El software

Francisco Hernández Ramírez
Juan Daniel Prades García

PID_00177262

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

Tal como se ha descrito en módulos anteriores, los sistemas empotrados son sistemas informáticos orientados a la ejecución de una tarea concreta con unas especificaciones de trabajo muy bien definidas. Este alto nivel de especificación permite simplificar los elementos de hardware al mínimo imprescindible para llevar a cabo la tarea minimizando su coste.

Ejemplo

Como especificaciones, podríamos hablar de la necesidad o no de trabajar en tiempo real o de las limitaciones en el consumo de energía.

Desde el punto de vista del software, este hecho provoca que, a menudo, los programas que gobiernan el funcionamiento de los sistemas empotrados, el microsoftware, se tengan que ejecutar en unos recursos de hardware limitados: poca memoria, potencia de cálculo moderada, interfaces de entrada y salida inexistentes o con limitaciones importantes, etc. Por lo tanto, en el mundo de los sistemas empotrados resulta especialmente importante elaborar código compacto y eficiente que permita un aprovechamiento máximo del hardware disponible. En el apartado 2, "Conceptos básicos del software para sistemas empotrados", describiremos algunas de las técnicas y herramientas de programación que permiten incrementar la eficiencia de nuestros programas.

En particular, las aplicaciones en las que se utilizan sistemas empotrados están relacionadas con el control de procesos de características muy diferentes, desde el control de un ascensor, pasando por la gestión del motor de un coche hasta un dispositivo de electrónica de consumo. Todas estas aplicaciones tienen dos cosas en común.

Por un lado, es necesario proveer al sistema empotrado de información sobre el entorno que ha de gobernar. Desde el punto de vista del hardware, esto se consigue mediante la integración de diferentes periféricos conectados a los elementos de proceso por medio de diferentes buses, tal como se ha descrito en módulos anteriores. Desde el punto de vista del software, es necesario que las señales provenientes de los periféricos modifiquen el desarrollo del flujo de ejecución del programa. Los elementos que permiten controlar el flujo de software se describen en el subapartado 2.1, "Interrupciones".

Por otro lado, se espera que este tipo de sistema trabaje en **tiempo real**, es decir, que hayan de cumplir especificaciones estrictas en cuanto al tiempo de ejecución de las diferentes tareas.

Ejemplo

Es obvio que retardos de varios milisegundos en la ejecución de un programa, que pueden ser aceptables, por ejemplo, en un ordenador convencional, no lo sean en el sistema empotrado responsable de disparar el airbag de un automóvil.

Computación en tiempo real

Entendemos por *computación en tiempo real* o *computación reactiva* todos los sistemas informáticos que deben cumplir, de una manera estricta, alguna restricción en el tiempo

de respuesta, es decir, el tiempo desde la sucesión de un acontecimiento y la ejecución de la respuesta por el sistema.

En el subapartado 2.2, "Funciones y punteros", se estudiarán las características del software que permite trabajar en tiempo real y en el 2.3, "Particularidades de la programación de sistemas empotrados", estrategias para organizar y priorizar la ejecución de las diferentes tareas del sistema.

Finalmente, no hay que olvidar que la prioridad del programador debe ser conseguir que el sistema empotrado se comporte de acuerdo con sus especificaciones funcionales. Hay diferentes modelos de programación o arquitecturas de software que facilitan la síntesis de programas con comportamientos muy determinados que se estudiarán en el apartado 3, "Modelos de programación".

Todos estos factores añaden un grado de complejidad a la programación para sistemas empotrados que no existe en la programación de ordenadores de propósito general convencionales. Por este motivo, es recomendable partir de unos conocimientos sólidos en programación antes de iniciar el estudio de este módulo.

Objetivos

Los materiales didácticos de este módulo contienen las herramientas necesarias para lograr los objetivos siguientes:

1. Saber elegir las herramientas de programación y el lenguaje más adecuado para desarrollar software para sistemas empotrados.

2. Conocer el concepto de *interrupción* como mecanismo básico para comunicarse con los periféricos de un sistema empotrado en tiempo real.

3. Entender los punteros como herramientas para una utilización eficiente de la memoria y saberlos utilizar.

4. Conocer las funciones *inline* y externas como técnicas de programación que permiten compactar y hacer más modular el código.

5. Comprender la necesidad y el funcionamiento de sistemas empotrados basados en bucles de control simple con interrupciones o sin ellas.

6. Saber diseñar sistemas basados en máquinas de estados para procesos de monitorización y control de complejidad intermedia.

7. Introducirse en el funcionamiento de los sistemas multitarea.

1. Cuestiones preliminares

A continuación abordaremos algunos aspectos introductorios.

1.1. Elección del lenguaje de programación

Como en la programación convencional, antes de empezar a programar sistemas empotrados, hay que elegir con cuidado el lenguaje de programación que usaremos. En los sistemas empotrados esta es una decisión particularmente relevante, puesto que, en muchos casos, programar de manera eficiente requiere poder acceder a las características de bajo nivel del hardware. Estas son algunas de las consideraciones que hay que tener en cuenta a la hora de tomar esta decisión:

- Las unidades de procesamiento (ya sean microprocesadores de propósito general o microcontroladores o DSP de los sistemas empotrados) solo aceptan instrucciones en código máquina.

- El código máquina es, por definición, el lenguaje del ordenador y no el del programador. Por lo tanto, la interpretación del código máquina es complicada para los programadores y fácilmente induce a errores.

- Cualquier programa, ya esté escrito en ensamblador, C, C++, Java, etc., tiene que ser traducido a código máquina antes de su ejecución. Por este motivo, no tiene mucho sentido crear un código fuente "perfecto" si se usa un traductor poco eficiente que haga que el código no funcione como se pretendía.

- Comparados con los microprocesadores de ordenadores convencionales, los sistemas empotrados suelen tener una potencia de cálculo limitada y poca memoria disponible; por lo tanto, el lenguaje utilizado debe ser eficiente.

- Para programar sistemas empotrados, a menudo hay que acceder a un bajo nivel a los recursos de hardware; es decir, como mínimo, poder leer y escribir de una posición de memoria concreta direccionable de manera directa. Esto lo hacen posible los punteros u otros mecanismos equivalentes que estudiaremos en el subapartado 2.2, "Funciones y punteros".

También hay otros aspectos que no son puramente técnicos a la hora de elegir un lenguaje de programación:

- Desde un punto de vista de ingeniería de software, hay que poder reaprovechar el código desarrollado en proyectos previos. Por lo tanto, el lengua-

je elegido debe permitir emplear bibliotecas que faciliten la reutilización de componentes de código validados anteriormente. Además, es deseable que se puedan adaptar **códigos antiguos a nuevas versiones del hardware** con un esfuerzo mínimo.

- Muchos sistemas empotrados tienen un largo tiempo de vida en el mercado, durante el cual hay que introducir mejoras y revisiones. Por lo tanto, el lenguaje elegido debe **facilitar el mantenimiento del código**, tanto después de haber transcurrido mucho tiempo como por la acción de diferentes programadores.

- Para asegurar la integrabilidad en el sistema de componentes de software desarrollados por terceros, garantizar la compatibilidad con tecnologías más o menos estándar, tener disponibles amplias fuentes de información y reducir el tiempo de formación del personal implicado en el proyecto, el lenguaje elegido debe ser de **uso común**.

Esta breve lista de las características deseables en un lenguaje de programación para sistemas empotrados ya hace ver que existe una paradoja entre las necesidades de eficiencia y de control a bajo nivel propias del código máquina y la inteligibilidad y facilidad de uso de los lenguajes de alto nivel.

> Inevitablemente, aquí hay que tomar una decisión de compromiso y que se aproxime tanto como sea posible al ideal:
>
> "[...] de un lenguaje eficiente, de alto nivel, que proporcione acceso de bajo nivel al hardware y que esté muy definido y, obviamente, que sea compatible con las plataformas de hardware que queremos utilizar".

Una elección habitual, que satisface buena parte de estos aspectos, ha sido el C. A continuación, se resumen algunas de sus características principales.

- Es un lenguaje de "nivel medio", con características de alto nivel (como el soporte de funciones, módulos y bibliotecas) y características de bajo nivel (como un buen acceso al hardware mediante punteros).

- Es muy eficiente.

- Hay buenos compiladores disponibles optimizados para la mayoría de los procesadores de uso habitual en sistemas empotrados (desde 8 bits hasta 32 bits o más).

- Es fácil de aprender para programadores acostumbrados a lenguajes de propósito general como Java o C++.

- Es de uso común y está muy bien documentado.

1.2. Herramientas para la programación de sistemas empotrados

Los diseñadores de sistemas empotrados, como cualquier otro programador, necesitan compiladores, enlazadores y depuradores para desarrollar el microsoftware del sistema. Sin embargo, como la programación se hace en una plataforma (PC) diferente de la que ejecutará finalmente el código (sistema empotrado), hay que emplear algunas herramientas específicas que facilitan esta transición:

- Depuradores sobre hardware final o emuladores.
- Utilidades para añadir sumas de comprobación al programa, de manera que el sistema empotrado pueda validar su autenticidad e integridad antes de ejecutarlo.
- Para sistemas que integren un DSP, los desarrolladores pueden necesitar herramientas de procesamiento matemático, como MATLAB/Simulink, Scilab/Scicos, MathCad, Mathematica, etc., para simular el comportamiento de los algoritmos que hay que implementar.
- En estos mismos casos, suele ser necesario el uso de bibliotecas compatibles con el hardware elegido que implementen estas rutinas matemáticas.
- En algunos casos, puede ser necesario el uso de compiladores y enlazadores personalizados que mejoren la optimización para un hardware determinado.
- Aun así, algunos sistemas empotrados basados en hardwares avanzados pueden disponer de su lenguaje de programación propio.
- Alternativamente, se puede integrar un sistema operativo empotrado en el sistema.
- Para los sistemas empotrados basados en máquinas de estados, hay herramientas que permiten simularlas y validar su comportamiento, y también generar el código fuente que las implementa de manera automática.

Véase también

Encontraréis más información sobre esto en el módulo "Simulación y test".

Los proveedores de estas herramientas de software pueden ser muy variados:

- Empresas de software especializadas en el mercado de los sistemas empotrados.
- Herramientas libres procedentes de algún proyecto GNU.

- En ocasiones, también es posible emplear herramientas de programación de propósito general gracias a las similitudes de la unidad de proceso elegida con los microprocesadores de ordenadores convencionales.

En cualquier caso, la elección final de una herramienta o de otra viene condicionada por las necesidades del proyecto, su complejidad, la experiencia del personal implicado y el presupuesto disponible.

2. Conceptos básicos del software para sistemas empotrados

La necesidad de simplificar los elementos de hardware que conforman los sistemas empotrados para reducir su coste y complejidad técnica, al mismo tiempo que se ha de trabajar en tiempo real con unas especificaciones muy condicionadas a la aplicación final, hacen imprescindible desarrollar arquitecturas de software con unas características muy determinadas.

En esta sección repasaremos conceptos básicos de programación no exclusivos de este tipo de sistemas, pero que, aplicados convenientemente y de manera inteligente, permiten mejorar el funcionamiento global, como son la utilización de interrupciones y punteros. A pesar de que la utilización óptima de los recursos de un microprocesador integrado en un sistema empotrado únicamente sería posible desarrollando software en lenguaje máquina, la complejidad en su interpretación y también los numerosos errores humanos derivados de trabajar con él hacen inviable esta aproximación desde un punto de vista técnico y económico.

Finalmente, en la tercera y última parte de la sección veremos que la utilización de funciones *inline* y funciones externas pueden ser útiles para optimizar, cuando menos de una manera parcial, el funcionamiento de estos sistemas.

2.1. Interrupciones

Las interrupciones recogidas dentro del software de control de sistemas que funcionan en tiempo real son a menudo uno de los factores clave que permite su funcionamiento correcto en una aplicación determinada.

Podemos definir una interrupción como la señal que recibe el microprocesador para redirigir el flujo del programa que se está ejecutando en un momento determinado.

Ejemplo

Las señales que los periféricos de un ordenador envían a la CPU y que hacen que empiece una nueva tarea para, posteriormente, retomar la primera una vez finalizada la condicionada por el periférico sería un ejemplo típico de interrupción.

En dispositivos reales, la mayoría de los microprocesadores integrados en sistemas empotrados están conectados a componentes capaces de generar interrupciones muy variadas. En cambio, en cuanto al desarrollo de software, las mismas interrupciones, y también las rutinas de interrupción de servicio o ISR[1] necesarias para gestionarlas de manera adecuada, son a menudo pasadas por alto por los programadores noveles, puesto que el error más pequeño en su configuración y en su diseño suele originar un fallo general de los sistemas muy difícil de corregir a posteriori. De hecho, estos programadores suelen usar el esquema más simple posible de programación para evitar utilizarlos y que

(1)Sigla de *interrupt service routine*. A veces, también se emplea el término *interrupt service process* (ISP).

se basa en preguntar de una manera recursiva a los dispositivos generadores de interrupciones, por ejemplo, los periféricos, si necesitan algún servicio del microprocesador. Obviamente, esta solución, conocida en inglés como *polled communication*, funciona bien cuando el procesador es rápido y potente, pero en el caso de sistemas empotrados podemos identificar las desventajas siguientes:

- **Consumo de muchos recursos de microprocesador**, puesto que, independientemente de si el periférico solicita servicio o no, el procesador ha de preguntar de manera recursiva sobre su estatus.

- **Código de software poco ordenado**. A cada iteración del programa principal, este tiene que preguntar al periférico sobre su estatus y, en caso de originarse una necesidad de servicio, se debe incluir dentro del programa principal el código para tratarla. Todo ello acaba complicando bastante el código final del programa.

- **Estados de latencia elevados de los periféricos**. Si el microprocesador está ocupado haciendo otra función, el dispositivo periférico es ignorado hasta que el primero le pregunte sobre su estatus. En caso de que el microprocesador sea lento, este punto puede llegar a ser crítico para determinadas aplicaciones.

Por lo tanto, es evidente que la programación basada en *polled communication* no es la aproximación más adecuada para controlar sistemas empotrados con capacidades de cálculo limitadas. Todo ello hace imprescindible diseñar su software de control considerando la necesidad de incluir en él interrupciones.

La gestión correcta de interrupciones en sistemas en tiempo real requiere una relación estrecha entre software y hardware, hecho que determina el lenguaje de programación más adecuado para programar las ISR. Por lo tanto, la pregunta directa que se deriva es: ¿qué es mejor, lenguaje máquina o de alto nivel? En el primer caso, se hace relativamente sencillo estimar el tiempo de ejecución de la ISR, puesto que para un microprocesador determinado con una capacidad de cálculo conocida a priori, cada instrucción necesita aproximadamente *x* microsegundos para ser ejecutada (dependiendo de factores intrínsecos al hardware, como el reloj interno o los tiempos de espera entre estados). Así, podemos llegar a acotar el tiempo máximo que necesitará el sistema para gestionar la interrupción formada por *y* instrucciones.

Por el contrario, la programación de ISR en un lenguaje de nivel medio-alto resulta mucho más problemática. De hecho, y a pesar de lo que a menudo afirman los proveedores de compiladores de estos tipos de lenguajes, no hay ninguna manera óptima de escribir una ISR en un lenguaje como C, puesto que no podemos conocer ni siquiera cuánto tiempo necesitará nuestra máquina para ejecutar una única línea de código, tal como demuestra J. Ganssle en su libro *The Art of Designing Embedded Systems*. Incluso para una instrucción simple

Bibliografía recomendada

J. G. Ganssle (2000). *The Art of Designing Embedded Systems* (1.ª ed.). Woburn, Massachusetts: Newnes (Elsevier).

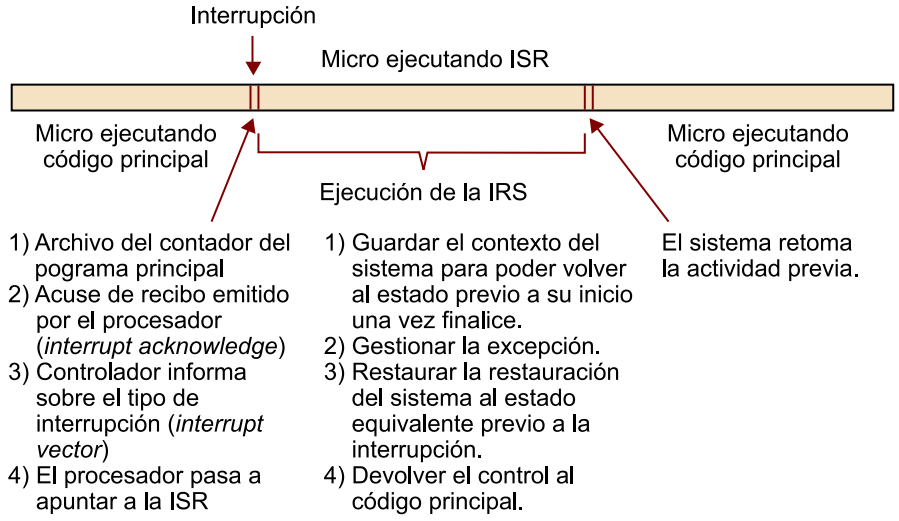
como es una iteración FOR, el tiempo de ejecución puede ir desde unos pocos milisegundos hasta consumir muchos recursos de máquina desde un punto de vista de tiempo y memoria. Esta dispersión dependerá fundamentalmente de cómo el compilador realice la conversión al lenguaje máquina del código en lenguaje de alto nivel. Así pues, en una primera aproximación podemos decir que no hay ningún impedimento para desarrollar ISR en lenguajes de alto nivel, como C, pero su tiempo de ejecución deberá ser convenientemente verificado para comprobar si satisface las especificaciones requeridas al sistema empotrado que se está programando. En el peor de los casos, cuando haya un riesgo de generación rápida de interrupciones concatenadas, se recomienda optar directamente por el lenguaje máquina como solución para evitar el colapso del sistema. Huelga indicar que aquí el término *rápido* es totalmente cualitativo e indefinido, que depende del significado fundamentalmente de la aplicación, del dispositivo final o, incluso, de las preferencias personales del programador.

Vemos, pues, que la gestión optimizada de interrupciones se convierte en un trabajo solo al alcance de desarrolladores experimentados. En este texto introductorio, evidentemente el objetivo no es conseguir este nivel de conocimiento y complejidad, sino repasar la secuencia de eventos que acaban configurando una interrupción estándar, y también familiarizarnos con los conceptos que, independientemente del lenguaje de programación empleado, se usan para describirlas.

2.1.1. Fundamentos conceptuales de las interrupciones

La generación de una interrupción provoca que el sistema empotrado siga una serie de eventos a lo largo del ciclo asociado a su gestión, tal como se puede ver en la figura siguiente:

Esquema de las acciones típicas de un proceso de interrupción



Primero, la activación de una interrupción y el aviso posterior del controlador que la recoge en el microprocesador del sistema genera los eventos y las acciones siguientes en el hardware:

- **Archivo del contador del programa principal en ejecución en la pila del procesador**. La dirección de memoria que el procesador estaba ejecutando antes de la interrupción queda guardada junto con otra información, como, por ejemplo, los contenidos de sus registros internos. Este último punto dependerá del modelo de procesador utilizado.

- **Generación de un acuse de recibo (*interrupt acknowledge*) por el procesador**. El acuse de recibo es enviado al periférico o controlador que genera la interrupción solicitando un vector de interrupción (*interrupt vector*), cuyo contenido dependerá del tipo de interrupción. Un vector de interrupción no es más que una indicación de la dirección de memoria donde el microprocesador tiene que ir a buscar la ISR o el código de programa asociado a la interrupción y que se tiene que ejecutar para poderla gestionar.

- **Ejecución de la ISR**. El microprocesador, una vez recibido el vector de interrupción, apunta a la dirección de memoria donde está la ISR y ejecuta las funciones previstas para gestionar la interrupción.

Una vez la interrupción ya sido tratada convenientemente, el procesador ejecuta las tareas siguientes:

- **Recuperación de la dirección de retorno y otra información archivada en la pila**. El procesador, para poder volver al estado previo a la generación de la interrupción, recupera la información archivada en la pila. Generalmente, la dirección de retorno (*return address*, en inglés) es casi siempre la dirección de memoria donde está la instrucción posterior que se habría ejecutado según el programa principal si la interrupción no hubiera existido.

- **Continuación de la ejecución del programa principal**. El procesador continúa ejecutando el programa principal. Si el programador ha gestionado correctamente la interrupción, es evidente que este código principal ni siquiera notará que ha habido una interrupción por el medio, hecho que por regla general habrá ocupado unos pocos milisegundos de CPU.

Analizando la secuencia de eventos anterior que constituyen una interrupción estándar, podemos afirmar que los puntos críticos del proceso son la generación del vector de interrupción y su interpretación correcta por parte del procesador y la ejecución de la ISR. En los apartados siguientes, veremos con más detalle estos dos aspectos y los requisitos generales que han de satisfacer los vectores y las ISR para evitar problemas con el funcionamiento final de dispositivos empotrados.

2.1.2. Vectores de interrupción

Todos los procesadores necesitan recibir un vector de interrupción cuando se les notifica la existencia de una interrupción para poder ejecutar la ISR correspondiente. Tal como hemos visto antes, los vectores indican al procesador dónde ir a buscar el servicio necesario para poder gestionar una interrupción de manera adecuada. Por regla general, los vectores son simplemente un número que el procesador traduce a una dirección de memoria donde está la primera instrucción de la ISR (podéis ver la tabla siguiente).

Direcciones de memoria contenidas en los vectores asociados a cada una de las cuatro interrupciones que puede procesar un microprocesador 80188

Tipo de interrupción	Dirección de memoria en el vector correspondiente
INT0	00030h
INT1	00034h
INT2	00038h
INT3	0003Ch

Sin embargo, algunos procesadores más antiguos, también podían contener código ejecutable, pero esta aproximación cada vez es menos utilizada.

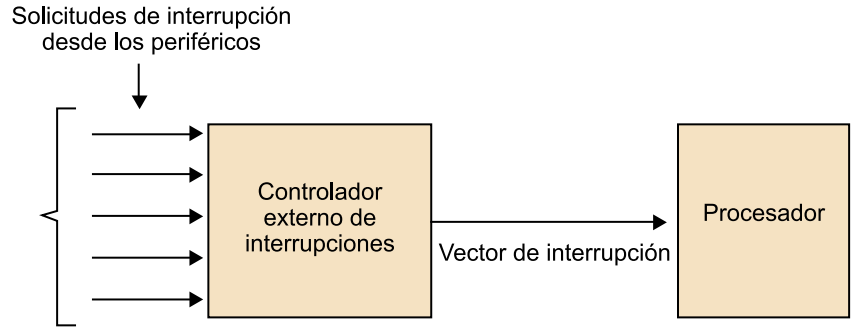
Actualmente, muchos procesadores están preparados para gestionar una interrupción especial y no ordinaria. Este tipo de interrupción, conocido como ***nonmaskable interrupt*** (NMI), se utiliza cuando se produce un error fatal en el sistema y no puede ser ignorado por el procesador, que, una vez recibe un aviso de interrupción, solicita el vector correspondiente y finaliza el funcionamiento del dispositivo. A efectos prácticos, esto puede provocar pérdidas de datos o situaciones no deseadas en determinadas aplicaciones; por lo tanto, muchos programadores prefieren diseñar sistemas empotrados que traten interrupciones críticas dejando el dispositivo inactivo a la espera de una revisión técnica.

Hay tres métodos para generar los vectores de interrupción solicitados por el procesador: a partir de un controlador externo, desde un controlador interno o desde los mismos periféricos. La figura siguiente esquematiza el funcionamiento de cada uno de los tres modelos. A efectos prácticos, los tres producen el mismo resultado, a pesar de que en el hardware existen diferencias significativas, en las que no nos detendremos.

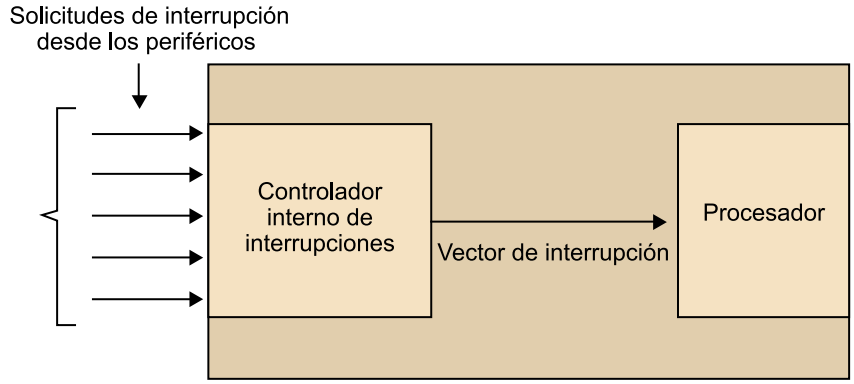
Ejemplo

En el caso de controladores internos al procesador, no se llega a generar ningún vector de interrupción real, puesto que toda la información necesaria para gestionar la interrupción se trata dentro del mismo procesador, pero el programador acaba teniendo la impresión de que se ha usado un vector de interrupción virtual que modifica el funcionamiento del sistema.

