando la primera llamada libera el mútex, la segunda llamada ya puede acceder.

Así, las interrupciones pueden seguir funcionando y con ello los periféricos asociados. Para que el efecto sea pequeño, las rutinas propias del núcleo deben estar muy optimizadas.

Si pese a todo existen aspectos que son lentos, por ejemplo el acceso a algún periférico, se pueden crear varios mútex, en función de lo que se desee bloquear. El incremento del número de mútex (granularidad) también implica un incremento de la complejidad del Kernel. Esta es la solución utilizada por la mayoría de los SO Unix, así como el FreeRTOS.

La segunda opción es hacer uso de dos colas. La primera, asociada al núcleo, trabajaría por eventos. Por lo tanto, las tareas asociadas al núcleo deberían estar muy optimizadas para evitar incrementos en la latencia. Una de las tareas del núcleo sería a su vez el gestor de tareas anticipativo, sobre el cual correrían las tareas del usuario. Por lo tanto, las tareas del usuario vivirían en un entorno multitarea, a diferencia de las del núcleo, que sería basado en eventos.

Cuando una tarea de usuario (presentar datos en pantalla) quiere acceder al núcleo (enviar los datos a la pantalla), crea un mensaje, que no deja de ser solicitar que se añada una tarea (addCharLCD) del núcleo al gestor por eventos, quedando la tarea del usuario bloqueada. Cuando le llega el turno a la tarea del núcleo, esta realiza su operación y, una vez finalizada, le indica al gestor de tareas anticipativo que la tarea del usuario puede continuar. Este modelo es el utilizado por las TOSThreads de TinyOS.

Desde el punto de vista de la sencillez, probablemente la solución basada en bloqueo de Kernel es la más sencilla al requerir únicamente un gestor de tareas. Sin embargo, desde el punto de vista de la modularidad, la opción de trabajar con dos colas y mensajes puede ser óptima.

3.3.5. Un abrazo mortal

Un abrazo mortal es una situación semejante a un intercambio de rehenes. Si cada bando dice que solo liberará a sus rehenes si el otro lo hace primero, nunca se llevará a cabo el intercambio. Es necesario buscar una solución alter-

nativa para asegurar que no se produzca el bloqueo. En la realidad, se suele utilizar un terreno neutral, como un puente, que permite que ambos procesos de liberación se realicen de manera simultánea.

Lo mismo sucede cuando dos procesos tienen bloqueados recursos –por ejemplo periféricos– y necesitan que el otro libere el que tiene bloqueado para poder continuar. Por lo tanto, no se puede salir de dicha situación y quedan ambas tareas bloqueadas. En este caso, la situación es más compleja al no haber el equivalente a un puente. Sin embargo, las tareas no son tan "malpensadas", por lo que se pueden buscar alternativas en las que una tarea libere primero y después lo haga la otra.

Este sería el caso de trabajar con dos colas, con la de núcleo basada en eventos. En este caso, al tenerse que finalizar una tarea antes de realizarse la siguiente, se evita este bloqueo. No así por lo que respecta a las tareas de usuario, donde todavía podría suceder. En este caso, es la pericia del programador la que permitirá dar con la solución más adecuada.

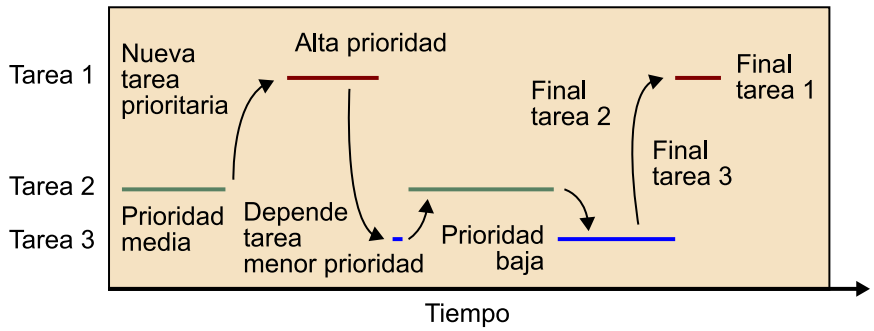
3.3.6. Prioridades

Del mismo modo que en el caso del gestor basado en eventos, los gestores multitarea también pueden trabajar por prioridades. En este caso, el contexto es algo más complejo debido al gestor anticipativo.

A modo de ejemplo, cuando se realiza una planificación por prioridades estática, puede suceder que una vez ha comenzado una tarea, aparezca una nueva que tenga más prioridad (de manera equivalente a una interrupción). En este caso, es necesario que la tarea que se está ejecutando actualmente deje paso a la más prioritaria, para lo cual se debe llevar a cabo un cambio de contexto.

Este paso se puede realizar del mismo modo que en el caso de planificación por ranuras de tiempo, pero de manera asíncrona. Dicho de otra manera, en el momento en el que la tarea de mayor prioridad entra en la cola, esta debe pasar al estado "RUN" y la que estaba corriendo deberá pasar al estado "ESPERA". En el momento en el que la tarea prioritaria finalice, la que estaba esperando, si no hay una nueva de prioridad superior, pasará otra vez al estado "READY" para poder correr.

Diagrama de tiempos cuando aparece una tercera de prioridad intermedia



Cuando se trabaja con prioridades, la principal dificultad es que aparezca una inversión de prioridades. Esto se produce cuando una tarea de alta prioridad requiere una segunda de baja prioridad y hay una tercera tarea de prioridad intermedia en la cola. En este caso, la tarea de baja prioridad no puede ejecutarse hasta que finalice la tarea intermedia. El resultado es que la tarea de alta prioridad pasa a tener la misma prioridad efectiva que la tarea de menor prioridad de la que depende. O, lo que es equivalente, las prioridades se han invertido.

La inversión de prioridades puede implicar más problemas que los debidos a la planificación de tareas. En sistemas como los basados en tiempo real, que tienen requisitos temporales muy estrictos, la prioridad de inversiones puede conllevar un retraso importante en la ejecución de una tarea. Si esto sucede, se puede producir un error en cascada, dejando totalmente inutilizado el sistema.

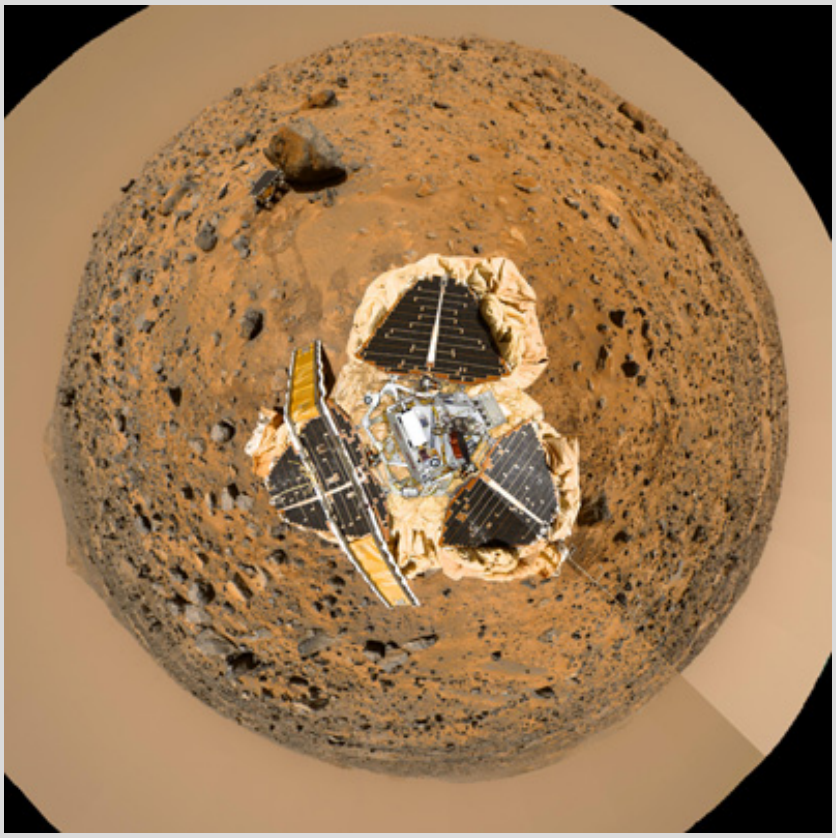
Esta situación se produjo en la misión Mars Pathfinder de la NASA. En esta misión, se utilizaba el robot Sojourner, que podía recorrer la superficie marciana, pero que adolecía de continuos resets. Los resets en cuestión obligaban a reinicializar todo el robot, con los inconvenientes que supone.

