Microcontroladores en tiempo real

E. Spinelli y A.Veiga
Laboratorio de Electrónica, Dto. de Física, Fac. de Cs. Exactas,
Universidad Nacional de La Plata
(electro@venus.fisica.unlp.edu.ar)

Resumen:

Se presenta una estrategia para la programación de microcontroladores de propósitos generales en la implementación de sistemas de tiempo real utilizando los conceptos básicos de un sistema multitarea basado en prioridades. Esta estrategia aplicada al escribir los programas de aplicación puede proveer la solucion a diferentes casos con estrictos requerimientos en tiempo, sin utilizar software adicional ni microcontroladores sofisticados.
Se analizan dos implementaciones de software de fácil programación, que permiten organizar las tareas que lleva a cabo el microcontrolador. Esta organización redunda en tiempos de respuesta predecibles, lo cual es altamente deseable en sistemas que se encuentran en contacto con el mundo real.
Ambas implementaciones se basan en asociar las diferentes tareas que debe realizar el microcontrolador con interrupciones de hardware. De esta manera se utiliza el manejo de interrupciones del cual dispone el microcontrolador como parte del scheduler, resultando un kernel compacto y simple de programar.

Microcontroladores en tiempo real

1. Introducción

La aparición de los microcontroladores (dispositivos que integran un microprocesador, RAM, ROM, timers y periféricos de entrada/salida en un único chip), su versatilidad y bajo costo, han llevado a que, en una gran cantidad de casos, sean la solución mas viable para la implementación de sistemas que interaccionan con el mundo real. Su capacidad de entrada/salida hace que sean una solución rápida y efectiva para el manejo de dispositivos externos, los cuales, en general, tienen restricciones estrictas en tiempo.
La forma mas común de escribir los programas de aplicación es hacerlo en forma tal que las tareas que debe realizar el microcontrolador se ejecuten secuencialmente, siempre en el mismo orden (polled loop). Para solucionar los problemas de sincronización que puedan presentarse, la técnica más común es hacer que el microcontrolador ejecute dichas tareas lo mas rápido posible. En la mayoría de los casos puede encontrarse una solución por esta vía, y en caso de no ser posible se busca trasladar el desarrollo hacia un modelo de microcontrolador mas potente.
Pero si se analiza en detalle el funcionamiento de un microcontrolador mientras ejecuta dicha secuencia fija de tareas, podrá observarse que éste emplea la mayor parte del tiempo esperando la finalización de eventos externos, sin realizar ningún procesamiento útil. Por ello, en la mayor parte de los casos en que no se consigue que el microcontrolador cumpla con requisitos de tiempo, la solución no es hacerlo trabajar mas rápido, sino hacer que realice las tareas en forma más ordenada.
Una solución es asociar las tareas con interrupciones y administrarlas adecuadamente. Una de las opciones es utilizar prioridades y asignar las más altas a las tareas que tiene mayores restricciones de tiempo. Debe permitirse que estas tareas interrumpan la ejecucion de tareas menos cruciales para el sistema (sistema preemptivo basado en prioridades). En general, este tipo de programación es adecuado para sistemas que tienen restricciones en tiempo, cuyo no cumplimiento resulta en la falla total de dicho sistema (hard real-time systems).
La estrategia preemptiva basada en prioridades es la utilizada por algunas aplicaciones comerciales que se presentan como sistemas operativos de tiempo real para embedded systems. Estas aplicaciones tienen la desventaja de ser programas voluminosos que deben colocarse en memoria. Son voluminosos debido a que están escritos para cubrir diferentes casos de aplicación, lo cual los lleva irremediablemente a tener más funciones que las necesarias para un caso particular.
El objetivo de este trabajo es presentar una estrategia para la programación de un microcontrolador de propósitos generales, que pueda emplearse en el momento de escribir un programa particular de aplicación, utilizando los conceptos básicos de un sistema multitarea de tiempo real. Se sugiere una forma ordenada de escribir los programas de aplicación, que puede proveer la solucion a diferentes casos sin utilizar software adicional o microcontroladores sofisticados.
En los puntos 2 y 3 se analizan dos metodologías diferentes de programación que permiten organizar las tareas que lleva a cabo el microcontrolador, resultando un sistema con tiempos de respuesta acotados, lo cual es altamente deseable en sistemas que se encuentran en contacto con el mundo real.
Para los ejemplos de aplicación de utilizó el microntrolador 8051 de Intel, programado en assembler, pero lo expuesto no pierde generalidad al ser trasladado a otras plataformas, ya que los recursos disponibles son similares para las diferentes familias

de microcontroladores de propósitos generales.
Los sistemas fueron implementados y actualmente son utilizados en instrumentación y control en experiencias de física nuclear.

1.1. Principios básicos de funcionamiento

Un sistema operativo multitarea tal como ha sido descripto anteriormente, debe proveer al menos tres funciones específicas:

1. Dispatcher de tareas: toma los recaudos necesarios para iniciar un nuevo proceso en el entorno multitarea.
2. Procesos de comunicación y sincronización entre los diferentes procesos que se ejecutan simultáneamente en el microcontrolador.
3. Scheduler de tareas: determina en qué orden y bajo qué condiciones se ejecutarán dichas tareas. Recordemos lo siguiente: para su aplicación en el campo del tiempo real, es deseable que la estrategia de scheduling sea preemptiva y basada en prioridades. El principio de funcionamiento de esta estrategia se basa en que cada proceso tiene un nivel de prioridad fijo, asignado a priori. Las tareas de más alto nivel de prioridad deben ser capaces de interrumpir la ejecución de las tareas de nivel mas bajo. Con esta estrategia se consiguen los mejores resultados en la implementación de sistemas que tienen una estricta dependencia del tiempo, y es en general la solución mas adecuada al planteo de un sistema multitarea que debe estar en contacto con el 'mundo real' [1]. La ventaja de utilizar este tipo de estrategia es que generalmente las tareas con restricciones de tiempo mas estrictas son las tareas con menor tiempo de ejecución.