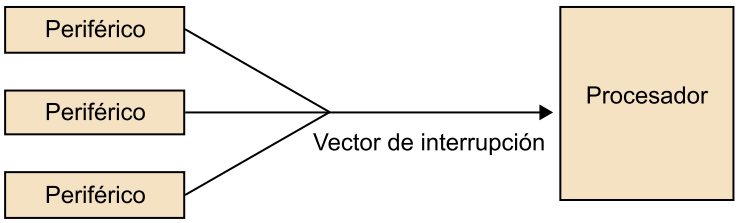
Mecanismos de generación de vector de interrupción



Procesador con controlador externo de interrupciones



Procesador con controlador externo de interrupciones



Periféricos que solicitan interrupciones directamente al procesador

2.1.3. Rutinas de servicio a la interrupción (ISR)

En el momento en el que una interrupción sucede, el procesador ejecuta una ISR. Para poder garantizar el funcionamiento correcto del sistema empotrado, todas las ISR deben satisfacer los tres puntos siguientes:

1) Ofrecer el servicio necesario para gestionar la interrupción generada.

2) Permitir al sistema aceptar nuevas interrupciones durante su ejecución.

Excepciones NMI

Las excepciones NMI son la excepción, puesto que por definición abortan el funcionamiento del sistema debido a un error fatal. Por lo tanto, solo satisfacen el primero de los tres puntos, puesto que tienen prioridad máxima.

3) Una vez finalizada la ISR, permitir y garantizar que el sistema vuelva al estado previo a la interrupción.

Ejemplo

Por ejemplo, la familia de los 68.000 permite a un periférico que emite una interrupción evaluar su prioridad, y corresponde al programador la responsabilidad de gestionar los conflictos potenciales entre dos requisitos de igual grado de prioridad que puedan llegar al procesador. Por el contrario, el procesador Intel 8259 permite asignar al programador las prioridades de todas las posibles interrupciones que se pueden generar en el sistema.

Los puntos primero y tercero son obvios y constituyen dos de las acciones básicas para poder gestionar una interrupción, tal como hemos visto anteriormente. En cambio, para justificar el segundo punto hemos de introducir el concepto de prioridad. Por regla general, todas las interrupciones tienen asignada una prioridad determinada. Este parámetro determina cuándo un procesador responde a la solicitud de interrupción de un periférico. Así, una interrupción de alta prioridad debería poder interrumpir la ejecución de una de prioridad más baja. Por el contrario, una de baja prioridad será ignorada siempre que una de alta esté siendo gestionada. Qué determina la prioridad dependerá del microprocesador que se emplee.

Así pues, para justificar la condición de que toda ISR ha de permitir aceptar nuevas interrupciones durante su ejecución, debemos pensar en el caso extremo de que una segunda interrupción de alta prioridad se genera mientras una primera de baja se está ejecutando. En caso de que el programador no lo haya previsto, lo más probable, exceptuando el caso de una NMI, es que la segunda sea ignorada hasta la finalización de la primera, puesto que la mayoría de los procesadores desconectan todos los interruptores y entradas desde los periféricos que lo pueden avisar de la existencia de nuevas solicitudes de interrupciones. El lector puede imaginar que esta situación puede originar situaciones no deseadas. Por este motivo, cada vez más procesadores permiten el tratamiento de interrupciones simultáneas o asíncronas.

En la aproximación más simple y tampoco recomendable, cualquier ISR finaliza si una nueva interrupción se genera durante su ejecución, independientemente de sus prioridades respectivas. Por el contrario, en el segundo esquema de funcionamiento, conocido como interrupciones imbricadas (*nested interrupts*), evidentemente más complicado desde un punto de vista de programación, una ISR solo puede ser interrumpida por otra de más alta prioridad para después ser retomada una vez la segunda ha finalizado. Nos podemos imaginar que para poder retomar la primera ISR en su punto de interrupción hay que guardar registros y direcciones de memoria en la pila. Por lo tanto, que este modo de trabajar sea factible dependerá fundamentalmente de la capacidad y de los recursos del procesador, puesto que debemos recordar que en la pila tenemos guardada previamente la información relativa al programa principal del sistema que ha sido interrumpido. Además, no se puede obviar el grado de complejidad que se deriva del diseño de interrupciones imbricadas, hecho que restringe su uso correcto a programadores experimentados y muy familiarizados con el tema.

2.1.4. Interrupciones: consideraciones finales

Las interrupciones son señales que reciben los microprocesadores para redirigir el flujo del programa que se está ejecutando y de este modo satisfacer los requisitos de partes del sistema, como los periféricos. El código que se ejecuta para gestionar esta demanda se conoce como *rutina de interrupción de servicio* o ISR .

La programación de interrupciones es compleja y puede inducir fácilmente a errores críticos en los sistemas. Por este motivo, los desarrolladores noveles a menudo utilizan el método de programación alternativo basado en lo que se conoce como *polled communication*. Esta solución funciona bien con procesadores potentes, pero en el caso de los sistemas empotrados resulta necesario usar interrupciones para conseguir un funcionamiento óptimo.

No hay ningún impedimento en programar las ISR con un lenguaje de alto nivel como C, pero la solución óptima es usar lenguaje máquina. Todo ello dificulta su implementación.

Por regla general, cuando un procesador proporciona un servicio a una interrupción, sigue el proceso siguiente:

1) Un componente hardware genera una solicitud de interrupción que se envía al microprocesador.

2) El microprocesador prioriza las diferentes solicitudes que recibe y selecciona una.

3) El microprocesador responde al componente hardware con un acuse de recibo.

4) El componente hardware o un controlador responsable envía un vector de interrupción al microprocesador.

5) El microprocesador lee la dirección de memoria indicada en el vector, guarda la información necesaria para recordar su estado presente y salta a la ISR.

6) Se ejecuta la ISR y se gestiona la solicitud de interrupción.

7) La ISR finaliza y el microprocesador retoma su actividad inicial.

En aplicaciones reales, los programadores deben tener presentes las diferentes prioridades de las interrupciones que se pueden generar y también las características del software que integra su dispositivo para conseguir optimizar su funcionamiento.

2.2. Funciones y punteros

A diferencia de las interrupciones, que conceptualmente son fáciles de entender pero difíciles de implementar en sistemas reales, los punteros pueden ser utilizados por programadores noveles, pero el conocimiento completo de qué hace la máquina con ellos solo se consigue después de años de experiencia.

La combinación de punteros y funciones dentro de un programa ofrece una serie de ventajas desde un punto de vista de uso de microprocesador y gestión de la memoria de la máquina, hecho que los hace atractivos para ser utilizados en numerosos lenguajes de programación de alto nivel. De hecho, los punteros se han convertido en una herramienta fundamental para desarrollar software con lenguaje C.

Desde un punto de vista de sistemas empotrados, los punteros se usan exactamente igual que en el caso de ordenadores con procesadores más potentes. En cambio, su utilización es comparativamente más beneficiosa, puesto que permiten contrarrestar la pobre capacidad de cálculo de los microprocesadores que los integran.

En esta sección repasaremos la definición de *puntero*, veremos algunos ejemplos de cómo operan en el nivel de memoria de la máquina y finalmente analizaremos en detalle cómo pueden ser combinados con funciones para optimizar el software. En todos los casos, los ejemplos ilustrativos serán en lenguaje C.

2.2.1. Fundamentos de los punteros. Definiciones

Un puntero es una variable que representa la posición (no el valor) de otro dato. Es decir: una variable cuyo valor es una dirección de la memoria. Dentro de la memoria de los computadores, cada dato está almacenado ocupando una o más celdas contiguas (palabras o bytes adyacentes). El número de celdas requeridas para almacenar un dato depende básicamente de su tipo.

Ejemplo

Un carácter necesitará un byte (8 bits) de memoria.

Punteros

Supongamos que *v* es una variable que representa un dato determinado. El compilador asignará automáticamente celdas de memoria para guardarlo. En la práctica, podremos acceder al dato siempre que conozcamos la localización o dirección de la primera celda de memoria donde está guardado *v*.

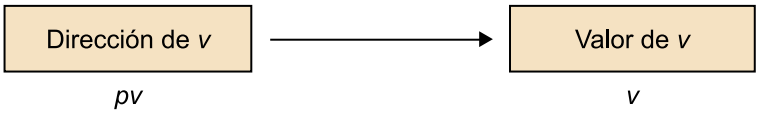
En lenguaje C, la dirección de memoria de *v* se representa mediante la expresión *&v*, donde & es el operador dirección que proporciona la dirección del operando. Si asignamos esta dirección de *v* a otra variable, *pv*. Así:

(2)Siempre que *u* y *v* hayan sido declarados como el mismo tipo de dato.

$$pv = \&v$$

Esta nueva variable es un puntero a *v* porque *apunta* a la posición de memoria donde está *v*. Hay que recordar que *pv* representa la dirección de *v* y no su valor. Por lo tanto, *pv* es definida como una variable apuntadora que está relacionada con *v* según el esquema de la figura siguiente:

Relación entre *pv* y *v* (donde *pv* = *&v* y *v* = **pv*)



Al dato representado por *v* se puede acceder fácilmente mediante la expresión **pv*, donde * es el operador indirección, que solo opera sobre una variable puntero. Así pues, **pv* y *v* representan el mismo dato (contenido de las celdas de memoria). Además, si escribimos *pv* = *&v* y *u* = **pv*, entonces *u* y *v* representan el mismo valor[2].

Esta relación entre variables, direcciones de memoria y punteros asociados se puede visualizar de manera clara a partir del programa siguiente en C:

```
#include <studio.h>

void main() {
      int u = 3;
      int v;
      int *pu;       /* puntero */
      int *pv;       /* puntero */

      pu = &u;       /* asignar dirección de u a pu */
      v = *pu;       /* asignar valor de u a v */
      pv = &v;       /* asignar dirección de v a pv */

      printf("\nu=%d &u=%X pu=%X *pu=%d", u, &u, pu, *pu);
      printf("\n\nv=%d &v=%X pv=%X *pv=%d", v, &v, pv, *pv);
}
```

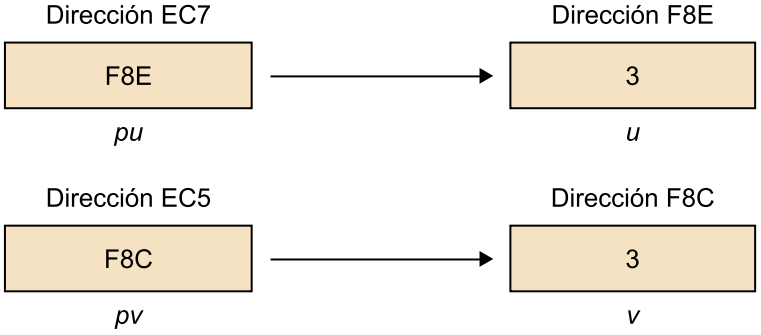
Según este sencillo código, *pu* es un puntero a *u* y *pv* un puntero a *v*. Por lo tanto, *pu* representa la dirección de *u* y *pv* la dirección de *v*.

Una posible salida producida por la ejecución del programa anterior sería:

```
uno=3       &u=F8E      pu=F8E      *pu=3
v=3         &v=F8C      pv=F8C      *pv=3
```

La figura siguiente representa la relación existente entre todas estas variables. Cabe indicar que las posiciones en memoria de los dos punteros no son escritas por el programa anterior.

Esquema gráfico con la relación entre las variables utilizadas en el ejemplo propuesto.



Los punteros, como cualquier otra variable, deben ser declarados antes de ser usados en un programa en C. Sin embargo, la interpretación correcta de la declaración de un puntero es diferente de la declaración de otras variables. Cuando se declara una variable puntero, el nombre de la variable debe ir precedido por un asterisco (*). Además, el tipo de dato que aparece se refiere al objeto del puntero, es decir, al dato que se almacena

en la dirección de memoria representada por el puntero y no el puntero en sí mismo. Veamos un ejemplo sencillo:

```
float u, v;
float *pv = &v;
```

Aquí, las variables *u* y *v* son declaradas como variables en coma flotante y *pv* es declarada como una variable puntero que apunta a otras variables también en coma flotante.

Una vez dadas las principales definiciones necesarias para recordar qué es un puntero, y visto cómo se trabaja en lenguaje C, el resto del apartado analizará cómo los punteros pueden interactuar con las funciones.

Bibliografía recomendada

Si queréis ampliar el resto de funcionalidades de los punteros, os recomendamos que consultéis las obras sobre programación en lenguaje C que aparecen en la bibliografía.

2.2.2. Paso de punteros a una función. Argumentos por referencia

A veces los punteros son pasados a las funciones como argumentos. Esto permite que datos del programa desde el cual se llama a la función sean objeto de acceso y alterados directamente por esta, para poder ser finalmente devueltos al programa ya modificados. Esta manera de trabajar según la cual los argumentos de las funciones son punteros se conoce como **argumentos por referencia**, en contraposición a pasar argumentos por valor, que suele ser la solución habitual.

Si se pasan argumentos por valor a una función, el dato es copiado a su interior. Por lo tanto, cualquier alteración hecha dentro de la función no queda reflejada en el programa principal, salvo que el programador así lo especifique, introduciendo líneas de código adicional y ralentizando de este modo el funcionamiento del programa. Por el contrario, cuando un argumento se pasa por referencia, la dirección de memoria es indicada a la función. Al contenido de esta dirección se puede acceder libremente desde dentro de la misma función y, por lo tanto, cualquier cambio que se haga será reconocido por la función, pero también desde fuera.

Vemos que el uso de punteros como argumentos de la función permite alteraciones y modificaciones globales de los datos desde dentro de la función, lo que ayuda a ahorrar líneas de código y aligera la cantidad de estas.

En sistemas empotrados en los que la velocidad de ejecución del software y su tamaño son críticos, la utilización de los punteros es una opción atractiva.

A continuación, tenemos un ejemplo sencillo en C que ilustra la diferencia entre pasar argumentos por valor o referencia:

```
#include <studio.h>


void func1(int u, int v);       /* prototipo de función*/
```

```
void func2(int *pu, int *pv);       /* prototipo de función*/

void main() {

      int u = 1;
      int v = 3;

      rintf("\ Antes de la func1; u=%d v=%d", u, v);
      func1(uno, v);
      printf("\ Después de la func1; u=%d v=%d", u, v);

      printf("\ Antes de la func2; u=%d v=%d", u, v);
      func2(&uno, &v);
      printf("\ Después de la func2; u=%d v=%d", u, v);

}

void func1(int u, int v) {
      u = 0;
      v = 0;
   printf("\ Dentro de func1; u=%d v=%d", u, v);
   return;

}

void func2(int *pu, int *pv) {
      *pu = 0;
      *pv = 0;
      printf("\ Dentro de func2; *pu=%d *pv=%d", *pu, *pv);
      return;

}
```

Este programa tiene dos funciones, denominadas `func1` y `func2`. La primera recibe dos variables enteras como argumentos que tienen originariamente asignados los valores 1 y 3, respectivamente. Los valores son alterados a 0,0 en su interior. Sin embargo, los nuevos valores no son reconocidos en main porque los argumentos llegaron por valor y cualquier cambio sobre los argumentos es local a la función en la que se han producido los cambios.

Analizamos ahora la segunda función, `func2`. Esta recibe dos punteros a variables enteras como argumentos. Los argumentos son identificados como punteros por los operadores de indirección (asterisco) que hay en la declaración de argumentos. Además, la declaración de argumentos indica que los punteros representan direcciones de memoria de cantidades enteras.

Dentro de `func2`, los contenidos de direcciones apuntadas son asignados con valores 0,0. Como las direcciones son reconocidas en la `func2` y en main, las variables enteras *u* y *v* también habrán cambiado a 0,0 en todos los lugares.

Las seis instrucciones `printf` ilustran los valores de *u* y *v* y sus valores asociados ***pu* y ***pv* dentro de main y dentro de las dos funciones. Por lo tanto, cuando se ejecute el programa se generará la salida siguiente:

```
Antes de la func1:      u=1      v=3
Dentro de func1:             u=0      v=0
Después de la func1:    u=1      v=3

Antes de la func2:      u=1      v=3
Dentro de func2:            *pu=0     *pv=3
Después de la func1:    u=1      v=3
```

El ejemplo anterior es ilustrativo del potencial de los punteros para modificar los datos de memoria dentro de un programa con pocas líneas de código y funciones simples. Evidentemente, estas variables ofrecen potencialidades más avanzadas que permiten operar de manera controlada con formaciones unidimensionales y multidimensionales, asignar dinámicamente la memoria del dispositivo o construir formaciones con ellas mismas.

Consulta recomendada

Os aconsejamos la consulta de un libro introductorio a la programación en lenguaje C para saber más sobre este tema.

2.2.3. Paso de funciones a otras funciones

Además de las operaciones enumeradas anteriormente, los punteros permiten una funcionalidad extra de interés especial en la programación y en el diseño software de los sistemas empotrados: el paso de una función como argumento a otra función como si la primera fuera una variable. Definiremos la primera función como la **función huésped** y la segunda como la **función anfitriona**. De este modo, la huésped es pasada a la anfitriona, desde donde se puede acceder libremente a ella. Llamadas sucesivas a la función anfitriona dentro del programa pueden pasar diferentes punteros (diferentes funciones huéspedes) a la anfitriona. Es evidente que este esquema de programación permite compactar bastante los códigos de software y aligerar la carga de trabajo del microprocesador, siempre que tengamos presente aquello que realmente está haciendo la máquina para evitar errores conceptuales.

Punteros (continuación)

Aquí entendemos que el lector ya tiene conocimientos previos de programación en lenguaje C para poder seguir el esbozo ilustrativo siguiente. Este programa consta de cuatro funciones: main, procesar, func1 y func2. Procesar es una función anfitriona para func1 y func2. Cada una de las tres funciones subordinadas retorna un valor entero.

```
int procesar(int(*)(int,int)); /* prototipo función anfitriona*/
int func1(int, int);           /* prototipo función huésped*/
int func2(int, int);           /* prototipo función huésped*/

void main() {
      int i, j;
      ......
```

```
      i = procesar(func1);     /* pasamos func1 a procesar; retorna
                                   un valor para i */
      ......
      j = procesar(func2);     /* pasamos func2 a procesar; retorna
                                   un valor para j */
      ......
      }

procesar(int (*pf)(int,int))   /*definición función anfitriona*/
{
      int a, b, c;
      ......
      c = (*pf) (a,b);          /* acceso a la función huésped */
                                /* retorna un valor de c */
      ......
      return(c);
}

func1(int a, int b) { /*definición función huésped*/
      int c;
      c = ....   /* usa a y b para evaluar c*/
      return(c);
}

Func2(int x, int y) { /*definición función huésped*/
      int z;
      z = ....   /* usa x e y para evaluar z*/
      return(z);
}
```

Así pues, este esbozo de programa contiene tres declaraciones de funciones. Las declaraciones para `func1` y `func2` son directas. Por otro lado, la declaración de procesar necesita una pequeña aclaración. Esta declaración la define como una función anfitriona que retorna un valor entero y al mismo tiempo tiene un argumento. El argumento es un puntero a una función huésped que retorna un valor entero y tiene dos argumentos enteros.

Algunas aplicaciones de programación se pueden formular más fácilmente en términos de paso de una función a otra, con la simplificación consiguiente en código y cálculo que esto implica. Esto resulta particularmente útil si el programa contiene diferentes ecuaciones matemáticas, de las cuales el usuario selecciona una cada vez que se ejecuta el programa.

Ejemplo

Una función puede representar una ecuación matemática y la otra puede contener una estrategia computacional de resolución. En este caso, la función que representa la ecuación puede ser pasada a la función que procesa la ecuación.

2.2.4. Funciones y punteros: consideraciones finales

Los punteros son variables que se utilizan en lenguajes de nivel medio y alto y que representan la posición en memoria de otro dato. Su utilización correcta dentro de los programas permite modificar los contenidos de la memoria de la máquina ahorrando líneas de código en comparación con otras opciones de programación menos sofisticadas. Entre las ventajas de usar punteros, es especialmente interesante su combinación con las funciones existentes dentro de un programa.

Véase también

Hay un ejemplo resuelto de este esquema de funcionamiento en la sección de actividades.

2.3. Particularidades de la programación de sistemas empotrados

La programación de los sistemas empotrados, tal como hemos visto hasta ahora, es básicamente idéntica a la de sistemas estándar. Sin embargo, algunas soluciones que pueden ser interesantes para los segundos deben ser utiliza-

das de manera adecuada en los primeros si su funcionalidad se quiere garantizar a consecuencia de sus recursos limitados. Como ejemplo, en este último apartado dedicado a los conceptos básicos del software para sistemas empotrados, veremos las consideraciones necesarias para integrar satisfactoriamente las funciones *inline* y las funciones externas en este tipo de sistemas.

2.3.1. Funciones *inline*

En lenguaje C y C++, el programador puede solicitar al compilador que introduzca el código completo de una función en cualquier parte del programa donde esta función sea llamada a actuar, en vez de ir a buscar la función a la posición de memoria donde ha sido definida inicialmente. A pesar de que el compilador no está obligado a respetar esta solicitud y de hecho muchas veces no lo hace, esta estrategia de programación conocida como ***inline expansion*** o simplemente ***inlining*** es interesante porque en primera aproximación ayuda a mejorar el tiempo de ejecución del software. Como contrapartida, aumenta el tamaño total del programa.

Inlining es una estrategia habitual para optimizar los funcionamientos de programas en los que hay muchas llamadas a funciones, especialmente si estas son pequeñas (pocas líneas de código). Utilizándola, el programador consigue las ventajas siguientes:

- Se elimina el coste de usar las instrucciones function call y return cada vez que se llama a la función de interés. Adicionalmente, también desaparece el código prólogo y epílogo que el compilador añade al programa de manera automática.

- Al mantener todo el código que hay que ejecutar junto a una posición de memoria determinada, la velocidad de ejecución del programa aumenta considerablemente, como mínimo desde un punto de vista teórico.

En cambio, trabajar con funciones *inline* puede no ser la solución más adecuada por diferentes motivos; presentamos los inconvenientes siguientes:

- Exceptuando el caso particular de programadores muy experimentados, el compilador suele conocer mejor que el ser humano cuándo generar funciones *inline*. Incluso, muchas veces las solicitudes de *inlining* hechas por el desarrollador de software son imposibles de hacer por razones técnicas. Todo ello genera problemas de compilación o rendimientos del software no previstos inicialmente.

- La proliferación no controlada de funciones *inline* dentro de un programa en C dispara el tiempo de compilación, puesto que el código es copiado en cada punto del código en el que la función es llamada a actuar.

- El aumento en el tiempo de compilación es directamente proporcional al aumento del tamaño del programa compilado.

- El aumento del tamaño del programa principal puede hacer que este no quepa correctamente en la memoria caché de la máquina, lo cual genera pérdida de datos o simplemente la hace funcionar de manera más lenta.

Por lo tanto, vemos que la conveniencia o no de usar funciones *inline* dependerá fundamentalmente del tipo de sistema empotrado que se esté programando. El desarrollador de software, antes de optar por el *inlining* como mecanismo para optimizar su sistema deberá considerar los tres puntos siguientes:

- En aplicaciones en las que el tamaño del programa que hay que ejecutar es más importante que la velocidad de ejecución, el *inlining* se suele evitar.

- El *inlining* consume registros de máquina adicionales, que se acaba traduciendo en accesos no deseados a la memoria RAM.

- Si el código aumenta mucho su tamaño, la capacidad limitada de la memoria RAM en sistemas empotrados puede ser superada. El programa simplemente no funcionará o se generará hiperpaginación o *trashing*.

A efectos prácticos, los desarrolladores de software y programadores, a pesar de que tengan presente este problema cuando trabajan, acaban incorporando heurísticas en sus compiladores, que deciden qué funciones serán *inline* y cuáles no después de evaluar el funcionamiento del programa final.

2.3.2. Funciones externas

En sistemas reales, el código de control normalmente está formado por diferentes archivos. Esto es especialmente válido en programas que usan funciones muy grandes o complejas, donde cada una de ellas puede llegar a ocupar un único archivo. Esta solución de programación también se suele usar cuando hay muchas funciones pequeñas relacionadas entre ellas, de las que unas pocas se agrupan en un único archivo.

Estos archivos se compilan de manera individual y a continuación se enlazan para generar un único programa. Todo ello hace más fácil la redacción y depuración del programa, puesto que los archivos tienen un tamaño más razonable. Además, los programas multiarchivo suelen ser más flexibles cuando son modificados o actualizados a posteriori.