Analizando este inconveniente, se descubrió que el problema se debía a que una tarea (bc_dist) requería más tiempo del esperado. Este retraso se debía a que esta tarea precisaba hacer uso de un recurso que era controlado por la tarea ASI/MET, que tenía una prioridad muy inferior. Como bc_dist no podía acceder a dicho recurso, finalmente el planificador detectaba el bloqueo de la tarea y reiniciaba el sistema.

El SO era VxWorks, que activa la opción de herencia de prioridad modificando una variable global. Una vez confirmado que el cambio resolvía el problema, este se llevó a cabo en el robot (en Marte) y se resolvió el problema. Se puede encontrar más información sobre esto en la web:

http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html

Imagen del *Mars Pathfinder* (NASA)



En este caso, una posible solución es que la segunda tarea herede la prioridad de la primera para que se ejecute antes que la tercera tarea. Con ello, conseguimos que la primera tarea se ejecute a tiempo.

4. Sistemas en tiempo real

Hasta ahora, hemos tratado sistemas operativos empotrados. En este tipo de sistemas, es muy común encontrar el concepto de tiempo real; en multitud de casos se indica que un determinado sistema operativo es en tiempo real. Sin embargo, sus formas de trabajar son completamente diferentes, por lo tanto, cabe plantearse la pregunta siguiente: ¿qué es lo que define un sistema como en tiempo real?

Para responder a esta pregunta, vamos a definir tres tipos de sistemas a partir del concepto de tiempo real:

1) Sistemas que no trabajan en tiempo real. Son aquellos que aseguran que se llevará a cabo una acción a partir de un determinado estímulo, de manera correcta y en algún momento del futuro. Un ejemplo de este sistema es un termómetro digital, ya que el tiempo que tarda en dar la temperatura correcta no está prefijado.

2) Sistemas en tiempo real relajado. Son aquellos que llevarán a cabo una determinada acción a partir de un estímulo, de manera correcta y que, por lo general, se terminará en un tiempo predeterminado. Un posible ejemplo de este tipo de sistemas es un reproductor de vídeo: en general, siempre reproducirá correctamente la película, pero si en un momento determinado la película se para, no resulta especialmente grave.

3) Sistema en tiempo real estricto. En este caso, se asegura que se llevará a cabo la acción frente a un estímulo de manera correcta y en un tiempo inferior al predeterminado. Este sería el caso de un airbag de automóvil, en el que un retraso puede resultar mortal.

> La diferencia entre un sistema en tiempo real estricto y uno que no lo es no es la velocidad con la que realiza las tareas, sino que sea determinista y predecible.

Por lo tanto, cualquier opción de gestor de tareas y planificador de las presentadas anteriormente podrían ser validas para llevar a cabo un sistema en tiempo real.

Sin embargo, si buscamos la facilidad desde el punto de vista del programador para conseguir que su aplicación sea en tiempo real, parece evidente que con un sistema operativo anticipativo basado en prioridades será más fácil que

con uno Round-Robin. Y el motivo es fundamentalmente que nos proporciona más herramientas para conseguirlo. Por otra parte, también necesita más recursos, y eso se debe tener en cuenta.

En general, consideraremos que un SO está pensado para trabajar en tiempo real si proporciona prioridades y es anticipativo.

5. Controladores de periféricos

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

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

Ejemplo

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

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

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

Evidentemente, puede haber peculiaridades específicas para un periférico que lo hacen diferir de la interfaz genérica. En este caso, se pueden añadir interfaces que amplíen la interfaz original, pero es importante que estas añadan nueva funcionalidad e intentar que sean también genéricas a su vez.

Para realizar estos controladores, es recomendable seguir una serie de pasos:

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

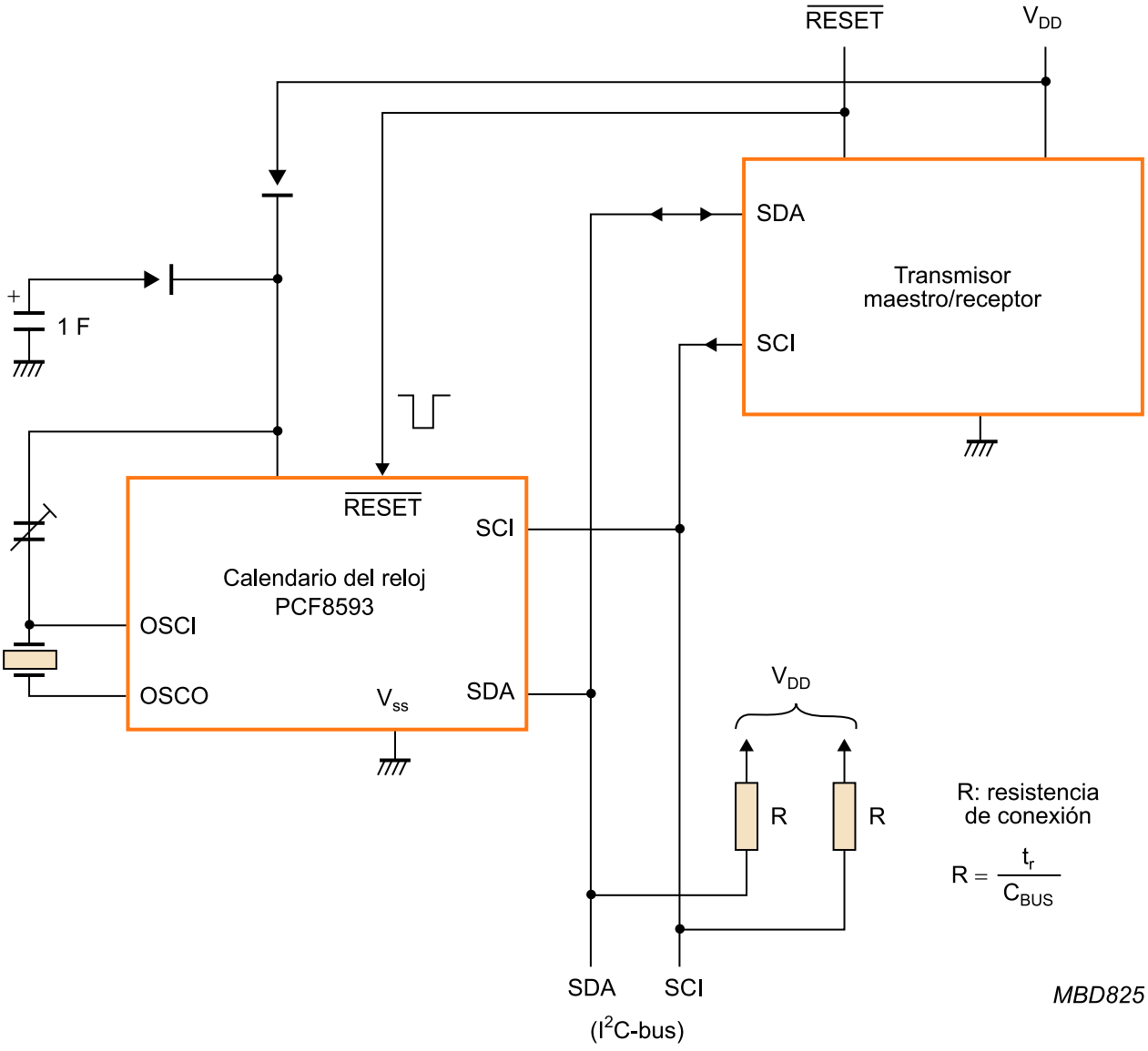
A continuación, vamos a verlos con detalle.