El kernel (o núcleo) del sistema operativo es la porción mas pequeña de éste que provee esas tres funciones.
Una forma simple de implementación de este kernel en un microcontrolador es asociar cada una de las tareas que éste debe realizar (y que tengan restricciones de tiempo) con las interrupciones disponibles en dicho microcontrolador. Esto nos permite utilizar el manejo de interrupciones por hardware que provee el microcontrolador para implementar partes del dispatcher y el scheduler.

1.2 Clasificación de las tareas

Como primer paso, debe procederse a realizar una detallada clasificacion de las tareas que debe realizar el microcontrolador, para cada aplicación en particular.
Estas pueden ser divididas en tareas que tienen restricciones estrictas en tiempo (adquisición de datos, por ejemplo) y tareas que pueden ser interrumpidas sin consecuencias graves para el conjunto.
El grupo de tareas con restricciones de tiempo puede subdividirse en tareas sincrónicas y asincrónicas. Las sincrónicas pueden ser relacionadas, en general, con las interrupciones provenientes de los timers internos del microcontrolador, los cuales se utilizarán para proveer la base de tiempo. Las tareas asincrónicas están asociadas, en general, a eventos externos que pueden utilizarse para disparar las interrupciones externas del microcontrolador.
Se propone la implementación de un sistema del tipo foreground/background, donde las tareas con restricciones de tiempo (tareas de tiempo real) estén asociadas a interrupciones de hardware (foreground) y las tareas sin restricciones de tiempo constituyan el loop de background del programa, que es constantemente interrumpido por las tareas de foreground. Esto permite utilizar el manejo de interrupciones del microcontrolador para administrar las tareas a ejecutar simultáneamente.
En el grupo de tareas de background deben incluirse las tareas que puedan tolerar interrupciones de larga duración. Estas pueden ser, por ejemplo, la atención de la

interfaz con el usuario o almacenamiento de datos. Incluso puede no haber ninguna tarea en el background.
En el grupo de tareas de foreground deben considerarse las tareas que se realizarán dentro de un esquema preemptivo basado en prioridades, lo cual significa que dichas tareas deben estar asociadas a interrupciones de hardware, ya sean externas o provenientes de los timers del microcontrolador, y que deben asignarse prioridades a estas tareas.
Debe tenerse en cuenta que en esta configuración, la llegada de gran cantidad de interrupciones puede llegar a retardar excesivamente los procesos de background, incluso puede suceder que estos se suspendan por completo. De la misma manera debe chequearse la tasa de arribo de las interrupciones de mas alto nivel de prioridad, pues retardan las de prioridad mas baja.
Dentro del esquema preemptivo, una interrupción no puede suspender a otra de igual nivel de prioridad (o sea que no se puede interrumpir a sí misma), y se desea que el tiempo de respuesta a una interrupción se mantenga siempre constante a lo largo del tiempo. Por lo tanto, no pueden acumularse interrupciones del mismo tipo. Si una tarea está siendo realizada y llega nuevamente un pedido para realizar la misma tarea, esto indica que la capacidad de respuesta ha sido agotada y debe replantearse nuevamente el problema. Si la segunda llegada fuera encolada y atendida luego, el tiempo de respuesta de la primera atención sería diferente de el de la segunda.
En caso que se desee incluir en el esquema preemptivo una tarea asincrónica que no esté directamente asociada a una interrupción de hardware, sino a una condición de software, puede procederse de la siguiente manera: se asocia esta tarea a una interrupción externa y se activa dicha interrupción a través de un contacto eléctrico entre la entrada de dicha interrupción al microcontrolador y un bit cualquiera de uno de los ports de salida. Así, activando un bit por software, se produce una interrupción de hardware que ejecuta la tarea asociada.
En el punto siguiente se describen los recursos que generalmente estan disponibles en los microcontroladores, y que nos permiten proceder con esta implementación.

1.3. Recursos disponibles

Para la implementación se dispone de un microcontrolador de propósitos generales que en general incluye los recursos enumerados a continuación, relacionados con el manejo de interrupciones [2][4][5]. Estos recursos son similares para distintas familias de microcontroladores de distintos fabricantes. En el desarrollo se hará referencia al microcontrolador 8051 de Intel.

1. Cinco fuentes de interrupción: dos interrupciones externas, dos interrupciones internas provenientes de timers, una interrupción de puerta serie. Las interrupciones son atendidas por medio de un llamado a una subrutina, que a su vez puede llamar a otras rutinas, las cuales pueden ser implementadas reentrantes.
2. Un registro de manejo de prioridad de dichas interrupciones (IP) que permite solamente dos niveles de prioridad. Cuando llega una interrupción de menor o igual prioridad que la que está siendo atendida, ésta no es descartada, sino que queda pendiente su atención. Dentro de un mismo nivel, las diferentes interrupciones tienen diferentes prioridades, con el objeto de resolver interrupciones simultáneas. En orden decreciente, estas prioridades son: Interrupción Externa 0, Interrupción del Timer 0, Interrupción Externa 1, Interrupción del Timer 1.
3. Un registro de habilitación/deshabilitación de interrupciones (IE). Incluye un bit (EA) que deshabilita todas las interrupciones a la vez.

Estos recursos no son suficientes para implementar un scheduler y un dispatcher, ya que se desean varios niveles de prioridad y no solo dos como provee el microcontrolador. Por lo tanto debe escribirse una parte del programa que se

encargue de la administración.
Antes de pasar a las dos experiencias realizadas, se describe a continuación la plataforma sobre la cual se realizaron.

