ESCUELA DE PREPARACIÓN DE OPOSITORES

E. P. O.

C/. La Merced, 8 – Bajo A Telf.: 968 24 85 54

30001 MURCIA

INF28

Programación en tiempo real. Interrupciones. Sincronización y comunicación entre tareas. Lenguajes.

SAI30

Programación en tiempo real. Interrupciones. Sincronización y comunicación entre tareas.

Esquema.

1 Introducción.

Un sistema de tiempo de real es "cualquier actividad de proceso de información o sistema que tiene que responder a estímulos generados externamente dentro de un plazo especificado y finito". Consecuentemente, la correctitud de un sistema de tiempo real depende no sólo del resultado lógico de la computación, sino también del tiempo en el que este resultado tarda en generarse.

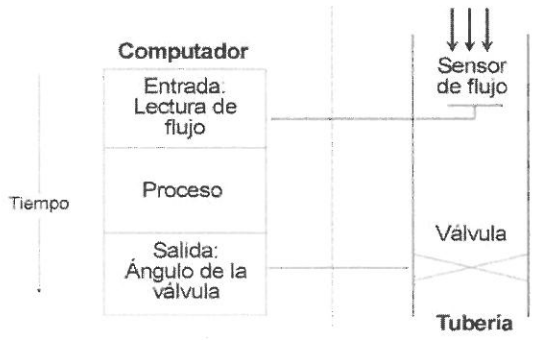
La computación de tiempo real no es equivalente a la computación rápida. El objetivo de una computación rápida es minimizar el tiempo de respuesta medio de un conjunto dado de tareas. En contraste, el objetivo de la computación de tiempo real es garantizar los requisitos temporales individuales de cada tarea. Más que ser rápido, que es un término relativo, la propiedad más importante de un sistema de tiempo real es la predecibilidad.

En los sistemas de tiempo real el computador actúa generalmente junto con dispositivos físicos. Está dedicado al control y la monitorización de estos equipos y forma parte de un sistema de ingeniería más amplio, bien un horno microondas, bien un sistema de guía de misiles, un automóvil o una central nuclear. Por esta razón, los sistemas de tiempo real se denominan asimismo sistemas empotrados.

Un sistema empotrado es un sistema informático cuyo propósito no es resolver ecuaciones matemáticas o proporcionar una base de datos, sino que forma parte de un sistema más grande (de ahí el nombre de empotrado), un sistema físico al que controla y adapta al entorno.

Las aplicaciones de tiempo real en el sector del automóvil y de la electrónica de consumo están creciendo muy rápidamente. Otros campos de aplicación de los sistemas de tiempo real son los siguientes: aviónica, control del tráfico aéreo, control de trenes, telecomunicaciones, sistemas de fabricación integrada, producción y distribución de energía eléctrica, control de edificios, y sistemas multimedia.

Un ejemplo típico de sistema de tiempo real es la aplicación de control de procesos mostrada en la siguiente figura.



En la figura, el computador realiza una única actividad, la de asegurar que por la tubería circule un flujo de líquido constante haciendo uso de una válvula. Mediante un sensor de flujo, el computador conoce las variaciones de éste y debe responder en tiempo real alterando el ángulo de la válvula de forma que la respuesta sea lo suficientemente rápida como para asegurar un flujo constante con independencia de las variaciones del entorno.

Los sistemas de tiempo real son más complejos que los sistemas convencionales, por lo que exigen requisitos adicionales a los lenguajes en que son programados.

2 Programación en tiempo real.

2.1 Características de los sistemas de tiempo real.

Un sistema de tiempo real posee muchas características, bien inherentes, bien impuestas. Por lo tanto, un sistema operativo o un lenguaje que pretenda ser utilizado para construir sistemas de tiempo real debe disponer de facilidades que soporten estas características.

2.1.1 Grandes y complejos.

Todo ingeniero del software conoce que la mayoría de los problemas asociados a la construcción de una aplicación están relacionados con su tamaño y su complejidad. Un programa pequeño puede ser diseñado, codificado y mantenido por una sola persona. Si esta abandona la institución o la compañía, otra persona puede aprenderlo rápidamente.

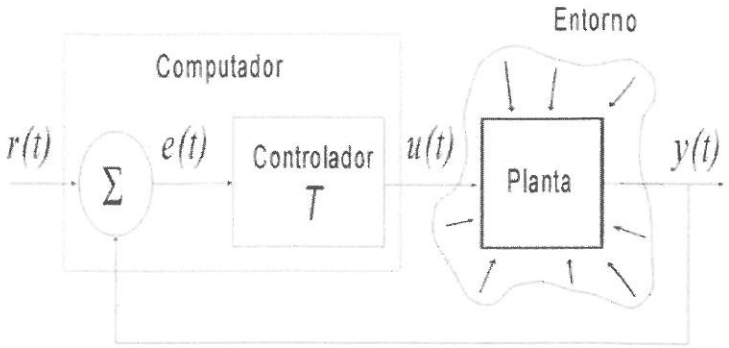
Desgraciadamente, no todos los programas son pequeños. La complejidad de una aplicación no es proporcional a su número de instrucciones. En su lugar, la complejidad va ligada a la variedad de la misma. Una aplicación es de una variedad grande cuando responde a un entorno diverso. Un tamaño grande es un síntoma de variedad.

Los sistemas de tiempo real deben responder, por definición, a eventos del mundo real. La variedad de estos eventos suele conducir a aplicaciones de gran tamaño. Ya que el entorno de una aplicación es continuamente cambiante, la aplicación, grande o pequeña, debe evolucionar continuamente. El costo de un rediseño y una reescritura constante de una aplicación grande tiene un costo prohibitivo, por lo que los sistemas de tiempo real deben ser extensibles.

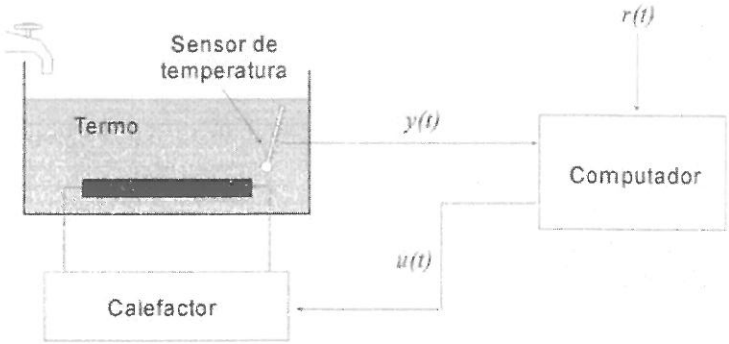
Los sistemas de tiempo real son muy a menudo grandes y complejos por lo que los lenguajes de tiempo real deben proporcionar facilidades para dividirlos en partes más pequeñas y manejables.