5.1. Información del hardware

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

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

Ejemplo de aplicación del PCF8593



Podemos observar dos aspectos fundamentales:

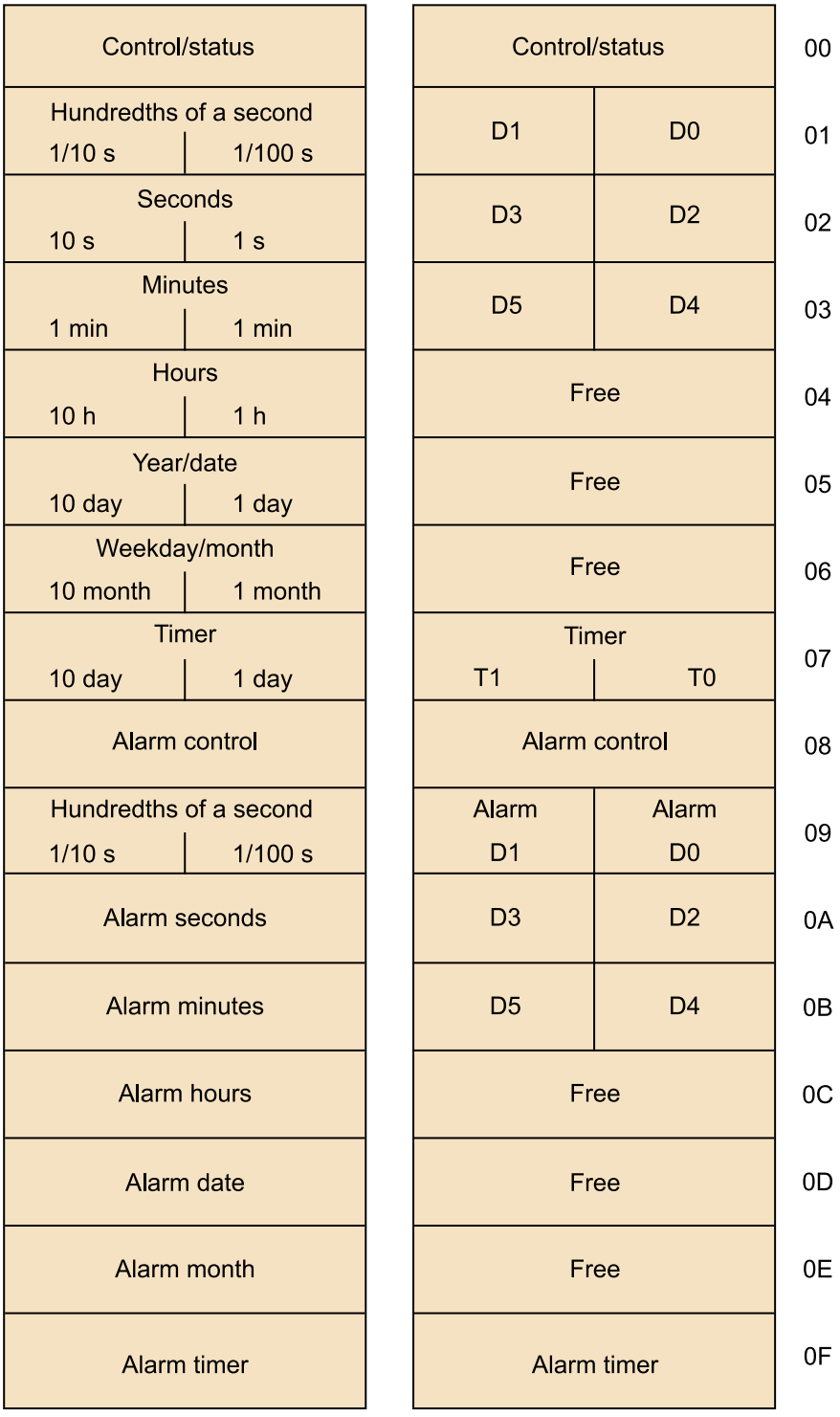
1) La comunicación con el microcontrolador se basa en un protocolo I^2C[1].

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

2) Mediante una capacidad grande, puede continuar funcionando en caso de que se pierda la alimentación en un momento determinado.

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

Registros del PCF8593



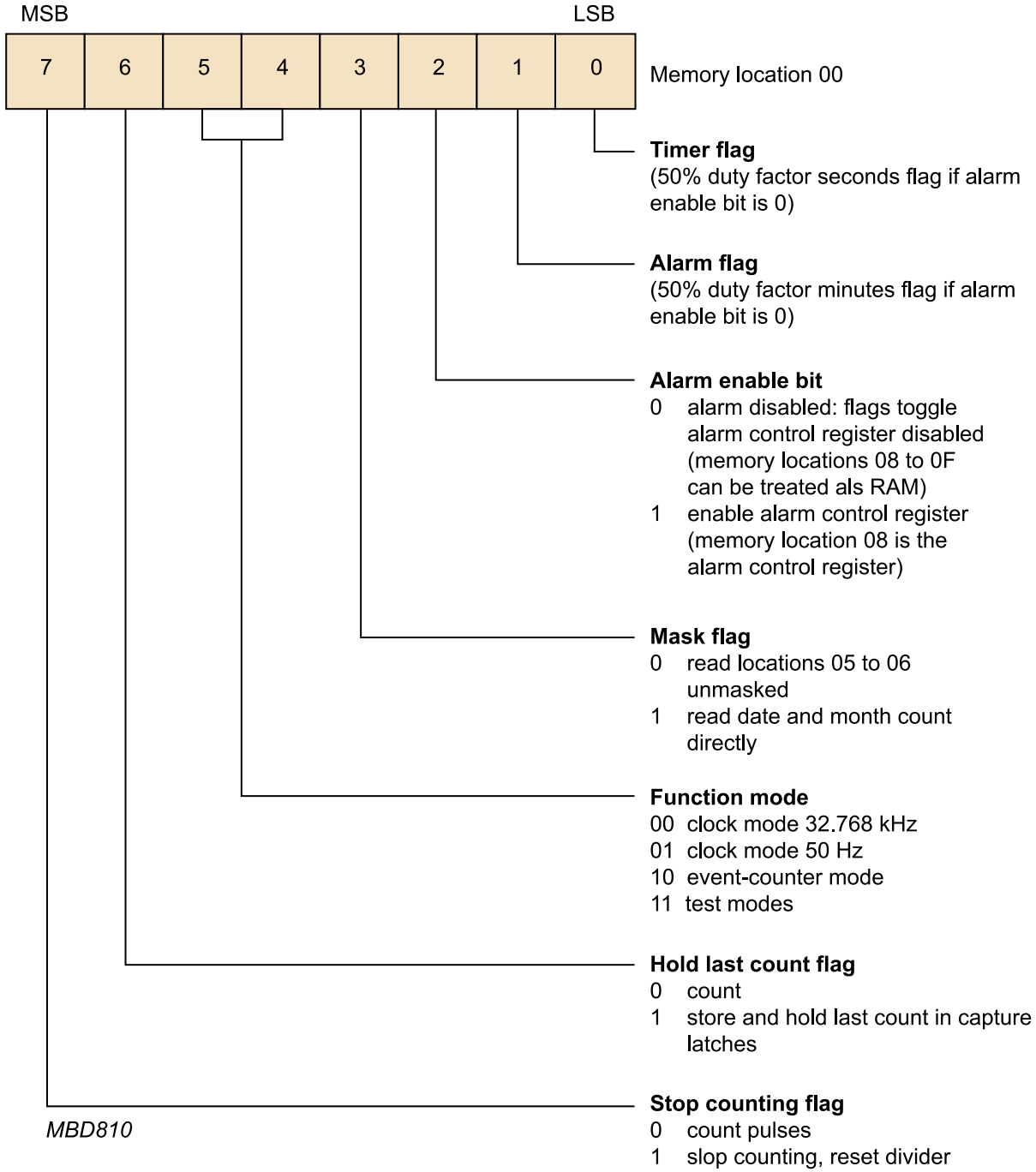
El dispositivo tiene 16 registros. El cero es de configuración y estado. En función de la configuración, puede trabajar en diferentes modos:

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

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

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

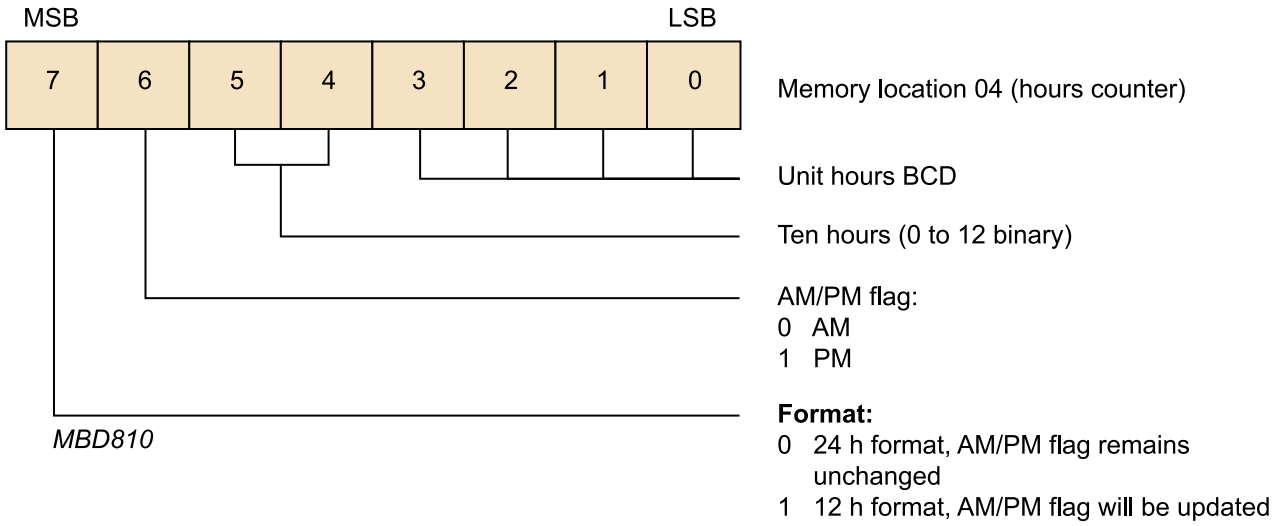
Registro de configuración del PCF8593



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

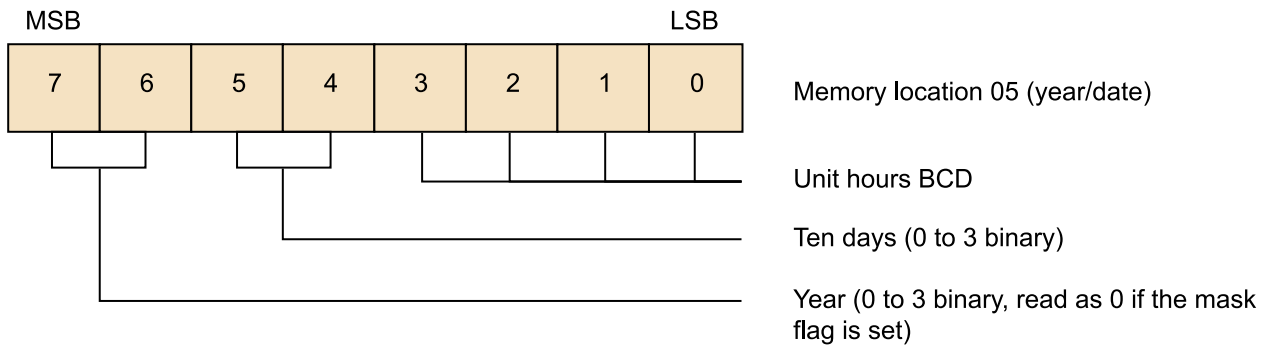
Para poner en hora el integrado, hemos de tener en cuenta las características especiales de los registros 4, 5 y 6 (podéis ver las siguientes tres figuras, respectivamente).

Funcionalidad del registro 4 del PCF8593



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

Funcionalidad del registro 5 del PCF8593

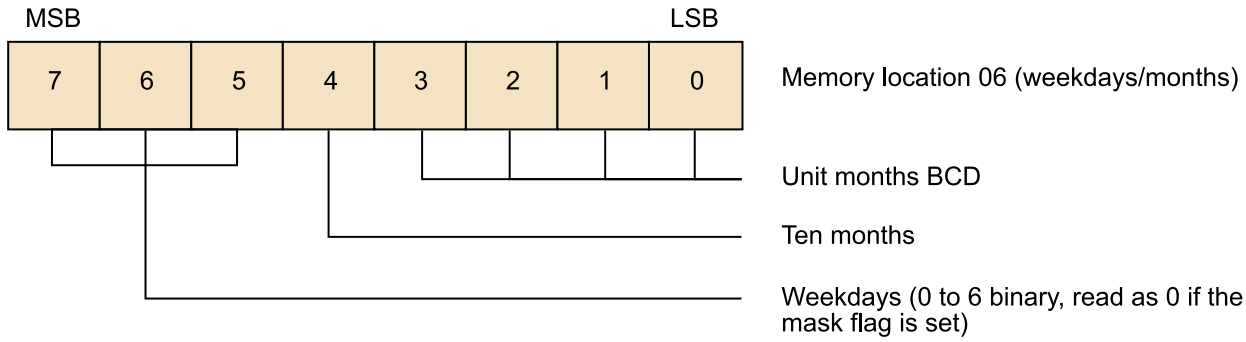


Este registro proporciona el día del mes en los bits 0-5 en BCD. En los dos bits más altos, encontramos el valor del año dentro de un ciclo de año bisiesto. Esto permite determinar el número de días del mes de febrero. A la hora de inicializar el valor de este registro, deberemos tenerlo en cuenta.

Ejemplo

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

Funcionalidad del registro 6 del PCF8593



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

Valores de inicialización de los registros del PCF8593

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

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

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

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

Trama I2C inicialización del PCF8593

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

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

Trama I2C de lectura de minutos y horas para el PCF8593

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

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

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

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

5.2. Programación del controlador

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

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

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

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

```
#define REGISTER_NUMBER ((uint8_t) 0x10)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

        date->weekday = temp;
}
```

Para el método de envío de la trama, se sigue un esquema semejante al de la tabla "Trama I2C inicialización del PCF8593". En primer lugar, se realiza la conversión del formato reloj interno al de datos del RTC con generateFrameData. Seguidamente, empezamos la transmisión de la trama. Para ello, enviamos en primer lugar una marca de inicio de I^2C. A continuación, enviamos la dirección de escritura del RTC y la dirección del primer registro al que queremos acceder (en este caso, el 0). Posteriormente enviamos los datos para programar el reloj. Finalmente, enviamos una marca de finalización.

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

        generateFrameData(date, data);

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

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

Para la recepción, el esquema es el de la tabla "Trama I2C de lectura de minutos y horas para el PCF8593". Empezamos la transmisión de la trama enviando en primer lugar una marca de inicio de I^2C. A continuación, enviamos la dirección de lectura del RTC y la dirección del primer registro al que queremos acceder, en este caso el 0. Posteriormente, leemos los datos para programar el reloj. Finalmente, enviamos una marca de finalización. Una vez recibida toda la trama, procesamos los datos guardándolos en una estructura de tipo date_t como la del reloj interno.

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

      startCondition();
      putByte(0xA3);
      putByte(0x00);

      for (i = 0; i < REGISTER_NUMBER; i++) {
            data[i] = getByte();
      }

      stopCondition();

      restoreFrameData(data, date);
}
```

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

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

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

      i2cInit();
      startTime = time(NULL);

      internalClk.cs = 0;
```

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

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

        writeClk(&internalClk);

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

        readClk(&internalClk);

