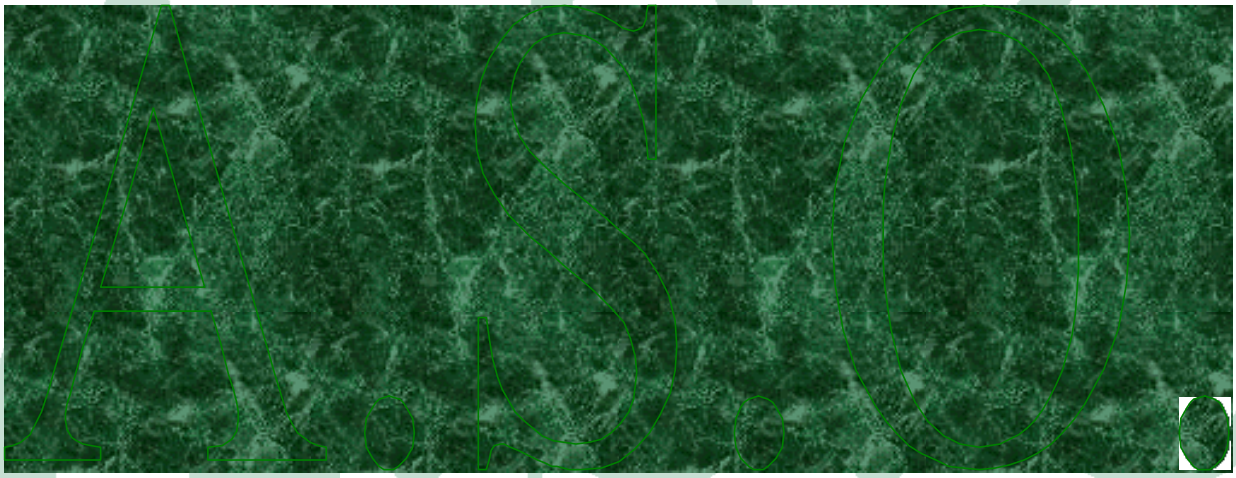


El Reloj

Patricia Afonso Godoy
María Teresa Cardona Perdomo

Indice

1.- INTRODUCCIÓN

Los relojes o temporizadores son un elemento imprescindible para el funcionamiento de todo sistema de tiempo compartido. Entre sus tareas más importantes se encuentran:

- mantener la hora del día.
- evitar la monopolización de la CPU por parte de un proceso.

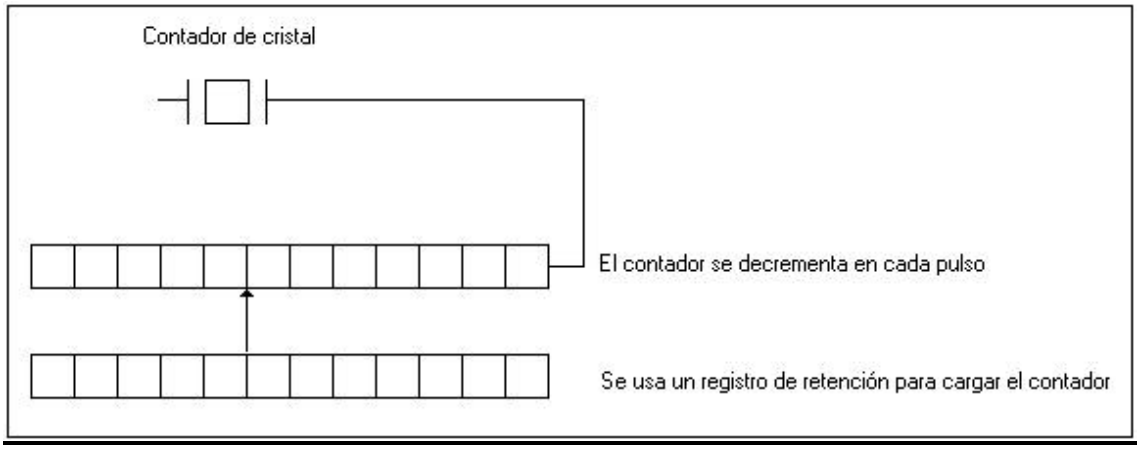
Aunque el reloj no sea un dispositivo por bloques, el software que lo implementa puede seguir el patrón de un controlador de dispositivo.