2.1.2 Manipulación de números reales.

Muchos sistemas de tiempo real llevan a cabo el control de procesos industriales. La siguiente figura muestra un sistema de control realimentado. El bloque planta es un sistema físico que, dada una entrada, $u(t)$ produce una salida $y(t)$. Por otra parte, el sistema global recibe una señal de referencia $r(t)$. El objetivo del sistema de control es que el sistema físico emita una señal $y(t)$, que sea lo más parecida posible a $r(t)$, es decir, minimizar la diferencia entre ambas. Para lograr el objetivo es preciso aplicar a la planta la señal apropiada $u(t)$.



Pongamos un ejemplo: *r(t)* puede ser la temperatura que debe tener el agua de un termo y el valor real que muestra el termómetro del termo es *y(t)*, según ilustra la siguiente figura. La función del controlador es obtener periódicamente la diferencia *e(t)* entre los valores real *r(t)* y deseado *y(t)* y minimizarla. ¿Cómo? Activando el calentador del termo con una intensidad *u(t)* en función de la diferencia *e(t)*. Esta intensidad *u(t)* puede ser calculada por el controlador ya que éste dispone de un modelo matemático del sistema físico o planta. Este modelo relaciona calor aplicado al termo e incremento de temperatura experimentado en la misma. El modelo es conocido en ingeniería de control como la función de transferencia de la planta, *T*. Podemos decir que $u(t+1) = T[e(t)]$.



La función de transferencia *T* de un sistema de control es simple en el caso del calentador de agua, generalmente una ecuación de diferencias, pero puede llegar a ser realmente compleja. A causa de estas dificultades y otras como el número de entradas y salidas (*r(t)* e *y(t)* pueden ser magnitudes vectoriales *r*(t) e *y*(t)), el controlador se implementada como un computador. El diseño de los algoritmos de las distintas funciones de transferencia es un tópico fuera del alcance de los sistemas de tiempo real. No así la implementación de los mismos. Un requisito imprescindible de los lenguajes de tiempo real es la habilidad para manipular números reales, sean de punto fijo o de punto flotante.

2.1.3 Fiabilidad y seguridad.

Cada vez más, la sociedad abandona el control de sus funciones vitales a los computadores, de modo que, cada vez más, se hace imperativo que los computadores no fallen. Muchos ejemplos dramáticos ilustran que el hardware y el software de los computadores debe ser fiable y seguro. Por otra parte, cuando la intervención de un operador humano se hace necesaria, se debe poner cuidado en que el diseño de la interfaz hombre-máquina minimice la posibilidad del error humano.

El mismo tamaño y complejidad de los sistemas de tiempo real exacerban el problema de la fiabilidad. No sólo deben considerarse los fallos inherentes a la ejecución de la aplicación sino también posibles errores introducidos en su diseño. El

tópico de "tolerancia a fallos" considera el problema del desarrollo de software fiable junto con las facilidades del lenguaje en materia de tratamiento de condiciones de error esperadas y no esperadas.

2.1.4 Interacción con el hardware.

Ya hemos percibido que la naturaleza de los sistemas empotrados exige al computador interactuar con el mundo exterior monitorizando sensores y activando actuadores. Ambos dispositivos son accedidos a través de sus registros hardware y a menudo elevan interrupciones al computador para indicar la necesidad de ser atendidos.

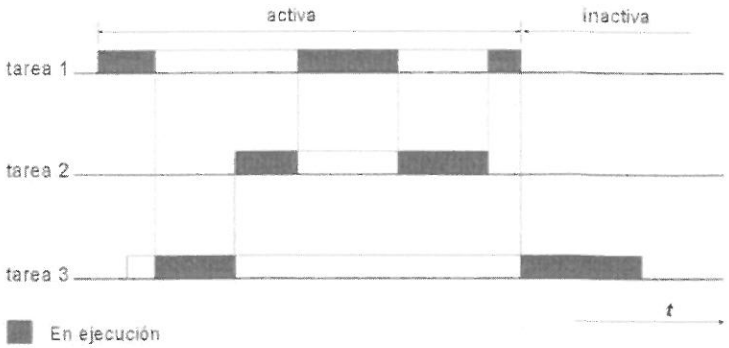
En el pasado, el control de dispositivos de llevaba a cabo por medio de invocaciones al sistema operativo o bien lo realizaba directamente la aplicación insertando código ensamblador en la misma. Hoy en día, el carácter crítico de la interacción con los dispositivos aconseja un control directo sin el apoyo de la capa del sistema operativo. El uso del ensamblador, sin embargo, se desaconseja debido a los requerimientos de fiabilidad de los nuevos sistemas. Así, los lenguajes de tiempo real deben proporcionar primitivas de acceso a los dispositivos y soporte de interrupciones.

2.1.5 Determinismo temporal.

El tiempo de respuesta es crucial en los sistemas empotrados, pero desgraciadamente es muy difícil diseñar e implementar sistemas que garanticen todos los plazos en todas las circunstancias posibles. Una aproximación al problema es dotar al sistema de una potencia de cómputo bien sobrada para asegurar que la situación más desfavorable no provoque situaciones de fallo. Entonces, dada una potencia de cómputo suficiente, se exige al lenguaje de tiempo real y a su núcleo de ejecución que proporcionen al programador ciertas características que hacen que el sistema sea predecible. A estas características se las denomina facilidades de tiempo real. Son: especificar los tiempos en que las operaciones han de realizarse, especificar los tiempos en que las operaciones han de completarse, responder a las situaciones donde no pueden ser atendidos todos los plazos, y responder a las situaciones donde los requisitos de temporización cambian.

2.2 Tiempo compartido y tiempo real.

Como en los sistemas de tiempo compartido, en un sistema de tiempo real el concepto de concurrencia es importante. Ya que dos estímulos pueden llegar muy cercanos en el tiempo y ambos exigen ser atendidos en un plazo máximo, los conceptos de concurrencia y planificación adquieren incluso mayor protagonismo que en los sistemas convencionales de tiempo compartido. Un sistema de tiempo real debe ejecutar simultáneamente distintas actividades que atienden a los distintos estímulos del entorno. Cada actividad se traduce en una o más tareas. Una tarea es una secuencia de instrucciones (sea un hilo o un proceso) que ejecuta en concurrencia con otras tareas. La ejecución de las tareas se multiplexa en el tiempo en uno o más procesadores. La siguiente figura muestra la ejecución concurrente de tres tareas y cómo la tarea 1, en su plazo de ejecución, debe ceder ciclos de procesador a las tareas 2 y 3.



Una tarea puede encontrarse en distintos estados a lo largo de su ejecución, inactiva, cuando ha completado su servicio al evento y un nuevo evento está por llegar, y activa, cuando se encuentra dando servicio al evento. Una tarea activa puede estar: en ejecución, preparada para ejecutarse, o bloqueada esperando alguna condición.