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

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

        return EXIT_SUCCESS;
  }
```

De esta manera, podemos intercambiar información con el RTC. En el anexo, se encuentra el código para poder emular en un entorno Posix.

Actividad

Modificad el método restoreFrameData para que corrija el año y tenga en cuenta el año actual, y consiguiendo que no se pierda el valor al recibir los datos del RTC.

6. Sistemas operativos y librerías de soporte

A continuación se describen algunos de los SO y entornos de trabajo más comunes dentro de este ámbito.

6.1. TinyOS

TinyOS es un sistema operativo libre y de código abierto basado en componentes y orientado a plataformas para redes de sensores inalámbricos (WSN). TinyOS está pensado para sistema empotrados y se programa en lenguaje nesC, como un conjunto de tareas y procesos cooperativos.

El nesC es un dialecto del C, optimizado para trabajar con microcontroladores con pocos recursos de memoria. Se basa en componentes que podrían ser considerados como objetos. Estos proporcionan abstracciones de los dispositivos disponibles, de manera equivalente a los controladores de dispositivos de un SO de sobremesa.

TinyOS facilita un entorno de desarrollo basado en línea de comandos. Dicho entorno integra los diferentes componentes escritos en nesC, modificándolos para generar un único fichero en C. Este fichero incorpora el código de los diferentes componentes, entre los que se incluye el gestor de tareas. El programa puede ser compilado de manera cruzada haciendo uso de GCC.

El gestor de tareas está basado en eventos y no incorpora mecanismos predictivos. De este modo, se minimizan los requisitos de memoria. Teniendo en cuenta que los eventos asociados a E/S suelen estar relacionados con interrupciones, es importante llevar a cabo la división entre el evento y una tarea asociada por los motivos indicados en el subapartado "Sistemas operativos basados en eventos".

Existe también un gestor de tareas anticipativo, denominado TOSThreads. Este hace uso de dos colas, tal como se explica en el subapartado "Acceso al núcleo" (perteneciente al bloque "Sistemas operativos multitarea"). Este permite cargar programas de manera dinámica, al poder tener disponible un núcleo de TinyOS en la plataforma, si se considera necesario.

La gran ventaja de TinyOS es el gran número de plataformas disponibles, así como código y documentación, lo cual reduce la curva de aprendizaje.

6.2. FreeRTOS

FreeRTOS es un sistema operativo en tiempo real para dispositivos empotrados. Hay adaptaciones para diferentes microcontroladores. Se distribuye bajo licencia GPL, con una excepción que permite "código propietario a los usuarios siguen siendo de código cerrado, manteniendo el núcleo en sí como de código abierto", circunstancia que facilita el uso de FreeRTOS en aplicaciones propietarias.

FreeRTOS está diseñado para ser pequeño y simple. El núcleo en sí consta de tres o cuatro archivos en C. Para hacer el código legible y simplificar la adaptación a nuevas plataformas, y su mantenimiento, está escrito fundamentalmente en C. Hay algunas rutinas en ensamblador, que son dependientes de la arquitectura del microcontrolador (cambios de contexto, mútex, etc.). Las fuentes que se pueden descargar contienen configuraciones preparadas y demostraciones para todas las adaptaciones del compilador, lo que permite el diseño rápido de aplicaciones.

Enlace recomendado

La web http://www.FreeRTOS.org también dispone de tutoriales para su uso, detalles de su diseño, así como comparativas de rendimiento en diferentes plataformas.

En función de la programación, permite trabajar de manera anticipativa o colaborativa, por lo que resulta un buen modo de adentrarse en este tipo de sistemas operativos.

6.3. Contiki

Contiki se define como un sistema operativo de código abierto, multitarea y fácilmente portable para sistema empotrados en red y redes de sensores inalámbricas. Está concebido para microcontroladores con una reducida capacidad de memoria. Una configuración normal de Contiki requiere 2 kBytes de RAM y 40 kBytes de ROM.

Contiki está desarrollado por personas tanto del ámbito industrial como del académico. Lo dirige Adam Dunkels, del Instituto Sueco de Ciencias de la Computación. Este coordina un equipo formado por miembros de SICS, SAP, Cisco, Atmel, NewAE y TU Munich.

Como sistema operativo, está basado en eventos, aunque soporta múltiples hilos (*threads* o tareas según nuestra nomenclatura), para lo cual utiliza un planificador. También permite utilizar lo que denomina protohilos (*protothreads*), que son una implementación simplificada que no requiere una pila para cada hilo, ya que cada tarea se debe finalizar antes de continuar con la siguiente.

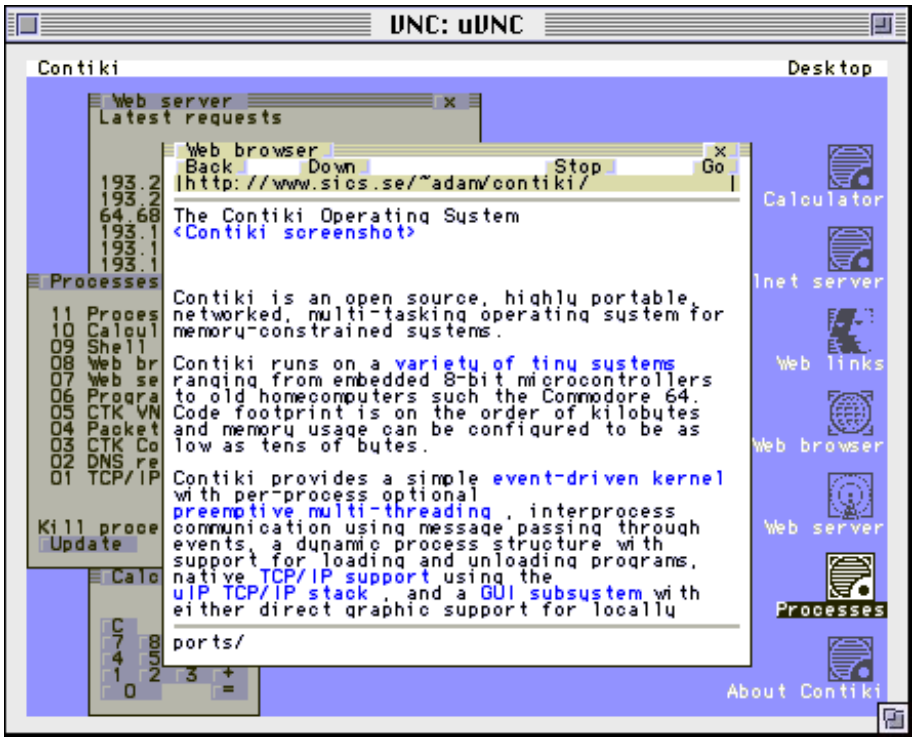
Contiki permite seleccionar para cada proceso si se utilizan múltiples. También permite intercambiar información entre procesos mediante mensajes. Por último, incluye la posibilidad de que el usuario acceda al sistema mediante línea de comando por Telnet, o de manera gráfica, haciendo uso del protocolo *virtual network computing* (VNC).

Una instalación completa de Contiki incluye las siguientes características:

- Núcleo multitarea.
- Subscripción opcional por tarea a multihilo.
- *Protothreads*.
- Protocolo de red TCP/IP, incluyendo IPv6.
- Interfaz gráfica para el usuario y sistema de ventanas.
- Acceso mediante *virtual network computing*.
- Un navegador web (decía ser el más pequeño del mundo).
- Servidor de web personal.
- Cliente sencillo de Telnet.
- Protector de pantalla.

Se ha transferido a múltiples arquitecturas, incluyendo la AVR de Atmel. En la figura siguiente se muestra el aspecto de este SO en un terminal VNC.

Aspecto de Contiki en un terminal VNC corriendo sobre un AVR



Como se puede observar, el resultado es notable trabajando sobre un pequeño microcontrolador.

6.4. QNX

QNX es un SO Unix comercial en tiempo real dirigido fundamentalmente a sistemas empotrados comerciales. El producto fue originalmente diseñado por QNX Software Systems, compañía que fue comprada por Research In Motion (los fabricantes de los teléfonos BlackBerry).

QNX se autodefine como el único SO verdaderamente basado en *microkernel*. El motivo aducido es que el núcleo de QNX únicamente contiene el gestor de procesos, comunicación entre procesos, redirección de interrupciones y temporizadores. El resto de los procesos del SO corre en modo usuario, incluyendo el proceso que crea nuevos procesos y gestiona la memoria (proc), trabajando de manera conjunta con el núcleo.

Los procesos del SO se conocen como **servicios**. Su utilización permite una gran modularidad, al poder activar o desactivar dicha funcionalidad ejecutando, o no, el servicio correspondiente. Es más, si aparece un problema en un servicio, su proceso se reinicia, sin afectar al *microkernel*.

QNX Neutrino ha sido adaptado a una serie de plataformas y ahora funciona en prácticamente cualquier CPU que se utiliza en el mercado empotrado. Esto incluye el PowerPC, la familia x86, MIPS, SH-4 y los relacionados con la familia ARM.

La interfaz de trabajo con las aplicaciones es Posix, lo cual simplifica la adaptación desde otros sistemas que cumplan con dicho estándar. En concreto, el perfil 52[2].

(2)IEEE Std 1003.13-2003, IEEE Standard for Information Technology - Standardized Application Environment Profile (AEP) - POSIX® Realtime and Embedded Application Support.

6.5. RTEMS

El sistema ejecutivo multiprocesador en tiempo real[3] es un sistema operativo en tiempo real de código libre y abierto diseñado para sistemas empotrados.

(3)En inglés, *real-time executive multiprocesor system* (RTMS).

Las siglas RTEMS inicialmente representaban sistema ejecutivo para misiles en tiempo real, que pasó con el tiempo a ser sistema ejecutivo para el ámbito militar en tiempo real, antes de cambiar a su significado actual. El desarrollo de RTEMS comenzó a finales de 1980 y las primeras versiones disponibles por FTP aparecieron en 1993. OAR Corporation es actualmente la gestora del proyecto RTEMS, en cooperación con un comité directivo que incluye a representantes de los usuarios.

RTEMS está diseñado para soportar varios estándares abiertos, incluyendo API Posix y uITRON. La API, conocida actualmente como la API clásica de RTEMS, se basó originalmente la especificación de la definición de interfaz para tiempo real ejecutivo (RTEID). RTEMS incluye un puerto de la pila TCP/IP FreeBSD. También dispone de soporte para varios sistemas de ficheros incluyendo NFS y el sistema de archivos FAT.

RTEMS no proporciona ningún tipo de gestión de memoria o procesos. En la terminología Posix, implementa un único proceso, que puede contener varios hilos o tareas. En este sentido, se asume un único procesador que no dispone

de unidad de gestión de memoria. También incluye un sistema de ficheros y acceso asíncrono a periféricos, por lo que entroncaría directamente con el perfil 52 de Posix, como el QNX.

RTEMS se utiliza para muchas aplicaciones. La comunidad experimental Physics and Industrial Control System colabora continuamente con él y también es muy utilizado en proyectos para espacio, ya que proporciona soporte para la mayor parte de las arquitecturas utilizadas en este entorno.

RTEMS se distribuye bajo una modificación de la licencia GPL, lo que permite la vinculación de objetos RTEMS con otros archivos sin necesidad de que los últimos sean GPL, de modo semejante a FreeRTOS. Esta licencia se basa en la actualización GNAT de GPL con una modificación para no ser específicas para el lenguaje de programación Ada.

Resumen

El módulo ha presentado, desde un punto de vista práctico, las necesidades de la gestión de procesos y abstracciones de sistema operativo para entornos empotrados. Este bloque nos ha guiado a través de ejemplos hacia las técnicas más relevantes para la gestión de procesos en entornos de propósito específico. Desde las técnicas más sencillas, basadas en colas *round-roben*, hasta sistemas preemptivos o de tiempo real, hemos visto muestras de cómo programar estas abstracciones, pensando en las restricciones propias de un sistema de propósito específico. Por otro lado, se han visto los beneficios de una programación orientada a acontecimientos, haciendo uso de las interrupciones hardware y las particularidades que deben tener las rutinas de servicio a la interrupción. El módulo también nos ha presentado los sistemas operativos más utilizados actualmente en entornos empotrados. Podemos destacar, entre otros, FreeRTOS y TinyOS como dos exponentes con un uso bastante extendido. Finalmente, a modo de guía de aprendizaje, el módulo nos ha adentrado en la creación de controladores de hardware o drivers a partir del esquema de características (*datasheet*) de un controlador de reloj en tiempo real (RTC).

Bibliografía

Barr, M.; Oram, A. (ed.). (1998). *Programming Embedded Systems in C and C++* (1.ª ed.). Sebastopol, California: O'Reilly and Associates.

Kamal, R. (2008). *Embedded Systems: Architecture, Programming and Design*. Nueva York: McGraw-Hill.

Labrosse, J. J. (2000). *Embedded Systems Building Blocks* (2.ª ed.). San Francisco, California: CMP Books.

Anexo

Para poder probar de manera sencilla los diferentes ejemplos, incluimos unas modificaciones que permiten comprobar dicho programa en un ordenador de sobremesa con una plataforma Posix. Las modificaciones que se requieren están fundamentalmente orientadas a las entradas y las interrupciones.

1) Gestor Round-Robin

El primer ejemplo es el del lector Round Robin. El primer elemento necesario es la lectura del teclado. Se utiliza un método sin bloqueo, de modo que la aplicación continúe trabajando, de manera equivalente a como funcionaría en un microcontrolador. Para ello, modifica el comportamiento del terminal Posix:

```
ssize_t kbread(void *buf, size_t count) {
      struct termios oldt, newt;
      int ch;
      ssize_t length;

      // Reading the actual terminal attributes
      tcgetattr(STDIN_FILENO, &oldt);
      newt = oldt;
      // Modifying the attributes to avoid blocking the stdin
      newt.c_lflag &= ~(ICANON | ECHO);
      // Modifying the terminal parameters
      tcsetattr(STDIN_FILENO, TCSANOW, &newt);
      // Reading the stdin stream
      length = read(STDIN_FILENO, buf, count);
      // Returning to the initial terminal attributes
      tcsetattr(STDIN_FILENO, TCSANOW, &oldt);

      // Returning the number of read keys
      return length;
```

Ahora solo es necesario adaptar el método de lectura de los botones para trabajar con esta función. Para ello, modificamos el código de getButtons.

```
void getButtons(struct buttons_t *buttons) {
      /*
      * int data = IOPORT;
      *
      * if ((data & UP_BUTTON) != 0) buttons->up = true;
      * else buttons->up = false;
```

```
        * if ((keys & DOWN_BUTTON) != 0) buttons->down = true;
        * else buttons->down = false;
        * if ((keys & MODE_BUTTON) != 0) buttons->mode = true;
        * else buttons->mode = false;
        */

        // Keys buffer size
        const int keysLength = 80;
        // Keys buffer definition
        unsigned char keys[keysLength];
        // Button value
        int button;
        // Pointer
        int i;
        // Keys buffer length result
        ssize_t length;

        // Get the available keys
        length = kbread(keys, keysLength);

        // Reset all the buttons
        buttons->up = false;
        buttons->down = false;
        buttons->mode = false;

        // Loop the buffer keys
        for (i = 0; i < length; i++) {
              // Copy to button the key
              button = keys[i];

              // Depending on the button set the
              // corresponding variable
              switch (button) {
              case 'u':
                    buttons->up = true;
                    break;
              case 'd':
                    buttons->down = true;
                    break;
              case 'm':
                    buttons->mode = true;
                    break;
              }
        }
  }
```

El siguiente paso es la lectura de la temperatura. Esta la emulamos con una sinusoide que varía en el tiempo. Para ello, definimos primero las constantes que definen la sinusoide:

```
const double TEMP_MEAN = 25.0;
const double TEMP_AMP = 5.0;
const int TEMP_FREQUENCY = 10;
```

A partir de ellas definimos la función que nos simula las variaciones de temperatura:

```
uint16_t getADCValue(void) {
      // Seconds from the Epoch
      time_t seconds;
      // Simulated temperature
      double tBase;
      // Simulated ADC value
      double vADC;
      // Resulting function value
      uint16_t result;

      // Get the clock value and make the module to avoid sin function saturation
      seconds = clock() % frequency;
      // Temperature evolution function
      tBase = TEMP_MEAN + TEMP_AMP *
                  sin( (2.0 * M_PI * seconds) / TEMP_FREQUENCY );

      // ADC value based on the sensor function
      vADC = 18.43 * tBase + 1126.125;

      // Rounding to get the integer value
      result = lround(vADC);

      return result;
}
```

Seguidamente, modificamos getTemp para utilizar en lugar de un registro, el resultado de la función anterior.

```
int16_t getTemp(void) {
      uint16_t adcValue16 = getADCValue();
      int32_t adcValue32 = adcValue16;
      int16_t result;
      adcValue32 <<= 5;
      adcValue32 /= 59;
      result = (int16_t) (adcValue32 - 610);
```

```
        return result;
}
```

Con ello, ya es posible simular el comportamiento de la aplicación básica.

2) Gestor síncrono Round-Robin

Para el gestor síncrono es necesario incluir la emulación de interrupciones, además de métodos de gestión del consumo. Empezamos por esos últimos, y en concreto, el de lowPowerMade. Para ello, hacemos uso del método pause, el cual nos permite dejar dormida. Este método se tiene que utilizar con cuidado, ya que deja completamente parado el método donde se encuentra, y sólo una señal del sistema operativo puede continuarla.

```
inline void lowPowerMode(void) {
        pause();
}
```

Además, necesitamos crear el método standardMode, que en este caso no hace ninguna acción, pero que en un microcontrolador probablemente incluiría alguna instrucción.

```
inline void standardMode(void) {
        ;
}
```

Para generar la interrupción de reloj, necesitamos realizar unos pasos previos. El motivo se debe a que para trabajar con señales, que sería el equivalente de las interrupciones en un sistema Posix, necesitamos un formato específico de método. Sin embargo, en nuestro caso, estamos utilizando uno más propio de los microcontroladores. Por este motivo, hemos de crear un método "contenedor" que llame a nuestro método interrupción. Para ello, definimos nuestro propio tipo de método (interrupt_t), una variable global con este tipo, para poder acceder a ella desde diferentes métodos (clock_handler), y el método contenedor (clockHandler), que únicamente llama al método indicado por clock_handler. Los diferentes elementos se muestran a continuación.

```
typedef void (*interrupt_t)(void);

interrupt_t clock_handler;

void clockHandler(int signum) {
        clock_handler();
}
```

Seguidamente, debemos dar de alta y de baja la interrupción de reloj (en el caso de Posix, la señal del reloj). Para ello, creamos en primer lugar una estructura donde guardaremos el método que tiene actualmente asignado la señal (old_action).