A continuación pasaremos a ver el hardware y el software de dicho dispositivo.

2.- HARDWARE DEL RELOJ

Desde el punto de vista del hardware podemos encontrarnos dos clases de reloj. La primera clase son los **conectados a la línea de potencia**. Estos son los más sencillos y se conectan a la línea de potencia de 110 o 220 V, causando una interrupción a cada ciclo de voltaje a 50 o 60 Hz. La segunda clase denominada **reloj programable**, consta de 3 componentes:

- un oscilador de cristal,
- un contador,
- un registro de retención.



Cuando el contador llegue a cero causa una interrupción de la CPU.

Toda computadora incluye al menos un circuito de este tipo, que proporciona una señal de sincronización a los diferentes circuitos de la computadora.

Existen 2 modos de operación importantes en los relojes programables. En el **modo de una acción**, al iniciar el proceso, se copia en el contador el valor del registro de retención, decrementando dicho contador en cada pulso del cristal. Cuando el valor del contador sea 0, causara una interrupción y se detendrá el reloj, esperando a que el software lo inicie otra vez.

En el **modo de onda cuadrada**, una vez el contador llegue a 0 y produzca la interrupción, se copiara nuevamente en el contador el registro de retención, repitiendo el proceso indefinidamente. A estas interrupciones periódicas se les denominan **tics** de reloj.

La ventaja de los relojes programables es que su frecuencia de interrupciones puede ser controlada por el software.

A fin de evitar que la hora actual se pierda cuando la computadora se apaga, se tiene un **reloj de respaldo de batería**, implementado con los tipos de circuito de baja potencia empleados en los relojes de pulsera digitales. El reloj de batería se leerá en el momento de arranque.

3.- SOFTWARE DEL RELOJ

Todo lo que hace el hardware del reloj es generar interrupciones en intervalos conocidos. Todo lo demás en que intervenga el tiempo, debe ser realizado por el software, el **controlador del reloj**.

Dependiendo del sistema operativo, pueden variar las obligaciones del controlador del reloj, aunque casi siempre incluyen:

1) Mantener la hora del día.
2) Evitar que los procesos monopolicen la CPU.
3) Contabilizar la utilización de la CPU.
4) Manejar la llamada al sistema ALARM emitida por procesos de usuario.
5) Proveer temporizadores de vigilancia.
6) Preparar perfiles, vigilar y recabar datos estadísticos.

3.1.- MANTENER LA HORA DEL DÍA (TIEMPO REAL):

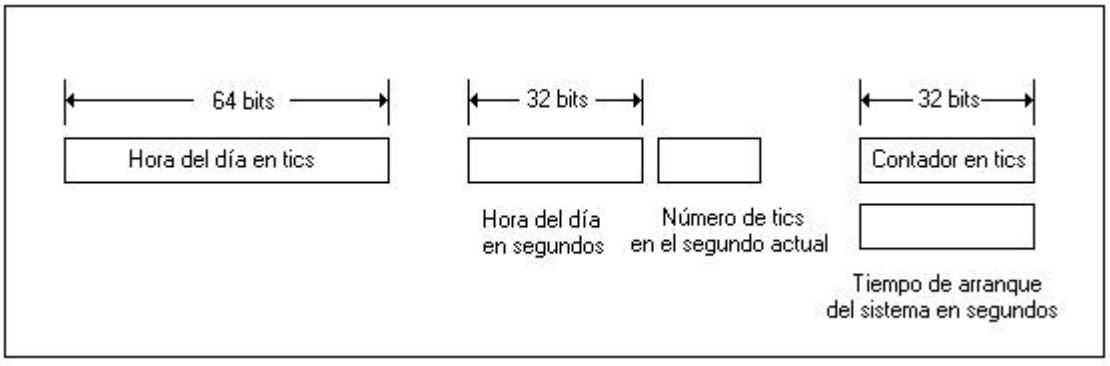
Para llevar a cabo este proceso, simplemente se requiere incrementar un contador en cada tic del reloj. Hay que saber el número de bits que tiene el contador de la hora del día. Sin embargo, vemos que con una tasa de reloj de 60 Hz y un contador de 32 bits, este se desbordaría en poco mas de 2 años.