La planificación de la ejecución de las tareas concurrentes debe asegurar el cumplimiento de algunas propiedades que no se exigen en los sistemas convencionales de tiempo compartido. Son las siguientes:

1. Garantía de plazos. Un sistema de tiempo real funciona correctamente cuando los plazos de todas las tareas están garantizados. Esto significa que todas las tareas ejecutan su actividad dentro de plazo cada vez que se activan. En contraste, en un sistema de tiempo compartido, lo importante es asegurar un flujo (número medio de transacciones por segundo) lo más elevado posible.
2. Estabilidad. Si a causa de una sobrecarga del sistema no se pueden ejecutar todas las tareas dentro de plazo, se deba garantizar que al menos un subconjunto de tareas críticas cumplen sus plazos. En contraste, en un sistema de tiempo compartido, el criterio es asegurar la equidad de en la ejecución de las tareas. Por ejemplo, que ninguna se vea indefinidamente postergada.
3. Tiempo de respuesta máximo. En un sistema de tiempo real se trata de acotar el tiempo de respuesta en el peor caso de todas las tareas. En un sistema de tiempo compartido, se trata de conseguir que el tiempo de respuesta medio sea lo más corto posible.

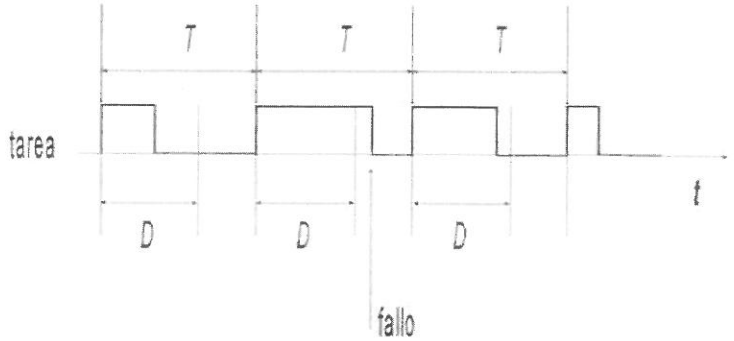
Para asegurar estas propiedades en los sistemas de tiempo real, hay que utilizar un método de planificación de tareas adecuado. Para ello es preciso contar con el hecho de que existen dos clases de tareas: periódicas y esporádicas.

2.2.1 Tareas periódicas.

Una tarea es periódica cuando se ejecuta en instantes de tiempo espaciados regularmente, denominados ciclos. Generalmente se requiere que la ejecución de cada ciclo se complete dentro de un plazo determinado. Así, una tarea periódica queda caracterizada por tres parámetros:

- *T*. Es el período de activación de la tarea. Cada *T* unidades de tiempo comienza un ciclo de la misma.

- *D*. Es el plazo de respuesta, medido desde el comienzo del ciclo o instante de activación. Si la tarea termina después de vencer el plazo, se produce un fallo, según muestra la siguiente figura.



- *C*. Es el máximo tiempo de utilización de CPU que se permite a la tarea en cada ciclo. Al realizar la planificación, *C* depende de las demás tareas que se ejecutan concurrentemente en el ciclo. Si en un ciclo deben ejecutarse muchas tareas, lógicamente debe exigirse que su tiempo de utilización de la CPU sea lo más corto posible.

2.2.2 Tareas aperiódicas.

Una tarea es aperiódica cuando se ejecuta en respuesta a eventos que ocurren de forma asíncrona, en instantes difíciles o imposibles de prever. En ocasiones se requiere que, todas las veces que se ejecuten, terminen dentro de un plazo de respuesta determinado. En este caso se denominan tareas esporádicas y puede especificarse en los requisitos del sistema una separación mínima en el tiempo entre dos eventos consecutivos. Una tarea esporádica queda caracterizada por tres parámetros:

- *T*. Es la separación mínima entre eventos. Cada vez que ocurre un evento, se activa la tarea asociada al mismo.
- *D*. Es el plazo de respuesta, medido desde el instante de activación.
- *C*. Es el tiempo de cómputo máximo que emplea el servicio al evento, igual que en el caso de las tareas periódicas.

Un ejemplo de sistema de tiempo real es el control de un conjunto de subsistemas en un automóvil. Por ejemplo, podemos considerar las tareas de control de inyección, medida de la velocidad y control de frenado o ABS. Este sistema puede ser planteado como un conjunto de tareas periódicas con los parámetros especificados por la siguiente tabla. Por lo que respecta a la medida de la velocidad, ésta debe realizarse cada $T = 20$ ms con unos cálculos que duren $C = 4$ ms. Si, por la razón que fuese (tal vez una tarea más prioritaria expulsa de la CPU a la tarea de la velocidad), estos cálculos no se han completado en $D = 5$ ms, entonces se produce un fallo en ese ciclo.

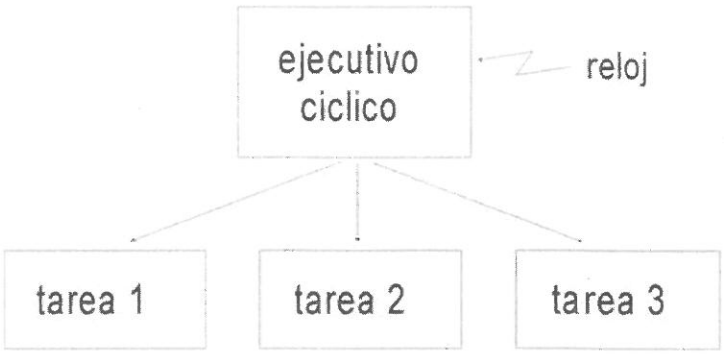
Tarea	Tipo	*C (ms)*	*T (ms)*	*D (ms)*
Control de inyección	Period.	40	80	80
Medida de la velocidad	Period.	4	20	5
Control de frenado (ABS)	Period.	10	40	40

2.3 Planificación.

Según el método de planificación de tareas que se utilice, hay dos tipos de arquitecturas de software que se utilizan en los sistemas de tiempo real: El ejecutivo cíclico y el conjunto de tareas.

2.3.1 El ejecutivo cíclico.

La siguiente figura muestra el denominado ejecutivo cíclico, la única entidad activa en el sistema. Invoca la ejecución de las tareas en secuencia, con arreglo a un plan de ejecución estático. Las tareas se codifican como procedimientos.



El ejecutivo cíclico se emplea en sistemas muy críticos pero de reducido tamaño. Su desarrollo es laborioso y de mantenimiento difícil, por lo que no es apropiado para sistemas grandes. El ejecutivo cíclico es una técnica utilizada antes de que los lenguajes de programación concurrente fuesen populares. Consiste en un programa con un único bucle de control en el que está insertado el código de cada tarea. Desgraciadamente, si las tareas tienen periodos no relacionados por un común denominador (20 ms. en el caso del automóvil), el código para cada una de ellas no puede mantenerse agrupado en un único bloque dentro del bucle de control. Esto conduce a programas no estructurados difíciles de comprender y de mantener.