```
struct sigaction old_action;
```

Seguidamente, creamos el método para dar de alta la interrupción (startInterruptHandler) y asignar el método de ésta, así como indicar el período con el que se debe llevar a cabo. Para ello, debemos llevar a cabo diferentes pasos y, al estar accediendo a puntos críticos del SO, realizarlo con todas las precauciones necesarias. El primer paso es mirar si hay alguna acción asignada a la señal que queremos utilizar –en nuestro caso SIGALRM, ya que se comporta de manera equivalente a una interrupción de un temporizador en un microcontrolador–. Si esta es distinta de SIG_IGN, la sustituimos por nuestra interrupción. Como hemos dicho anteriormente, en este caso será el método contenedor de nuestra interrupción, clockHandle.

Una vez dado de alta el método, ponemos en marcha la interrupción, para lo cual asignamos a la variable timerActualStatus el tiempo que debe transcurrir hasta el próximo salto en la estructura it_value. En el campo it_interval, indicamos el período con el que se repetirá dicha señal. Una vez dados los valores, estos son asignados al temporizador ITIMER_REAL mediante el método setitimer.

```
void startInterruptHandler(interrupt_t handler, const uint32_t usperiod) {
      struct itimerval timerActualStatus;
      struct sigaction new_action;

      /* Specify the interrupt handler */
      clock_handler = handler;
      /* Set up the structure to specify the new action. */
      new_action.sa_handler = clockHandler;
      sigemptyset (&new_action.sa_mask);
      new_action.sa_flags = 0;

      /* Read de old action */
      sigaction(SIGALRM, NULL, &old_action);

      /* If no action is expected, set the interrupt as the new action. */
      if (old_action.sa_handler != SIG_IGN) {
            sigaction(SIGALRM, &new_action, NULL);

            timerActualStatus.it_value.tv_sec = usperiod/1000000;
            //Conversion to microsecond
            timerActualStatus.it_value.tv_usec = usperiod%1000000;
            //Periodic Wake time
```

```
            timerActualStatus.it_interval.tv_sec = timerActualStatus.it_value.tv_sec;
            timerActualStatus.it_interval.tv_usec = timerActualStatus.it_value.tv_usec;

            /* Set the time period */
            setitimer(ITIMER_REAL, &timerActualStatus, NULL);
      }
}
```

Para detener la señal, seguimos el proceso inverso. Primero, paramos el temporizador asignando valores nulos a it_value e it_interval y, seguidamente, volvemos a asignar la antigua interrupción a la señal.

```
void stopInterruptHandler(void) {
      struct itimerval timerActualStatus;

      timerActualStatus.it_value.tv_sec = 0; //Stop the timer
      timerActualStatus.it_value.tv_usec = 0; //And no microsecond
      timerActualStatus.it_interval.tv_sec = 0; //Next wake up time would be 1 second
      timerActualStatus.it_interval.tv_usec = 0; //And no microsecond

      /* Stop the interrupt setting the time to 0 */
      setitimer(ITIMER_REAL, &timerActualStatus, NULL);
      /* Set the old action as the action that has to be done */
      sigaction(SIGALRM, &old_action, NULL);
}
```

Por último, conviene indicar que, en general, este último método no será utilizado, ya que en un microcontrolador nunca deberíamos parar el reloj principal. En cualquier caso, lo incluimos como ejemplo, por si se desea utilizar en otro escenario.

3) Gestor basado en eventos

Para el gestor basado en eventos, es necesario incluir la emulación de interrupciones como en el gestor Round-Robin síncrono. Para ello, creamos una nueva interrupción que mira si se ha pulsado algún botón del teclado y, en ese caso, modifica el valor de la estructura asociada con setButtonsValue. Además, llama a buttonsInterrupt para emular una interrupción de botones. A su vez, en cada ciclo de reloj modifica el valor del ADC con setADCValue y llama a la interrupción del ADC.

```
void timeInterrupt(void) {
      if (kbhit()) {
            setButtonsValue();
            buttonsInterrupt();
      }
```

```
        setADCValue();
        adcInterrupt();
}
```

El método de inicio de interrupción da de alta la interrupción de temporizador, que llamará a la rutina timeInterrupt.

```
void interruptInit(void) {
        startInterruptHandler(timeInterrupt, US_PERIOD); // 0.1 sec
}
```

Además, creamos la función getADCValue, que lee el valor de la variable global adcRegister, y el método setADCValue, que modifica de manera continua el valor de la temperatura, de modo equivalente a getADCValue en el punto 1 de este anexo.

```
inline uint16_t getADCValue(void) {
        return adcRegister;
}

void setADCValue(void) {
        // Seconds from the Epoch
        time_t seconds;
        // Simulated temperature
        double tBase;
        // Simulated ADC value
        double vADC;
        // Resulting function value
        uint16_t result;
        // Temperature evolution frequency
        const int frequency = 10;

        // Get the clock value and make the module to avoid sin function saturation
        seconds = clock() % frequency;
        // Temperature evolution function
        tBase = TEMP_MEAN + TEMP_AMP * sin( (2.0 * M_PI * seconds) / TEMP_FREQUENCY);

        // ADC value based on the sensor function
        vADC = 18.43 * tBase + 1126.125;

        // Rounding to get the integer value
        result = lround(vADC);

        adcRegister = result;
}
```

Para los botones, seguimos un esquema equivalente, aunque la función setButtonsValue procesa las posibles teclas (u, d, m) para determinar qué bit de la variable buttonsRegister debe modificar.

```
inline uint16_t getButtonsValue(void) {
      return buttonsRegister;
}

void setButtonsValue(void) {
      // Keys buffer size
      const int keysLength = 80;
      // Keys buffer definition
      unsigned char keys[keysLength];
      // Button value
      int button;
      // Pointer
      int i;
      // Keys buffer length result
      ssize_t length;

      // Get the available keys
      length = kbread(keys, keysLength);

      // Reset all the buttons
      buttonsRegister = 0;

      // Loop the buffer keys
      for (i = 0; i < length; i++) {
            // Copy to button the key
            button = keys[i];

            // Depending on the button set the
            // corresponding variable
            switch (button) {
            case 'u':
                  buttonsRegister |= (1 << UP_BUTTON);
            break;
            case 'd':
                  buttonsRegister |= (1 << DOWN_BUTTON);
            break;
            case 'm':
                  buttonsRegister |= (1 << MODE_BUTTON);
                  break;
            }
      }
}
```

Con estas rutinas es posible emular el sistema operativo basado en eventos en un ordenador basado en un SO Posix.

4) Controlador de periféricos

En este caso, hacemos un emulador del PCF8593 para poder simular su comportamiento. Empezamos definiendo el número de registros (REGISTER_NUMBER) y el tiempo de período (US_PERIOD).

```
#define REGISTER_NUMBER     ((uint8_t) 0x10)
#define US_PERIOD           ((uint32_t) 10000)
```

Seguidamente se define el conjunto de elementos que permiten emular el PCF8593. En este caso, los registros asociados al tiempo con las variables centésima de segundo (cs) a año (year). De la misma manera, el control del estado de la trama I^2C (framePosition), el modo de trabajo (read_write), los índices y los *buffers* para enviar y recibir datos.

```
volatile struct {
      struct {
            uint8_t cs;
            uint8_t sec;
            uint8_t min;
            uint8_t hour;
            uint8_t day;
            uint8_t month;
            uint8_t year;
            uint8_t weekday;
            bool h12_24;
      } RTCtime;
      enum {
            START, ADDRESS, REGISTER, DATA, ERROR
      } framePosition;
      bool read_write;
      uint8_t offset;
      uint8_t index;
      uint8_t regs[REGISTER_NUMBER];
      uint8_t latches[REGISTER_NUMBER];
} pcf8593;
```

Las siguientes funciones son copias de las del gestor síncrono, por lo que solo se indica el nombre. Estas permiten lanzar cada cierto período de tiempo un método, semejante al proceso que seguiría el contador del PCF8593.

```
struct sigaction old_action;


typedef void (*interrupt_t)(int);
```

```
void startInterruptHandler(interrupt_t handler, const uint32_t usperiod);

void stopInterruptHandler(void);
```

El siguiente método permite copiar los datos de los campos de fecha a los registros que se transfieren mediante I^2C. Tiene en cuenta los diferentes bits que utiliza el integrado para transferir toda la información.