1.4. Descripción del caso de aplicación

Se desea utilizar el microcontrolador 8051 de Intel para controlar la temperatura de una planta. Para ello debe realizar varias tareas simultáneamente: adquisición de la temperatura actual implementando un convertidor doble rampa discreto y cálculo de la acción de control, implementación de un PWM por software, y por último manejo de display y teclado. El objetivo de esta implementación es la minimización de los periféricos con que debe contar el microcontrolador, con el objeto de minimizar costos de hardware haciendo un uso intensivo del software disponible.
Se desea utilizar el conversor A/D doble rampa que se muestra en la figura 1 [3]. Su característica principal es que fue implementado en forma discreta para su utilización con un microcontrolador, de forma tal de ocupar la mínima cantidad de recursos de éste: utiliza como base de tiempo uno de los timers internos del microcontrolador (Timer0), una de las interrupciones externas para detectar la finalización de la conversión (Int0), uno de los bits de cualquiera de los ports para comandar el cambio de sentido de la rampa (PA.1) y otro para el multiplexor de entrada (PA.3, no se presenta en la figura). El objetivo de dicho multiplexor es adquirir la señal proveniente de una termocupla y de un sensor de temperatura para la compensación de junta fría de la termocupla.
Para ello sólo utiliza los siguientes componentes externos: un integrado con cuatro llaves analógicas, un amplificador operacional cuádruple, una referencia de tensión, resistencias y capacitores.
Tiene implementado un procedimiento de autocero con pocos dispositivos adicionales y utilizando un bit más de un port (PA.0), lo que permite trabajar con la precisión requerida (15 bits), independizándose de los corrimientos de los amplificadores utilizados para la termocupla.
Para implementar el modulador de ancho de pulso (PWM) se utiliza como base de tiempo un timer (Timer 1) y un bit de salida de uno de los ports (PA.2) activa la etapa de potencia.

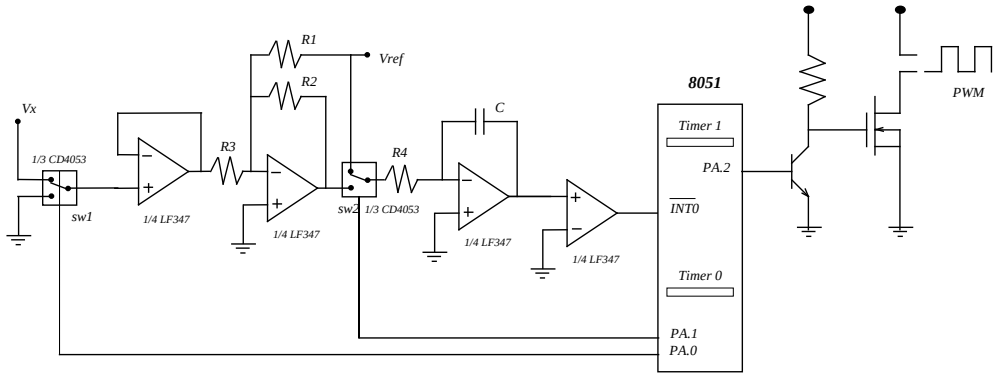
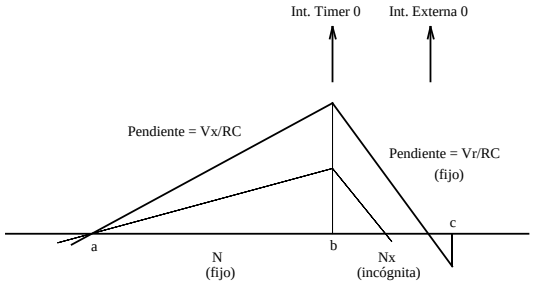


Fig. 1

El funcionamiento se describe a continuación. Se carga el Timer 0 con un número fijo N elegido de antemano, que es el fondo de escala del conversor A/D (por ejemplo 4096 para conversión a 12 bits) y se conmuta PA.1 para que aplique la tensión Vx de entrada a la rampa, generada por el capacitor. Cuando han transcurrido los N ciclos de clock, llega la interrupción del timer (la cual se genera al llegar la cuenta a cero). En la rutina de atención del timer se conmuta PA.1 para que aplique -Vref a la rampa.

La interrupción externa 0 aparece cuando la rampa ha llegado a cero. Esta llegada a cero funciona además como gate del timer 0, en uno de los diferentes modos de programación de éste. Un esquema de la evolución de la rampa con el tiempo puede verse en la figura 2.



$$\frac{Vx}{R.C}.N = \frac{Vr}{RC}.Nx \qquad Vx = Vr.\frac{Nx}{N}$$

Fig. 2

Para aclarar el funcionamiento de este sistema, se presenta en la figura 3 la red de Petri correspondiente, utilizando la extensión en tiempo de ésta. Se incluyen todas las tareas. Los tokens se encuentran en un estado tal que se está aguardando la finalización del conteo de ambos timers (este es el estado mas frecuente en que se encuentra el sistema) y se están ejecutando las tareas de background.

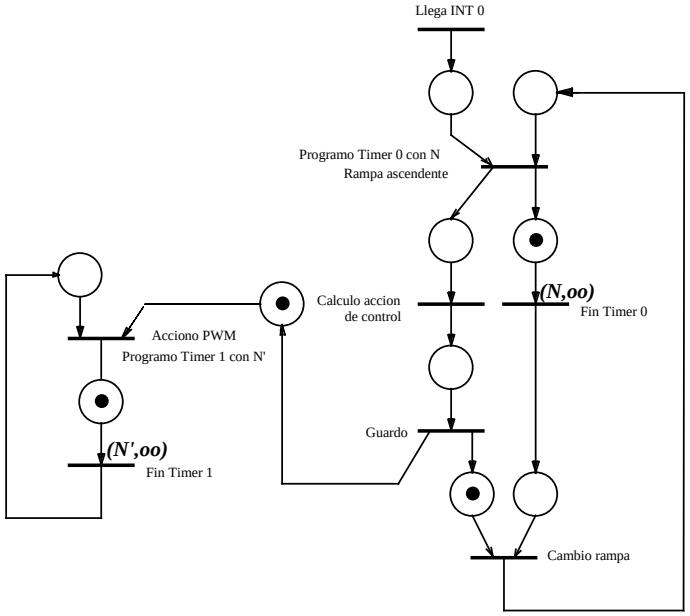


Fig. 3

El objetivo es implementar un sistema de software que sea capaz de manejar todas estas tareas simultáneamente, con tiempos de respuestas acotados.
Un primer intento fue la implementación de un sistema secuencial, del tipo polled loop, pero no se lograron resultados satisfactorios por dos motivos fundamentalmente. En primer lugar, el ciclo de conversión del A/D es muy diferente del correspondiente al PWM. En segundo lugar, el tiempo de conversión de un A/D doble rampa es variable, y por lo tanto la sincronización del sistema presentó inconvenientes.
A continuación se describen las dos estrategias basadas en interrupciones empleadas para lograr este objetivo. La descripción se realiza primero en términos que intentan ser generales, y luego se detalla la aplicación específica al sistema de hardware

descripto anteriormente.