Para solucionar este problema existen 3 estrategias:

A. Aumentar el contador a 64 bits, aunque esto provoca un mantenimiento más costoso de éste pues tiene que modificarse muchas veces cada segundo.

B. Mantener la hora del día en segundos, utilizando un contador alternativo que acumule el número de tics necesarios hasta alcanzar un segundo completo.

C. Contar en tics, pero hacerlo relativo al momento en que se arranco el sistema leyendo del reloj de respaldo o si el usuario introduce el tiempo real. De esta manera, cuando queramos saber la hora actual, sumaremos la hora almacenada más un contador inicializado en el arranque.



3.2.- EVITAR LA MONOPOLIZACIÓN DE LA CPU:

Cuando se inicia un proceso, el planificador del reloj inicializa un contador con el cuanto en tics para ese proceso.

Para cada interrupción del reloj, el controlador de éste decrementará en 1 el valor del contador. Cuando este contador llegue a tener valor 0, el controlador llamará al planificador que se encargará de poner en marcha otro proceso.

3.3.- CONTABILIZAR LA UTILIZACIÓN DE LA CPU:

Esto podemos realizarlo de dos maneras:

- Cada vez que se inicia un proceso se pone en marcha un segundo temporizador, distinto del temporizador principal del sistema. Cuando se detiene dicho proceso, se tiene la opción de leer el temporizador para determinar el tiempo de ejecución del mismo. Este segundo temporizador deberá guardarse cada vez que ocurra una interrupción para reestablecerse posteriormente y así determinar ciertamente el tiempo de ejecución del proceso.

- Una forma menos exacta, aunque mucho más simple, consiste en tener almacenado en una variable global un puntero a la tabla de procesos, que indique cual de ellos se esta ejecutando. En cada tic de reloj se incrementará el valor de un campo de la entrada del proceso. De esta forma, cada tic de reloj se "cobra" al proceso que se estaba ejecutando en ese momento. Un problema de esta estrategia es que si ocurren muchas interrupciones, se contabilizará al proceso en ejecución un tiempo que no ha utilizado (porque cuando ocurre una interrupción se seguirá incrementando este valor). Como nota final, decir que llevar la contabilidad del uso de la CPU durante las interrupciones es demasiado costoso, por lo que nunca se hace.

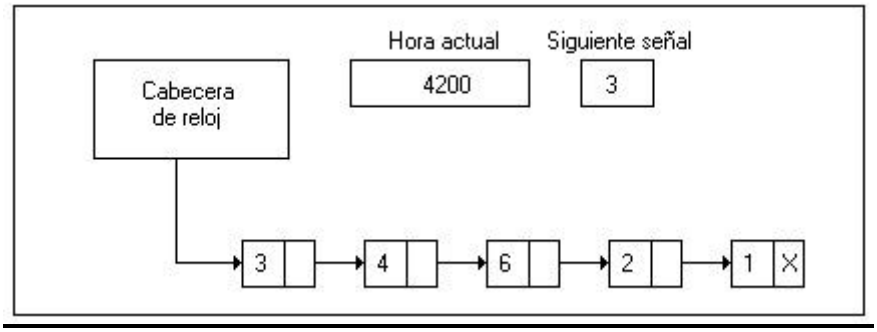
3.4.- MANEJAR LA LLAMADA AL SISTEMA ALARM:

En MINIX, un proceso puede solicitar ser interrumpido por el sistema operativo, después de un cierto tiempo.