Dentro de un programa multiarchivo, una función extern es reconocida a lo largo de todo el programa, mientras que una función estática solo lo es dentro del archivo en el que se haya definido. En todo caso, su tipo se establece

colocando externo o *static* a comienzos de la definición de la función. Hay que recordar que, por defecto y en caso de no indicarlo, todas las funciones serán siempre externas.

Funciones externas

A continuación, podemos ver un programa sencillo en lenguaje C que genera el mensaje "Hola" desde una función externa. El programa está formado por dos funciones: main y salida. Cada función está dentro de un archivo diferente.

Primer archivo (ARCHIVO1.C)

```
/* programa simple multiarchivo para escribir "Hola" */

#include <stdio.h>

extern void salida(void);    /* prototipo de función */

void main() {
      salida();
}
```

Segundo archivo (ARCHIVO2.C)

```
extern void salida(void);      /* definición de función externa */
{
      printf("¡Hola!");
      return;
}
```

En el momento de compilar el programa anterior, simplemente generaremos un proyecto común que incluya los dos archivos. Cómo hacerlo dependerá exclusivamente del compilador que se utilice.

Las principales ventajas de usar funciones externas y, por extensión, la programación de código con diferentes archivos son las siguientes:

- El trabajo se puede distribuir entre diferentes programadores. Cada uno de ellos se puede ocupar de uno diferente.

- Se puede usar un estilo orientado a objetos. Cada archivo define un tipo particular de objeto como un tipo de dato, y las operaciones en este objeto como funciones. La implementación del objeto se puede mantener en privado en el resto del programa.

- Se consiguen programas muy estructurados que son fáciles de mantener, actualizar y, en caso de que sea necesario, mejorar.

- Los archivos pueden contener el conjunto de las funciones de un grupo relacionado, por ejemplo todas las operaciones con matrices.

- Objetos bien implementados o definiciones de funciones pueden ser reutilizados por otros programas, lo que a la larga reduce el tiempo de desarrollo.

- Cuando se hacen cambios en un único archivo, solo hay que volver a compilar este archivo para poder reconstruir el programa, con el consiguiente ahorro de tiempo y recursos que esto implica.

Por lo tanto, podemos afirmar que el uso de funciones externas es una herramienta atractiva para desarrollar software de control modular para sistemas empotrados, fácil de modificar y mejorar en el futuro. Sin embargo, su implementación requiere una planificación y estructuración previa de su diseño que a menudo no se hace, especialmente por parte de los programadores noveles.

3. Modelos de programación

Hay muchas maneras de implementar el software que gobierna un sistema empotrado. Sin embargo, habitualmente se sigue una arquitectura o modelo de programación en particular, que está muy probado y facilita que otros programadores comprendan y reutilicen el código. En esta sección, describiremos solo algunos de estos modelos.

3.1. Estrategias básicas

Los modelos de programación o estrategias básicas son los que indicaremos a continuación.

3.1.1. Bucle de control simple

Los sistemas empotrados se caracterizan por estar orientados a la ejecución de una tarea específica que se hace continuamente. Este hecho provoca que, en muchos casos, el núcleo básico del programa consista, simplemente, en un bucle continuo que se encarga de llamar de manera sucesiva a las diferentes subrutinas que gestionan los diversos elementos de hardware o software.

En este paradigma tan básico, si se quiere introducir la capacidad de interactuar con el mundo exterior, puede ser necesario comprobar en algún momento el estado de ciertos periféricos. En estos casos, habrá que implementar alguna rutina que verifique su estado.

Por lo tanto, un sistema así de elemental solo podrá detectar cambios del entorno y responder a ellos durante la ejecución de esta rutina de control. Así, el tiempo de ejecución de cada una de las iteraciones de este bucle es el que determina el tiempo máximo de respuesta del sistema.

En conclusión, a pesar de ser un modelo de programación muy simple y eficiente, presenta limitaciones importantes en cuanto al funcionamiento en tiempo real. Veamos un ejemplo:

Queremos implementar un equipo que controle un letrero luminoso mediante un sistema empotrado. El funcionamiento debe ser el siguiente:

> "Inicialmente, hay que iluminar todo el letrero, que consiste en las letras de la palabra *OFERTA* durante 3 segundos. A continuación, hay que apagarlo durante 0,5 segundos, e ir encendiendo cada letra a intervalos de 1 segundo. Cuando se ha completado el proceso, hay que apagarlo durante 0,5 segundos y repetir la secuencia".

Se trata de un funcionamiento cíclico y, por lo tanto, resulta adecuado plantearse un funcionamiento basado en un bucle simple de control. Observamos que se trata de una situación en la que no se espera recibir ninguna señal de los periféricos que pueda alterar el funcionamiento del sistema durante todo el ciclo de funcionamiento. Además, no se trata de una aplicación crítica, de manera que, a pesar de tener que ofrecer una respuesta en tiempo real, no requiere una precisión extrema.

A continuación, se presenta una posible solución al problema:

```
#define TIEMPO_CICLO 50      //constante que recoge el tiempo de ejecución
                            //de un ciclo del bucle de control (en ms)

#define TODO_ON             3000      //intervalos secuenc. encendida (ms)
#define TODO_OFF            3500
#define O_ON                4500
#define OF_ON               5500
#define OFE_ON              6500
#define OFER_ON             7500
#define OFERT_ON            8500
#define OFERTA_ON           9500
#define TODO_FINAL_OFF      10000

void main() {

      int tiempo = 0;
      int j = 0;

      while (1) {            // el bucle de control se ejecuta SIEMPRE

            if (tiempo > 0 && tiempo < TODO_ON) {
                  display.ofertaON();
            } else if (tiempo > TODO_ON && tiempo < TODO_OFF) {
                  display.OFF();
            } else if (tiempo > TODO_OFF && tiempo < O_ON) {
                  display.oON();
            } else if (tiempo > O_ON && tiempo < OF_ON) {
                  display.ofON();
            } else if (tiempo > OF_ON && tiempo < OFE_ON) {
                  display.ofeON();
            } else if (tiempo > OFE_ON && tiempo < OFER_ON) {
                  display.oferON();
            } else if (tiempo > OFER_ON && tiempo < OFERT_ON) {
                  display.ofertON();
            } else if (tiempo > OFERT_ON && tiempo < OFERTA_ON) {
                  display.ofertaON();
            } else if (tiempo>OFERTA_ON && tiempo<TODO_FINAL_OFF) {
```

```
                    display.OFF();
                } else {
                    j = 0;
                }
                j++;
                tiempo = j * TIEMPO_CICLO;
            }
        }
```

En primer lugar, hay que prestar atención al hecho de que el ciclo de control se ejecuta siempre. Por lo tanto, solo se para cuando se apaga el equipo.

En segundo lugar, observamos que se ha establecido un sistema de control de tiempo basado en el tiempo de ejecución medio del bucle de control simple. Según prueba y error, se ha llegado a la conclusión de que el tiempo necesario para ejecutar el bucle es de cerca de 50 milisegundos en el hardware disponible. De acuerdo con este resultado, un simple contador de iteraciones (*j*) se encarga de mantener actualizada una variable (tiempo) que recoge el tiempo estimado en la secuencia de iluminación actual. Si bien esta solución puede ser adecuada para esta aplicación, es evidente que no sería conveniente en sistemas en los que el tiempo de ejecución de este bucle fuera incierto.

En tercer lugar, en cada uno de los intervalos de la secuencia de encendido, se llama a la subrutina correspondiente (por ejemplo, display.OFF[]) que se encarga de ejecutar las acciones necesarias. Esta es una manera de proceder habitual en los sistemas basados en bucle simple de control: la estructura del método principal se mantiene simple (el bucle y poca cosa más) y los detalles de las acciones que hay que llevar a cabo en cada iteración se programan, de manera modular, en subrutinas independientes.

3.1.2. Control por eventos

El control por eventos surge de la necesidad de diseñar sistemas empotrados capaces de interactuar con el mundo exterior en tiempo real. En este sentido, permite superar las limitaciones del bucle de control simple.

En este caso, se asume que todo el funcionamiento del sistema empotrado está determinado por la sucesión de eventos que esté previsto detectar y procesar. O, de manera equivalente, por la activación sucesiva de diferentes interrupciones de acuerdo con los mecanismos descritos en el subapartado 2.1, "Interrupciones".

¿Qué provoca las interrupciones?

El origen de estas interrupciones puede ser muy variado. Por ejemplo, pueden ser generadas por un temporizador interno a una frecuencia determinada o por un byte recibido en el puerto serie. Esto quiere decir que idealmente las tareas del sistema se ejecutan únicamente a instancia de diferentes tipos de eventos internos o externos al sistema.

Hay que tener en cuenta la naturaleza impredecible de la secuencia de interrupciones (tanto en cuanto a su orden y número, como al retardo entre ellas): si no se hace un tarea exhaustiva de planificación de la respuesta del sistema en las diferentes situaciones, el sistema puede no ser resistente, ni estable, ni predecible.

La experiencia demuestra que los sistemas controlados por interrupciones son eficaces si: **a)** las subrutinas de gestión de interrupción son breves y simples (como en el caso de los tiempos de ejecución breve), o **b)** si se puede acceder a ellas rápidamente (como en el caso de un bajo tiempo de latencia). Estas dos características ayudan a prevenir retardos y colapsos en el sistema, por ejemplo, ante un alud de interrupciones.

Desde un punto de vista práctico, se suelen implementar sistemas híbridos entre los paradigmas de bucle de control simple y de control por interrupciones. Habitualmente, los sistemas ejecutan un tarea simple y no muy sensible a retardos imprevistos en el bucle principal. Paralelamente, las subrutinas de gestión de interrupciones solo se encargan de añadir tareas a la secuencia del bucle principal que se ejecutarán en la próxima iteración. Así, la atención a interrupciones resulta rápida, previniendo colapsos, a la vez que mantiene un flujo de ejecución simple, periódico y predecible. En el subapartado 3.3.1, "Diseño y captura de funcionalidad", mostramos un ejemplo de esto.

3.2. Máquinas de estado

En sistemas empotrados orientados a aplicaciones de control, a menudo resulta útil organizar la programación en función del estado del sistema que se ha de controlar. Sería equivalente a disponer de un único sistema de control con varios modos de funcionamiento según las circunstancias.

Las máquinas de estados se basan en la asunción de que el sistema solo se puede encontrar en un único estado en cada instante. Cada uno de los estados se asocia a una situación de funcionamiento determinada, de modo que resulta fácil restringir la ejecución de tareas y la atención de interrupciones solo a aquellas que sean propias del estado.

En un sistema controlado por bucle simple, con interrupciones o sin ellas, los diferentes estados se podrían visualizar como diferentes secuencias de subrutinas que hay que ejecutar de manera alternativa dentro del bucle principal.

Las máquinas de estados resultan, por lo tanto, una buena estrategia para la estructuración y la compartimentación del código y sus funcionalidades: se puede modificar el código de un estado determinado sin afectar al resto.

3.2.1. Diseño y captura de funcionalidad

Veamos un ejemplo de cómo se debe capturar la funcionalidad de un sistema mediante una máquina de estados, y también sus ventajas respecto a la programación secuencial tradicional.

Controlador de ascensor implementado con máquina de estados

Las especificaciones requieren que el sistema de control de un ascensor se comporte de la manera siguiente:

"Moved el ascensor ya sea arriba o abajo hasta llegar a la planta solicitada. Una vez hayáis llegado a ella, abrid la puerta, como mínimo durante 10 segundos, y dejadla abierta hasta que la planta solicitada cambie. Aseguraos de que la puerta no se abre nunca mientras se está en movimiento. No cambiéis de dirección, salvo que haya más plantas solicitadas en la dirección del movimiento actual (por ejemplo, plantas superiores si estáis subiendo, plantas inferiores si estáis bajando)".

Esta sería una posible implementación de este ejemplo empleando programación secuencial tradicional[3].

```
void UnitControl()
{
      up = down = 0; open = 1;
      while (1) {
      while (req == floor);
      open = 0;
      if (req > floor) { up = 1;}
      else {down = 1;}
      while (req != floor);
      up = down = 0;
      open = 1;
      delay(10);
      }
}
```

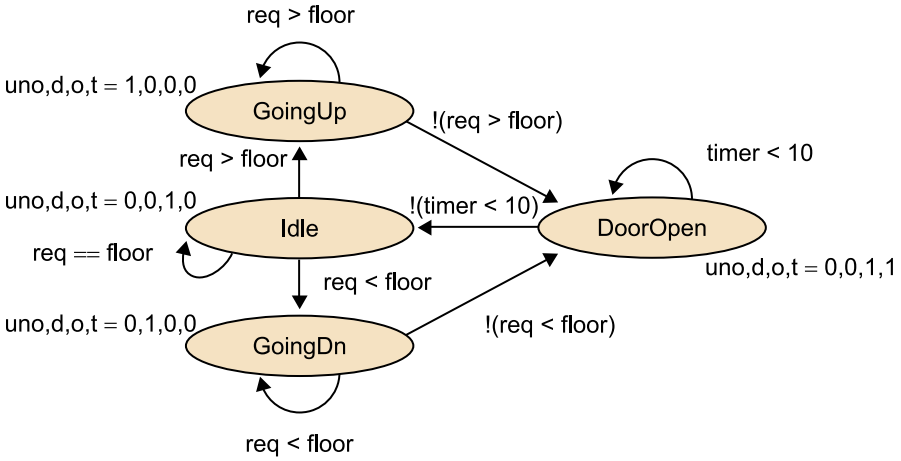
(3) Nota: `up`, `down` y `open` son variables booleanas que determinan la ejecución de las tareas de ascender, descender y abrir puertas, respectivamente. `floor` y `req` son enteros que almacenan la información de la planta actual y la planta solicitada, respectivamente.

Resulta bastante evidente que si utilizamos solo la programación secuencial convencional resultará fácil cometer errores y perderemos mucho tiempo para conseguir implementar la funcionalidad deseada. Una manera más simple, directa y sistemática de implementar este tipo de sistemas son las **máquinas de estados finitos**[4].

(4) En inglés, *finite state machine* (FSM).

La manera habitual de representar el funcionamiento de una FSM es el **diagrama de estados**. La figura siguiente muestra el diagrama de estados de la máquina que implementa el control del ascensor según la descripción anterior.

Diagrama de estados de una posible implementación del ascensor del ejemplo que acabamos de analizar



Los diagramas de estado se suelen representar con la convención de signos siguiente.

- Círculos: representan los estados. A cada uno de los estados se le asocia una secuencia binaria diferente del resto conocida como **código del estado**.

- Flechas: representan las transiciones permitidas entre estados. Suelen ir acompañadas de la **condición** que provoca este cambio.

Este tipo de diagrama se puede elaborar fácilmente siguiendo los pasos siguientes:

1) Elaborar una lista de todos los estados posibles.

2) Declarar todas las variables; suelen ser modificadas por las rutinas de atención a una interrupción al recibir una señal exterior indicativa de un cambio de estado. También pueden ser modificadas en ciertos estados de la máquina de estados.

3) Por cada estado, enumerar las posibles transiciones a otros estados, e identificar las condiciones.

4) Por cada estado y/o transición, hacer una lista de todas las acciones necesarias (tareas, cambios de variables, etc.).

5) Asegurarse de que, por cada estado, las condiciones de transición son mutuamente excluyentes y completas (es decir, que solo una condición puede ser cierta al mismo tiempo y que en todo momento hay, como mínimo, una condición cierta).

Ejemplo (continuación): controlador de ascensor implementado con máquina de estados

En el ejemplo que nos ocupa, estos cinco puntos se traducirían en:

1) Necesitamos cuatro estados:

- *Idle* = el ascensor espera con la puerta abierta.
- *GoingUp* = el ascensor cierra la puerta y sube.
- *GoingDown* = el ascensor cierra la puerta y baja.
- *DoorOpen* = el ascensor abre la puerta.

2) Necesitaremos tres variables:

- *req* = planta solicitada
- *floor* = planta actual.
- *timer* = hora del temporizador.

3) En la tabla siguiente identificamos las transiciones entre estados siguientes, asociadas a una serie de condiciones entre las variables:

Transiciones entre estados y condiciones

Transición	Condición
Idle → *Idle*	req == floor
Idle → *GoingUp*	req > floor
Idle → *GoingDown*	req < floor
GoingUp → *GoingUp*	req > floor
GoingUp → *DoorOpen*	!(req > floor)
GoingDown → *GoingDown*	req < floor
GoingDown → *DoorOpen*	!(req < floor)
DoorOpen → *DoorOpen*	timer < 10
DoorOpen → *Idle*	!(timer < 10)

4) Las tareas que debe llevar a cabo el sistema en los diferentes estados son las siguientes:

- *u* = desplazar el ascensor arriba.
- *d* = desplazar el ascensor abajo.
- *o* = abrir las puertas.
- *t* = iniciar el temporizador.

En la tabla siguiente se enumeran las tareas que hay que llevar a cabo en cada uno de los estados. Fijaos en que si se asocia una variable binaria a cada una de ellas, se genera automáticamente un código binario identificativo diferente para cada uno de los estados. Los códigos no pueden ser iguales puesto que dos estados que hagan las mismas tareas deberían ser, a la fuerza, los mismos.

Lista de los estados, acciones y códigos

Estado	u	d	o	t
Idle	0	0	1	0
GoUp	1	0	0	0
GoDown	0	1	0	0
DoorOpen	0	0	1	1

5) Finalmente, se deja como ejercicio para el lector verificar que las condiciones de transición impuestas son mutuamente excluyentes y completas teniendo en cuenta los cambios de variables asociados a cada una de las tareas.

3.2.2. Implementación con código

Una vez completado el diseño de la máquina de estados siguiendo los pasos anteriores, ya se puede pasar a su implementación en código fuente.

Ejemplo (continuación): controlador de ascensor implementado con máquina de estados

Veamos la implementación del ejemplo anterior:

```
#define IDLE 0
#define GOINGUP 1
#define GOINGDN 2
#define DOOROPEN 3

void UnitControl() {
      int state = IDLE;
      while (1) {
      switch (state) {
      IDLE: up=0; down=0; open=1; timer_start=0;
            if  (req==floor) {state = IDLE;}
            if  (req > floor) {state = GOINGUP;}
            if  (req < floor) {state = GOINGDN;}
            break;
      GOINGUP: up=1; down=0; open=0; timer_start=0;
            if  (req > floor) {state = GOINGUP;}
            if  (!(req>floor)) {state = DOOROPEN;}
            break;
      GOINGDN: up=1; down=0; open=0; timer_start=0;
            if  (req < floor) {state = GOINGDN;}
            if  (!(req<floor)) {state = DOOROPEN;}
            break;
      DOOROPEN: up=0; down=0; open=1; timer_start=1;
            if (timer < 10) {state = DOOROPEN;}
            if (!(timer<10)){state = IDLE;}
            break;
      }
      }
}
```

Fijémonos, en primer lugar, en que gracias al trabajo de diseño de una máquina de estado, esta implementación es mucho más clara que la basada en código secuencial.

En segundo lugar, observamos que el programa consiste en un bucle principal que se repite de manera indefinida, pero que hace diferentes acciones en cada iteración en función del estado en el que se encuentra el sistema.

En tercer lugar, hay que prestar atención al hecho de que las estructuras *switch-case* son particularmente adecuadas para implementar este tipo de software, puesto que favorecen la estructuración y la compartimentación del código mencionadas anteriormente. A continuación, presentamos una plantilla de código que puede ser empleada para implementar cualquier máquina de estado:

```
#define S0      0
```

```
#define S1      1
...
#define SN      N

void StateMachine() {
      int state = S0;       //indicar aquí el estado inicial.
      while (1) {
      switch (state) {
      S0:
                  //indicar las acciones del estado S0:
                  /*actions*/

                  //indicar transiciones Ti de salida de S0:
                  if(condition T0 == true ) {
                         state = estado siguiente según T0;
                         /*actions*/
                  }
                  if(condition T1 == true ) {
                         state = estado siguiente según T1;
                         /*actions*/
                  }
            ...
                  if(condition Tm == true ) {
                        state = estado siguiente según Tm;
                        /*actions*/
                  }
            break;
      S1:
                  //indicar las acciones del estado S1:
                  ...
                  //indicar transiciones Ti de salida de S1:
                  ...
            break;
      ...
      SN:
                  //indicar las acciones del estado SN:
                  ...
                  //indicar transiciones Ti de salida de SN:
                  ...
            break;
      }
      }
}
```

Finalmente, vale la pena reflexionar sobre el hecho de que el modelo de programación basado en máquinas de estado facilita al diseñador la visualización de todos los posibles estados y transiciones entre estados en función de las

diferentes entradas o variables del sistema. De este modo, resulta una manera bastante natural de pensar y resolver los problemas de secuenciación típicos de sistemas empotrados.

Como hemos visto, la arquitectura de máquina de estados es particularmente adecuada para diseños que llevan a cabo una única función. Sin embargo, a menudo hay que implementar sistemas complejos con múltiples funciones más o menos independientes gobernadas por diferentes máquinas de estado. Las máquinas de estado jerárquicas solucionan, de manera simple, esta necesidad.

3.2.3. Máquinas de estado jerárquicas

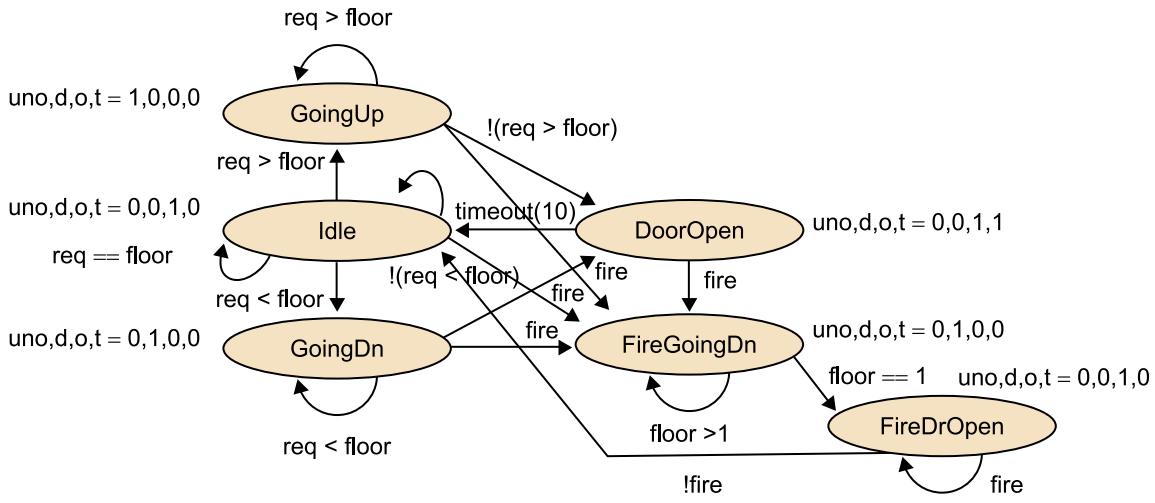
Ejemplo: controlador de ascensor con modo de incendios implementado con máquina de estados convencional

Continuando con el ejemplo anterior, supongamos que queremos añadir un modo de funcionamiento en caso de incendio al sistema de control del ascensor anterior. Una posible descripción funcional de este sistema sería:

"En caso de incendio, desplazad el ascensor a la primera planta y dejad las puertas abiertas".

Si nos basamos en una máquina de estados convencional, esta pequeña modificación complica notablemente el diagrama de estados, puesto que hay que añadir múltiples estados, variables, transiciones; podéis ver la figura siguiente:

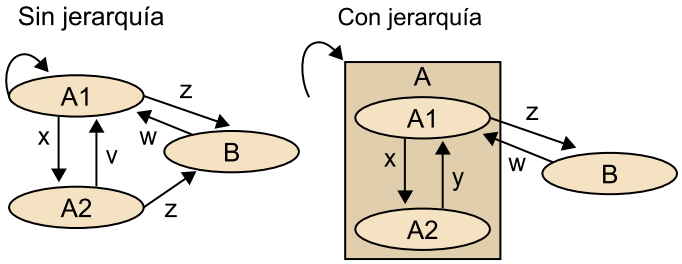
Diagrama de estados de un controlador de ascensor con modo de incendios



En las **máquinas de estados jerárquicas**, algunos estados pueden consistir en máquinas de estado completas e independientes que se ejecutan solo cuando se accede a estos estados.

La figura siguiente muestra un ejemplo de una máquina de estados con jerarquía y sin ella. En ambos casos, la funcionalidad es la misma, pero el uso de jerarquía permite visualizar fácilmente que la ejecución de B es independiente del funcionamiento normal de A y que solo se necesita una única transición *z* para acceder a ella.