2. Implementación de un sistema cooperativo

Esta implementación no utiliza un kernel. Se trata simplemente de varias tareas que se ejecutan en un esquema preemptivo basado en prioridades, pero sin un ente administrador principal. Las mismas tareas, en una base cooperativa, realizan la administración del scheduler y del dispatcher[1], de la forma que se describe a continuación.

2.1. Implementación del dispatcher de tareas

Se utiliza como dispatcher al mecanismo propio del que disponen los microcontroladores de asociar la ejecución de una subrutina a la llegada de una interrupción. Para ello no debe escribirse programa adicional. El microcontrolador, ante la llegada de una interrupción, produce un salto a una dirección prefijada y conocida para cada interrupción. En esa dirección puede incluirse un nuevo salto a la dirección de comienzo de la rutina de atención, que es la tarea que se desea realizar. El context switch que realiza el microcontrolador antes de producir el salto a la rutina de atención de una interrupción es únicamente del program counter (PC). El resto de los registros que sean modificados deben ser salvados en el stack por la propia subrutina. Esta estrategia presenta un tiempo de context switch variable para las diferentes rutinas, pero que puede ser calculado dentro del tiempo de ejecución éstas.

2.2. Implementación de la comunicación entre procesos

La implementación de la comunicación entre tareas se realiza por medio de la utilización de variables globales, que pueden se accedidas por todas las tareas. Es importante proteger el acceso a estas variables si son de mas de 8 bits. Aunque el acceso a un registro o una posición de memoria de 8 bits es atómica, si una variable global es de 16 bits, el acceso a sus dos bytes puede ser interrumpido, luego de la lectura del primero, por otra tarea que modifique esta variable, obteniéndose un dato incorrecto. Para ello puede optarse por la utilización de un semáforo o simplemente por la inhibición de las interrupciones durante el doble acceso (lo cual produce un retardo de tres ciclos de instrucción para las tareas de mas alta prioridad, en caso que lleguen en ese instante). Las variables globales no son el método mas elegante de comunicación entre procesos y deben utilizarse con precaución. Lo que es cierto es que son el método mas rápido y sencillo, el cual acompañado con una programación cautelosa, suele ser la solución mas adecuada para una gran parte de las aplicaciones que puedan presentarse.

2.3. Implementación del scheduler de tareas

Como dijimos anteriormente, es deseable que la estrategia de scheduling sea preemptiva y basada en prioridades. El microcontrolador dispone de un registro para el manejo de las prioridades de las interrupciones: IP. Este registro solo permite asignar prioridad 0 o 1 a las interrupciones, lo cual puede ser suficiente en la mayoría de los casos. Utilizar solo dos niveles de prioridades puede ser útil en los casos que una de las tareas sea crucial (se le asigna prioridad 1), un grupo de tareas tenga restricciones de tiempo menos estrictas (se les asigna prioridad 0), y un grupo de tareas no tenga restricciones en tiempo (background). En el caso que se necesiten más niveles de prioridad, la solución es que las propias tareas manipulen constantemente el registro IP en una forma cooperativa, habilitando o deshabilitando

la posibilidad de ser interrumpidas por otras tareas. Lo ideal es que se disponga de tantos niveles de prioridad como tareas deban ejecutarse.
El inconveniente de esta implementación es que requiere una cuidadosa programación, ya que cualquier tarea tiene el control total del sistema. La ventaja es que no utiliza un kernel ejecutivo adicional para la administración de las tareas, lo cual nos libra de retardos adicionales en el cambio de una tarea a la otra. Además el context switch no es automático, ya que debe realizarse manualmente al ejecutarse cada tarea, pero tiene la ventaja de no ser necesario el almacenamiento de todo el contexto, sino solo de los registros que serán utilizados, lo cual acelera el tiempo de respuesta.
Este tipo de configuración no siempre es aplicable, ya que se requiere un alto grado de conocimiento acerca de la frecuencia de ejecución de las tareas, como así también una importante sincronización entre éstas, ya que las propias tareas habilitan la ejecución de las demás a través de un manejo de prioridades. Para ello el nivel de cooperación entre las tareas debe ser alto.

2.4. Implementación práctica

En primer término, se asociaron las tareas con las interrupciones de la siguiente forma:

1. Background: manejo de display y teclado.
2. Interrupción del timer 0: cambio del sentido de la rampa del conversor A/D discreto utilizando un bit de uno de los ports de salida. Esta tarea es la que requiere mayor prioridad y debe ser capaz de interrumpir a las demás. Esto se debe a que un retardo en la ejecución del cambio de rampa produce un error en la adquisición de la temperatura actual. Esta tarea tiene la ventaja de ser la que menos instrucciones tiene (esto es muy común en sistemas de tiempo real)
3. Interrupción externa 0: detección del fin de conversión. La tarea asociada a ésta interrupción debe incluir el cálculo de la acción de control, y debe poner a disposición de la tarea que se encarga del PWM el valor de ésta, utilizando variables globales. Además debe presentar al proceso de background los valores de temperatura para enviar al display. No es necesario que esta tarea tenga una prioridad tan alta como la anterior, ya que al ponerse en cero el pin INT0 el timer se detiene, y no produce error. Debe permitir ser interrumpida por la interrupción del timer 0 y del timer 1.
4. Interrupción del timer 1: manejo del ciclo del PWM. Tiene una prioridad intermedia entre las dos anteriores. Tiene que ser capaz de interrumpir a INT_0.