2.3.2 Tareas concurrentes.

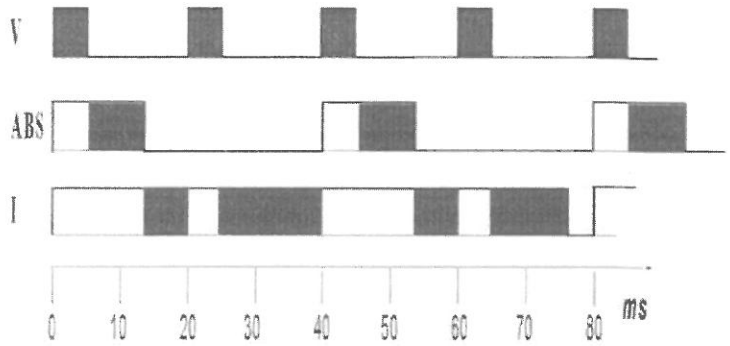
En contraste con el ejecutivo cíclico, las tareas se programan de forma desacoplada y es el sistema operativo o núcleo de ejecución del lenguaje el que planifica el tiempo de CPU entre las tareas activas. Estas pueden invocar servicios del sistema operativo o núcleo de ejecución para sincronizarse o comunicarse entre sí.

El conjunto de tareas puede ser empleado en toda clase de sistemas porque es más fácil de desarrollar y de mantener. La teoría de planificación de tiempo real proporciona algoritmos y métodos de análisis que permiten determinar si en todo momento se garantizan los plazos de respuesta de las tareas activas. Para ayudar en el proceso de planificación vamos a introducir el concepto de prioridad. La prioridad de una tarea indica al planificador el orden en el que las tareas deberían ser ejecutadas.

La asignación de prioridades a las tareas puede ser estática y dinámica. En el primer caso cada tarea tiene una prioridad fija y en cada momento se ejecuta la tarea activa con mayor prioridad. ¿A qué tareas se asigna la prioridad más alta? Por una parte, podemos asignar la prioridad más alta a las tareas más frecuentes. Este tipo de planificación se denomina monotónica en frecuencia. Por otra parte, podemos asignar una prioridad mayor a las tareas más urgentes. Este segundo tipo de planificación se denomina monotónica en plazo. Ambos tipos de planificación disponen de herramientas matemáticas que permiten analizar el sistema y comprobar si los plazos están

garantizados. Si se modifica el sistema, es preciso repetir el análisis de los tiempos de respuesta.

La siguiente figura muestra una planificación monotónica en frecuencia del sistema de control de automóvil antes presentado. Examinando el parámetro *T* de la tabla anterior, la tarea más frecuente es la del control de velocidad (*T*=20 ms), a continuación el control de frenado ABS (*T*=40 ms) y, finalmente, la tarea de control de inyección (*T*=80 ms). Por ejemplo, podemos ver cómo en los instantes 20 y 60 la tarea *V*, más prioritaria, se activa y expulsa de la CPU a la tarea I.



2.4 Sistemas críticos y acríticos.

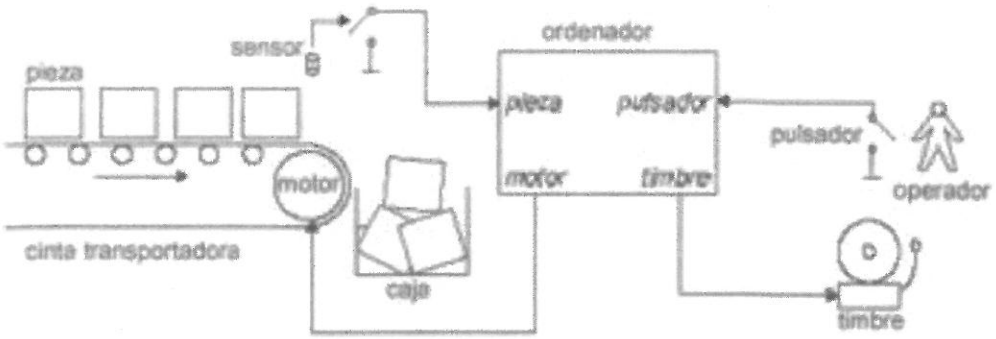
Una característica importante de un sistema de tiempo real es su criticidad. Pueden distinguirse dos tipos de sistemas de tiempo real, críticos y acríticos. Un sistema es acrítico cuando se puede tolerar que ocasionalmente un plazo de respuesta se sobrepase. Un ejemplo es un sistema de adquisición de datos donde estos llegan a intervalos regulares. En ocasiones el proceso del dato *n* hace que se pierda el dato *n+1*, sin que ello signifique una pérdida significativa en el conocimiento del entorno y respuesta al mismo. En contraste, un sistema es crítico cuando el incumplimiento de un plazo es inadmisible porque puede tener consecuencias catastróficas. Así ocurre, por ejemplo, en el sistema de frenado de un automóvil o en sistemas militares como aviones de combate o guía de misiles. Afortunadamente, muchos sistemas de interés práctico son acríticos.

3 Interrupciones en sistemas de tiempo real.

Una interrupción es una indicación a un sistema informático de la ocurrencia de un evento, tanto externo como interno. Puesto que un sistema de tiempo real tiene que responder a estímulos externos en tiempo finito y especificado (por definición), las interrupciones son uno de sus componentes básicos.

El objetivo de las interrupciones en un sistema de tiempo real es permitir a un ordenador responder de forma eficiente a eventos, normalmente externos, independientemente del código que está siendo ejecutado en ese momento.

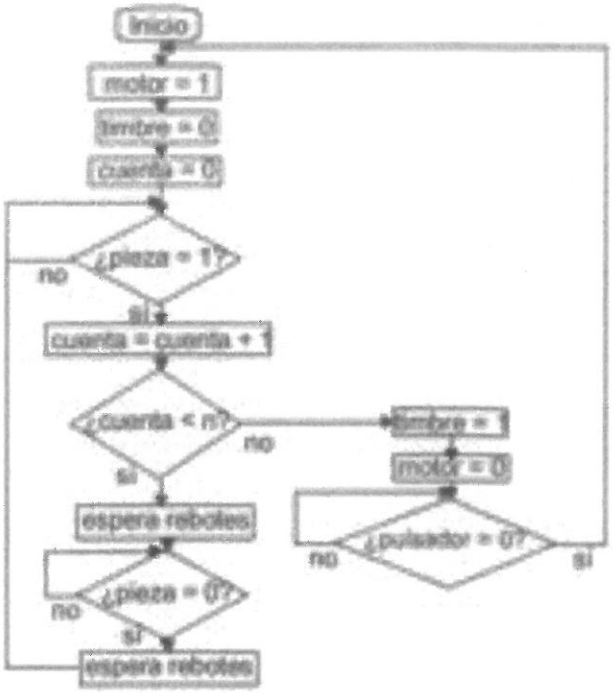
Veamos un ejemplo de sistema de tiempo real:



Los procesos son los siguientes:

1. Poner en marcha la cinta transportadora.
2. Contar las piezas que vayan cayendo en la caja.
3. Si el número de piezas llega a *n*:
   3.1. Detener la cinta transportadora.
   3.2. Hacer sonar el timbre para avisar al operador e ir al paso 5.
4. Si el número de piezas no llega a n, ir al paso 2.
5. Cuando el operador accione el pulsador, se irá de nuevo al paso 1.