Como solo disponemos de un reloj, y necesitaríamos uno para cada alarma pedida, simularemos múltiples relojes virtuales para poder llevar a cabo esta operación.

Esto se puede realizar con una tabla o una lista en el caso en que sean muchas señales, encadenando todas las peticiones de reloj pendientes, ordenadas por tiempo y una variable que guarda el momento en que expirará la siguiente alarma.

Cada vez que se produzca un tic, se comprobará si se ha agotado la alarma. Si es así, se realizara una búsqueda en la tabla o seleccionaremos la siguiente en la lista para localizar la siguiente alarma.



3.5.- PROVEER TEMPORIZADORES DE VIGILANCIA:

Estas alarmas se manejaran igual que las vistas anteriormente, pero en lugar de provocar una señal, llamaran a un procedimiento especificado por el solicitante.

Este procedimiento podrá realizar todo lo que estime necesario, hasta provocar interrupciones, cosa que no es conveniente realizar dentro del núcleo.

3.6.- PREPARAR PERFILES, VIGILAR Y RECABAR ESTADÍSTICAS:

Por último, el reloj también puede encargarse tanto de la realización de perfiles, como de la monitorización y recolección de estadísticas.

Esto permitirá al usuario indicarle al sistema que realice la construcción de un histograma de su contador de programa, lo que permitirá al usuario saber que partes de su programa emplean más tiempo en ejecutarse.

4.- GENERALIDADES DEL CONTROLADOR DE RELOJ EN MINIX

El controlador de reloj de MINIX está contenido en el fichero **clock.c**. Tiene un único punto de entrada, **clock_task()**, que realiza un bucle infinito que esperará por mensajes.

4.1.- TIPOS DE MENSAJES:

4.1.1.- HARD_INT:

Es el mensaje enviado al driver cuando ocurre una interrupción de reloj y hay que hacer algún trabajo. Por ejemplo, esto ocurre cuando se tiene que enviar una alarma o el tiempo de ejecución de un proceso ha expirado.

4.1.2.- GET_UPTIME:

Se usa para obtener el tiempo en tics desde el momento de arranque.

4.1.3.- GET_TIME:

Retorna la hora real actual, calculándola como el número de segundos transcurridos desde el 10 de Enero de 1970 a las 12:00 A.M. Cuando se invoca, convierte el valor actual del contador de tics a segundos y lo suma al tiempo de arranque almacenado.

El controlador almacena el tiempo en una variable desde el arranque del sistema.

4.1.4.- SET_TIME (NUEVO VALOR DEL TIEMPO EN SEGUNDOS) :

Fija el tiempo real del sistema. Este mensaje solo puede ser enviado por el super usuario.

4.1.5.- SET_ALARM (NÚMERO DE PROCESO, PROCEDIMIENTO A LLAMAR, RETARDO):

Permite a un proceso poner un temporizador de vigilancia que expira en un número especifico de tics de reloj. Cuando un proceso de usuario hace una llamada a ALARM , envía un mensaje al manejador de memoria y éste se lo enviará al driver de reloj. Cuando la alarma expire, el controlador de reloj retornará un mensaje al manejador de memoria, que lanzará la señal correspondiente.

4.1.6.- SET_SYN_AL (NÚMERO DE PROCESO, RETARDO):

Similar al SET_ALARM, pero se usa para establecer una alarma síncrona.

Una alarma síncrona envía un mensaje a un proceso, generando una señal o una llamada a un procedimiento. Por tanto, la tarea de una alarma síncrona es enviar mensajes a los procesos que lo requieran.

4.2.- VARIABLES IMPORTANTES EN LA IMPLEMENTACIÓN:

La tarea de reloj no utiliza estructuras de datos importantes, pero sí usa unas variables para controlar el tiempo, que mostramos a continuación.