Aquí estamos ante el caso en que se necesitan tres prioridades diferentes, además del background, lo cual se logró manipulando el registro IP como se muestra en el listado. En negrita pueden verse dichos cambios. Este programa incluye el redireccionamiento de las rutinas de atención de interrupciones y el procedimiento de inicialización de los recursos utilizados del microcontrolador. Algunas de las tareas (como por ejemplo la atención del display) no se describe en detalle debido a que varía notablemente con las necesidades de cada implementación. Estas se incluyen solamente como llamados a subrutinas.

```
                                ; REDIRECCIONAMIENTO DE LAS INTERRUPCIONES:
            ORG 0000h
            LJMP Init           ; Reset.
            ORG 0003h
            LJMP INT_0          ; Interrupción externa 0.
            ORG 000Bh
            LJMP INT_T0         ; Interrupción del Timer 0.
            ORG 001Bh
            LJMP INT_T1         ; Interrupción del Timer 1.

            ORG 0100h

Init:
            LCALL Init_All

Loop_Bkgnd:                     ; LOOP DE BACKGROUND:
                                ; Aquí ejecuto el background.
            LCALL Display       ; Refresco display tomando el dato de las variables globales.
            LCALL Teclado       ; Teclado no manejado por interrupciones.
            AJMP Loop_Bkgnd
```

```
INT_O:                          ; FIN DE CONVERSION:
                                ; Interrupción 0: routina de atención.
            LCALL Context_Switch    ; Salvo los registros que voy a utilizar.
            LCALL Multiplex         ; Preparo el multiplexor para la proxima conversión.
            LCALL Set_Timer_0       ; Reprogramo y arranco el timer.
            SETB  PA.1              ; Cambio a Vx.
            LCALL Calculo_PWM       ; Calculo la acción de control.
            LCALL Out_Data          ; Almaceno los cálculos en variables globales.
            MOV IP,#00000010B           ; Doy prioridad al timer 0 hasta que se produzca la
                                ; interrupcion INT_T0.
            LCALL Context_Switch    ; Recupero los registros que guardé.
            RETI


INT_TO:                         ; CAMBIO DE PENDIENTE:
                                ; Interrupción Timer 0: routina de atencion.
            CLR PA.1            ; Cambio a Vref.
            MOV IP,#00001000B           ; Doy prioridad al timer 1 hasta que se produzca la
                                ; interrupcion INT_0.
            RETI


INT_T1:                         ; MANEJO DEL PWM:
                                ;
            LCALL Context_Switch    ; Salvo los registros que voy a utilizar.
            CLR TR1             ; Paro el timer.
            LCALL On_Off ; Comando la salida del PWM en On u Off segun correspionda.
            LCALL Context_Switch    ; Recupero los registros que utilice.
            RETI
```

Listado 1

Nótese que las tres tareas conviven simultáneamente con tres niveles diferentes de prioridad. El máximo nivel de prioridad es para INT_T0. La INT_0 se deja interrumpir por INT_T1, manipulando IP en conjunto con INT_T0. El context switch es diferente en cada caso, y en INT_T0 no se realiza context switch en absoluto, pues no es necesario. Esto acelera el tiempo de respuesta de dicha interrupción.
Aquí puede verse el alto grado de conocimiento acerca del funcionamiento del sistema que es necesario, y algunas condiciones que deben cumplirse para lograr un adecuado funcionamiento.
En la figura 4 se muestra la secuencia de eventos y la manipulación del registro de prioridades para el caso cooperativo.

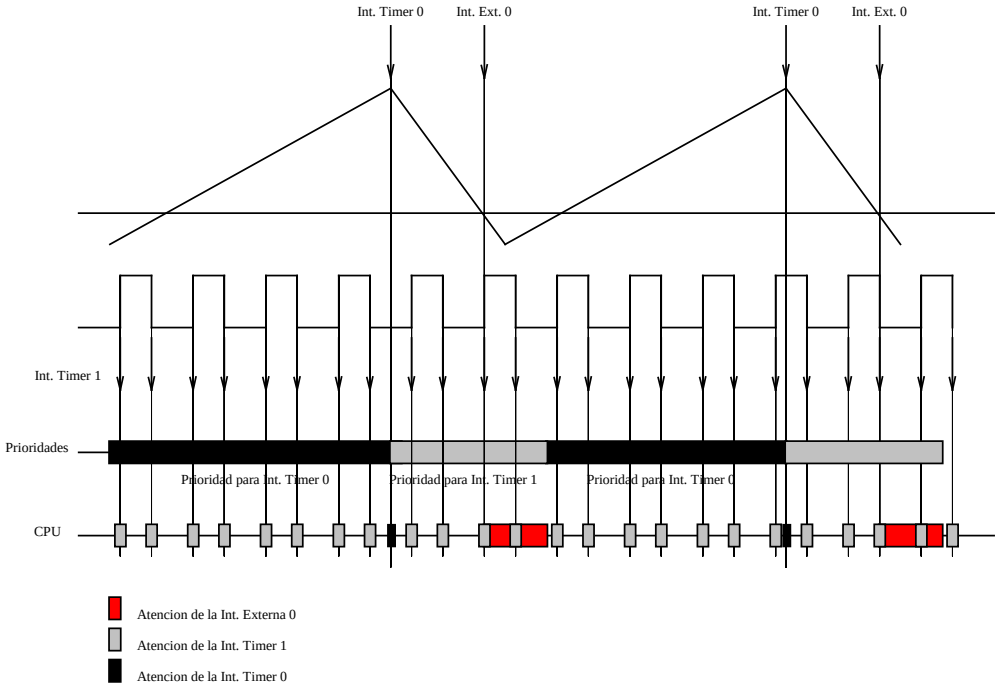


Fig. 4

En primer lugar es necesario que las interrupciones se manipulen con cuidado y en forma cooperativa, pues si, por ejemplo, INT_T0 no realiza el cambio de prioridades adecuadamente, INT_T1 no podrá interrumpir la ejecución de INT_0, lo cual se

evidenciará en una modificación del período del PWM.
También es indispensable un alto grado de sincronización entre INT_T0 e INT_0 para poder manipular IP en conjunto.
Por estos motivos, es posible encontrar casos en los que una implementación de este tipo sea imposible, dado que no se cumplen las condiciones para la cooperación. En ese caso es necesario agregar una nueva tarea con control sobre las demás, que se encarga de manipular las prioridades en función de los requisitos de tiempo establecidos.
Esta tarea se agregó en la implementación que se describe a continuación.