Una posible implementación utilizando muestreo (o *polling*) sería:



En este esquema la CPU no se puede utilizar para ninguna otra tarea ya que debe supervisar constantemente la operación del dispositivo mediante un bucle de lectura de un registro de estado. Una vez en el estado apropiado, la tarea realiza sobre el dispositivo las operaciones oportunas de transferencia de datos.

3.1 Interrupciones simples.

El uso de interrupciones permite que se interrumpa la ejecución del programa en la CPU cuando se produce un evento. Esta interrupción se realiza a través de una línea especial del bus de control llamada IRQ (*interrupt request*). La petición de interrupción,

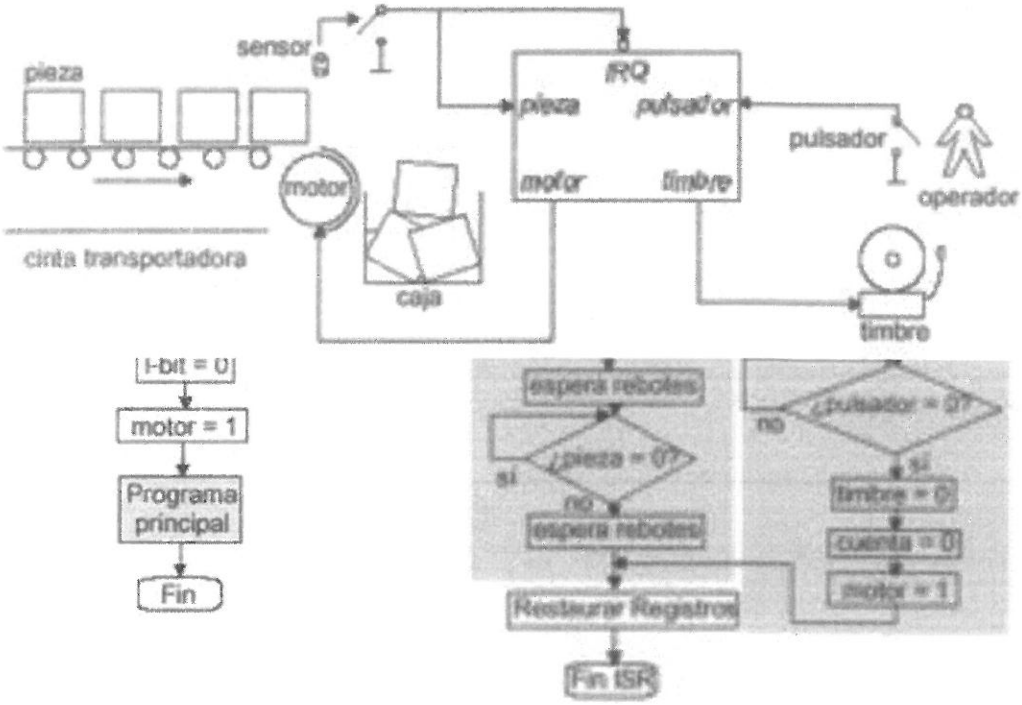
se traduce entonces en la llamada a una rutina servidora de interrupción (ISR). De este modo, la CPU no debe estar continuamente comprobando el estado de los sensores.

Para implementar este método, la CPU, además de la línea de entrada de petición de interrupción, debe disponer de un bit interno de inhibición de interrupciones (I-bit), para inhibir la entrada de IRQ.

Cuando llega una interrupción, (si el I-bit es 0), ocurre lo siguiente:

1. La CPU termina de ejecutar la instrucción actual.
2. Se guarda el estado de la CPU en la pila.
3. Se deshabilitan las futuras interrupciones poniendo I-bit a 1.
4. Se carga en el controlador del programa la dirección de comienzo de la ISR, con lo que se ejecuta.
5. La ISR retorna mediante una instrucción especial de fin de interrupción.
6. Se recupera el estado de la CPU de la pila.
7. El programa interrumpido continúa por donde se interrumpió.

Veamos cómo se resolvería con este método el ejemplo anterior:



Este sistema presenta las siguientes características:

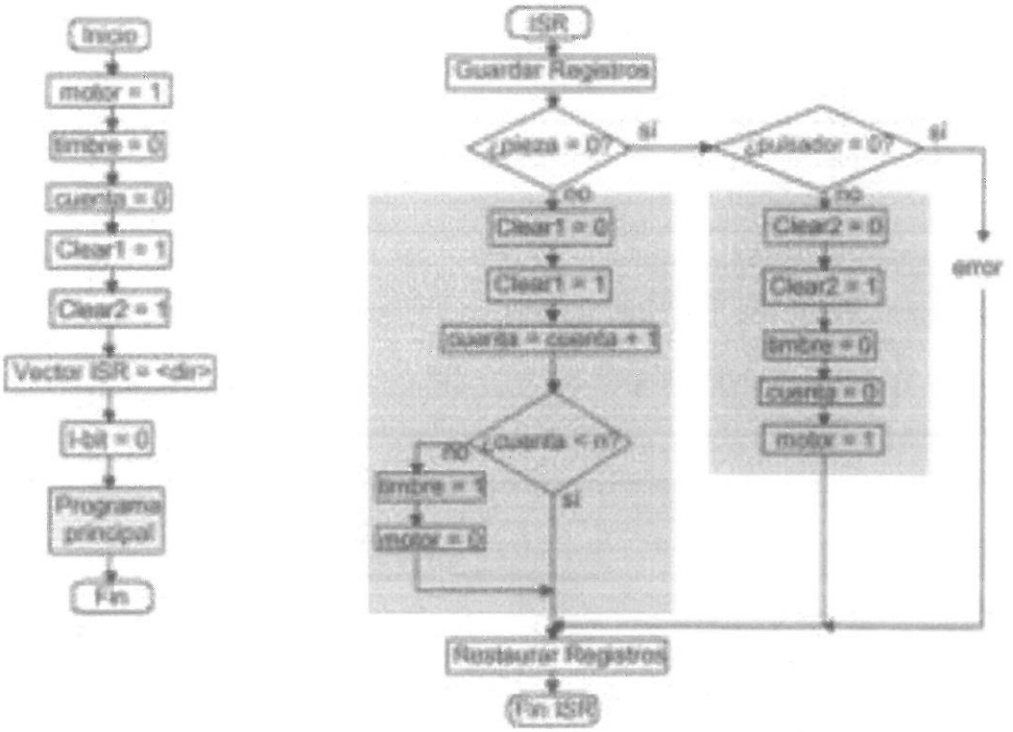
- La señal IRQ no se memoriza ni se borra, es instantánea.
- La espera de rebotes se realiza con espera de tiempo.
- El programa principal realiza cualquier otra actividad.

3.2 Interrupciones múltiples.

En muchas situaciones, el servicio de interrupción de un suceso puede a su vez ser interrumpido por otro suceso de mayor prioridad. Pueden establecerse niveles de prioridad de interrupciones.