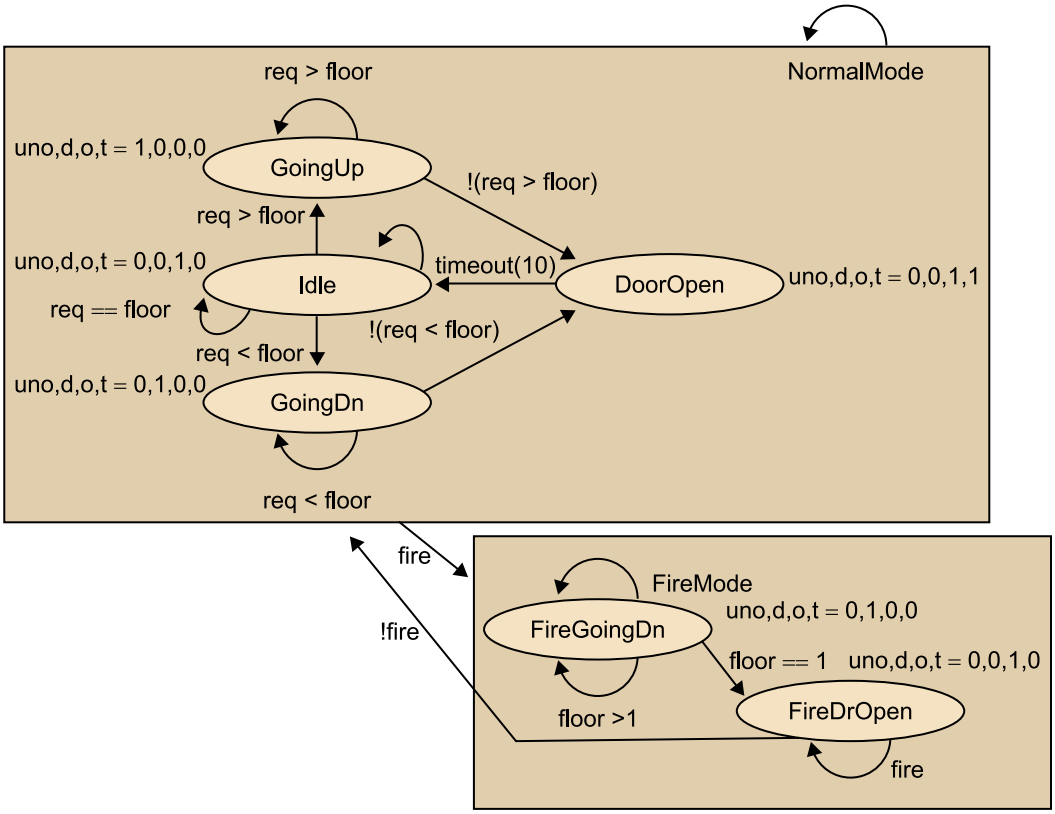
Diagrama de estado de una máquina de estados simple implementada con jerarquía y sin ella



Ejemplo: controlador de un ascensor implementado con una máquina de estados con jerarquía

Tal como se muestra en la figura, el ejemplo del control del ascensor con modo en caso de incendio se simplifica notablemente si usamos una máquina de estado jerárquica que atienda de manera específica la situación de incendio `(fire == 1)`.

Diagrama de estados de una posible implementación del ascensor del subapartado 3.2.1 con máquinas de estado jerárquicas



En este caso, el código fuente se vería modificado de la manera siguiente:

```
#define NORMAL 0            // definimos modos de trabajo
#define FIRE 1

#define N_IDLE 0            // definimos estados del primer modo
#define N_GOINGUP 1
#define N_GOINGDN 2
#define N_DOOROPEN 3
```

```
#define F_GOINGDN 0      // definimos estados del segundo modo
#define F_DOOROPEN 1

void UnitControl() {
        int modo = NORMAL;
        int state = N_IDLE;

        while (1) {
        switch (MODO) {
              NORMAL:
                    switch (state) {
              N_IDLE: up=0; down=0; open=1; timer_start=0;
                    ...
              N_GOINGUP: up=1; down=0; open=0; timer_start=0;
                    ...
              N_GOINGDN: up=1; down=0; open=0; timer_start=0;
                    ...
              N_DOOROPEN:up=0; down=0; open=1; timer_start=1;
                    ...
              }

              FIRE:
                   switch (state) {
              F_GOINGDN: up=0; down=1; open=0; timer_start=0;
                    ...
              F_DOOROPEN:up=0; down=0; open=1; timer_start=0;
                    ...
        }
        }
}
```

3.3. Sistemas multitarea

Las estrategias de programación vistas hasta el momento no resultan adecuadas para atender a un gran número de tareas que hay que realizar simultáneamente.

Veamos un ejemplo de lo que entendemos por *un sistema con múltiples tareas concurrentes*.

Sistemas multitarea elementales

Se pide programar un sistema capaz de llevar a cabo las acciones siguientes:

1) Leer dos números diferentes X e Y.
2) Imprimir el mensaje "Hola, Mundo" cada X segundos
3) Imprimir el mensaje "¿Cómo estás?" cada Y segundos.

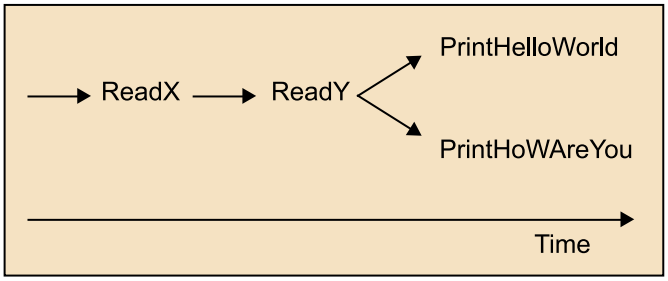
Se trata, pues, de una aplicación que requiere la ejecución de dos tareas de manera concurrente (por ejemplo, imprimir dos mensajes, cada uno con su ritmo de repetición independiente) que no sería obvio de capturar en función de máquinas de estado o cualquiera de las estrategias vistas anteriormente. Una posible implementación de esta funcionalidad podría ser la siguiente:

```
void main_ConcurrentTaskExample() {
       x = ReadX();
       y = ReadY();


       Call concurrently {
```

```
                PrintHelloWorld(x) and PrintHowAreYou(y)
        }
}

void PrintHelloWorld(x) {
        while( 1 ) {
                print("Hello world.");
                delay(x);
        }
}

void PrintHowAreYou(x) {
        while( 1 ) {
                print("How are you?")
                delay(y);
        }

}
```

Según este código, la secuencia de ejecución de las diferentes rutinas a lo largo del tiempo sería:

Secuencia de funcionamiento del código del ejemplo del apartado 3.3.3



Prestemos atención al hecho de que las distintas tareas de un sistema empotrado multitarea pueden y suelen estar basadas, cada una de ellas, en un bucle simple de control.

El aspecto de la salida del programa podría ser:

```
Enter X: 1
Enter Y: 2
Hello world. (Time = 1 s)
Hello world. (Time = 2 s)
How are you? (Time = 2 s)
Hello world. (Time = 3 s)
How are you? (Time = 4 s)
Hello world. (Time = 4 s)
...
```

Si bien la mayoría de los lenguajes de programación permiten elaborar programas con múltiples tareas, su correcta ejecución requiere tener en cuenta una serie de conceptos teóricos relacionados con la gestión de múltiples tareas.

Se denomina *gestión multitarea* el proceso de planificar tareas de manera que **parezca que se ejecutan simultáneamente**. Esta función incluye la activación/desactivación de tareas, el control de las prioridades y la planificación de su ejecución.

Dado que buena parte de las implementaciones reales de sistemas multitarea se basa en la utilización de sistemas operativos, en primer lugar se dedica una pequeña sección a su introducción. A continuación, ofrecemos varios apartados en los que se describen las características fundamentales de la gestión multitarea.

Ved también

El módulo "Controladores y sistemas operativos" se dedica íntegramente al estudio del uso de los sistemas operativos.

3.3.1. Introducción a los sistemas operativos en tiempo real

Una de las maneras más eficaces de implementar características de funcionamiento en tiempo real en sistemas empotrados con funcionalidad multitarea compleja es mediante el uso de sistemas operativos en tiempo real[5].

(5)En inglés, *real-time operating systems* (RTOS), también conocidos como *real-time executives* o *real-time kernels*.

Esta metodología de trabajo se basa en el uso de un software básico predesarrollado: el sistema operativo, que, entre otras cosas, gestiona el acceso al hardware para la ejecución de las diferentes tareas previstas. Tradicionalmente, esta necesidad se satisfacía implementando controladores específicos para cada sistema empotrado y cada periférico. A medida que estos periféricos han crecido en complejidad y que las interfaces estándar se han hecho más habituales, se ha demostrado que reescribir estos elementos de software desde cero para cada aplicación es una estrategia de programación muy poco eficiente. Los sistemas operativos incorporan estos controladores y ahorran mucho trabajo al programador, lo cual facilita notablemente el desarrollo de sistemas empotrados que interactúen con gran variedad de periféricos.

Al mismo tiempo, y a medida que la funcionalidad de los sistemas empotrados crece en complejidad, se hace necesario gestionar de manera eficiente la ejecución simultánea de diferentes tareas con los recursos disponibles.

Por lo tanto, del mismo modo que los sistemas operativos convencionales, los RTOS controlan los diferentes recursos del sistema empotrado y gestionan el acceso a estos de las diferentes tareas. Los RTOS tienen una característica adicional que los diferencia de los sistemas operativos convencionales: son **deterministas**, en el sentido de que el tiempo necesario para ejecutar una tarea determinada es conocido y está completamente especificado. Esta característica permite evaluar el tiempo que se necesita para ejecutar las diferentes tareas y, por lo tanto, permite cumplir requisitos de funcionamiento en tiempo real.

Habitualmente, los RTOS se distribuyen en forma o bien de núcleos (*kernels*), o bien de sistemas operativos completos. Las versiones núcleo normalmente implementan un sistema básico de gestión de tareas y de memoria. Las ver-

siones de sistema operativo completo pueden incluir controladores para una gran variedad de dispositivos, como unidades de disco, interfaces y puertos. Otra característica habitual de los RTOS es que suelen requerir que el hardware del sistema genere un interrupción periódica (denominada *contador de tiempo* o *temporizador*) que es utilizada como base de tiempo de todo el sistema de tiempo real y que permite planificar las tareas.

En resumen, las funciones principales de un RTOS son:

- Gestión multitarea.
- Comunicación y sincronización de tareas.
- Gestión de recursos y memoria.

En las secciones siguientes, se continuarán describiendo las metodologías y herramientas que permiten el funcionamiento multitarea desde una perspectiva cercana a las implementaciones reales, es decir, asumiendo en la mayoría de los casos que se trabaja con un RTOS.

3.3.2. Activación/desactivación de tareas

En esta sección se describe cómo se mantiene el control de las diferentes tareas que hay que llevar a cabo en entornos multitarea.

La mayoría de los sistemas multitarea, especialmente los basados en RTOS, mantienen una lista que recoge las diferentes tareas que hay que ejecutar, junto con su prioridad, llamada **lista de ejecución**.

Cuando una tarea pasa a estar lista (*ready*) para su ejecución, se añade a la lista y se ejecuta cuando corresponda, en función de los diferentes esquemas de priorización y planificación que se describen en los apartados siguientes.

Tareas activas

Se considera que una tarea está activa cuando se está ejecutando de manera efectiva en el procesador. Una tarea está lista cuando simplemente se encuentra en lista pendiente de su ejecución.

Si una tarea deja de estar lista (*not ready*), es eliminada inmediatamente de la lista de tareas.

Cuando una tarea está activa (por ejemplo, si se está ejecutando en el procesador), se puede dar alguna circunstancia que impida temporalmente su ejecución, como, por ejemplo, que dependa de la finalización de otras tareas no completadas. En este caso, la tarea dejaría de estar lista, se eliminaría de la lista de ejecución y se interrumpiría la ejecución hasta la finalización de las tareas de las que depende.

Ejemplo

Una tarea de gestión de datos introducidos por teclado solo estaría lista cuando se hubiera pulsado alguna tecla.

Los mecanismos lógicos que dirigen la inclusión o eliminación de una tarea determinada de la lista de ejecución se denominan *planificación de tareas*.

Esta capacidad de activar o desactivar tareas permite a los sistemas multitarea responder de manera muy compleja a los cambios en su entorno. En cierto modo, emulan el funcionamiento habitual del mundo real. Por ejemplo, po-

demos tener previsto trabajar todo el día en la oficina, pero un accidente con el coche de camino al trabajo puede hacer que toda la jornada de trabajo se retrase. Más tarde, la reunión prevista se cancela y hay que ocupar el tiempo en alguna otra actividad.

3.3.3. Prioridad

Resulta obvio que no todas las tareas que tiene que llevar a cabo un sistema empotrado requieren el mismo grado de diligencia.

Ejemplo

En caso de accidente, es más importante que el sistema de control del airbag garantice su despliegue a tiempo, que atienda una rutina de autoverificación que, casualmente, esté programada para ejecutarse al mismo tiempo.

Se denomina *prioridad* la urgencia relativa de una tarea o interrupción en comparación con otra. Se suele indicar con un número entero asociado a cada tarea o a cada tipo de interrupción.

En el caso de tareas, determina el tiempo y el turno de proceso que le asignará el planificador para su ejecución.

Nota

Cabe recordar que hay interrupciones con prioridad máxima siempre, como la NMI destinada a avisar de un error fatal del sistema.

En el caso de interrupciones, determina cómo se debe proceder en caso de que se produzcan otras interrupciones mientras es atendida por la ISR.

3.3.4. *Time-slicing*

Si se dispone de una única unidad de proceso, el hardware solo podrá atender la ejecución de una única tarea en cada instante de tiempo. Este es el caso de la mayoría de los sistemas empotrados. No obstante, esto no impide que muchos de estos sistemas puedan ofrecer un comportamiento aparentemente de multitarea.

El *time-slicing* es una de las posibles maneras de simular el funcionamiento multitarea en una única unidad de proceso. Se basa en dividir el tiempo de ejecución en intervalos regulares (o franjas de tiempos) y, durante este tiempo, dedicar el procesador y el resto de los recursos del sistema a la ejecución de una única tarea.

Para determinar estos intervalos, se utilizan las interrupciones generadas por el contador de tiempo del sistema.

Al recibir la interrupción de final de intervalo, el software encargado de la gestión multitarea, habitualmente un RTOS, detiene la ejecución de la tarea actual y pasa el control a la siguiente, que se ejecuta durante el intervalo siguiente.

Este proceso se repite cíclicamente recorriendo todas las tareas incluidas en la lista de ejecución (por ejemplo, tareas listas). Así pues, el tiempo necesario para completar una tarea determinada depende en gran medida del número de tareas que se ejecutan simultáneamente en el sistema.

Con esta metodología, resulta muy simple tener en cuenta también la prioridad de las diferentes tareas: el planificador de tareas puede prever que las que disfruten de una alta prioridad sean ejecutadas durante más de un intervalo de tiempo consecutivo.

Aun así, si una tarea finaliza su actividad antes de acabar el intervalo asignado, puede dar el tiempo sobrante a la tarea siguiente, lo que posibilita un mayor aprovechamiento del tiempo.

Para garantizar, en el momento de retomar una tarea, que esta continúe exactamente en el punto donde había finalizado, es necesario guardar una cantidad de información.

Se denomina **contexto** los datos contenidos en los diferentes registros y pilas del sistema en el momento de la interrupción.

Por lo tanto, hay que restituir este contexto para poder continuar trabajando en iguales condiciones. El contexto incluye, entre otros datos, la información relativa a la siguiente instrucción pendiente de ejecución (puntero de instrucciones) y todos los datos asociados a la tarea (puntero de pila, espacio de memoria asignado, registros de datos, etc.).

El gestor de multitarea, a menudo una parte de la RTOS, es el responsable de guardar y restituir el contexto correspondiente en el momento de activar y desactivar las diferentes tareas. Para ello, utiliza las herramientas de control de tareas.

Véase también

Podéis ver que se describen en el subapartado 3.4.6, "Herramientas para la comunicación y sincronización de tareas".

Si se dispone de una única unidad de procesamiento, el hardware solo podrá atender la ejecución de una única tarea en cada instante de tiempo. Este es el caso de la mayoría de los sistemas empotrados. Esto no impide que muchos de estos sistemas puedan ofrecer un comportamiento aparentemente de multitarea.

3.3.5. Planificación de tareas

Veamos ahora diferentes técnicas de planificación de tareas.

Planificación secuencial

La planificación secuencial implica que las diferentes tareas que hay que llevar a cabo se ejecutan de manera sucesiva, una detrás de la otra, pasando el control a la siguiente solo cuando la primera ha acabado.

Esta estrategia es muy simple de implementar, pero resulta particularmente ineficiente cuando hay que hacer tareas que dejan el sistema en espera, como, por ejemplo, esperar a que el usuario pulse una tecla o esperar a la estabilización de la temperatura dentro de una habitación. Resulta obvio que, en estos casos, la planificación secuencial lleva al sistema a largos períodos de inactividad que podrían ser utilizados en tareas más provechosas.

Otro inconveniente de los sistemas de planificación secuenciales es que atienden a todas las tareas con la misma prioridad.

Ejemplo

Este hecho puede llegar a ser problemático; por ejemplo, si cuando se recibe una interrupción urgente, el sistema debe esperar a finalizar la tarea en curso para poder atenderla.

Un tercer inconveniente de la planificación secuencial es su ineficiencia para gestionar un número elevado de tareas. Pueden darse situaciones en las que los recursos del sistema no den abasto para atender un alud de tareas listas para su ejecución en un intervalo de tiempo corto. Los sistemas secuenciales no suelen implementar mecanismos que eviten el desbordamiento de las capacidades computacionales del sistema por un exceso de tareas pendientes.

Por todos estos motivos:

> Los sistemas que únicamente implementan planificadores secuenciales se considera que no son capaces de trabajar en multitarea.

En cualquier caso, hay que reconocer la principal ventaja que tiene. En un sistema con planificación secuencial, es posible predecir qué instrucciones serán ejecutadas por el procesador, con qué orden y en qué momento lo hará. De hecho, si se conocen todas las entradas en el programa, es posible predecir sus resultados y su comportamiento, independientemente de que el sistema ejecute las instrucciones con más o menos celeridad. Por esta misma razón, en este tipo de sistema resulta simple analizar el tiempo necesario para la finalización de las tareas; y, por lo tanto, facilitan la implementación de sistemas en tiempo real.

Planificación prioritaria

En la práctica, el funcionamiento secuencial es raro. Incluso en sistemas basados en bucles de control simple, es muy frecuente tener que atender interrupciones. Cuando se recibe una interrupción, la ISR se pone en marcha para atenderla y altera el flujo normal de la ejecución.

Por lo tanto, a pesar de que siempre hay que garantizar el funcionamiento correcto del sistema, el paradigma de secuencialidad y predictibilidad de la serie de instrucciones ejecutadas en el procesador raramente se cumple.

Una de las estrategias más comunes para conseguir que la rotura de la secuencialidad prevista no vaya en contra de la funcionalidad que se quiere es la utilización de la prioridad de las diferentes tareas cuando se planifica la ejecución.

La planificación **prioritaria** (*preemptive schedulling*) es el método más común de planificar tareas en RTOS y se basa en el principio siguiente:

> "Una tarea se ejecuta hasta que se acaba y hasta que una tarea de prioridad más alta lo requiere".

- Idealmente, en un sistema basado en planificación prioritaria, una tarea de baja prioridad quedaría interrumpida en el momento en el que una tarea de prioridad alta entrara en la lista de ejecución.

- De manera similar, si hubiera que ejecutar una tarea de más prioridad (como, por ejemplo, la ISR), el planificador le cedería el control.

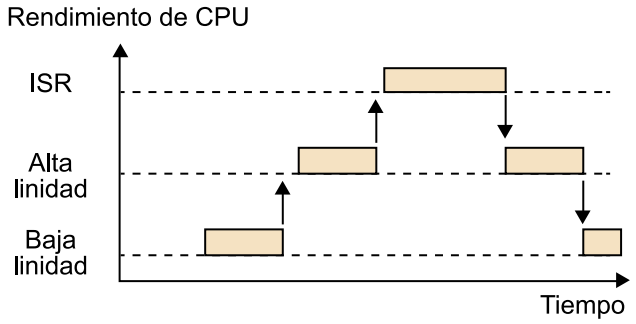
- Una vez acabada la tarea de más prioridad, se emprendería el resto de las tareas por orden inverso de prioridades (podéis ver la figura siguiente).

Este proceso de jerarquización de las prioridades se denomina *interrupciones imbricadas*, o *nested interrupts*, tal como se han definido en la sección destinada a describir las interrupciones.

Véase también

Os recomendamos repasar el apartado 2.1, "Interrupciones".

Secuencia temporal de la transferencia del control



Por ejemplo, tarea activa o en ejecución en cada instante de tiempo, en un sistema multitarea con planificación prioritaria que muestra prioridades imbricadas. Típicamente, la ISR suele tener asignada una prioridad mayor que el resto de las tareas. Cuando la ISR finaliza, devuelve el control a la tarea de más prioridad pendiente de ejecución.

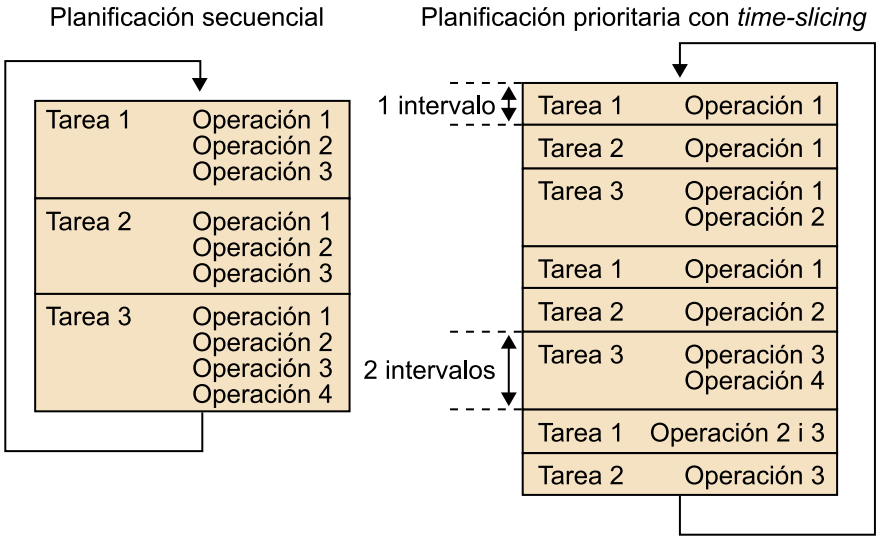
En la práctica, los planificadores prioritarios suelen utilizar esquemas de *time-slicing* para repartir el tiempo de procesador entre las diferentes tareas. Como regla general, a una tarea de alta prioridad simplemente se le asignan más intervalos de tiempos de ejecución que a una de baja prioridad. Solo en casos extremos, como cuando hay que atender una interrupción de muy alta prio-

ridad, se puede recurrir a detener la tarea en curso antes de que finalice su intervalo. A menudo, lo que se hace es atender las interrupciones al finalizar el intervalo en curso.

Hay numerosos algoritmos que permiten determinar el número de ciclos óptimo que se debe asignar a cada tarea en función de su prioridad, de modo que se optimice algún parámetro de funcionamiento del sistema, como, por ejemplo, el tiempo de ejecución de todas las tareas, la tarea más prioritaria, etc. Entre los más utilizados, destaca el *rate monothonic algortihm* (RMA), que permite maximizar el número de tareas que se completan dentro de los plazos previstos.

Tal como se muestra en la figura siguiente, mientras que en una planificación secuencial las tareas se ejecutan hasta su finalización, en una planificación prioritaria con *time-slicing* las tareas se ejecutan hasta que se les acaba el tiempo disponible. Esta diferencia soluciona la mayoría de los problemas habituales de la planificación secuencial y permite desplegar todo el potencial de los sistemas multitarea.