3. Implementación de un kernel simple de tiempo real

Un kernel es una tarea de alta prioridad que tiene la capacidad de administrar la ejecución del resto de las tareas del sistema. Es un sistema operativo reducido. Debe proveer tres funciones fundamentales, tal cual se describió anteriormente: scheduling, dispatching y comunicación entre procesos.
Existen productos comerciales muy versátiles de carácter general que permiten implementar kernels de tiempo real en microcontroladores. Pero es esa generalidad y versatilidad lo que los hace voluminosos, pues proveen funciones que muchas veces no son utilizadas. En la mayoría de los casos, una programación que utilice los conceptos básicos de funcionamiento de un kernel ayuda a solucionar los problemas de tiempo sin necesidad de incluir un sistema operativo completo al sistema.
Se propone una *estrategia* de programación que utilice los conceptos básicos del funcionamiento de un kernel, con el objeto de cumplir los requerimientos de tiempo del sistema. Con una programación ordenada y un buen uso de las interrupciones y prioridades puede lograrse, sin la adición de una gran cantidad de código, un verdadero funcionamiento multitarea con un esquema preemptivo basado en prioridades. Este esquema es, como ya dijimos, el más adecuado para la mayoría de las aplicaciones con restricciones de tiempo importantes (hard real-time systems).

3.1. Implementación del dispatcher de tareas

Puede implementarse como una nueva subrutina que automáticamente envíe a un stack todas las variables utilizadas por las rutinas y mantenga un registro de las tareas que están en ejecución.
Nótese que en el ejemplo de aplicación, la atención de la rutina con mayores restricciones en tiempo (INT_T0) no necesita context switch (solo el program counter), mientras que la atención de INT_0 necesita que sean salvados varios registros. Si automatizáramos el context switch, en la atención de INT_T0 (que sólo tiene una instrucción) estaríamos ejecutando instrucciones innecesarias.
Guardar automáticamente el contexto implicaría guardar más variables de las necesarias y por lo tanto desmejorar el tiempo de atención de las interrupciones. Por ello se optó por continuar con un context switch ajustado para cada tarea, perdiéndose generalidad pero ganando en velocidad.
El registro de las tareas que se están ejecutando se implementó en el scheduler.

3.2. Implementación de la comunicación entre procesos

Al igual que en la implementación anterior, la comunicación entre tareas se realiza por medio de la utilización de variables globales, que pueden se accedidas por todas las tareas y que deben estar protegidas si son de mas de un byte.

3.3. Implementación del scheduler de tareas

Se escribió una nueva tarea que es la encargada de decidir cuál es la próxima tarea que debe ejecutarse y (ya que no se implementó un dispatcher) de mantener un registro del estado de las distintas rutinas. Para ello se direccionaron todas las interrupciones a la tarea Kernel (ver listado 2).

Al llegar una interrupción, dicho kernel debe verificar si la tarea que está corriendo tiene mayor o menor prioridad que la tarea asociada a la interrupción. De ser menor, debe interrumpir la tarea que está ejecutando y debe iniciar la nueva, para luego retomar la suspendida. En el caso que llegue una de menor prioridad debe marcar como pendiente dicha interrupción, para que sea ejecutada al finalizar las de prioridad mayor. Finalmente debe retornar de la interrupción.

Cada vez que se termina de ejecutar una tarea, el scheduler debe verificar la existencia de tareas pendientes en orden de prioridad, antes de devolver el control al loop de background. La ejecución de esta tarea adicional implica un aumento del tiempo de respuesta a las interrupciones, ya que todas las interrupciones deben pasar antes por el kernel. Debe procurarse que este tiempo sea lo mas pequeño posible y, fundamentalmente, que sea constante para todas las tareas, a fin de poder acotar los tiempos de respuesta.

Esta estrategia se implementó de la siguiente manera: la atención de todas las interrupciones se hace llamando a la rutina Kernel, utilizando un arreglo de flags para indicar cuál es la interrupción que ha llegado. La tarea Kernel hace un poll por los flags, de mayor a menor prioridad. Cuando detecta que ha llegado una interrupción verifica que no haya una tarea de mayor prioridad ejecutándose, y ejecuta la tarea asociada. Si está siendo ejecutada una de mayor prioridad la marca como pendiente y retorna de la interrupción. Cuando la tarea de mayor prioridad sea terminada, el Kernel se encontrará con la de menor prioridad pendiente y la ejecutará antes de retornar de la interrupción anterior.

Para que todas las tareas puedan ser interrumpidas por el Kernel, éste debe utilizar el registro de prioridades de interrupciones del microcontrolador para asignarle mínima prioridad a la tarea que está siendo ejecutada.