3.2.1 Compartición de una línea de interrupción simple.

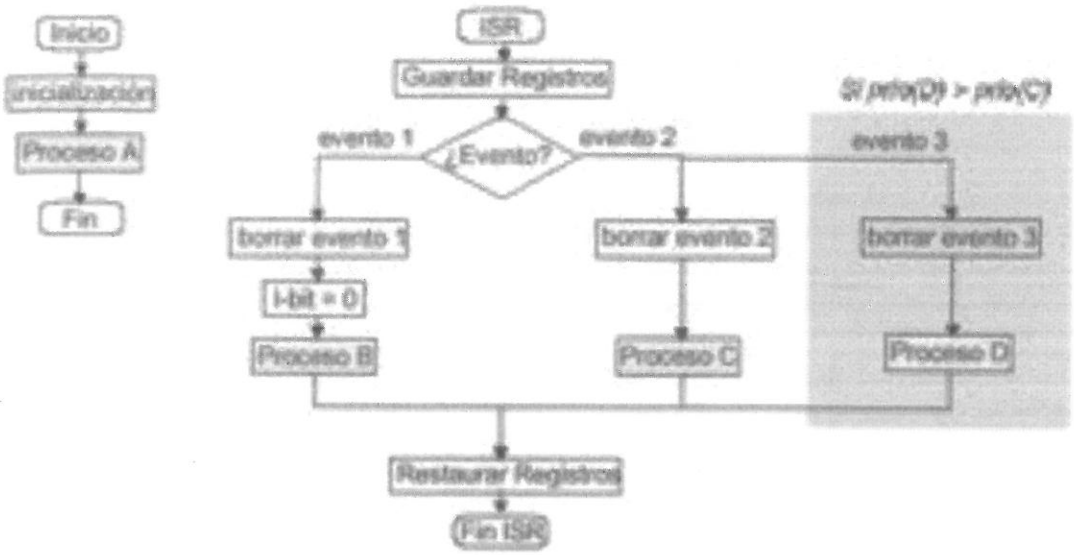
Es posible compartir para dos o más eventos una misma línea de interrupción simple.



En el ejemplo, la IRQ se activa tanto por el pulsador, como por la llegada de una pieza. La prioridad implícita en el código es: prioridad(principal) < prioridad(pieza) = prioridad(pulsador).

3.2.2 Anidamiento de interrupciones.

Supongamos tres tareas A, B, y C con prioridades tales que prioridad(A)<prioridad(B)<prioridad(C).



3.2.3 Enmascaramiento de interrupciones.

El enmascaramiento permite seleccionar, mediante una máscara, qué peticiones de interrupción van a ser atendidas en cada momento. El mecanismo de enmascaramiento es el siguiente:

1. Al entrar en la ISR se almacena el registro de máscara en la pila.
2. Se identifica el evento causante de la interrupción mediante el registro de evento.
3. Se establece la máscara correspondiente al evento.
4. Se borra la solicitud de interrupción correspondiente al evento.
5. Se pone el I-bit a 0 para permitir interrupciones.
6. Se ejecuta el proceso correspondiente al evento.
7. Se deshabilitan las interrupciones (I-bit a 1).
8. Se recupera el registro de máscara de la pila.
9. Se retorna de la ISR.

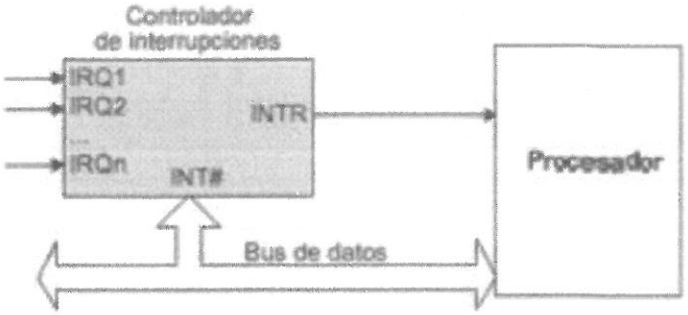
3.2.4 Interrupciones jerarquizadas.

El mecanismo de interrupciones jerarquizadas es el siguiente:

1. Al entrar en la ISR se almacena el registro de nivel en la pila.
2. Se identifica el evento causante de la interrupción mediante el registro de evento.
3. Se establece el registro de nivel al nivel correspondiente al evento.
4. Se borra la solicitud de interrupción correspondiente al evento.
5. Se pone el I-bit a 0 para permitir interrupciones de nivel superior o igual al actual.
6. Se ejecuta el proceso correspondiente al evento.
7. Se deshabilitan las interrupciones (I-bit a 1).
8. Se recupera el registro de nivel de la pila y se establece como nuevo valor de nivel.
9. Se retorna de la ISR.

3.2.5 Interrupciones vectorizadas.

Se basan en una tabla de direcciones de ISR en memoria, indexada por el número de interrupción.



4 Sincronización y comunicación entre tareas.

Raras veces los procesos de un sistema son independientes unos de otros. Más a menudo cooperan para un fin común o compiten por la utilización de recursos. Para ello es necesario realizar operaciones de comunicación y sincronización entre procesos.

- Dos procesos se comunican cuando hay una transferencia de información de uno a otro.
- Dos procesos están sincronizados cuando hay restricciones en el orden en que ejecutan algunas de sus acciones.

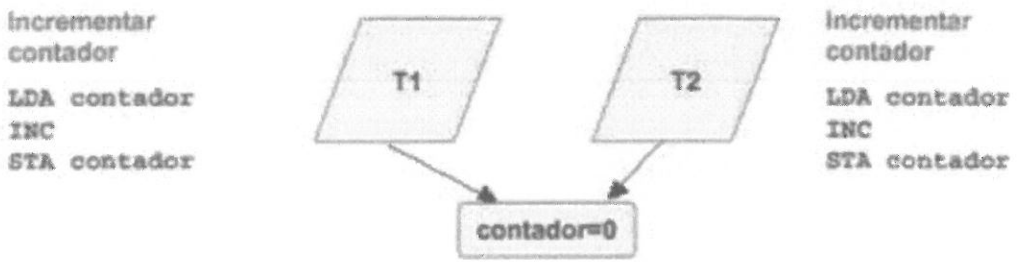
4.1 Comunicación con variables comunes.

En un sistema monoprocesador, la forma más directa de comunicación entre dos o más procesos consiste en compartir datos comunes. Sin embargo, el acceso incontrolado a variables comunes puede producir resultados anómalos.

Se dice que hay una condición de carrera cuando el resultado de la ejecución depende del orden en que se intercalan las instrucciones de dos o más procesos. Se trata de una situación anómala que hay que evitar.

En la siguiente figura podemos ver un ejemplo donde el resultado final puede ser 1 ó 2 dependiendo de las velocidades relativas de los procesos.

Para evitar las condiciones de carrera hay que asegurar que las operaciones con variables comunes se ejecutan de forma atómica (indivisible).



4.1.1 Exclusión mutua.

Una secuencia de instrucciones que se debe ejecutar de forma indivisible se denomina sección crítica. Las operaciones de dos secciones críticas respecto a la misma variable no se entremezclan.

La forma de sincronización que se usa para proteger una sección crítica se llama exclusión mutua.

4.1.2 Sincronización condicional.

Cuando una acción de un proceso solo se puede ejecutar si otro proceso está en determinado estado o ha ejecutado ciertas acciones, hace falta un tipo de sincronización denominada sincronización condicional.