Esquemas de dos metodologías de ejecución de múltiples tareas

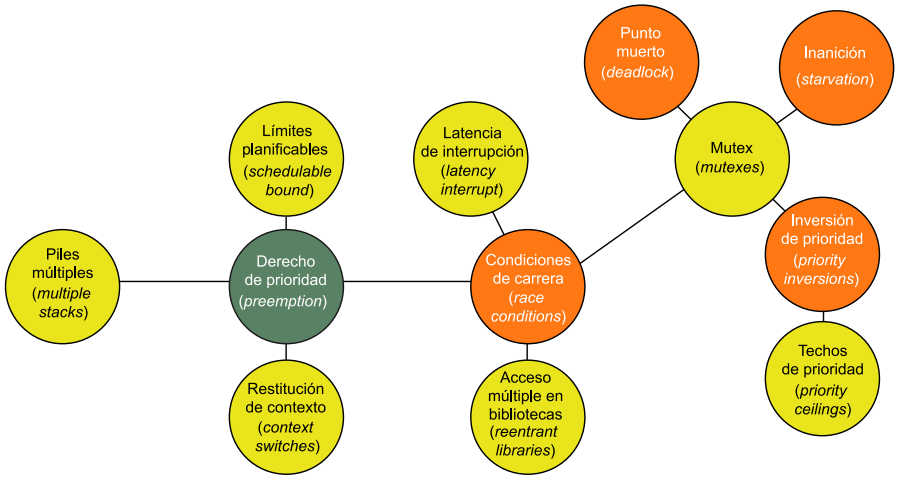


Supongamos que el sistema tiene que ejecutar tres tareas de 2, 3 y 4 operaciones, respectivamente. La última tarea tiene una prioridad más alta que las anteriores. En un sistema basado en un planificador secuencial, cada tarea está activa (por ejemplo, se ejecuta) hasta que se completa la ejecución de todas sus operaciones. En cambio, en un sistema basado en *time-slicing*, el tiempo de ejecución se divide en intervalos y se va repartiendo cíclicamente entre las diferentes tareas. Fijaos en que a las tareas de prioridad más alta se les puede asignar más de un intervalo. En este caso, las operaciones de cada tarea se ejecutan hasta que se acaba el tiempo del intervalo asignado. Fijaos en que si una operación no tiene tiempo de finalizar, continúa en el intervalo siguiente (podéis ver "Tarea 1, operación 1"). Si la operación no utiliza todo el intervalo asignado, el resto del tiempo se puede destinar a completar otras operaciones de la misma tarea (podéis ver "Tarea 1, operaciones 2 y 3").

Peligros de la planificación prioritaria

Hay que tener presente que el hecho de no controlar de manera directa la conmutación entre las diferentes tareas puede ser una fuente de problemas. La figura siguiente recoge hasta diez posibles problemas asociados a una mala utilización de la planificación con prioridad.

Esquema de los principales problemas que puede generar la planificación prioritaria



En amarillo (ved la versión electrónica del PDF o la web) se indican los problemas que pueden ser considerados simples inconvenientes. En naranja, se identifican los aspectos que implican errores de funcionamiento.

A continuación, analizaremos las situaciones más relevantes:

- **Pérdida de rendimiento por exceso de planificación vinculante**: se puede demostrar que para conseguir un máximo grado de cumplimiento de los plazos de ejecución de un conjunto arbitrario de tareas mediante algoritmos RMA puede ser necesario desaprovechar (por ejemplo, dejar un procesador en estado inactivo) hasta el 31% del tiempo de proceso disponible, en el peor de los casos. Este hecho implica llegar a disponer solo del 69% de las presentaciones del hardware si se necesita disponer de funcionamiento multitarea planificado en tiempo real.

- **Pilas múltiples**: es una necesidad intrínseca de los sistemas multitarea planificados con prioridad, puesto que la necesidad de retomar tareas pausadas en el punto en el que se pararon implica la obligación de almacenar en memoria toda la información de la pila de cada una de las tareas mientras no se hayan finalizado completamente. En sistemas con un gran número de tareas concurrentes, este hecho da todavía más relevancia a las limitaciones de memoria habituales en sistemas empotrados.

- **Restitución de contexto**: relacionado con el problema anterior, la continuación de una tarea pausada implica la restitución completa del contexto del sistema; es decir, de los valores de múltiples registros tal como estaban inicialmente. Además de ocupar espacio de memoria, la restitución de esta información implica ejecutar una serie de instrucciones adicionales que toman tiempos de proceso en la ejecución efectiva de las tareas.

En la mayoría de las aplicaciones, las tareas están fuertemente relacionadas e intercambian información. A menudo, este intercambio de información requiere manipular los datos de una misma región de la memoria en los diferentes intervalos de ejecución asignados a cada tarea. Esto implica un riesgo intrínseco de corrupción de los datos si no se trabaja con cuidado.

- **Condiciones de carrera**: son las situaciones en las que los efectos combinados de dos o más tareas concurrentes varían según el orden preciso en el que se ha ejecutado la secuencia de instrucciones combinadas de una y otra. A continuación, veremos algunos ejemplos de esto.

Ejemplo: condiciones de carrera en tareas accediendo a variables globales

a) Supongamos que *x* e *y* son variables globales:

```
global int x;
global int y:

void main() {
      Call concurrently {
            TaskA() and TaskB()
      }
}

void TaskA {
      x = 1;
      y = 2;
      if ( y != 0 ) {
           z = x/y;
      }
}

void TaskB {
      y = 0;
}
```

Si la tarea B interrumpe la ejecución de la tarea A justo después de evaluar la condición y antes de efectuar la división, el regreso a la tarea A llevará a una división por cero.

b) Supongamos ahora que hay cinco tareas concurrentes que ejecutan el mismo código sobre la variable global *x*:

```
global int x;

void main() {
      Call concurrently {
            5*ConcurrentTask();
      }
}

void ConcurrentTask {
      for ( int i = 0; i 100000;< i++ ) {
            x = x + 1;
      }
}
```

En la mayoría de los casos, este programa no dará la solución esperada ($x = 500.000$). La razón es que cada adición implica acceder al valor de *x*, efectuar el incremento y volverlo a guardar en la variable global. En un esquema de planificación prioritaria, las diferentes tareas se pueden ir interrumpiendo mutuamente, de manera incierta, entre cada uno de estos pasos de modo que se efectúen adiciones sobre valores no actualizados.

- **Latencia de interrupción**: el tiempo de espera entre el envío de una señal de interrupción y la ejecución efectiva de la ISR (tiempo de latencia de interrupción) suele ser más grande en sistemas basados en planificación prioritaria por el hecho de que estos pueden desencadenar condiciones de carrera en las tareas que se encontraran en ejecución. Se denominan *secciones críticas* los bloques de código que se deben ejecutar de manera estrictamente secuencial, puesto que, en caso de no hacerlo, pueden producir resultados erróneos. En estos casos, la solución pasa por proteger es-

tas secciones críticas deshabilitando temporalmente la anotación a interrupciones durante su ejecución. Esta estrategia, obviamente, puede tener consecuencias en la capacidad del sistema para responder en tiempo real.

- **Acceso múltiple a bibliotecas**: es posible que las condiciones de carrera se produzcan entre dos accesos concurrentes a una misma biblioteca compartida. Se trata de situaciones especialmente preocupantes, puesto que son condiciones de carrera entre dos secciones de código idénticas que pueden intentar hacer lo mismo sobre los mismos recursos, simultáneamente. En este caso, no suele ser posible deshabilitar las interrupciones, puesto que se trata de secciones de código cerradas y de propósito muy general en las que hay que evitar introducir comportamientos inesperados. La solución suele ser hacer que estas partes del código se ejecuten de manera anormalmente lenta, y evitar así que puedan llegar a ser concurrentes.

- **Mútex**: como solución para evitar corrupciones de datos asociados con condiciones de carrera, se suele recurrir a implementar indicadores binarios (los mútexs) que permitan bloquear el acceso a un conjunto de datos en memoria. Forman parte de las herramientas incluidas habitualmente en los RTOS. La tarea del programador consiste en identificar los datos que pueden ser corrompidos y las secciones críticas del código para, a continuación, crear un mútex asociado a estos datos. A partir de aquí, es necesario que el código dé a cada sección crítica indicaciones para reservarse el acceso al mútex antes de iniciar la ejecución y de liberarlo al finalizar.

Ejemplo: condición de carrera evitada con un mútex

El primer ejemplo del subapartado 3.4 se podría proteger mediante un mútex de la manera siguiente.

```
global int x;
mutex mutex1(x);      // asociamos un mútex a la variable x

void main() {
      Call concurrently {
            5*ConcurrentTask()
      }
}


void ConcurrentTask {
      for ( int i = 0; i 100000;< i++ ) {
            mutex.lock();      // bloqueamos el mútex
            x = x + 1;
            mutex.unlock();    // liberamos el mútex
      }
}
```

En la práctica, los mútexs suelen causas otros problemas y, por lo tanto, hay que evitarlos.

- **Inanición**: si dos tareas de prioridad muy diferente comparten un mismo mútex, se puede dar el caso de que la de menos prioridad no pueda nunca finalizar en el plazo esperado, puesto que la de más prioridad (que puede

tener asignado mucho más tiempo de ejecución) no libera nunca el mútex en el momento de ceder el control a la de baja prioridad y, por lo tanto, se ve forzada a devolver el control a la anterior.

- **Punto muerto (*deadlock*)**: en este caso, dos tareas de prioridad similar reservan un mútex en su turno de ejecución, pero si su intervalo de ejecución finaliza antes de poderlo liberar, la otra siempre lo encuentra bloqueado. Esto implica una situación indefinida en la que, a las dos tareas, se les asignan turnos de ejecución a los que van renunciando alternativamente puesto que encuentran un mútex bloqueado. En estos casos, la solución suele ser reiniciar el sistema. A continuación, veremos un ejemplo de punto muerto.

Ejemplo: mútexs múltiples que causan una situación de punto muerto

```
mutex mutex1, mutex2;

void main() {
      Call concurrently {
            TaskA() and TaskB()
      }
}

void TaskA() {
      while( 1 ) {
            ...
            mutex1.lock();
            /* sección crítica 1 */
            mutex2.lock();
            /* sección crítica 2 */
            mutex2.unlock();
            /* sección crítica 1 */
            mutex1.unlock();
      }
}

void TaskB() {
      while( 1 ) {
            ...
            mutex2.lock();
            /* sección crítica 2 */
            mutex1.lock();
            /* sección crítica 1 */
            mutex1.unlock();
            /* sección crítica 2 */
            mutex2.unlock();
      }
}
```

En este ejemplo, se necesitan dos mútexs (mutex1, mutex2) llamados en dos tareas concurrentes (TaskA y TaskB). Siguiendo la secuencia de ejecución se observa que:

- TaskA bloquea mutex1 para ejecutar la sección crítica 1 de manera segura.
- TaskB bloquea mutex2 para ejecutar la sección crítica 2 de manera segura.
- Task y TaskB ejecutan las secciones críticas 1 y 2, respectivamente.
- TaskA intenta bloquear mutex2 para trabajar ahora sobre la sección crítica 2 y queda bloqueada a la espera de que TaskB libere mutex2.
- TaskB intenta bloquear mutex1 para trabajar ahora sobre la sección crítica 1 y queda bloqueada a la espera de que TaskA libere mutex1.

Vemos, por lo tanto, que esta forma de programación supone una situación de punto muerto.

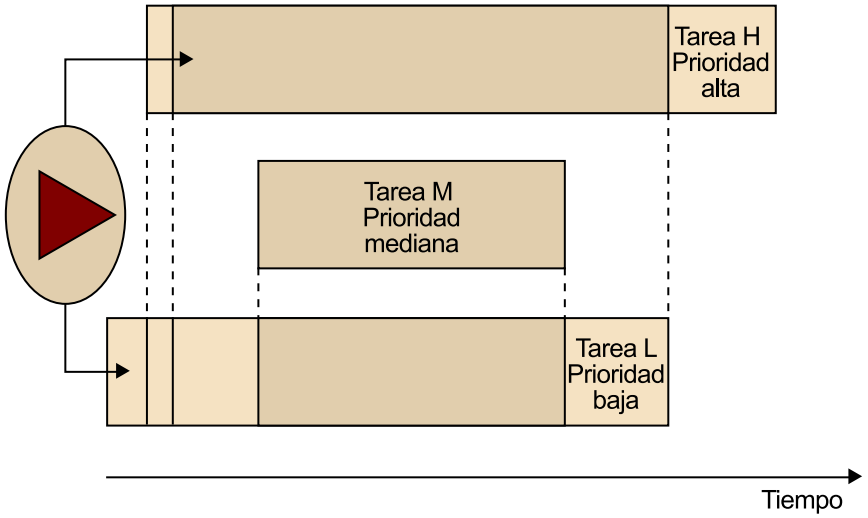
- **Inversión de prioridad**: esta situación es un poco más sutil que las dos anteriores, puesto que requiere el concurso de tres tareas de prioridades crecientes en las que las de más y menos prioridad comparten un mútex. Si la de más prioridad queda a la espera de que la de menos prioridad libere el mútex, se podría dar la situación de que una tarea de prioridad intermedia impidiera la ejecución de la de menos prioridad y, por lo tanto, impidiera la liberación del mútex. El resultado final es que una tarea de menos prioridad (la intermedia) evita que otra de más prioridad pueda acceder al procesador.

  La inversión de prioridad suele ser considerada un error del sistema, ya que viola las prioridades naturales.

Hay dos soluciones habituales. La primera, denominada **herencia de prioridad**, consiste en asignar a la tarea de baja prioridad la misma prioridad que a la tarea de alta prioridad durante el tiempo que bloquea el recurso compartido con esta.

La segunda, el establecimiento de **techos de prioridad**, consiste en asignar a cada recurso compartido (que son los que, bloqueados mediante mútexs, generan las situaciones de inversión de prioridad) una prioridad tan elevada, como mínimo, como la tarea de más prioridad que lo utilice. Desgraciadamente, esta solución también viola las prioridades naturales, puesto que puede hacer que una tarea de baja prioridad, que utiliza un recurso considerado de alta prioridad, avance a una tarea de prioridad intermedia.

Una tarea de baja prioridad (Tarea L) y una de alta prioridad (Tarea H) comparten un recurso controlado con un mútex



Suponemos que justo después de que L bloquee el mútex, H queda lista para ejecutarse. Sin embargo, H tiene que esperar a que L deje de necesitar el recurso (lo libere) para poder continuar. Por lo tanto, queda como no lista y pendiente de ejecución. Si, en estas circunstancias, una tarea de prioridad intermedia (Tarea M) queda lista para ejecutarse, avanzará a L y le impedirá continuar con su actividad, gracias a su prioridad. El resultado global es que la tarea M de prioridad intermedia se ha ejecutado antes de la tarea H de alta prioridad. Por lo tanto, se ha producido el fenómeno de la inversión de prioridad.

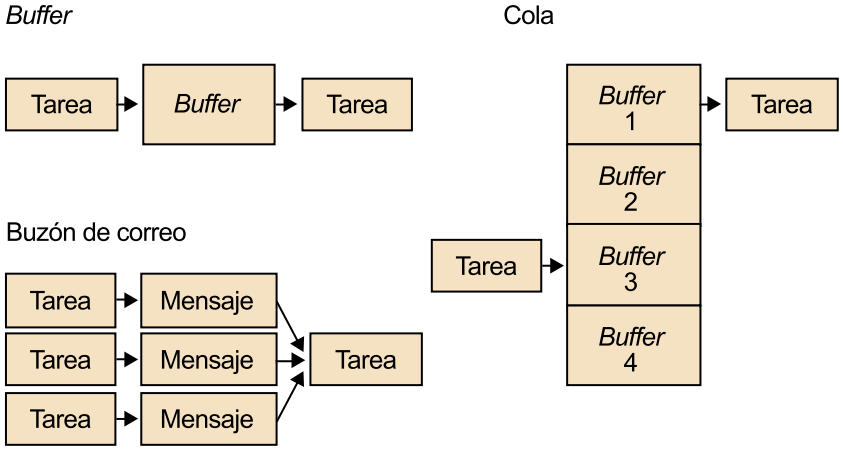
Como se verá más adelante, si se utiliza un RTOS moderno, se tiene acceso a herramientas alternativas (los buzones de correo), que son el método seguro y correcto de intercambiar información entre tareas, puesto que evitan parte de los problemas de bloqueo de recursos y también cualquier corrupción de los datos.

Todo lo que se ha descrito en esta sección tiene su origen en el carácter incierto (a efectos prácticos es casi aleatorio) con el que se intercalan las tareas en los sistemas basados en planificación con prioridad. En las próximas secciones se describen algunas de las herramientas que permiten prevenir y evitar los inconvenientes de la planificación con prioridad. En cualquier caso, es responsabilidad del programador decidir si este tipo de planificación es el más adecuado para su aplicación o si habría suficiente con otras aproximaciones más simples.

3.3.6. Herramientas para la comunicación y sincronización de tareas

Hay varias herramientas de programación que permiten satisfacer las necesidades de intercambiar información entre tareas (comunicación) y controlar el ritmo y el orden de ejecución (sincronización). En sistemas basados en RTOS, el propio sistema operativo las pone a disposición del programador. Pueden ser también implementadas por el programador en sistemas diseñados desde cero. Veamos las más habituales. La figura siguiente muestra unos esquemas de su funcionamiento básico:

Representación esquemática de las herramientas para la comunicación entre tareas más habituales



Los semáforos permiten regular la sincronía entre tareas. En el ejemplo, la tarea A llama a la tarea B y queda a la espera de su finalización. Por ello, permanece inactiva hasta que el semáforo, activado por la finalización de la tarea B, se pone en verde. Los *buffers* permiten intercambiar datos entre diferentes tareas de manera simple. Para el intercambio de informaciones permanentes, se pueden emplear las colas, consistentes en secuencias de *buffers* que se van llenando sucesivamente. Cada vez que un *buffer* queda lleno, puede ser leído por otra tarea. Finalmente, los buzones de correo permiten el envío de mensajes entre diferentes tareas.

Buffers

Los *buffers* permiten intercambiar datos entre tareas. Típicamente, la tarea fuente de datos solicita un *buffer* al RTOS (o a cualquiera otro software que se encargue de la gestión de los *buffers*), le pasa los datos y pide que sean transferidos a la tarea de destino. En este momento, el RTOS pasa a la tarea de destino un puntero al *buffer* de datos con información sobre la longitud de los datos que debe leer.

Ejemplo: intercambio de datos entre tareas mediante un *buffer*

El código siguiente muestra un ejemplo de cómo hay que transferir datos entre dos procesos concurrentes mediante un *buffer*:

```
int N = bufferSize;
dataArray buffer[N];
int count = 0;

void main() {
      Call concurrently {
            TaskA() and TaskB()
      }
}

void TaskA() {
      int y;
```

```
        while( 1 ) {
              produce(&data);
             while( count == N );      //espera si el buffer está lleno
              buffer[i] = data;
              i = (i + 1) % N;
              count = count + 1;
        }
  }

  void TaskB() {
        int y;
        while( 1 ) {
              while( count == 0 );      //espera el buffer está vacío
              data = buffer[i];
              i = (i + 1) % N;
              count = count - 1;
              consume(&data);
        }
  }
```

Observemos que la TaskA es solo fuente de datos, mientras que la TaskB es la usuaria.

Dado que cada tarea genera o consume datos continuamente, se ha implementado una verificación (basada en un bucle *while*) que comprueba que o bien el *buffer* tiene posiciones disponibles (TaskA), o bien está completamente vacío.

Esta solución no es la mejor, puesto que es posible que se corrompan los datos de la variable global *count* (que registra el número de posiciones ocupadas en el *buffer*). En el último ejemplo del subapartado 3.4.5, "Mútexs múltiples que causan una situación de punto muerto", se muestra una solución basada en los mútexs.

Colas

Las colas son cadenas de *buffers* que permiten el intercambio de un volumen más grande de datos de una manera secuencial. Por ejemplo, mientras la tarea destino procesa la información del primer *buffer* de la cola, la tarea fuente puede empezar ya a llenar de datos el segundo.

Los *buffers* y las colas representan las formas más básicas de intercambiar información entre tareas. En combinación con las herramientas de sincronización, que se describen a continuación, se pueden proteger los datos del *buffer* ante la corrupción (puesto que a menudo son recursos compartidos).

Mútexs

Los mútexs son indicadores binarios utilizados usualmente para proteger recursos compartidos de accesos simultáneos durante la ejecución de secciones críticas de código.

Para ilustrar su funcionamiento, podemos pensar que el mútex funciona de manera análoga a como se gestiona la llave del lavabo en un bar:

- Cuando un cliente (tarea) quiere acceder a este recurso, pide la llave al camarero (RTOS).

- Si este no tiene la llave, significa que hay otro cliente que lo utiliza y, por lo tanto, tiene que esperar.

- En el momento en el que la llave se devuelve (se libera el mútex), el recurso queda disponible para el resto de los clientes.

De este modo, el mútex actúa a la vez como bloqueador de un recurso y como indicador de que está ocupado.

¿Cómo habría que proceder en el caso de disponer de varios recursos similares, pero no equivalentes (como es el caso de disponer de un lavabo para hombres y otro para mujeres)? Para gestionar correctamente estos dos recursos, habría que establecer dos mútexs diferentes, uno para cada uno de los recursos, de manera que cada tipo de tarea supiera cuándo puede acceder al que le corresponde.

Ejemplo: control de acceso a *buffers* mediante mútex

El código siguiente muestra cómo se puede proteger el contador de posiciones de *buffer* ocupadas (*count*) ante la corrupción de datos mediante un mútex.

```
int N = bufferSize;
dataArray buffer[N];
int count = 0;
mutex mutex_count(count);             // asociamos un mútex a count

void main() {
      Call concurrently {
            TaskA() and TaskB()
      }
}

void TaskA() {
      int i;
      while( 1 ) {
            produce(&data);
            while( count == N );    //espera si el buffer está lleno
            buffer[y] = data;
            i = (i + 1) % N;
            mutex_count.lock();      // count bloqueada
            count = count + 1;
            mutex_count.unlock();      // count liberada
      }
}

void TaskB() {
      int i;
      while( 1 ) {
            while( count == 0 );    //espera si el buffer está vacío
            data = buffer[i];
            i = (i + 1) % N;
            mutex_count.lock();      // count bloqueada
            count = count - 1;
            mutex_count.unlock();      // count liberada
            consume(&data);
      }
}
```

Hay que recordar que, tal como se describe en apartados anteriores, el uso de mútexs puede suponer situaciones de inversiones de prioridad.

Semáforos

Los semáforos son estructuras de datos que tienen un comportamiento similar a un contador y que se utilizan para la sincronización entre tareas. Su finalidad es intercambiar indicaciones de sincronía entre tareas.

Dado que los semáforos pueden tener más de dos estados, son adecuados para intercambiar varios tipos de mensaje entre tareas (siguiendo el ejemplo, indicar cuántas veces se debe pulsar el botón).

Para garantizar que funcionen correctamente, se deben organizar de modo que cada tarea interactúe con un mismo semáforo de una **única** manera: o bien incrementándolo o bien leyéndolo.

Sincronización entre tareas mediante un semáforo

Imaginemos dos tareas: la primera, responsable de detectar la pulsación del botón de puesta en marcha de un sistema mediante una interrupción (llamémosla `power_button_pushed`) y la segunda, encargada de activar un periférico como consecuencia de esta acción. Utilizando semáforos esta funcionalidad se podría implementar haciendo que:

- la primera tarea incremente el valor del semáforo en el momento en el que se produzca la pulsación/interrupción, o que
- la segunda esté a la espera de detectar este incremento para iniciar su actividad.

Esta funcionalidad se podría implementar de la manera siguiente:

```
interrupt irq(power_button_pushed);      // creamos una interrupción
semaphore power_button;                  // creamos un semaphore

void main() {
      Call concurrently {
            TaskA() and TaskB()
      }
}

void TaskA() {
      while(1) {
           while(irq == false);     // esperamos a que pulse el botón
            power_button.incremento();
      }
}

void TaskB() {
      int i;
      while( 1 ) {
            if (i < power_button.value()) {
                  /* actions */
            }
            i = power_button.value();
            delay(10);

      }
}
```

La TaskA permanece a la espera de recibir la interrupción. Cada vez que la recibe, incrementa el valor del semáforo.

La TaskB mantiene un registro del valor del semáforo. Si se ha producido un incremento, ejecuta /* actions */. Si no se ha incrementado durante la última iteración, no se hace nada. En este tipo de implementación, en la que los plazos de ejecución no se han es-

pecificado previamente, se puede añadir un pequeño retardo para dar tiempo a que el semáforo incremente su valor de manera segura.

Observemos que las dos tareas se basan en un bucle básico de control que itera continuamente. Por lo tanto, ambas tareas se mantienen activas mientras el sistema esté puesto en marcha.

Actualmente, hay herramientas avanzadas que permiten comunicar y sincronizar tareas al mismo tiempo y que evitan buena parte de los problemas de las herramientas descritas hasta el momento. A continuación las describiremos.

Buzones de correo

Los buzones de correo[6] permiten enviar mensajes entre tareas que contengan tanto datos como información de sincronización.

(6)En inglés, *mailboxes*.

Se comportan como el correo habitual:

- El software gestor de este correo (por ejemplo, el RTOS) se encarga de almacenar los mensajes en buzones hasta que son leídos por la tarea correspondiente.

- Es posible enviar mensajes a múltiples tareas y recibir también de varias, pero no es posible modificar su contenido una vez enviados.

- También suele ser posible establecer prioridades en los mensajes.