En el listado 2 se presenta una implementación en pseudo-código para demostrar su funcionamiento.

```
Int1:       Setear_Ready1                       ; Llega Interrupción 1 (maxima prioridad)
            JUMP Kernel

Int2:       Setear_Ready2                       ; Llega Interrupción 2 (prioridad intermedia)
            JUMP Kernel

Int3:       Setear_Ready3                       ; Llega Interrupción 1 (minima prioridad)
            JUMP Kernel



Kernel:                                         ; NOTA: El orden de este polling es el orden de
                                                ; prioridades de las rutinas.
                                                ; Utilizo los flags Ready* y Running* como Taks Status.

poll1:      IFNOT Ready1 JUMP poll2  ; Miro si llego la interrupcion 1, si no miro la proxima.
            IF Running1 JUMP scherror           ; Si llego y esta todavia ejecutando, salgo con error.
            Limpiar_Ready1                      ; Paso la tarea a modo ejecutando.
            Setear_Running1
            Bajar_Prioridad_1                   ; Mientras esta corriendo le bajo la prioridad.
            EXEC Tarea_1            ; Ejecuto la tarea asociada a la interrupción 1.
            Subir_Prioridad_1                   ; Repongo la prioridad alta.
            Limpiar_Running1                    ; Aviso que termino de correr la tarea 1.
            JUMP Kernel                         ; Termino. No debo salir hasta que no queden
                                                ; interrupciones pendientes.

poll2:      IFNOT Ready2 JUMP poll3  ; Miro si llego la interrupcion 2, si no miro la proxima.
            IF Running2 JUMP pollerror          ; Si llego y esta todavia ejecutando, salgo con error.
            IF Running1 JUMP pollexit           ; Verifico si está corriendo la 1.
                                                ; De ser asi marco la tarea como pendiente y salgo.
            Limpiar_Ready2                      ; Idem anterior.
            Setear_Running2
            Bajar_Prioridad_2
            EXEC Tarea_2
            Subir_Prioridad_2
            Limpiar_Running2
            JUMP Kernel

poll3:      IFNOT Ready3 JUMP pollexit          ; Miro si llego la interrupcion 3, si no termino.
            IF Running3 JUMP pollerror          ; Si llego y esta todavia ejecutando, salgo con error.
            IF Running1 JUMP pollexit           ; Verifico si están corriendo la 2 o la 1. De ser asi
            IF Running2 JUMP pollexit           ; marco la tarea como pendiente y salgo.

            Limpiar_Ready3                      ; Idem anterior.
            Setear_Running3
            Bajar_Prioridad_3
            EXEC Tarea_3
            Subir_Prioridad_3
            Limpiar_Running3
```

```
            JUMP Kernel

pollerror:  EXEC Tarea_Error
pollexit:   Return_from_Interrupt              ; Devuelvo el control al background.
```

Listado 2

En la figura 5 se presentan los cuatro estados en los que pueden encontrarse cada una de las tareas, junto con las condiciones que deben cumplirse para que se modifiquen dichos estados.

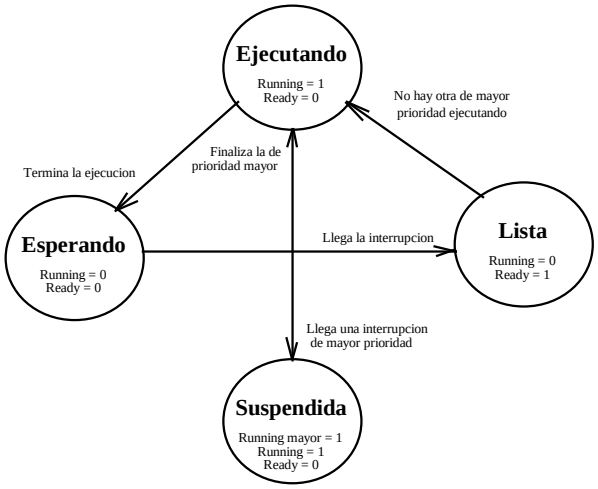


Fig. 5

3.4. Implementación práctica

El software para el ejemplo anterior de aplicación se replanteó con el objeto de utilizar este pequeño kernel de tiempo real. Se utilizó la misma asignación de tareas a interrupciones que en el punto 2.4.

Nótese que se eliminaron los manejos del registro IP dentro de las rutinas de atención de las interrupciones.

El listado 3 se presenta en assembler para el microcontrolador 8051 de Intel, y las aclaraciones que siguen a continuación son específicas para éste.

Antes de ejecutar una tarea debe bajarse el nivel de prioridad de ésta, para que pueda ser interrumpida por las demás. Para ello, debe cambiarse el nivel de prioridad de la interrupción durante su ejecución. Los cambios en las prioridades no son validados hasta el fin de una interrupción, por lo cual se ejecuta una instrucción RETI con un CALL, para forzar esta actualización de prioridades.

```
REDIRECCIONAMIENTO DE
LAS INTERRUPCIONES:

            ORG 0000h
            LJMP Reset              ; Reset.

                                    ; Todas las interrupciones saltan al kernel,
                                    ; previo aviso de cual llego con Ready.*
                                    ; (asignar Ready.1 a la interrupcion de maxima
                                    ; prioridad y Ready.4 a la de minima)
            ORG 0003h
            SETB Ready.3; Minima prioridad para la
            LJMP Kernel             ; interrupcion externa 0.

            ORG 000Bh
            SETB Ready.1; Maxima prioridad para la
            LJMP Kernel             ; interrupcion del Timer 0.

            ORG 001Bh
            SETB Ready.2; Prioridad intermedia para la
            LJMP Kernel             ; interrupcion del Timer 1.


            ORG 0100h

Kernel:
                                    ; NOTA: El orden de este polling es el orden de
                                    ; prioridades de las rutinas.

poll1:      JNB Ready.1,poll2       ; Miro si llego la interrupcion del timer 0, si no miro la proxima.
```

```
            JB Run.1,pollerr        ; Si llego y esta todavia ejecutando, salgo.
            CLR Ready.1             ; Atiendo la tarea asociada al timer 0.
            SETB Run.1              ; Aviso que esta corriendo.
            CLR IP.1                ; Mientras esta corriendo le bajo la prioridad.
            CALL pollex             ; Para que se active el cambio de IP debo ejecutar un RETI.
            LCALL INT_T0; Mando a ejecutar.
            SETB IP.1               ; Repongo la prioridad alta.
            CALL pollex             ; Ejecuto RETI.
            CLR Run.1               ; Aviso que termino de correr.
            AJMP Kernel             ; Termino. No debo salir hasta que no queden
                                    ; interrupciones pendientes.

poll2:      JNB Ready.2,poll3       ; Miro si llego la interrupcion del timer 1, si no miro la proxima.
            JB Run.2,pollerr        ; Si llego y esta todavia ejecutando, salgo.
            JB Run.1,pollex         ; Si esta Run.2 vuelvo luego.
            CLR Ready.2             ; Atiendo la tarea asociada al timer 1.
            SETB Run.2              ; Aviso que esta corriendo.
            CLR IP.3                ; Mientras esta corriendo le bajo la prioridad.
            CALL pollex             ; Para que se active el cambio de IP debo ejecutar un RETI.
            LCALL INT_T1; Mando a ejecutar.
            SETB IP.3               ; Repongo la prioridad alta.
            CALL pollex             ; Ejecuto RETI.
            CLR Run.2               ; Aviso que termino de correr.
            AJMP Kernel             ; Termino. No debo salir hasta que no queden
                                    ; interrupciones pendientes.

poll3:      JNB Ready.3,pollex      ; Miro si llego la interrupcion externa 0, si no termino.
            JB Run.3,pollerr        ; Si llego y esta todavia ejecutando, salgo.
            JB Run.1,pollex         ; Verifico si esta Run.2 o Run.3. De ser asi
            JB Run.2,pollex         ; vuelvo luego.
            CLR Ready.3             ; Atiendo la tarea asociada a la interrupcion externa 0.
            SETB Run.3              ; Aviso que esta corriendo.
            CLR IP.0                ; Mientras esta corriendo le bajo la prioridad.
            CALL pollex             ; Para que se active el cambio de IP debo ejecutar un RETI.
            LCALL INT_0             ; Mando a ejecutar.
            SETB IP.0               ; Repongo la prioridad alta.
            CALL pollex             ; Ejecuto RETI.
            CLR Run.3               ; Aviso que termino de correr.
            AJMP Kernel             ; Termino. No debo salir hasta que no queden
                                    ; interrupciones pendientes.

pollerr:    LCALL  Error; Hacer algo que indique overflow!

pollex:     RETI                    ; Regreso de la interrupcion.
                                    ; Tambien uso esta instruccion como habilitador del cambio
                                    ; de prioridades, llamandola con CALL.


PROGRAMA:

Reset:      LCALL Init_All

Loop_Bkgnd:                         ; LOOP DE BACKGROUND:
                                    ; Aquí ejecuto el background.
            LCALL Display           ; Refresco display tomando el dato de las variables globales.
            LCALL Teclado           ; Teclado no manejado por interrupciones.
            AJMP Loop_Bkgnd

INT_O:                              ; FIN DE CONVERSION:
                                    ; Interrupción 0: routina de atención.
            LCALL Context_Switch    ; Salvo los registros que voy a utilizar.
            LCALL Multiplex         ; Preparo el multiplexor para la proxima conversión.
            LCALL Set_Timer_0       ; Reprogramo y arranco el timer.
            SETB  PA.1              ; Cambio a Vx.
            LCALL Calculo_PWM       ; Calculo la acción de control.
            LCALL Out_Data          ; Almaceno los cálculos en variables globales.
            LCALL Context_Switch    ; Recupero los registros que guardé.
            RETI


INT_TO:                             ; CAMBIO DE PENDIENTE:
                                    ; Interrupción Timer 0: routina de atencion.
            CLR PA.1                ; Cambio a Vref.
            RETI


INT_T1:                             ; MANEJO DEL PWM:
                                    ;
            LCALL Context_Switch    ; Salvo los registros que voy a utilizar.
            CLR TR1                 ; Paro el timer.
            LCALL On_Off; Comando la salida del PWM en On u Off segun correspionda.
            LCALL Context_Switch    ; Recupero los registros que utilice.
            RETI
```