Un ejemplo de sincronización condicional es el problema del productor y consumidor con tampón limitado.

En este ejemplo, no se debe hacer *Put* cuando el tampón está lleno, y no se debe hacer *Get* cuando el tampón está vacío. Además, hay exclusión mutua en el acceso al tampón.

4.2 Mecanismos de sincronización.

4.2.1 Espera ocupada y suspensión síncrona.

Una forma de realizar la exclusión mutua es usar un indicador compartido. Este método se puede utilizar si el acceso al indicador es atómico (por ejemplo, mediante una instrucción *test- and- set*).

```
while Test_ and_ Set(Flag) loop
   null;
end loop;
-- sección crítica
Flag := False;
```

Es un método poco eficiente y complicado. Además, no es fácil imponer un orden de acceso cuando hay varios procesos esperando.

Una solución más eficiente que la espera ocupada consiste en disponer de un indicador que permita que un proceso se detenga o continúe su ejecución de forma atómica. En este caso, la atomicidad se asegura en el núcleo de multiprogramación.

4.2.2 Semáforos.

Un semáforo es una variable que toma valores enteros no negativos. Además de asignarle un valor inicial, sólo se pueden hacer dos operaciones (indivisibles) con un semáforo S:

- Wait: Si S>0, decrementa S en 1. Si S=0, la tarea se suspende hasta que S>0.
- Signal: Incrementa S en 1.

Otros nombres comunes para las operaciones de un semáforo son: Down/Up, P/V, Secure/Release.

La verificación y modificación del valor, así como la posibilidad de suspender la tarea se realiza en conjunto, como una sola e indivisible acción atómica. Se garantiza que al iniciar una operación con un semáforo, ningún otro proceso puede tener acceso al semáforo hasta que la operación termine o se bloquee. Esta atomicidad es absolutamente esencial para resolver los problemas de sincronización y evitar condiciones de competencia.

4.2.3 Regiones críticas condicionales.

Una región crítica es una secuencia de instrucciones que se ejecuta en exclusión mutua:

- El compilador produce el código de bajo nivel necesario
- Se identifica la variable compartida a la que se accede en una sección crítica

La sincronización condicional se consigue mediante guardas en las regiones críticas de forma que sólo se puede entrar en la región crítica cuando la guarda es verdadera.

El problema de las regiones críticas es su complicada implementación:

- Cada vez que un proceso sale de una región crítica hay que volver a evaluar las guardas de los procesos que están esperando en otras regiones con la misma variable, para lo que hay que reanudar el proceso que espera, que se puede volver a suspender si la guarda es falsa.
- Muchos cambios potenciales de contexto.
- No es obligatorio que estén confinadas en módulos.
- El programa puede ser difícil de comprobar y mantener.

4.2.4 Monitores.

Implementar la comunicación entre procesos con los semáforos es complicado y puede producir bloqueos. Un pequeño error y todo se vendría abajo, puesto que los errores son condiciones de competencia, bloqueos y otras formas de comportamiento impredecible e irreproducible.

Para facilitar la escritura de programas correctos, existe una primitiva de sincronización de alto nivel, llamada monitor. Un monitor es una colección de procedimientos, variables y estructuras de datos que se agrupan, en cierto tipo particular de módulo o paquete. Los procesos pueden llamar a los procedimientos de un monitor cuando lo deseen, pero no tienen acceso directo a las estructuras de datos internas del monitor desde los procedimientos declarados fuera de él.

Los monitores tienen una propiedad importante que los hace útiles para conseguir la exclusión mutua: sólo uno de los procesos puede estar activo en un monitor en cada momento. Los monitores son construcción del lenguaje de programación, por lo que el compilador sabe que son especiales y puede controlar las llamadas a los procedimientos del monitor de forma distinta a las llamadas de los demás procedimientos. Por lo general, cuando un proceso llama a un procedimiento de monitor, las primeras instrucciones de éste verificarán si hay otro proceso activo dentro del monitor. En caso afirmativo, el proceso que hace la llamada será suspendido hasta

que el otro proceso salga del monitor. Si no hay otro proceso que esté utilizando el monitor, el que hace la llamada podrá entrar.

El compilador tiene la tarea de implantar la exclusión mutua sobre los datos del monitor, pero una forma usual de conseguir esto es mediante un semáforo binario. Puesto que es el compilador, y no el programador, el que ordena las cosas para la exclusión mutua, es menos probable que algo malo ocurra. En todo caso, la persona que escribe el monitor no es consciente de la forma en que el compilador logra la exclusión mutua. Basta con saber que al transformar todas las secciones críticas en procedimientos del monitor, no ocurrirá que dos procesos ejecuten sus secciones críticas al mismo tiempo.

Con la exclusión mutua automática de las regiones críticas, los monitores hacen que la programación en paralelo tenga menos tendencia a los errores que en el caso de uso de semáforos.

Como hemos dicho, un monitor es un módulo cuyas operaciones se ejecutan en exclusión mutua. El compilador produce el código de bajo nivel necesario de forma que la sincronización condicional se obtiene con variables de condición. Para ello, podemos definir dos operaciones primitivas: *signal* y *wait*.

- *wait* suspende el proceso de forma incondicional y lo pone en una cola asociada a la condición que espera.
- *signal* reanuda el primer proceso que espera en la señal

4.3 Comunicación mediante mensajes.

Este método de comunicación entre procesos utiliza dos primitivas, SEND y RECEIVE, las cuales, al igual que los semáforos y a diferencia de los monitores, son llamadas al sistema en vez de comandos del lenguaje. Como tales, se pueden colocar con facilidad en procedimientos de librería, como por ejemplo, *send(destino,&mensaje)* y *receive(fuente,&mensaje)*. El primer procedimiento envía un mensaje a un destino dado y el segundo recibe un mensaje desde cierto origen (o desde cualquier origen, si al receptor no le importa este punto). Si no hay mensajes disponibles, el receptor se puede bloquear hasta que llegue alguno.

5 Lenguajes de tiempo real.

Debido a los requisitos especiales de rendimiento y de fiabilidad demandados por los sistemas de tiempo real, podemos enumerar las características que un lenguaje de programación debe poseer para el desarrollo de estos sistemas:

- Concurrencia. Un lenguaje de tiempo real debe permitir la definición de tareas concurrentes. Es importante debido a que los sistemas de tiempo real deben responder a procesos asíncronos que ocurren simultáneamente.
- Control de tiempo. El lenguaje debe permitir acceder a un reloj de tiempo real para la ejecución de tareas en instantes concretos de tiempo.
- Planificación de la ejecución de tareas.
- Comunicación con dispositivos hardware. Debe ser posible el manejo de los dispositivos hardware, así como las interrupciones.
- Control de errores y situaciones excepcionales. El lenguaje debe contener instrucciones para la detección y recuperación de fallos.

- Facilidades para la programación fiable. Debido a que los programas de tiempo real son frecuentemente grandes y complejos, el lenguaje debe soportar la programación modular, estructuras de control y definición datos adecuadas, etc.