Por todo ello, los buzones pueden emular las funcionalidades de los *buffers*, colas, semáforos y mútexs y prevenir cualquier riesgo intrínseco, puesto que en ningún momento utilizan espacios de memoria compartidos.

Comunicación entre tareas mediante buzones de correo

A continuación, se muestra una implementación alternativa del segundo ejemplo del subapartado 3.4.5, "Condición de carrera evitada con un mútex", basada en buzones de correo.

```
int N = bufferSize;
dataArray buffer[N];
int count = 0;
mutex mutex_count(count);             // asociamos un mútex a count

void main() {
      Call concurrently {
            TaskA() and TaskB()
      }
}

void TaskA() {
      while( 1 ) {
            produce(&data);
            send(TaskB, &data);      // se envían datos
            /* actions 1 */
            receive(TaskB, &data); // se espera a recibir datos
            consume(&data);
      }
}

void TaskB() {
      while( 1 ) {
            receive(TaskA, &data);      // se espera a recibir datos
            transform(&data)
            send(Task A, &data);      // se envían datos
            /* actions 2 */
      }
}
```

El subapartado 3.4.5, "Condición de carrera evitada con un mútex", se ha ampliado haciendo que TaskB devuelva los datos, una vez transformados, a TaskA para su reutilización. La instrucción send() manda un mensaje con el puntero a data a la tarea destinataria indicada. La instrucción receive() espera recibir un mensaje proveniente de la tarea indicada.

Fijémonos en que la implementación, a pesar de tener una funcionalidad mayor, es más simple que las anteriores y, sobre todo, más segura.

3.3.7. Gestión de recursos

Memoria

En entornos multitarea, buena parte de los recursos del sistema, y no solo el tiempo de procesador, son compartidos por las diferentes actividades que hay que llevar a cabo. La memoria es otro recurso intrínsecamente compartido. Además de gestionar los problemas derivados de la corrupción de datos vistos anteriormente, sea cual sea la arquitectura de nuestro sistema, hay que establecer mecanismos que permitan repartir la memoria disponible entre las diferentes tareas.

La solución más frecuente consiste en hacer que las tareas soliciten memoria al software encargado de su gestión (por ejemplo, el RTOS) cada vez que la necesitan. Aun así, este software se encarga de liberarla cuando las tareas finalizan.

Gracias a este proceso continuo de reserva y liberación de pequeñas porciones de memoria (**asignación dinámica**), el sistema puede funcionar, ejecutando múltiples tareas, con unas necesidades de memoria moderadas.

Por simplicidad, se suele optar por dirigir la memoria disponible por bloques. Un bloque de memoria es una porción de memoria a la que se accede con una dirección lógica única. De este modo, forma el mínimo conjunto de posiciones de memoria contiguas accesibles y asignables a una única tarea.

Si los bloques son demasiado grandes para las necesidades habituales de las tareas del sistema, se producirá un derroche importante de espacio de memoria.

Ejemplo

Cada tarea bloquea un bloque, independientemente del uso que haga de él.

Al mismo tiempo, habrá pocos bloques disponibles para asignar y, por lo tanto, puede limitar innecesariamente el número de tareas activas simultáneamente.

Por el contrario, si los bloques son demasiado pequeños, las tareas que requieran más de un bloque típicamente necesitarán que sean contiguos y, por el mismo proceso de asignación/liberación, puede ser que esto ya no sea posible al cabo de un cierto tiempo de funcionamiento. Este fenómeno se conoce como **fragmentación de la memoria** y puede suponer un retardo innecesario del sistema.

Véase también

Podéis ver, en el subapartado 3.3.2, la tercera figura: "Diagrama de estados de una posible implementación del ascensor del ejemplo que acabamos de analizar".

La conclusión de todo esto es que para conseguir un máximo aprovechamiento de la memoria y un máximo rendimiento, es necesario dimensionar correctamente los bloques en función de las necesidades de nuestro sistema.

Interrupciones

Es obvio que un sistema en tiempo real necesita gestionar interrupciones. Como se ha descrito anteriormente, la ejecución de las tareas se ve detenida en el momento en el que se recibe una interrupción.

En este instante, se cede el control a la ISR, que identifica el tipo y actúa en consecuencia, y añade la tarea de atención correspondiente a la lista de ejecución.

A partir de este momento, ya es responsabilidad del planificador de tareas decidir si esta nueva tarea tiene suficiente prioridad para alterar sus planes o, simplemente, será atendida en un momento posterior y se continúa con la tarea que se había interrumpido inicialmente.

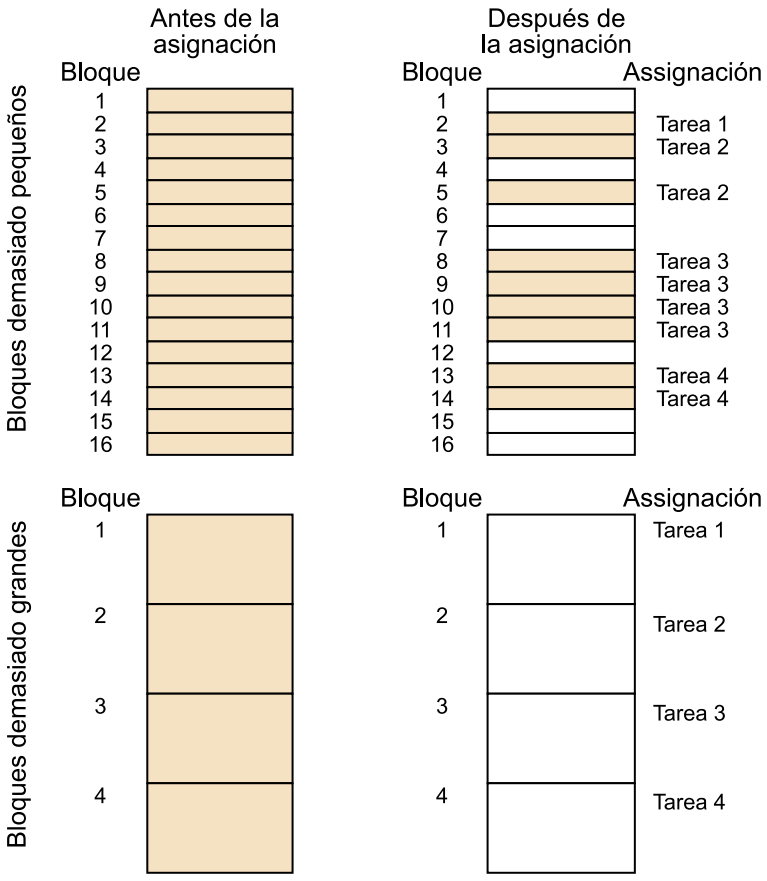
Para hacer el seguimiento de este proceso se suelen implementar, como mínimo, los servicios de atención a interrupciones siguientes:

- **Entrada de interrupción**: notifica al sistema la llegada de una interrupción y almacena el contexto de la tarea que queda interrumpida de manera que sea posible que continúe.

- **Atención de interrupción**: consulta el vector de interrupciones e identifica la tarea de atención a interrupción que hay que añadir a la lista de tareas.

- **Salida de interrupción**: notifica al sistema que se ha completado la atención a la interrupción y le indica que puede continuar la ejecución, y hace que el planificador reevalúe la nueva lista de tareas.

Esquema de los problemas habituales en la asignación de bloques de memoria



En el caso de emplear bloques demasiado pequeños, se produce una fragmentación. Si los bloques son demasiado pequeños, se derrocha mucha memoria.

Resumen

En este módulo didáctico hemos repasado conceptos básicos relativos al software y a su arquitectura, y hemos visto que su aplicación en los programas que gobiernan los sistemas empotrados puede ayudar a trabajar de una manera más optimizada en condiciones de memoria limitada, potencia de cálculo moderada o requisitos para trabajar en tiempo real. Lo hemos hecho analizando los aspectos siguientes:

a) En primer lugar, hemos repasado las características propias de los sistemas empotrados prestando atención a:

- La **elección del lenguaje** de programación para desarrollar su software de control.
- **Identificar las herramientas** existentes para desarrollar este software.

b) En segundo lugar, hemos analizado conceptos básicos de programación que ayudan a agilizar el código de control de los sistemas empotrados. Los principales conceptos que hemos presentado son los siguientes:

- Las **interrupciones**, como un mecanismo básico de comunicarse con los periféricos y otros elementos del sistema en tiempo real.
- Los **punteros**, como variables que nos permiten manipular la memoria de una manera más eficiente.
- **Funciones *inline*** y funciones **externas**, como ejemplos de técnicas de programación que permiten compactar y hacer más modular el código de control.

c) Finalmente, hemos visto los modelos de programación existentes y hemos identificado las ventajas e inconvenientes de su utilización, siempre intentándolos relacionar con ejemplos prácticos. Los modelos de programación analizados han sido:

- Los **simples**, basados en la utilización de un bucle de control simple o el control de eventos.
- Las **máquinas de estado**, como alternativa que permite simplificar el código cuando el potencial de los modelos simples se vuelve insuficiente.
- Los **sistemas multitarea**, como solución para intentar acercarnos lo máximo posible a sistemas operativos ideales trabajando en tiempo real. Se han

identificado los conceptos necesarios para entender su funcionamiento, y también la metodología para poder planificar y ejecutar las tareas de una manera adecuada. Se ha prestado atención a la necesidad de sincronizarlas y también a gestionar los recursos de la máquina de la mejor manera posible.

A lo largo de todo el módulo se han intercalado problemas y ejercicios prácticos con la voluntad de aclarar al máximo posible los conceptos aquí presentados. Finalmente, hemos creído necesario incluir ejercicios de autoevaluación corregidos al final del módulo, puesto que la programación es una disciplina que solo se llega a controlar mediante la práctica continuada.

Actividades

1. Elaborad un programa en lenguaje C que calcule los valores futuros de depósitos de ahorros mensuales suponiendo que aplicamos el cálculo de interés compuesto, y que este puede variar del 10 al 20% anual. Utilizad el formalismo de programación consistente en pasar una función a otra mediante punteros.

El programa debería considerar los tres casos que hay dependiendo de la frecuencia con la que este interés se suma al depósito:

a) Por períodos superiores al mes.

b) Diariamente.

c) De manera continua.

En cada caso, la cantidad futura F se relaciona con las imposiciones mensuales A, la tasa de interés anual i, el número de años n y el número de composiciones m (m = 1 para períodos anuales, m = 2 para semestrales, m = 4 para trimestrales, m = 12 para mensuales y m = 360 para diarios) mediante las fórmulas siguientes:

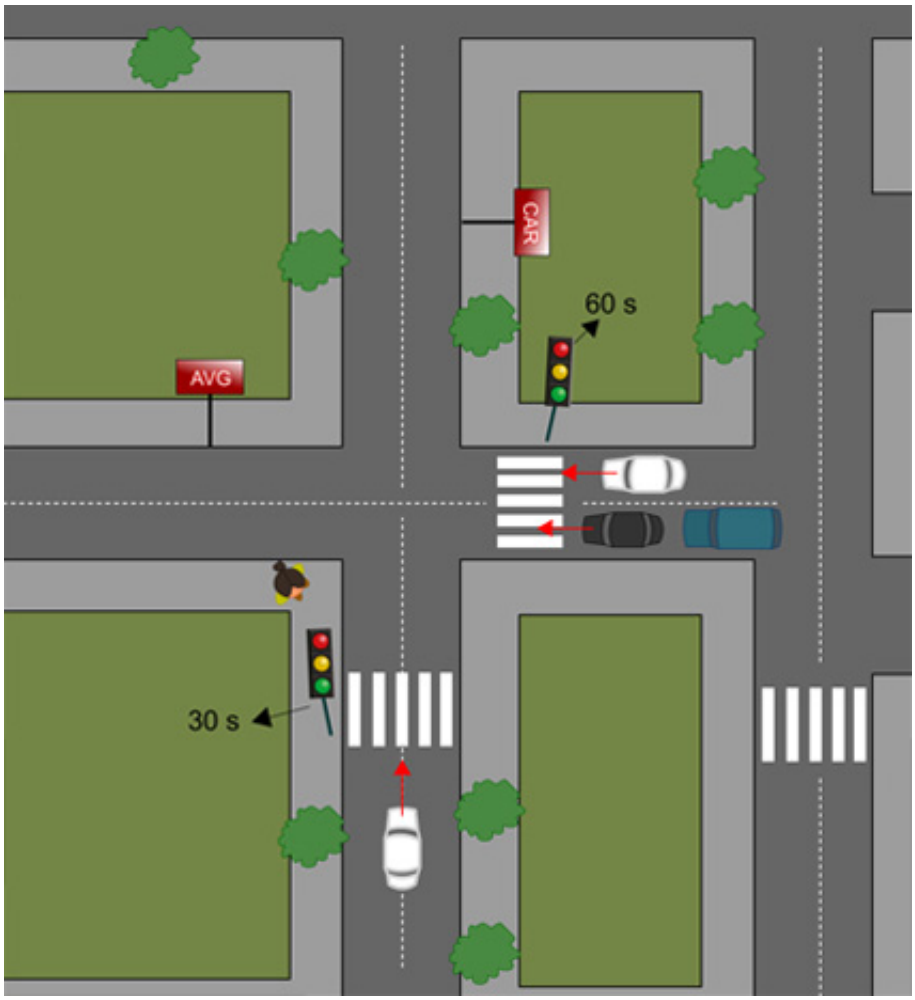
a) $F = 12A \cdot \left[\frac{(1+i/m)^{mn}-1}{i}\right]$

b) $F = A \cdot \left[\frac{(1+i/m)^{mn}-1}{(1+i/m)^{m/12}-1}\right]$

c) $F = A \cdot \left[\frac{e^{n}-1}{e^{i/12}-1}\right]$

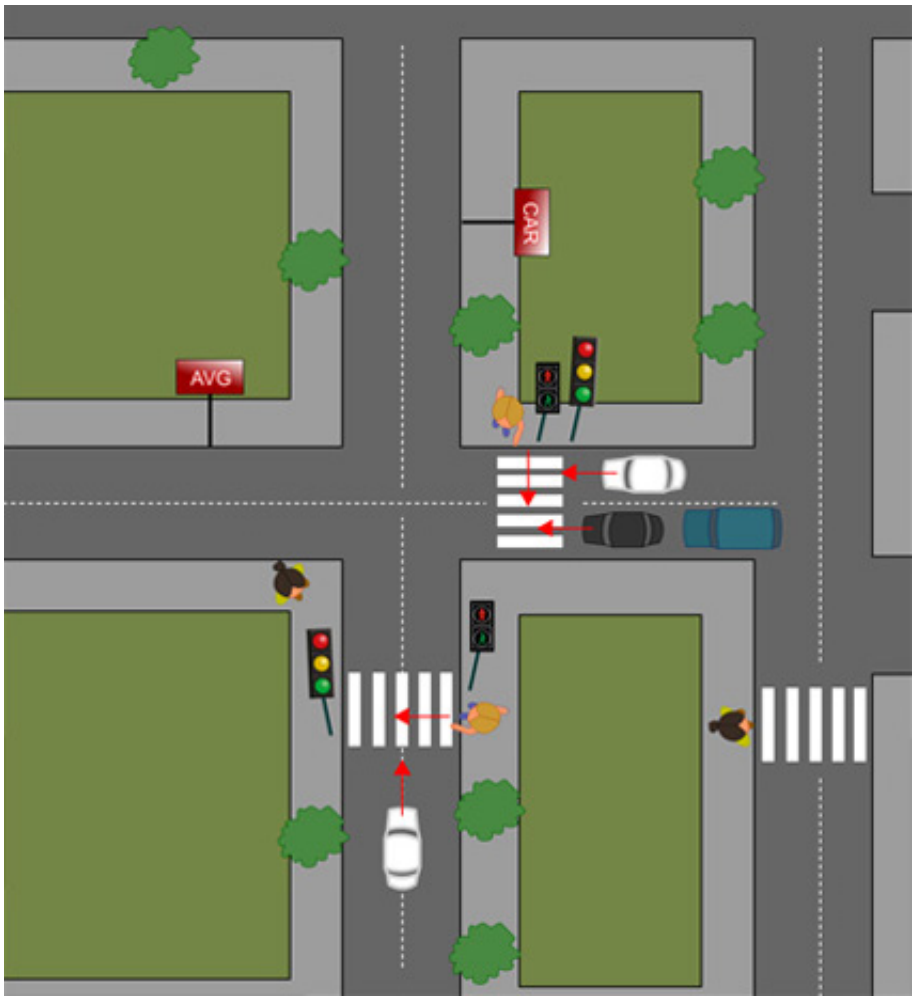
2. Diseñad una máquina de estados que permita implementar la descripción funcional siguiente:

"Se tiene que regular la circulación de vehículos en el cruce entre una avenida principal (AV) y una calle secundaria (CLL) mediante semáforos de tres posiciones (verde, naranja y rojo). Para garantizar la fluidez del tráfico, se debe permitir el paso de vehículos por AVG durante 60 segundos y por CAR durante 30 segundos, en cada turno. Por seguridad, la señal naranja (que indica cambio a rojo inminente) se debe mantener durante 2 segundos y hay que esperar 3 segundos entre que se corta la circulación por una de las calles (luz roja) y se permite el paso por la otra (luz verde)".



Identificad los estados y las variables necesarios, y también las transiciones y tareas que hay que realizar. Elaborad el diagrama de estados correspondiente y proponed una implementación en código fuente.

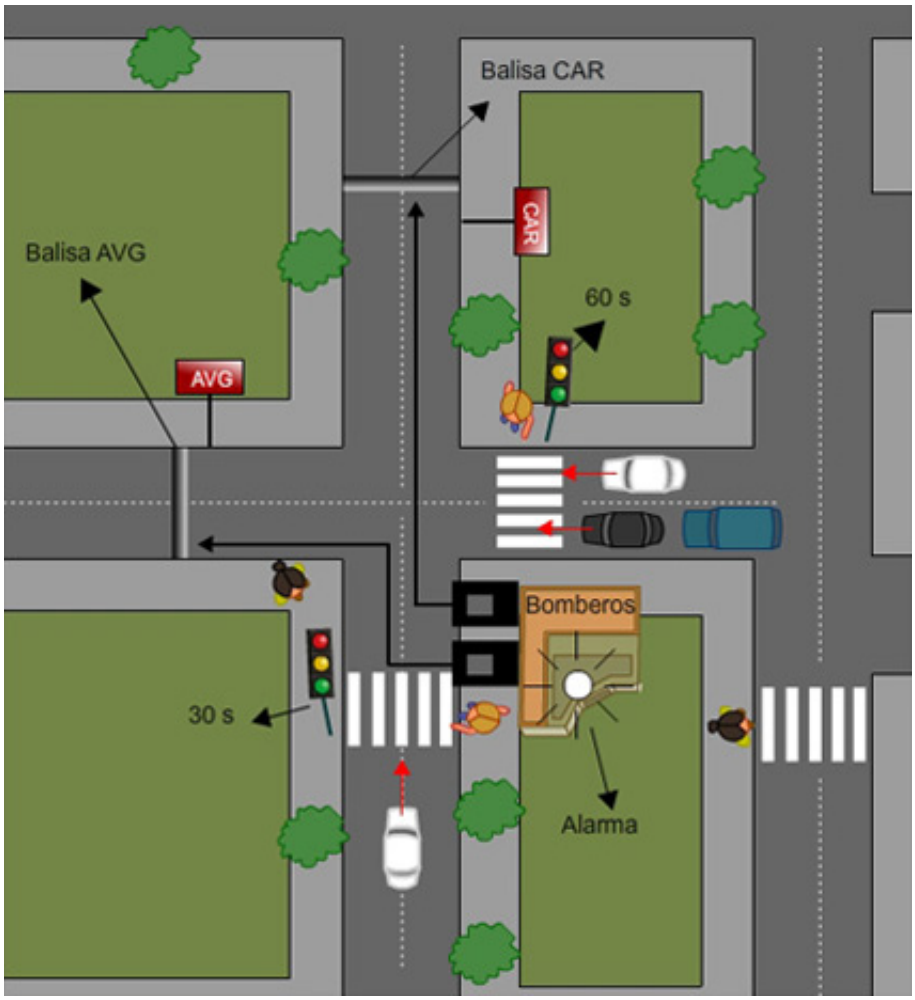
3. Modificad el diseño de la actividad 2 para que implemente también un semáforo que regule el paso de peatones por la avenida AV. Suponed que el semáforo de peatones dispone de 2 luces y que la transición de verde a rojo hay que indicarla haciendo parpadear la luz verde durante 2 segundos. Enumerad únicamente los cambios que habría que hacer en el diseño anterior.



4. Se ha instalado una estación de bomberos en la calle CAR tal como indica la figura. Modificad el diseño de la actividad 2 para que implemente un sistema de salida de emergencia de los vehículos de bomberos según la descripción siguiente:

"Al sonar la alarma en la estación de bomberos, hay que cortar la circulación en AV y en CLL para permitir el paso de vehículos de bomberos. Los semáforos vuelven al funcionamiento normal después de que el vehículo de bomberos haya superado las balizas de seguridad situadas en los dos calles. Hay que parar el tráfico de los dos calles con seguridad".

Implementad el sistema con máquinas de estado jerárquicas.



5. Diseñad una máquina de estados que permita implementar la descripción funcional siguiente.

"Se debe controlar el funcionamiento de un contestador automático. Hay que reproducir un mensaje de bienvenida de 7 segundos, seguido de un pitido de 1 segundo después de recibir tres tonos consecutivos. A continuación, hay que activar la grabación del mensaje durante no más de 60 segundos. Al completarse la grabación, hay que incrementar la pantalla que cuenta los mensajes grabados en una unidad. La grabación se para en el momento en el que el interlocutor cuelga. También hay que prever un modo que reproduzca los mensajes, cuando los haya grabado, y que solo se active al pulsar un botón del equipo. En este modo, el contestador no puede atender llamadas".

Emplead máquinas de estados jerárquicas. Identificad los estados y las variables necesarios, y también sus transiciones y las tareas que hay que llevar a cabo. Elaborad el diagrama de estados correspondiente y proponed una implementación en código fuente.

6. Responded a las cuestiones siguientes sobre conceptos fundamentales del módulo:

a) ¿Cómo se define una interrupción?

b) ¿Qué es una ISR?

c) ¿Qué son las acciones y los eventos que definen un proceso de interrupción?

d) ¿Qué son las principales ventajas de usar interrupciones en el funcionamiento de los sistemas empotrados?

e) ¿Qué es un vector de interrupción?

f) Enumerad los cinco pasos que hay que seguir en el diseño de una máquina de estados.

g) Enumerad hasta diez posibles problemas causados por la planificación prioritaria.

Ejercicios de autoevaluación

Indicad si las interrupciones siguientes son verdaderas o falsas, y justificad vuestra respuesta:

1. Los vectores de interrupción no contienen código ejecutable.

2. Las interrupciones se deben programar en un lenguaje de bajo nivel.

3. Hay tres métodos para generar vectores de interrupción.

4. La prioridad de las interrupciones siempre puede ser definida por el programador.

5. En un sistema empotrado en el que la velocidad de ejecución no es prioritaria, la utilización de la técnica de *inlining* puede ayudar a optimizar su funcionamiento.

6. Un programador experimentado siempre incluye funciones *inline* en su código.

7. La utilización de funciones externas permite reducir el coste económico de desarrollo del software de sistemas empotrados.