```
void generateRegisters(bool latch) {
       uint8_t hour12;
       uint8_t temp;
       uint8_t *buffer;

       if (latch) buffer = pcf8593.latches;
       else buffer = pcf8593.regs;

       temp = ((pcf8593.RTCtime.cs / 10) << 4);
       temp |= (pcf8593.RTCtime.cs % 10);
       buffer[1] = temp;

       temp = ((pcf8593.RTCtime.sec / 10) << 4);
       temp |= (pcf8593.RTCtime.sec % 10);
       buffer[2] = temp;

       temp = ((pcf8593.RTCtime.min / 10) << 4);
       temp |= (pcf8593.RTCtime.min % 10);
       buffer[3] = temp;

       if (pcf8593.RTCtime.h12_24 == false) {
              temp = ((pcf8593.RTCtime.hour / 10) << 4);
              temp |= (pcf8593.RTCtime.hour % 10);
              buffer[4] = temp;
       } else {
              temp = pcf8593.RTCtime.hour;
              hour12 = (temp / 2) + 1;
              temp /= 12;

              temp <<= 6;
              temp |= ((hour12 / 10) << 4);
              temp |= (hour12 % 10);
              temp |= 0x80;
              buffer[4] = temp;
       }

       temp = ((pcf8593.RTCtime.day / 10) << 4);
       temp |= (pcf8593.RTCtime.day % 10);
```

```
        temp |= (pcf8593.RTCtime.year << 6);
        buffer[5] = temp;

        temp = ((pcf8593.RTCtime.month / 10) << 4);
        temp |= (pcf8593.RTCtime.month % 10);
        temp |= (pcf8593.RTCtime.weekday << 5);
        buffer[6] = temp;
}
```

Del mismo modo, el siguiente método realiza la tarea contraria, convirtiendo los datos del formato de los registros a variables.

```
void generateVariables(void) {
      uint8_t hour12;
      uint8_t temp;

      for( ; pcf8593.offset < pcf8593.index; pcf8593.offset++) {
            switch (pcf8593.offset) {
            case 1:
                  pcf8593.RTCtime.cs = ((pcf8593.regs[1] >> 4) * 10) +
                              (pcf8593.regs[1] & 0x0F);
                  break;
            case 2:
                  pcf8593.RTCtime.sec = ((pcf8593.regs[2] >> 4) * 10) +
                              (pcf8593.regs[2] & 0x0F);
                  break;
            case 3:
                  pcf8593.RTCtime.min = ((pcf8593.regs[3] >> 4) * 10) +
                              (pcf8593.regs[3] & 0x0F);
                  break;
            case 4:
                  temp = pcf8593.regs[4];
                  if ((temp & 0x80) != 0) {
                        hour12 = pcf8593.regs[4] & 0x1F;

                        pcf8593.RTCtime.hour = ((hour12 >> 4) * 10) + (hour12 & 0x0F);

                        if ((temp & 0x40) != 0) {
                              pcf8593.RTCtime.hour += 11;
                        } else {
                              pcf8593.RTCtime.hour -= 1;
                        }
                        pcf8593.RTCtime.h12_24 = true;
                  } else {
                        pcf8593.RTCtime.h12_24 = false;
                        pcf8593.RTCtime.hour = ((pcf8593.regs[4] >> 4) * 10) +
                        (pcf8593.regs[4] & 0x0F);
```

```
                }
                break;
            case 5:
                pcf8593.RTCtime.day = (((pcf8593.regs[5] & 0x30) >> 4) * 10) +
                        (pcf8593.regs[5] & 0x0F);
                pcf8593.RTCtime.year = (pcf8593.regs[5] >> 6);
                break;
            case 6:
                pcf8593.RTCtime.month = (((pcf8593.regs[6] & 0x10) >> 4) * 10) +
                        (pcf8593.regs[6] & 0x0F);
                pcf8593.RTCtime.weekday = (pcf8593.regs[6] >> 5);
                break;
            default:
                break;
            }
        }

        pcf8593.index = 0;
        pcf8593.offset = 0;
}
```

El siguiente método se encarga de incrementar los diferentes contadores actualizando la fecha.

```
void timeInterrupt(int n) {
        uint8_t monthDays;

        pcf8593.RTCtime.cs++;

        if (pcf8593.RTCtime.cs >= 100) {
            pcf8593.RTCtime.cs = 0;
            pcf8593.RTCtime.sec++;

            if (pcf8593.RTCtime.sec >= 60) {
                pcf8593.RTCtime.sec = 0;
                pcf8593.RTCtime.min++;

                if (pcf8593.RTCtime.min >= 60) {
                    pcf8593.RTCtime.min = 0;
                    pcf8593.RTCtime.hour++;

                    if (pcf8593.RTCtime.hour >= 24) {
                        pcf8593.RTCtime.hour = 0;
                        pcf8593.RTCtime.day++;
                        pcf8593.RTCtime.weekday++;

                        if (pcf8593.RTCtime.weekday >= 7) {
```

```
                                    pcf8593.RTCtime.weekday = 0;
                              }

                              switch (pcf8593.RTCtime.month) {
                              case 1:
                              case 3:
                              case 5:
                              case 7:
                              case 8:
                              case 10:
                              case 12:
                                    monthDays = 31;
                                    break;
                              case 4:
                              case 6:
                              case 9:
                              case 11:
                                    monthDays = 30;
                                    break;
                              case 2:
                                    if (pcf8593.RTCtime.year == 0) {
                                          monthDays = 29;
                                    } else {
                                          monthDays = 28;
                                    }
                                    break;
                              default:
                                    pcf8593.RTCtime.month = 1;
                              }

                              if (pcf8593.RTCtime.day > monthDays) {
                                    pcf8593.RTCtime.day = 1;
                                    pcf8593.RTCtime.month++;

                                    if (pcf8593.RTCtime.month > 12) {
                                          pcf8593.RTCtime.month = 1;
                                          pcf8593.RTCtime.year++;

                                          if (pcf8593.RTCtime.year >= 4) {
                                                pcf8593.RTCtime.year = 0;
                                          }
                                    }
                              }
                        }
                  }
            }
      }
```

```
        generateRegisters(false);
}
```

Por último, el método i2cInit inicializa los diferentes elementos que emularían el integrado, así como la interrupción.

```
void i2cInit() {
        int i;



        for (i = 0; i < REGISTER_NUMBER; i++)
            pcf8593.regs[i] = 0;

        pcf8593.RTCtime.cs = 0;
        pcf8593.RTCtime.sec = 0;
        pcf8593.RTCtime.min = 0;
        pcf8593.RTCtime.hour = 0;
        pcf8593.RTCtime.day = 1;
        pcf8593.RTCtime.month = 1;
        pcf8593.RTCtime.year = 0;
        pcf8593.RTCtime.weekday = 0;
        pcf8593.RTCtime.h12_24 = false;

        startInterruptHandler(timeInterrupt, US_PERIOD);
}
```

Con todo ello, se consigue emular el PCF8593, de modo que el programa puede leer datos. Además, necesitamos el *driver* que emula el periférico de I^2C del microcontrolador. Empezamos con el método startCondition para poder enviar el mensaje.

```
void startCondition(void) {
        if (pcf8593.framePosition == START) {
              pcf8593.framePosition = ADDRESS;
        } else {
              pcf8593.framePosition = ERROR;
        }
}
```

Si la posición actual es START, se inicia el procesado de la trama; en caso contrario, se indica un error. En la siguiente posición hay que añadir la trama en sí, cuyos datos dependen de si se quiere escribir o leer del RTC.

```
void putByte(uint8_t value) {
        switch (pcf8593.framePosition) {
        case ADDRESS:
```

```
            if (value == 0xA2) {
                  pcf8593.framePosition = REGISTER;
                  pcf8593.read_write = false;
            } else if (value == 0xA3) {
                  pcf8593.framePosition = REGISTER;
                  pcf8593.read_write = true;
                  generateRegisters(true);
            } else {
                  pcf8593.framePosition = ERROR;
            }
            break;
      case REGISTER:
            if (value < REGISTER_NUMBER) {
                  pcf8593.offset = value;
                  pcf8593.index = value;
                  pcf8593.framePosition = DATA;
            } else {
                  pcf8593.framePosition = ERROR;
            }
            break;
      case DATA:
            if (pcf8593.read_write == false) {
                  if (pcf8593.index < REGISTER_NUMBER) {
                        pcf8593.regs[pcf8593.index] = value;
                        pcf8593.index++;
                  } else {
                        pcf8593.framePosition = ERROR;
                  }
            } else {
                  pcf8593.framePosition = ERROR;
            }
            break;
      default:
            pcf8593.framePosition = ERROR;
      }
}

uint8_t getByte(void) {
      uint8_t value;

      switch (pcf8593.framePosition) {
      case DATA:
            if (pcf8593.read_write == true) {
                  if (pcf8593.index < REGISTER_NUMBER) {
                        value = pcf8593.latches[pcf8593.index];
                        pcf8593.index++;
                  } else {
```

```
                        pcf8593.framePosition = ERROR;
                  }
            } else {
                  pcf8593.framePosition = ERROR;
            }
            break;
      default:
            pcf8593.framePosition = ERROR;
      }

      return value;
}

void stopCondition(void) {
      pcf8593.framePosition = START;
      if (pcf8593.read_write == false) {
            generateVariables();

            pcf8593.read_write = true;
      }
}
```