4.2.1.- LOST_TICKS:

- Tiempo que ha permanecido parado el proceso a causa de una interrupción.
- Actualmente no se utiliza, pero se ha preparado por si el usuario desea implementar algún tipo de driver por su cuenta que pudiera inhabilitar las interrupciones durante un tiempo tan largo que pudieran perderse algún tic de reloj.

4.2.2.- TICKS Y PENDING_TICKS:

- **Ticks** es una variable local del controlador que se actualiza cuando se retorna de una interrupción.
- La variable ticks se utilizará para actualizar **pending_ticks**, que llevara la cuenta de los ticks transcurridos en el tratamiento de cada mensaje tratado en *clock_task*.
- Resetea *lost_ticks* a cero.

4.2.3.- SCHED_TICKS:

- Se decrementa en cada tick para controlar el cuanto de ejecución de un proceso en la CPU.
- El manejador de interrupciones envía un mensaje a la tarea de reloj sólo si se vence una alarma o ha expirado el cuanto de ejecución.

4.2.4.- REALTIME:

- Permite calcular la hora del día actual. Esto se realiza cuando la tarea del reloj recibe un mensaje de cualquier parte del sistema para obtener la hora.
- Suma la variable **pending_ticks** a con **realtime** y luego pone a cero **pending_ticks**.
- Junto con ***boot_time*** (tiempo en segundos de arranque del sistema) permite calcular la hora actual.

4.2.5.- NEXT_ALARM:

- Esta variable registra cuando podrá ocurrir la próxima alarma.
- El driver debe tener cuidado porque el proceso puede terminar o ser matado antes de que la señal se produzca. Por este motivo, cuando la señal se genera se comprueba si aún se necesita.

- Un proceso de usuario sólo puede tener una alarma pendiente.
- Al ejecutar una nueva llamada a ALARM mientras el temporizador está todavía activado se cancelara la alarma anterior.
- Una posible solución es almacenar los temporizadores en un campo de la entrada del proceso (de la tabla de procesos).

4.2.6.- WATCH_DOG:

- Para no perder la información relativa a la invocación de una función en una determinada tarea, esta variable contendrá la función que será llamada cuando la tarea genere un **alarm**.

4.2.7.- SYN_TABLE:

- Array que almacena flags que indican para cada proceso si se espera recibir una alarma síncrona.

4.3.- TAREA DE ALARMA SÍNCRONA:

Una *alarma síncrona* es muy parecida a una alarma, pero en lugar de enviar una señal o llamar a una función cuando el tiempo expira, envía un mensaje.

El mecanismo de alarma síncrona fue añadido al MINIX para soportar el servidor de red, el cual, como el manejador de memoria y el servidor de ficheros, corre como un proceso separado.

Cualquier proceso que necesite esperar un cierto intervalo por un evento y transcurrido éste desbloquearse, solicitará una alarma síncrona. (Por ejemplo, implementación de los *timeouts* de reconocimiento).

Examinando el tipo o origen del mensaje, se puede determinar si ha llegado un paquete o se ha perdido.

Una alarma síncrona es más rápida que una alarma implementada mediante señales, la cual requiere diversos mensajes y una cantidad considerable de procesamiento.

Una función *watch_dog* también es rápida, pero sólo es útil para tareas compiladas en el mismo espacio de direcciones, como por ejemplo la tarea de reloj.

Cuando un proceso espera por un mensaje, una alarma síncrona es más apropiada que señales asíncronas o funciones *watch_dog*, siendo más sencillo su manejo con un pequeño procesamiento adicional.