8. Las máquinas de estado son adecuadas para implementar sistemas muy complejos con necesidades multitarea.

9. La planificación multitarea es siempre mejor que la secuencial.

Solucionario

Actividades

1. La solución aquí propuesta en ningún caso es única y, por lo tanto, podéis encontrar aproximaciones al problema presentado por el enunciado igualmente válidas. Usaremos una función que denominaremos *tabla*, a la que, y en función de los datos que el usuario introduzca por teclado, pasaremos un puntero a otra función que se basará en la fórmula adecuada de interés compuesto para hacer el cálculo solicitado. Se recomienda que ejecutéis este código para comprobar el resultado final.

```
/* Programa de cálculo de depósitos mediante interés compuesto */
#include <studio.h>
#include <studio.h>
#include <studio.h>
#include <studio.h>

/* Prototipos de funciones que usaremos en el programa */
void tabla (double (*pf) (double i, int m, double n), double a,
int m,  double n);
double md1 (double i, int m, double n);
double md2 (double i, int m, double n);
double md3 (double i, int m, double n);

/* Función principal del programa */
main() {
      int m;       /* # períodos de composición por año */
      double n;       /* # años */
      double a;       /* Cantidad imposición mensual */
      char freq;       /* Indicador frecuencia de la composición */
      /* Entrada de datos */
    printf("\n VALOR FUTURO DE UNA SERIE DE DEPÓSITOS MENSUALES\n\n");
      printf("\n Cantidad de los pagos mensuales: ");
      scanf("%lf", &a);
      printf("Número de años: ");
      scanf("%lf", &n);
      /* Introducción de la frecuencia de la composición */
      do {
            printf("Freq. Composición (A,S,Q,M,D,C: ");
            scanf("%lf", &freq);
            freq = toupper(freq); /* convertimos a mayúsculas */
            if (freq == 'A') {
                  m = 1;
                  printf("\n Composición anual\n");
            }
            else if (freq == 'S') {
                  m = 2;
                  printf("\n Composición semestral\n");
            }
            else if (freq == 'Q') {
                  m = 4;
                  printf("\n Composición trimestral\n");
            }
            else if (freq == 'M') {
                  m = 12;
                  printf("\n Composición mensual\n");
            }
            else if (freq == 'D') {
                  m = 360;
                  printf("\n Composición diaria\n");
            }
            else if (freq == 'C') {
                  m = 0;
                  printf("\n Composición continua\n");
            }
            else {
                  printf("\n ERROR\n\n");
            }
} while(freq != 'A' && freq != 'S' && freq != 'Q' && freq != 'M'
&& freq != 'D' && freq != 'C');
```

```
/* Hagamos los cálculos */
if (freq == 'C')
                  tabla(md3, a, m, n); /* composición continua */
            else if (freq == 'D')
                  tabla(md2, a, m, n); /* composición diaria */
            else
                  tabla(md1, a, m, n); /* otros casos */
}

/* generador de "tabla". Esta función acepta un puntero a otra
función tal como pide el enunciado del problema */
void tabla(double(*pf)(double i, int m, double n), double a,
int m, double n) {
      int cont; /* Contador del bucle */
      double i; /* tasa de interés */
      double f; /* valor futuro */
      printf("\n Interés      Cantidad futura\n\n");
      for(cont = 1; cont <=20; ++cont){
            i = 0.01 * cont
           f = a * (*pf)(i,m,n); /* LA FUNCIÓN PASA COMO UN PUNTERO */
            printf("      %2d            %.2f\n", cont, f);
      }
      return;
}

/* Depósitos mensuales, composición periódica */
double md1(double i, int m, double n){
      double factor, rao;
      factor = 1 + i/m;
      rao = 12 * (pow(factor, m*n)-1)/y;
      return(rao);
}

/* Depósitos mensuales, composición diaria */
double md2(double i, int m, double n){
      double factor, rao;
      factor = 1 + i/m;
      rao = (pow(factor, m*n)-1)/(pow(factor, m/12)- 1);
      return(rao);
}

/* Depósitos mensuales, composición continua */
double md3(double i, int nulo, double n){
      double rao;
      rao = (exp(i*n)-1)/(exp(i/12)-1);
      return(rao);
}
```

2. Presentamos una posible realización. Seguiremos el esquema de solución propuesto en el apartado 3.3, "Máquinas de estado".

1) Necesitaremos cinco estados, correspondientes a las diferentes situaciones de encendido y apagado de las luces del semáforo de cada calle:

- *AVVerde:* AV abierto, CLL cortado.
- *AVNaranja:* AV a punto de ser cortado, CLL cortado.
- *Corte:* AV y CLL cortados.
- *CLLVerde:* CLL abierto y AVG cortado.
- *CLLNaranja:* CLL a punto de ser cortado, AV cortado.

2) Necesitaremos una única variable:

- *Timer* = valor de temporizador que reiniciaremos al final de cada estado.
- *Prev* = variable booleana que almacena en qué calle se había permitido el paso anteriormente (1 = AV; 0 = CLL).

3) Tabla de transiciones:

Transición	Condición
AVGVerde → AVGNaranja	*timer* > 60 s
AVGNaranja → Corte	*timer* > 2 s
Corte → CLLVerde	*timer* > 3 s .AND. prev == 0
Corte → AVGVerde	*timer* > 3 s .AND. prev == 1
CLLVerde → CLLNaranja	*timer* > 30 s
CLLNaranja → Corte	*timer* > 2 s

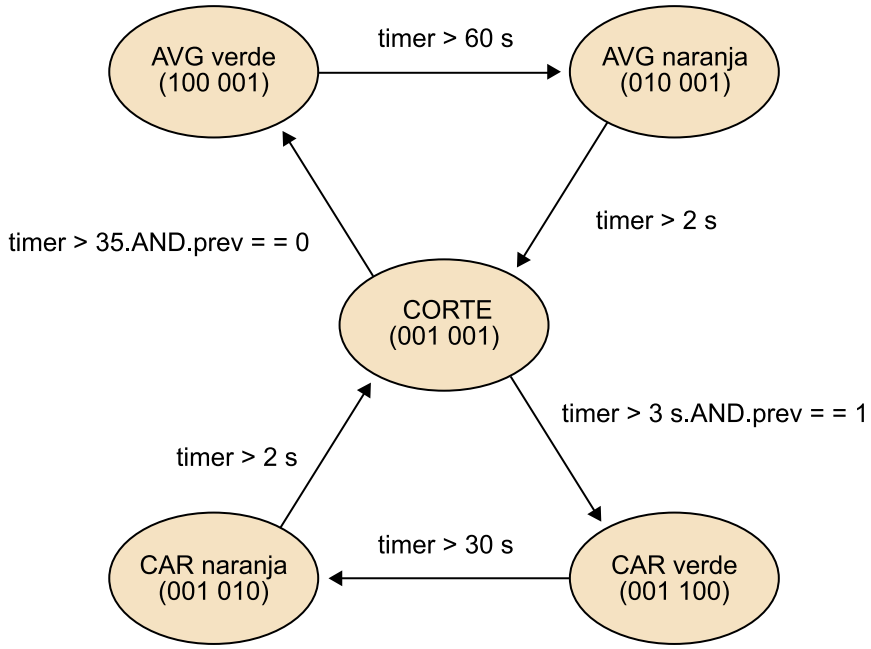
4) La tarea que hay que llevar a cabo en cada uno de los estados se reduce a encender o apagar las diferentes luces del semáforo; por lo que:

- *AVGg* = luz verde del semáforo de AVG.
- *AVGo* = luz naranja del semáforo de AVG.
- *AVGr* = luz roja del semáforo de AVG.
- *CLLg* = luz verde del semáforo de CLL.
- *CLLo* = luz naranja del semáforo de CLL.
- *CLLr* = luz roja del semáforo de CLL.

Por lo tanto, en cada uno de los estados hay que establecer las combinaciones siguientes:

Estado	AVg	AVo	AVr	CLLg	CLLo	CLLr
AVVerde	1	0	0	0	0	1
AVNaranja	0	1	0	0	0	1
Corte	0	0	1	0	0	1
CLLVerde	0	0	1	1	0	0
CLLNaranja	0	0	1	0	1	0

Con todo esto, el diagrama de estados resultante es:



Para los códigos de cada estado, usamos AVg, AV o AVr, o CLLg, CLL o CLLr

Y el código fuente:

```
#define CORTE 0
#define AVVerde 1
#define AVNaranja 2
#define CLLVerde 3
#define CLLNaranja 4


void UnitControl() {
      int state = CORTE;
      int timer = 0;
      boolean prev = 0;
      while (1) {
      switch (state) {
      CORTE: AVg=0;AVo=0;AVr=1; CLLg=0;CLLo=0;CLLr=1;
            if  (timer>3) {
                        if (prev==0) {
                              state = AVVerde;
                              timereset();
                        } else if (prev==1) {
                              state = CLLVerde;
                              timereset();
                        }
                  }
            break;

      AVVerde: AVg=1;AVo=0;AVr=0; CLLg=0;CLLo=0;CLLr=1;
            prev = 1;
                  if  (timer>60) {
                        state = AVNaranja;
                        timereset();
                  }
            break;

      AVNaranja:AVg=0;AVo=1;AVr=0;CLLg=0;CLLo=0;CLLr=1;
            if  (timer>2) {
                        state = CORTE;
                        timereset();
                  }
            break;

      CLLVerde: AVg=0;AVo=0;AVr=1; CLLg=1;CLLo=0;CLLr=0;
            prev = 0;
                  if  (timer>30) {
                  state = CLLNaranja;
                  timereset();
                  }
            break;

      CLLNaranja:AVg=0;AVo=0;AVr=1;CLLg=0;CLLo=1;CLLr=0;
            if  (timer>2) {
                        state = CORTE;
                        timereset();//reiniciamos contador tiempo
                  }
            break;
            }
            timer = currenttime(); //cargamos tiempo actual
        }
}
```

3. Observamos que se nos pide que añadamos una serie de acciones de control del semáforo de peatones que se llevan a cabo de manera totalmente síncrona (por ejemplo, simultánea) con los cambios de estado de los semáforos reguladores de vehículos. Por este motivo, no hay que añadir ningún estado más a la máquina; simplemente, hay que prever algunas acciones más:

- *AVvianG* = luz verde del semáforo de peatones.
- *AVvianInt* = luz verde intermitente del semáforo de peatones.
- *AVvianR* = luz roja del semáforo de peatones.

Con todo esto, la tabla de tareas se tiene que modificar de la manera siguiente:

Estado	AVg	AVo	AVr	CLLg	CLLo	CLLr	AV-vianG	AVvia-nInt	AVvianR
AVVerde	1	0	0	0	0	1	**0**	**0**	**1**
AVNa-ranja	0	1	0	0	0	1	**0**	**0**	**1**
Corte	0	0	1	0	0	1	**0**	**0**	**1**
CLLVerde	0	0	1	1	0	0	**1**	**0**	**0**
CLLNaran-ja	0	0	1	0	1	0	**0**	**1**	**0**

4. Presentamos una posible realización. Partiremos del diseño de la máquina de estados de la actividad 2 (modo NORMAL), a la que habrá que añadir un nuevo modo (ALARMA) de funcionamiento que se inicia al activarse la alarma de la estación de bomberos. Nuevamente, seguiremos el esquema de solución propuesto en el subapartado 3.3, máquinas de estado.

1) Necesitaremos dos estados nuevos:

- *BOMNaranja*: detención de los vehículos que circulan, tanto por AV como por CLL de manera segura; es decir, hay que poner todos los semáforos en naranja durante 2 segundos.
- *BOMCorte*: AV y CLL cortados, de manera que permiten el paso de los bomberos.

2) Necesitaremos tres variables más, que se modificarán mediante las rutinas ISR correspondientes a la interrupción proveniente de cada uno de los periféricos.

- *balizaAV* = variable booleana que recoge el paso del vehículo de bomberos por la baliza de AV.
- *balizaCLL* = variable booleana que recoge el paso del vehículo de bomberos por la baliza de CLL.

3) Tabla de transiciones:

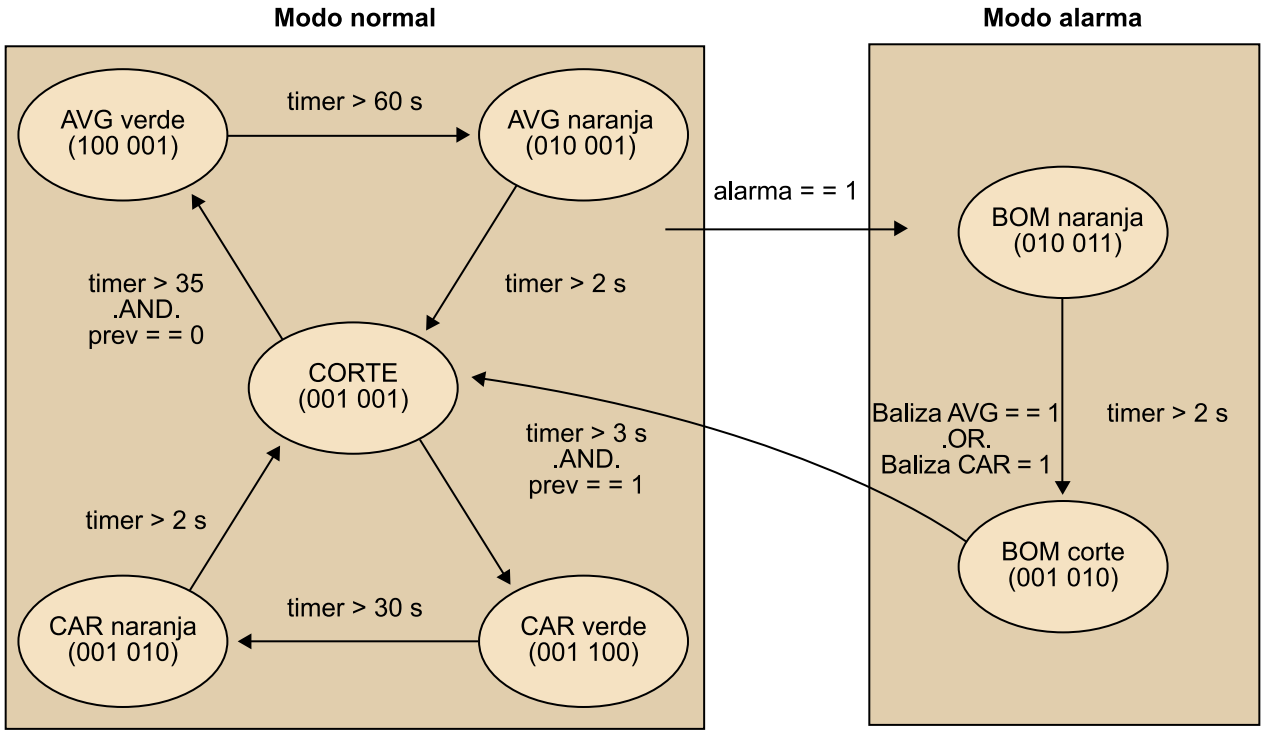
Transición	Condición
NORMAL → ALARMA	alarm == 1
BOMNaranja → BOMCorte	*timer* > 2 s
BOMCorte → NORMAL(AVVerde)	balizaAV == 1 .OR. balizaCLL ==1

4) En la tabla de tareas, solo hay que añadir las acciones correspondientes a los estados nuevos.

Modo	Estado	AVg	AVo	AVr	CLLg	CLLo	CLLr
NORMAL	*AVVerde*	1	0	0	0	0	1
	AVNaranja	0	1	0	0	0	1
	Corte	0	0	1	0	0	1
	CLLVerde	0	0	1	1	0	0
	CLLNaranja	0	0	1	0	1	0
ALARMA	*BOMNaranja*	0	1	0	0	1	0

Modo	Estado	AVg	AVo	AVr	CLLg	CLLo	CLLr
	BOMCorte	0	0	1	0	0	1

Con todo esto, el diagrama de estados resultante es:



Para los códigos de cada estado, usamos AVg, AV o AVr, o CLLg, CLL o CLLr

Y el código fuente:

```
#define NORMAL 0
#define ALARM 1

#define CORTE 0
#define AVVerde 1
#define AVNaranja 2
#define CLLVerde 3
#define CLLNaranja 4

#define BOMNaranja 0
#define BOMCorte 1

global boolean alarm = 0;
global boolean balizaAV = 0;
global boolean balizaCLL = 0;


void UnitControl() {
      int modo = NORMAL;
      int stateNORMAL = CORTE;
      int stateALARM = BOMNaranja;

      int timer = 0;
      boolean prev = 0;

      alarm = 0;
      balizaAV = 0;
      balizaCLL = 0;
```

```
      while (1) {
   switch (modo) {
      NORMAL:
            switch (stateNORMAL) {
      CORTE: AVg=0;AVo=0;AVr=1; CLLg=0;CLLo=0;CLLr=1;
            if  (timer>3) {
                        if (prev==0) {
                              stateNORMAL = AVVerde;
                              timereset();
                        } else if (prev==1) {
                              stateNORMAL = CLLVerde;
                              timereset();
                        }
                  }
            break;

      AVVerde: AVg=1;AVo=0;AVr=0; CLLg=0;CLLo=0;CLLr=1;
            prev = 1;
                  if  (timer>60) {
                        stateNORMAL = AVNaranja;
                        timereset();
                  }
            break;

      AVNaranja:AVg=0;AVo=1;AVr=0;CLLg=0;CLLo=0;CLLr=1;
            if  (timer>2) {
                        stateNORMAL = CORTE;
                        timereset();
            }
            break;

      CLLVerde: AVg=0;AVo=0;AVr=1; CLLg=1;CLLo=0;CLLr=0;
            prev = 0;
                  if  (timer>30) {
                        stateNORMAL = CLLNaranja;
                        timereset();
}
            break;

      CLLNaranja:AVg=0;AVo=0;AVr=1;CLLg=0;CLLo=1;CLLr=0;
            if  (timer>2) {
                        stateNORMAL = CORTE;
                        timereset();//reiniciamos contador tiempo
                  }
            break;
            }
            break;

      ALARM:
            switch (stateALARM) {
      BOMNaranja:AVg=0;AVo=1;AVr=0;CLLg=0;CLLo=1;CLLr=0;
            if  (timer>2) {
                        stateALARM = BOMCorte;
                        timereset();
                  }
            break;

      BOMCorte: AVg=0;AVo=0;AVr=1; CLLg=0;CLLo=0;CLLr=1;
            if  (balizaAV == 1) {
                        modo = NORMAL;
                        stateNORMAL = CORTE;
                        timer = 0;
                        prev = 0;
                        alarm = 0;
                        balizaAV = 0;
                        balizaCLL = 0;
                  } else if {balizaCLL == 1) {
                        modo = NORMAL;
                        stateNORMAL = CORTE;
                        timer = 0;
                        prev = 0;
```

```
                        alarm = 0;
                        balizaAV = 0;
                        balizaCLL = 0;
                  }
            break;
            }
            break;

      timer = currenttime(); //cargamos tiempo actual
      if (alarm == 1) {
            modo = ALARM
            stateALARM = BOMNaranja
      }
}

void ISRAlarm() {     // ISR de atención a la activación de la alarma
      alarm = 1;
}

void ISRbalizaAV() {  // ISR de atención a baliza en AV
      balizaAV = 1;
}

void ISRbalizaCLL() {  // ISR de atención a baliza en CLL
      balizaCLL = 1;
}
```

5. Presentamos una posible elaboración del contestador automático. Seguiremos el esquema de solución propuesto en el apartado 4.c, "Máquinas de estado".

1) Necesitaremos dos modos: CONTESTADOR, REPRODUCTOR con los estados siguientes:

MODO CONTESTADOR:

- *Wait*: espera la llegada de un tono de llamada.
- *Ring1*: se ha recibido un tono de llamada.
- *Ring2*: se han recibido dos tonos de llamada consecutivos.
- *Messg*: se han recibido tres tonos de llamada consecutivos y se reproduce el mensaje de bienvenida.
- *Beep*: se hace sonar un pitido durante 1 segundo.
- *Record*: se activa la grabación del mensaje.
- *IncCount*: se para la grabación y se incrementa el contador de mensajes.

MODO REPRODUCTOR:

- *Check*: se verifica el número de mensajes grabados.
- *Play*: se reproducen todos los mensajes grabados.

2) Necesitaremos hasta seis variables:

- *timer* = valor de temporizador que reiniciaremos cuando sea necesario.
- *msgcnt* = contador de mensajes grabados.

Las variables siguientes se modifican mediante la ISR de atención a periférico correspondiente.

- *tono* = contador del número de tonos que han sonado.
- *cuelga* = variable booleana que indica si el interlocutor ha colgado.
- *pulsador* = variable booleana que indica si se ha pulsado el botón de reproducción de mensajes.

3) Tabla de transiciones:

Transición	Condición
Wait → *Ring1*	tono = = 1
Ring1 → *Ring2*	tono = = 2
Ring2 → *Messg*	tono = = 3

Transición	Condición
Ring1 → Wait	cuelga = = 1
Ring 2 → Wait	cuelga = = 1
Messg → Wait	cuelga = = 1
Beep → Wait	cuelga = = 1
Messg → Beep	*timer* > 7 s
Beep → Record	*timer* > 1 s
Record → IncCount	*timer* > 60 s .OR. cuelga = = 1
IncCount → Wait	-
CONTESTADOR → REPRODUCTOR (Check)	Pulsador = = 1
Check → Play	msgcnt > 0
Play → Play	msgcnt > 0
REPRODUCTOR → CONTESTADOR (Wait)	msgcnt = = 0

4) La tarea que hay que llevar a cabo en cada uno de los estados se reduce a activar o desactivar los componentes siguientes:

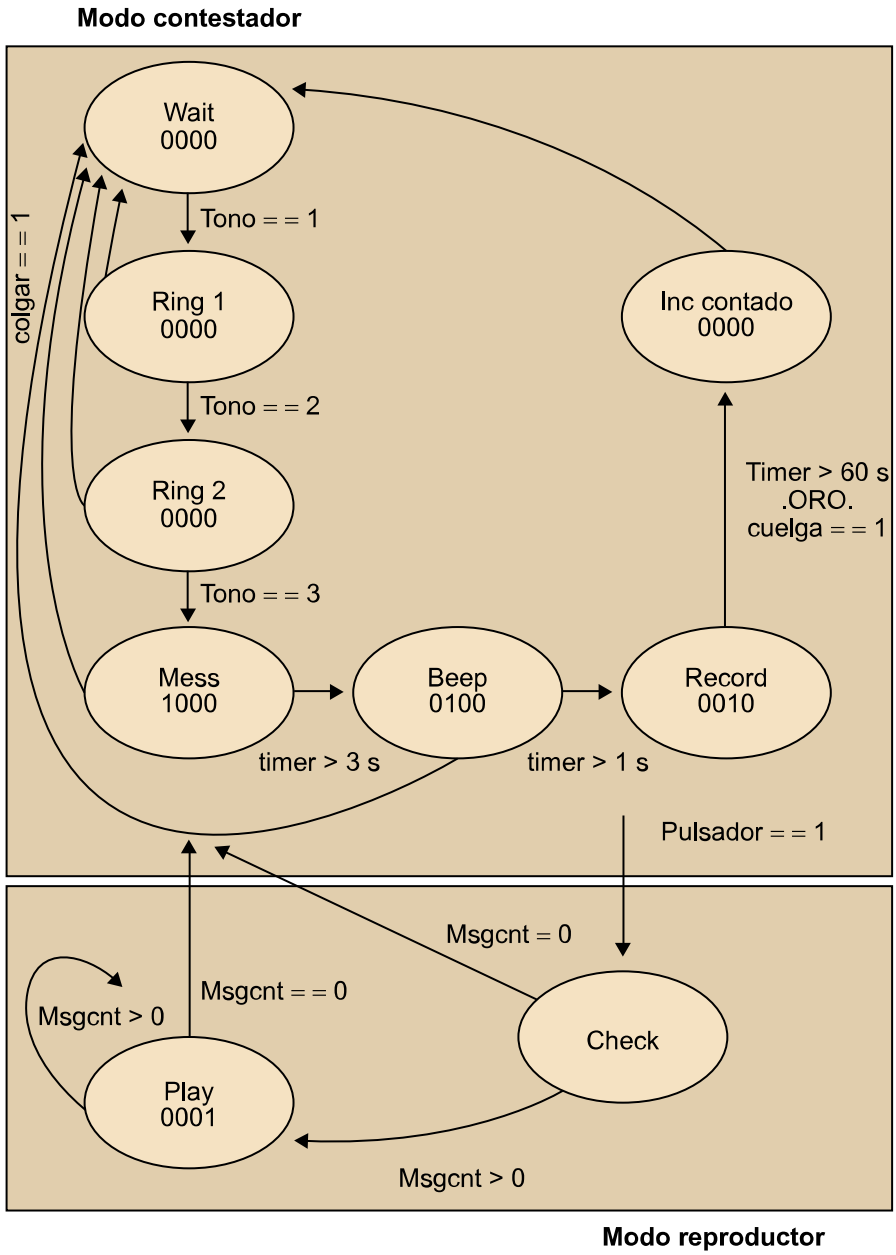
- *m* = reproductor de mensaje de bienvenida.
- *b* = generador del pitido.
- *r* = grabadora de mensajes.
- *p* = reproductor de mensaje.

Por lo tanto, en cada uno de los estados hay que establecer las combinaciones siguientes:

Estado	m	b	r	p
Wait	0	0	0	0
Ring1	0	0	0	0
Ring2	0	0	0	0
Messg	1	0	0	0
Beep	0	1	0	0
Record	0	0	1	0
IncCount	0	0	0	0
Check	0	0	0	0
Play	0	0	0	1

Observamos que muchos de los estados son diferentes solo lógicamente, es decir, se llevan a cabo acciones diferentes dentro de la memoria del sistema empotrado pero no corresponden a ningún cambio en el estado de los periféricos m, b, r y p.