5.1 Lenguajes convencionales y sistemas operativos de tiempo real.

El enfoque inicial de los sistemas de tiempo real consistió en utilizar un lenguaje secuencial (Pascal, C, etc.), siendo el sistema operativo (de tiempo real) quien debe manejar las contrucciones propias de los sistemas de tiempo real.

La ventaja de este enfoque radica en que no se necesita aprender un nuevo lenguaje. Sin embargo, resulta difícil la definición y validación de las características del sistema de tiempo real.

5.2 Lenguajes con tratamiento sistemático de la concurrencia.

En este esquema se incluyen lenguajes como PL/1, Algol 68, Pascal Concurrente o Modula. En estos lenguajes la concurrencia y los mecanismos de sincronización aparecen integrados en el propio lenguaje.

El principal problema que presentan es la falta de mecanismos de control para la planificación de los procesos, así como la imposibilidad de compilar por separado los diferentes componentes.

5.3 Modula-2 y ADA.

Estos dos lenguajes presentan buenas características de modularidad. Cada uno, presenta una serie de características particulares:

- Modula-2: modularidad, concurrencia, interrupciones, acceso a hardware.
- ADA: más potente, para aplicaciones empotradas de tiempo real, genericidad, sincronización por citas, excepeciones.

5.4 Lenguajes con especificaciones temporales.

Son lenguajes experimentales y de uso no industrial. Destacan por la posibilidad de especificar restricciones de tiempo real. Los lenguajes de este tipo más conocidos son Esterel y Euclid.

5.5 Características deseables de los lenguajes de tiempo real.

Podemos agrupar las características que sería deseable que presentara un lenguaje de tiempo real en los siguientes grupos o categorías:

- Seguridad. La detección de errores en tiempo de compilación reduce coste de desarrollo, no recarga el tiempo de ejecución, pero presenta un mayor coste del compilador y tiempo de compilación.
- Legibilidad. Una notación clara, sintaxis, modularidad, keywords, etc., reduce el tiempo de documentación proporcionando una mayor seguridad y menor mantenimiento, aun a costa de una mayor longitud de los programas.
- Simplicidad. Minimiza el coste del compilador y reduce el coste de entrenamiento y de corrección de errores.

- Flexibilidad. Permite expresar lo que el programador desea directa y coherentemente.
- Portabilidad. Independiza del hardware y del sistema operativo. Sin embargo, es difícil de conseguir en los sistemas de tiempo real.
- Eficiencia. Garantiza las restricciones temporales. Sin embargo, está en contraposición con la seguridad, flexibilidad y legibilidad.

5.6 Caracterización de algunos lenguajes de interés para sistemas de tiempo real.

5.6.1 Ensamblador.

Ha sido el "lenguaje obligado" para los sistemas de tiempo real hasta hace algunos años. Sus principales características son:

- Orientado a la máquina en vez de orientado al problema.
- Altos costos de desarrollo y mantenimiento.
- No es portable.

5.6.2 Lenguajes estructurados de alto nivel.

Surgen a partir de finales de los años 70. Ejemplos de estos lenguajes son: PASCAL, C, etc. Sus características más destacadas son:

- Estructuración ("Revolución contra el GOTO").
- Tipificación de datos.
- Cuando son aplicados a sistemas de tiempo real (multitareas) hacen uso de servicios del sistema operativo respectivo (a través de bibliotecas de servicios).

5.6.3 Lenguajes concurrentes de alto nivel

Aparecen como respuesta a la "crisis del software". Ejemplos de estos lenguajes son: ADA, Chill, JAVA, etc. Sus características más destacadas son:

- Implementan tipos abstractos de datos o son orientados a objeto.
- Poseen primitivas para especificar tareas y su sincronización.

6 Conclusiones.

Un sistema de tiempo de real genera alguna acción en respuesta a sucesos externos bajo varias restricciones de tiempo y fiabilidad. El objetivo de la computación de tiempo real es garantizar los requisitos temporales individuales de cada tarea.

Los sistemas de tiempo real son más complejos que los sistemas convencionales, por lo que exigen requisitos adicionales a los lenguajes en que son programados.

Los sistemas de tiempo real son muy a menudo grandes y complejos por lo que los lenguajes de tiempo real deben proporcionar facilidades para dividirlos en partes más pequeñas y manejables.

Un requisito imprescindible de los lenguajes de tiempo real es la habilidad para manipular números reales, sean de punto fijo o de punto flotante.

El mismo tamaño y complejidad de los sistemas de tiempo real exacerban el problema de la fiabilidad. No sólo deben considerarse los fallos inherentes a la ejecución de la aplicación sino también posibles errores introducidos en su diseño. El tópico de "tolerancia a fallos" considera el problema del desarrollo de software fiable junto con las facilidades del lenguaje en materia de tratamiento de condiciones de error esperadas y no esperadas.

Los lenguajes de tiempo real deben proporcionar primitivas de acceso a los dispositivos y soporte de interrupciones.

Un sistema de tiempo real debe especificar los tiempos en que las operaciones han de realizarse, especificar los tiempos en que las operaciones han de completarse, responder a las situaciones donde no pueden ser atendidos todos los plazos, y responder a las situaciones donde los requisitos de temporización cambian.

Un sistema de tiempo real debe ejecutar simultáneamente distintas actividades que atienden a los distintos estímulos del entorno. Cada actividad se traduce en una o más tareas. Una tarea es una secuencia de instrucciones (sea un hilo o un proceso) que ejecuta en concurrencia con otras tareas. Por ello, en los sistemas de tiempo real, hay que utilizar un método de planificación de tareas adecuado.

Según el método de planificación de tareas que se utilice, hay dos tipos de arquitecturas de software que se utilizan en los sistemas de tiempo real: El ejecutivo cíclico y el conjunto de tareas.

Una interrupción es una indicación a un sistema informático de la ocurrencia de un evento, tanto externo como interno. Puesto que un sistema de tiempo real tiene que responder a estímulos externos en tiempo finito y especificado, las interrupciones son uno de sus componentes básicos.

El uso de interrupciones permite que se interrumpa la ejecución del programa en la CPU cuando se produce un evento. Esta interrupción se realiza a través de una línea especial del bus de control llamada IRQ (*interrupt request*). La petición de interrupción, se traduce entonces en la llamada a una rutina servidora de interrupción (ISR).

En muchas situaciones, el servicio de interrupción de un suceso puede a su vez ser interrumpido por otro suceso de mayor prioridad. Pueden establecerse niveles de prioridad de interrupciones.

Raras veces los procesos de un sistema son independientes unos de otros. Más a menudo cooperan para un fin común o compiten por la utilización de recursos. Para ello es necesario realizar operaciones de comunicación y sincronización entre procesos. El acceso de dos o más tareas concurrentes a las variables comunes debe realizarse en exclusión mutua. Otra forma de sincronización está ligada al cumplimento de una condición. Algunos mecanismos que permiten conseguir ambas formas de sincronización son: semáforos, regiones críticas condicionales, y monitores.