5.- CÓDIGO DEL CLOCK.C

5.1.- DEFINICIÓN DE VARIABLES, CONSTANTES Y PROTOTIPADO DE FUNCIONES:

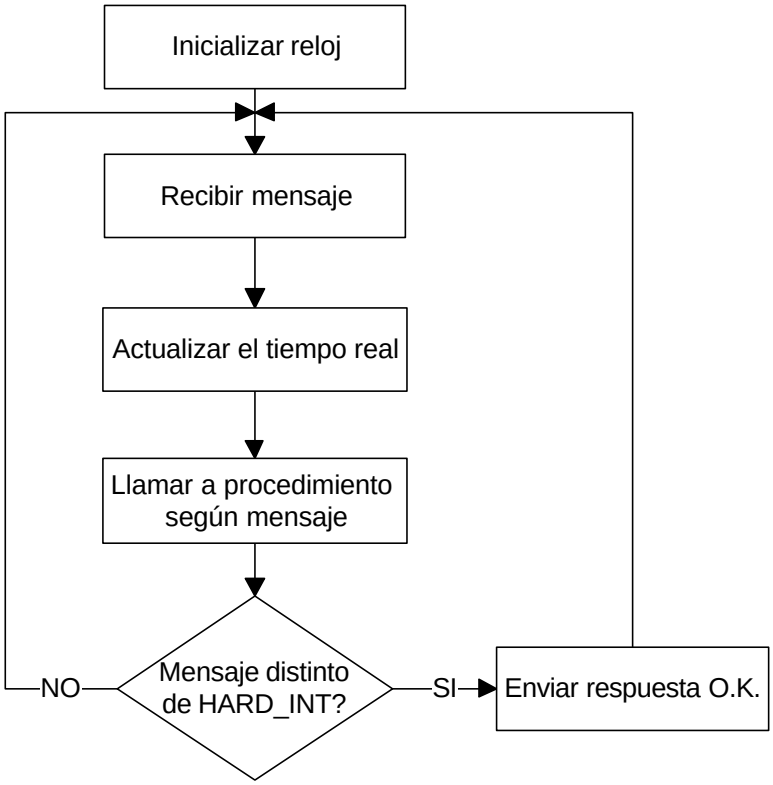
```
#include "kernel.h"
#include <signal.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"

/* DEFINICIÓN DE CONSTANTES */
#define MILLISEC 100
     /* Intervalo de llamada al planificador, en milisegundos */
#define SCHED_RATE (MILLISEC*HZ/1000)
     /* Nº de ticks por cuanto de tiempo */

/* PARÁMETROS DEL RELOJ */
#define COUNTER_FREQ (2*TIMER_FREQ)
     /* Frec. del contador en onda cuadrada */
#define LATCH_COUNT     0x00
     /* Fija el canal */
#define SQUARE_WAVE      0x36
     /* Modo de onda cuadrada y tipo de acceso */
#define TIMER_COUNT ((unsigned) (TIMER_FREQ/HZ))
     /* Valor inicial de contador */
#define TIMER_FREQ  1193182L
     /* Frecuencia del reloj para PC y AT */
#define CLOCK_ACK_BIT   0x80
     /*Bit de reconocimiento de interrup. de reloj PS/2*/

/* VARIABLE DE LAS TAREAS DE RELOJ */
PRIVATE clock_t realtime;
     /* realtime actual */
PRIVATE time_t boot_time;
     /* Tiempo en seg. desde que se cargó el sistema */
PRIVATE clock_t next_alarm;
     /* Próxima alarma */
PRIVATE message mc;   /* Buffer de mensaje entrada/salida */
PRIVATE int watchdog_proc; /*Contiene el nº de proc. en llamada a wactch_dog*/
PRIVATE watchdog_t watch_dog[NR_TASKS+NR_PROCS];
     /* Vector de punteros a funciones que deben realizar las tareas .*/

/* VARIABLES USADAS PARA LAS TAREAS DEL RELOJ Y PARA LA SINCRONIZACIÓN DE ALARMAS*/
PRIVATE int syn_al_alive= TRUE;
PRIVATE int syn_table[NR_TASKS+NR_PROCS];

/* VARIABLES EMPLEADAS DURANTE UNA INTERRUPCIÓN DE RELOJ */
PRIVATE clock_t pending_ticks;
     /* Ticks pendientes de actualizar debido a interrupciones de reloj */
PRIVATE int sched_ticks = SCHED_RATE;
     /* Contador del cuanto: cuando llega a cero se llama al planificador */
PRIVATE struct proc *prev_ptr;
     /* Último proceso ejecutado. */

FORWARD _PROTOTYPE (  void common_setalarm, (int proc_nr,
                        long delta_ticks, watchdog_t fuction) );
FORWARD _PROTOTYPE( void do_clocktick, (void) );
FORWARD _PROTOTYPE( void do_get_time, (void) );
FORWARD _PROTOTYPE( void do_getuptime, (void) );
FORWARD _PROTOTYPE( void do_set_time, (message *m_ptr) );
FORWARD _PROTOTYPE( void do_setalarm, (message *m_ptr) );
FORWARD _PROTOTYPE( void init_clock, (void) );
FORWARD _PROTOTYPE( void cause_alarm, (void) );
FORWARD _PROTOTYPE( void do_setsyn_alrm, (message *m_ptr) );
FORWARD _PROTOTYPE( int clock_handler, (int irq) );
```