Con todo esto, el diagrama de estados resultante es:



Y el código fuente:

```
#define CONTESTADOR 0
#define REPRODUCTOR 1

#define Wait 0
#define Ring1 1
#define Ring2 2
#define Messg 3
#define Beep 4
#define Record 5
#define IncCount 6

#define Check 0
#define Play 1

global int tono = 0;
global boolean cuelga = 0;
global boolean pulsador = 0;


void UnitControl() {
```

```
        int modo = CONTESTADOR;
        int stateCONTESTADOR = Wait;
        int stateREPRODUCTOR = Check;

        int timer = 0;
        int msgcnt = 0;

        tono = 0;
        cuelga = 0;
        pulsador = 0;

        while (1) {
     switch (modo) {
        CONTESTADOR:
              switch (stateCONTESTADOR) {
        Wait: m=0; b=0; r=0; p=0;
                    cuelga == 0
                    if  (tono == 1) {
                          stateCONTESTADOR = Ring1;
                    }
                    break;

        Ring1: m=0; b=0; r=0; p=0;
                    if (cuelga == 1) {
                          stateCONTESTADOR = Wait;
                    else if  (tono == 2) {
                          stateCONTESTADOR = Ring1;
                    }
                    break;

              Ring2: m=0; b=0; r=0; p=0;
              if (cuelga == 1) {
                          stateCONTESTADOR = Wait;
                    else if  (tono == 3) {
                          stateCONTESTADOR = Messg;
                          timereset();
                    }
                    break;

              Messg: m=1; b=0; r=0; p=0;
              tono = 0;
                    if (cuelga == 1) {
                          stateCONTESTADOR = Wait;
                    else if  (timer > 7) {
                          stateCONTESTADOR = Beep;
                          timereset();
                    }
                    break;

              Beep: m=0; b=1; r=0; p=0;
              if (cuelga == 1) {
                          stateCONTESTADOR = Wait;
                    else if  (timer > 1) {
                          stateCONTESTADOR = Record;
                          timereset();
                    }
                    break;

              Record: m=0; b=0; r=1; p=0;
              if (cuelga == 1) {
                          stateCONTESTADOR = IncCount;
                    else if  (timer > 60) {
                          stateCONTESTADOR = IncCount;
                    }
                    break;

              IncCount: m=0; b=0; r=0; p=0;
                    msgcnt = msgcnt + 1;
                    stateCONTESTADOR = Wait;
                    break;
              }
              break;
```

```
      REPRODUCTOR:
            switch (stateREPRODUCTOR) {
            Check: m=0; b=0; r=0; p=0;
                  pulsador = 0;
                  if (msgcnt == 0) {
                        modo == CONTESTADOR;
                        stateCONTESTADOR = Wait;
                        tono = 0;
            else if  (msgcnt > 0) {
                  stateREPRODUCTOR = Play;
            }
            break;

            Check: m=0; b=0; r=0; p=1;
            if (msgcnt == 0) {
                        modo == CONTESTADOR;
                        stateCONTESTADOR = Wait;
                        pulsador = 0;
                        tono = 0;
                  else if  (msgcnt > 0) {
                        playmeassage();   // acciones que permiten
                                          //reproducir un mensaje
                        msgcnt = msgcnt - 1;
                  }
                  break;
            }
            break;
      }

      timer = currenttime(); //cargamos tiempo actual
      if (pulsador == 1) {
            modo = REPRODUCTOR
            stateREPRODUCTOR = Check
      }

}

void ISRTono() {          // ISR de atención a la recepción de un tono
      tono = tono + 1;
}

void ISRCuelga() {         // ISR de atención al cierre de línea
      cuelga = 1;
}

void ISRPulsador() {      // ISR de atención a la pulsación
      pulsador = 1;
}
```

6. a) Cualquier señal que recibe un procesador para redirigir el programa en ejecución en un momento determinado y pasar a ejecutar un código específico.

b) Una ISR o rutina de interrupción de servicio es cualquier rutina que se ejecuta como consecuencia de la generación de una interrupción y que satisface tres requisitos:

- Gestiona la interrupción que la ha generado.
- Permite al procesador aceptar nuevas interrupciones durante su ejecución.
- Devuelve el sistema al estado previo a la generación de la interrupción asociada.

c)

- Archivo del contador del programa principal en ejecución en la pila del procesador.
- Generación de un acuse de recibo (*interrupt acknowledge*) por el procesador.
- Ejecución de la ISR asociada.
- Recuperación de la dirección de retorno y otra información archivada en la pila a la finalización de la ISR.
- Continuación de la ejecución del programa principal.

d)

- Reducción de la utilización total de recursos del microprocesador, puesto que desaparece la necesidad de comunicarse con los periféricos de manera recursiva.
- Eliminación de los estados de latencia de los periféricos.
- Trabajo con códigos de software más ordenado y compartimentado.

e) Un vector de interrupción es una dirección de memoria que indica al procesador dónde buscar la ISR asociada a una interrupción determinada.

f)

- Elaborar una lista de todos los estados posibles.
- Declarar todas las variables.
- Por cada estado, hacer una lista de las posibles transiciones a otros estados, identificando las condiciones.
- Por cada estado y/o transición, hacer una lista de todas las acciones necesarias (tareas, cambios de variables, etc.).
- Asegurarse de que, por cada estado, las condiciones de transición son mutuamente excluyentes y completas.

g)

- Pérdida de rendimiento por exceso de planificación vinculante.
- Pilas múltiples.
- Restitución de contexto.
- Condiciones de carrera.
- Latencia de interrupción.
- Acceso múltiple a bibliotecas.
- Mútex.
- Inanición.
- Punto muerto (*deadlock*).
- Inversión de prioridad.

Ejercicios de autoevaluación

1. Falso. Por regla general, los vectores de interrupción efectivamente no contienen código ejecutable, pero en algunos modelos antiguos de procesadores había algunos que sí.

2. Falso. No hay ningún requisito que obligue a programar las ISR de una interrupción en lenguaje de bajo nivel. No obstante, es cierto que estos tipos de lenguaje permiten controlar mejor su tiempo de ejecución a pesar de los inconvenientes a la hora de escribir el código.

3. Verdadero. Los vectores de interrupción se pueden generar desde un controlador externo, uno interno o desde los propios periféricos.

4. Falso. Esto dependerá del procesador que se utilice. Además, la interrupción NMI siempre tiene prioridad máxima.

5. Verdadero. Si la velocidad de ejecución no es prioritaria y se busca reducir el tamaño del código que controla un sistema empotrado, el uso de funciones *inline* puede ser una buena solución.

6. Falso. La utilización de *inlining* deviene compleja e incluso los programadores experimentados deben recurrir a heurísticas para decidir si usan o no esta técnica. A efectos prácticos, los compiladores deciden mejor que los programadores cuándo lo tienen que utilizar.

7. Verdadero. La posibilidad de desarrollar código en diferentes archivos que abre esta técnica permite distribuir el trabajo a diferentes programadores. Además, cualquier cambio correctivo o de actualización solo afectará a una parte del código, lo que simplifica estos pasos.

8. Falso. Las máquinas de estado son adecuadas para implementar sistemas encargados de controlar diferentes elementos extensos con estados de funcionamiento muy definidos. Este hecho las hace adecuadas para llevar a cabo tareas bastante específicas en cada uno de estos estados. Por lo tanto, no resultan adecuadas en entornos multitarea en los que haya que atender a diferentes necesidades de una manera concurrente. Por practicidad (hay que definir, entre otros, todos los estados de la máquina), no son convenientes en sistemas de extrema complejidad. En estos casos hay que recurrir a RTOS.

9. Falso. Si bien la planificación multitarea permite atender múltiples necesidades de una manera concurrente, las soluciones implementadas para emular este funcionamiento (por ejemplo, la planificación prioritaria) introducen una cierta impredictibilidad en el funcionamiento del sistema. En este sentido, los planificadores secuenciales son totalmente predecibles.

Glosario

address bus *m* Véase **bus de direcciones**.

dirección física *f* Datos que se envían mediante el bus de direcciones para indicar la posición de memoria a la que se quiere acceder.
en physical address

argumentos por referencia *m pl* Modo de trabajar en programación que se basa en pasar como datos punteros a una función.

argumentos por valor *m pl* Modo de trabajar en programación que se basa en pasar datos a una función, que son copiados dentro de estas. Su modificación no afecta al valor de los datos fuera de la función.

bloque de memoria *m* Porción de memoria que se reserva de manera íntegra y se asigna para una tarea concreta. Permite la asignación dinámica de la cantidad de memoria que necesita cada tarea en cada momento en entornos multitarea.
en memory block

bus de direcciones *m* Conjunto de líneas de comunicaciones que conectan un procesador con cualquier elemento de memoria con el fin de permitir seleccionar alguna de sus posiciones de memoria. Típicamente, si el bus está formado por n líneas, permite localizar hasta 2^n direcciones individuales. Estrategias avanzadas de gestión de memoria como la paginación permiten extender la cantidad de posiciones que hay que controlar con n líneas de bus de direcciones.
en address bus

central processing unit *f* Véase **unidad central de procesamiento**.

check *f* Véase **suma de comprobación**.

comunicación entre tareas *f* Paso de datos entre las tareas de un sistema multitarea. La mayoría de los RTOS incluyen multitud de recursos (buzones de correo, colas, etc.) para transferir datos entre tareas con seguridad.
en intertask communication

condición de carrera *f* Situación en la que los efectos combinados de dos o más tareas concurrentes varían según el orden preciso en el que se ha ejecutado la secuencia de instrucciones combinadas de una y de otra. Pueden ser evitadas mediante los mútexs y la identificación de secciones críticas.
en race condition

contexto *m* Datos contenidos en los diferentes registros y pilas del sistema en el momento de la interrupción y que, por lo tanto, hay que restituir para poder continuar trabajando en iguales condiciones. El contexto incluye, entre otros aspectos, la información relativa a la siguiente instrucción pendiente de ejecución (puntero de instrucciones) y todos los datos asociados a la tarea (puntero de pila, espacio de memoria asignado, registros de datos, etc.).

CPU *f* Véase **unidad central de procesamiento**.

critical section *f* Véase **sección crítica**.

deadline *m* Véase **plazo**.

digital signal processor *m* Véase **procesador de señales digitales**.

DSP *m* Véase **procesador de señales digitales**.

exclusión mutua *f* Garantía de acceso exclusivo a un recurso compartido. En los sistemas integrados, el recurso compartido es típicamente un bloque de memoria, una variable global, un periférico o un conjunto de registros. La exclusión mutua se consigue en general con el uso de un mútex.
en mutual exclusion

firmware *m* Véase **microsoftware**.

función externa *f* Función definida en un archivo diferente de donde está definido el código principal del programa.

función *inline* *f* sin. ***inlining***

inlining *m* Estrategia de programación destinada a optimizar el funcionamiento de programas en los que hay muchas llamadas a funciones. Cada vez que una función es llamada (función *inline*), el compilador introduce su código completo donde ha sido llamada en vez de ir a buscarla a la posición de memoria en la que ha sido definida inicialmente.
sin. **función *inline***

interrupción *f* Señal eléctrica asíncrona enviada por un periférico al procesador. Cada vez que se envía este tipo de señal (se dice que se ha producido una interrupción), se interrumpe la ejecución normal de la tarea que esté activa en el procesador en aquel momento. Cuando esto sucede, se guarda el contexto actual (puntero de instrucción, estado de los registros, etc.) y se ejecuta una rutina de servicio de interrupción. Cuando finaliza la rutina de servicio de interrupción, el procesador continúa con la ejecución normal de la tarea que había quedado interrumpida.
en interrupt

interrupción de software *f* Interrupción originada por un elemento de software. Habitualmente se utilizan para aplicar los puntos de interrupción y puntos de entrada del sistema operativo. A diferencia de las interrupciones convencionales (originadas en el hardware), estas se producen de manera síncrona con la ejecución del software; es decir, se producen a comienzos del ciclo de ejecución de una instrucción.
en software interrupt

interrupt *f* Véase **interrupción**.

interrupt latency *f* Véase **latencia de interrupción**.

interrupt service process *m* Véase **proceso de servicio de interrupción**.

interrupt service routine *f* Véase **rutina de servicio de interrupción**.

interrupt type *m* Véase **tipo de interrupción**.

interrupt vector *m* Véase **vector de interrupción**.

interrupt vector table *f* Véase **tabla de vectores de interrupción**.

intertask communication *f* Véase **comunicación entre tareas**.

intertask synchronization *f* Véase **sincronización entre tareas**.

inversión de prioridades *f* Situación de violación de prioridad inducida por la existencia de un mútex compartido entre tareas de prioridad muy diferente. En estas circunstancias, la tarea de otra prioridad puede ser avanzada por una tercera de prioridad intermedia si esta entra a ejecutarse en el momento en el que el mútex está bloqueado por la de baja prioridad.
en priority inversion

ISP *m* Véase **proceso de servicio de interrupción**.

ISR *f* Véase **rutina de servicio de interrupción**.

kernel *m* Véase **núcleo**.

latencia de interrupción *f* Cantidad de tiempo entre el envío de una señal de interrupción y el inicio de la rutina de servicio de interrupción asociada. La velocidad del procesador y el algoritmo de planificación empleado en el sistema son algunos de los factores que afectan a este retardo.
en interrupt latency

lista de ejecución *f* En entornos multitarea, estructura de datos mantenida por el sistema (por ejemplo, el núcleo de un RTOS) y que recoge todas las tareas listas para la ejecución junto con su prioridad. De acuerdo con esta información y con los recursos disponibles en el sistema en cada instante, el planificador gestiona el modo como se deben ejecutar.
en task list

memory block *m* Véase **bloque de memoria**.

microcontrolador *m* Microprocesador altamente integrado diseñado específicamente para su uso en sistemas empotrados. Los microcontroladores suelen incluir un procesador integrado, memoria (una pequeña cantidad de memoria RAM, ROM o ambas) y otros periféricos

en el mismo chip. Los ejemplos más comunes son PIC de Microchip, el 8051, 80196 de Intel y una serie de Motorola 68HCxx.
en microcontroller

microcontroller *m* Véase **microcontrolador**.

microsoftware *m* Conjunto de instrucciones que gobiernan el funcionamiento de los sistemas empotrados. Se trata de software elaborado para su ejecución en un hardware específico y que a menudo se encuentra almacenando en memorias tipo flash o ROM.
en firmware

multitarea *f* Ejecución de rutinas de múltiples rutinas de software de manera pseudosimultánea o pseudoparalela. El planificador del sistema operativo es el responsable de distribuir el tiempo de procesador entre las diferentes rutinas/tareas (que en realidad se ejecutan de manera individual y consecutiva/secuencial), de tal manera que se simula el procesamiento simultáneo/paralelo.
en multitasking

multitasking *f* Véase **multitarea**.

mútex *m* Acortamiento de *mutual exclusion*, se trata de un indicador binario que se puede utilizar para sincronizar las actividades de varias tareas en entornos multitarea. Como tal, puede proteger secciones críticas de interrupciones indeseadas y recursos compartidos de accesos simultáneos.

mutual exclusion *f* Véase **exclusión mutua**.

núcleo *m* Parte esencial de cualquier RTOS, formada por el planificador de tareas y las rutinas de cambio de contexto.
en kernel

operative system *m* Véase **sistema operativo**.
sigla **OS**

OS *m* Véase ***operative system***.

periférico *m* Elemento de hardware (excluyendo el procesador) que hace funciones de entrada y salida de información. Uno o varios periféricos se pueden encontrar integrados en un mismo chip (por ejemplo, en el procesador); en estos casos, se denominan *periféricos integrados en chip*.
en peripheral

peripheral *m* Véase **periférico**.

physical address *f* Véase **dirección física**.

pila *f* Conceptualmente, se trata de una lista abierta que permite la adición y eliminación de elementos de manera que el último en entrar (ser escrito) siempre es el primero accesible para salir (ser leído). En los sistemas informáticos, estas estructuras de datos permiten almacenar información que se debe recorrer de manera secuencial, como, por ejemplo, una serie de instrucciones en código máquina. Su funcionamiento particular la hace especialmente adecuada para llevar a cabo el seguimiento de tareas muy jerarquizadas. Por ejemplo, permite añadir instrucciones de atención a interrupción de modo que, una vez ejecutadas, el programa continúa exactamente en el punto en el que había quedado interrumpido. Nota: en este contexto, se entiende que la lectura de un dato de la cima de la pila implica su eliminación.
en stack

planificación prioritaria *f* Método de planificación de tareas que se basa en suspender la ejecución de una tarea cuando otra de más prioridad pasa a estar lista (o a una tarea de la misma prioridad se le concede un intervalo de ejecución). Son más complejas de implementar que los planificadores secuenciales, permiten multitarea, pero resultan más adecuadas para responder a eventos externos al sistema.
en preemptive scheduling

planificación secuencial *f* Método de planificación de tareas en el que todas las tareas se ejecutan de manera consecutiva, siguiendo estrictamente el orden con el que han estado listas para su ejecución. En principio, no tienen en cuenta la prioridad de las tareas. Son simples de implementar, no permiten multitarea real, ni resultan adecuadas para aplicaciones complejas en tiempo real.
en sequential scheduling

planificador *m* Elemento de software integrado en el núcleo del sistema operativo responsable de decidir qué tarea tiene que utilizar el procesador en un momento determinado. Diferentes metodologías de planificación tienen en cuenta diferentes factores, como el número y prioridad de las tareas, la presencia de recursos para compartir, etc.
en scheduller

pointer *m* Véase **puntero**, **puntero de instrucción**.

preemptive scheduling *f* Véase **planificación prioritaria**.

prioridad *f* Urgencia relativa de una tarea o interrupción en comparación con otra. En el caso de las tareas, la prioridad es un número entero y, en función de la prioridad del resto de las tareas concurrentes, determina el tiempo y el turno de proceso que le asignará el planificador para su ejecución.
en priority

priority *f* Véase **prioridad**.

priority inversion *f* Véase **inversión de prioridades**.

proceso de servicio de interrupción *m* sin. **rutina de servicio de interrupción**
en interrupt service process
sigla **ISP**

procesador de señales digitales *m* Dispositivo similar a un microprocesador, que ha sido dotado de una CPU interna optimizada para su uso en aplicaciones que involucran el procesamiento de señales en tiempo discreto. Además de las instrucciones del microprocesador estándar, los DSP suelen dar soporte a una serie de instrucciones especiales, como las multiplicaciones y acumulaciones, para hacer cálculos comunes de procesamiento de señal más deprisa. Normalmente, se basan en una arquitectura Harvard, que mantiene separados los espacios de memoria que contienen los programas y los datos, puesto que permite una velocidad de transferencia de datos superior. Algunas familias habituales de DSP son la 320Cxx de Texas Instruments y la 5600x de Motorola.
en digital signal processor
sigla **DSP**

puntero *m* Variable que representa la posición (no el valor) de otro dato. Variable cuyo valor es una dirección de memoria.
en pointer

puntero de instrucción *m* Registro en un procesador que contiene la dirección de la instrucción que hay que ejecutar después.
en pointer

race condition *f* Véase **condición de carrera**.

real-time operative system *m* Véase **sistema operativo en tiempo real**.
sigla **RTOS**

register *m* Véase **registro**.

registro *m* Posición de memoria integrada en un procesador o un dispositivo de entrada/salida. En comparación con la memoria convencional, permiten velocidades de lectura/escritura muy superiores. Normalmente, la referencia a un registro u otro está codificada como parte de la instrucción de bajo nivel que ejecuta el procesador, y no como una dirección de memoria discreta.
en register

RTOS *m* Véase **sistema operativo en tiempo real**.

rutina de servicio de interrupción *f* Pequeño programa que se ejecuta en respuesta a una interrupción determinada. Es equivalente al término *gestor de interrupciones*.
en interrupt service rutine
sigla **ISR**

scheduller *m* Véase **planificador**.

sección crítica *f* Secuencia de instrucciones que se deben ejecutar en secuencia y sin interrupción para garantizar el funcionamiento correcto del programa. Si las instrucciones son interrumpidas, el funcionamiento del programa deja de ser predecible y, por lo tanto, incorrecto.

en critical section

semáforo *m* Estructura de datos que se utiliza para la sincronización entre tareas. Es una de las herramientas que proporciona el sistema operativo para la sincronización entre tareas y puede ser de dos tipos: binario y con contadores (por ejemplo, tienen más de dos estados).
en semaphore

semaphore *m* Véase **semáforo**.

sequential scheduling *f* Véase **planificación secuencial**.

sincronización entre tareas *f* Coordinación de la temporalización y secuencialización entre las diferentes tareas de un sistema multitarea. La mayoría de los RTOS incluyen multitud de recursos (semáforos, mútex, etc.) para sincronizar tareas de manera segura.
en intertask synchronization

sistema operativo *m* En el contexto de los sistemas empotrados, elemento de software que posibilita el funcionamiento en multitarea distribuyendo los recursos disponibles entre las diferentes tareas que hay que llevar a cabo. Un sistema operativo consiste típicamente en un conjunto de llamadas al sistema, un planificador de tareas, una rutina de servicio de interrupciones, un contador/temporizador de sistema y un conjunto de controladores de los dispositivos periféricos.
en operative system (OS)

sistema operativo en tiempo real *m* Sistema operativo diseñado específicamente para su uso en sistemas de tiempo real. Se utilizan en los sistemas en los que resulta imprescindible cumplir unas especificaciones temporales determinadas. Por ejemplo, garantizar unos tiempos máximos en la resolución de un cálculo o en la respuesta a un acontecimiento externo.
en real-time operative system
sigla **RTOS**

software interrupt *f* Véase **interrupción de software**.

stack *f* Véase **pila**.

suma de comprobación *f* Método de control de la integridad de los datos que permite detectar si han sido corrompidos. Una manera simple de hacerlo, y que, típicamente, añade pocos datos redundantes al mensaje, consistiría en sumar cada uno de los componentes básicos de un sistema (generalmente cada byte) y almacenar el valor del resultado. Posteriormente, se seguiría el mismo procedimiento y se compararía el resultado con el valor almacenado. Si ambas sumas concuerdan, se asumirá que los datos probablemente no han sido dañados.
en check

tarea *f* Cualquier cálculo individual, conjunto de cálculos, la lógica de la toma de decisiones, intercambio de información o combinación de ellas que tiene que llevar a cabo en tiempo de ejecución el software. Es la abstracción central de los RTOS. Cuando se utiliza un RTOS, cada tarea debe mantener su propia copia del puntero de instrucciones de la CPU y los registros de propósito general. La diferencia principal con los procesos es que las tareas comparten un espacio de memoria común. Por lo tanto, hay que tener cuidado para evitar sobrescribir código, datos y pilas de otras tareas.
en task

task *f* Véase **tarea**.

task list *f* Véase **lista de ejecución**.

tabla de vectores de interrupción *f* Lista que contiene el vector de interrupción asociado a cada tipo de interrupción. Ha de ser inicializado antes de que las interrupciones estén habilitadas.
en interrupt vector table

temporizador/contador *m* Periférico que cuenta eventos externos (modo contador) o ciclos de procesador (modo automático). Prácticamente, todos los microcontroladores tienen uno o más contadores integrados en su hardware.
en time/counter

plazo *m* En un sistema en tiempo real, el momento en el que un conjunto particular de los cálculos o las transferencias de datos deben ser completados. Se denominan *plazos duros* los que se han de cumplir forzosamente, puesto que son imprescindibles para el funcionamiento correcto del sistema. Se denominan *plazos blandos* los que no tienen impacto en el funcionamiento correcto del sistema pero sí que pueden afectar a las prestaciones.

en deadline

time/counter *m* Véase **temporizador/contador**.

time-slicing *m* Estrategia que permite simular el funcionamiento multitarea con una única unidad de proceso. Se basa en dividir el tiempo de ejecución en intervalos regulares (o porciones de tiempos) y durante este tiempo, dedicar el procesador y el resto de los recursos del sistema a la ejecución de una única tarea.

tipo de interrupción *m* Un número único asociado a cada interrupción. Cuando se produce una interrupción, el procesador utiliza este identificador para consultar la tabla de vectores de interrupción y decidir cómo hay que proceder.
en interrup type

unidad central de procesamiento *f* sin. **procesador**
en central processing unit
sigla **CPU**

vector de interrupción *m* Dirección de la posición de memoria en la que se inicia una rutina de servicio de interrupción determinada.
en interrup vector

Bibliografía

Beltrán de Heredia, J. (2001). *Lenguaje ensamblador de los 80x86* (13.ª ed.). Madrid: Anaya Multimedia.

Ganssle, J. G. (2000). *The Art of Designing Embedded Systems* (1.ª ed.). Woburn, Massachusetts: Newnes (Elsevier).

Gottfried, B. (1996). *Programación en C* (2.ª ed.). México, DF: McGraw-Hill / Interamericana de España.

Stuart R. B. (2002). *Embedded Microprocessor Systems: Real World Design*. (3.ª ed.). Woburn, Massachusetts: Newnes (Elsevier).

Marvedel, P. (2003). *Embedded System Design* (1.ª ed.). Dordrecht: Kluwer Academic Publishers.