Listado 3

4. Limitaciones

El trabajo hasta aquí expuesto no intenta ser una implementación general de un kernel multitarea de tiempo real, sino que propone una estrategia de programación basada en los conceptos básicos de un kernel de este tipo. Conociendo los principios de funcionamiento de un kernel general, pueden aplicarse a una programación ordenada a fin de explotar al máximo los recursos disponibles en los microcontroladores de propósitos generales. Esta estrategia permite acotar los tiempos de respuesta de un sistema lo cual es altamente beneficioso para las aplicaciones de tiempo real.

La principal desventaja es que el número de tareas está limitado por el número de interrupciones disponibles en el microcontrolador. Además dicho número no puede ser modificado durante la ejecución del programa. Esto limita el campo de aplicación de lo anteriormente expuesto.

Por otro lado, no se justifica extenderse demasiado en el perfeccionamiento y la generalización del scheduler y el dispatcher de tareas, ya que se estaría reescribiendo un sistema operativo de tiempo real, lo cual no es la intención de este trabajo.

El campo de aplicación de esta estrategia de programación está limitado a aplicaciones en las que el microcontrolador efectúa tareas simples que tienen alta interrelación con dispositivos externos. En el caso que la programación se complique, es recomendable utilizar un sistema operativo de los que se encuentran disponibles para las diferentes arquitecturas de microcontroladores.

5. Conclusiones

Las experiencias realizadas aplicando esta estrategia a diferentes casos demostraron que con pocas lineas adicionales de programa es posible implementar una estrategia pereemptiva basada en prioridades en un microcontrolador de propósitos generales. Sólo es necesario un conocimiento a fondo de los recursos disponibles y una cuidadosa programación.
Esta estrategia permite optimizar la utilización de la CPU, evitando tiempos muertos indeseables y, por lo tanto, permitiendo la resolución de problemas complejos con dispositivos standard, que hubieran requerido microcontroladores mas potentes en el caso de utilizarse un polled loop.

6. Bibliografía

[1] Philip A. Laplante. ***Real-Time Systems Design and Analisis***. IEEE Press, 1992.
[2] ***MCS-51 Architectural Overview, Hardware Description of the 8051 and 8052, MCS-51 Programmers Guide and Instruction Set y MCS-51 Data sheets*****.** Intel Corp. 1983.
[3] E. Spinelli y A. Veiga. ***Dual Slope A/D converter for the 8051***. Electronic Engineering, Enero 1996.
[4] J. González Vázquez. ***Introducción a los Microcontroladores.*** McGraw Hill, 1992.
[5] J. Martínez Pérez. ***Prácticas con Microcontroladores***. McGraw Hill, 1993.
[6] Bill O. Gallmeister. ***POSIX.4 Programming for the real world***. O'Reilly & Associates, 1995.
[7] John A. Stankovic. ***Real-Time and Embedded Systems***. Departament of Computer Science, University of Massachusetts, Amherst, MA 01003.
[8] John A. Stankovic. ***Real-Time Computing***. Departament of Computer Science, University of Massachusetts, Amherst, MA 01003. Abril de 1992.
[9] Andrew S. Tanenbaun. ***Modern Operating Systems***. Prentice Hall Inc. 1992.