5.2.- CLOCK_TASK:

5.2.1.- CÓDIGO EN C:

```
PUBLIC void clock_task()
{
/* PROGRAMA PRINCIPAL QUE SE ENCARGA DE REALIZAR LAS TAREAS PROPIAS DEL RELOJ.  ACTUALIZA EL VALOR DE
REALTIME SUMANDO LOS POSIBLES TICKS DE RELOJ  PENDIENTES DEBIDO A INTERRUPCIONES DE RELOJ.  ADEMÁS,
DISCRIMA LOS DIFERENTES TIPOS DE MENSAJE, Y LOS ATIENDE REALIZANDO LLAMADAS  A LAS FUNCIONES APROPIADAS.
*/

     int opcode;

     init_clock();
          /* Inicializa el reloj */

     /* BUCLE PRINCIPAL. SE QUEDA ESPERANDO HASTA RECIBIR UN MENSAJE PARA TRATARLO */
     while (TRUE) {
          receive(ANY, &mc);
               /* Se recibe un mensaje */
          opcode = mc.m_type;
               /* Extraemos el tipo de mensaje */
          lock();
               /*Se actualiza realtime con los ticks pendientes (Región Crítica)*/
            realtime += pending_ticks;
            pending_ticks = 0;
          unlock();

          /* SE TRATA CONVENIENTE EL MENSAJE RECIBIDO */
          switch (opcode) {
          case HARD_INT:
                    do_clocktick();
                    break;
               case GET_UPTIME:
                    do_getuptime();
                    break;
               case GET_TIME:
                    do_get_time();
                    break;
               case SET_TIME:
                    do_set_time(&mc);
                    break;
               case SET_ALARM:
                    do_setalarm(&mc);
                    break;
               case SET_SYNC_AL:
                    do_setsyn_alrm(&mc);
                    break;
               default:
                    panic("clock task got bad message", mc.m_type);
          }

          /* SE RESPONDE CON UN MENSAJE OK, EXCEPTO EN EL CASO DE UN MENSAJE HARD_INT */
          mc.m_type = OK;
          if (opcode != HARD_INT) send (mc.m_source, &mc);
     }
}
```

5.2.2.- ALGORITMO:



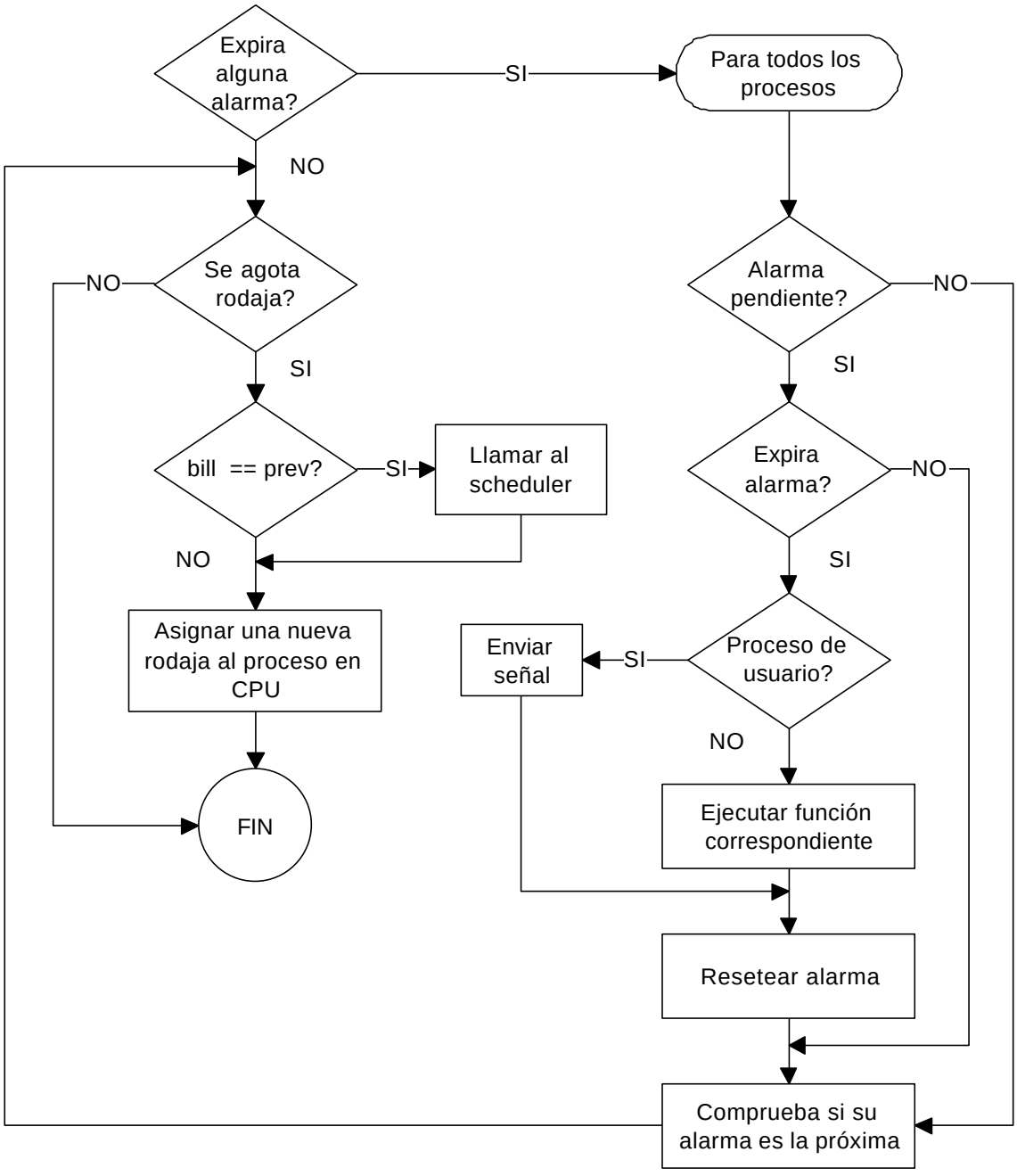
5.3.- DO_CLOCKTICK:

5.3.1.- CÓDIGO EN C:

```
PRIVATE void do_clocktick()
{
/*A PESAR DE SU NOMBRE, NO SE LLAMA EN CADA TICK DE RELOJ, SINO CUANDO HAY ALGO ESPECIAL QUE ATENDER, COMO POR EJEMPLO UNA ALARMA QUE HA EXPIRADO O UNA RODAJA DE CPU QUE HA FINALIZADO */

     register struct proc *rp;
     register int proc_nr;

     /* LA ALARMA MÁS CERCANA HA EXPIRADO. SE COMPRUEBA TODA LA TABLA DE PROCESOS Y TAREAS PARA COMPROBAR SI HA EXPIRADO MÁS DE UNA SIMULTÁNEAMENTE. */
     if (next_alarm <= realtime) {
          next_alarm = LONG_MAX;
          for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++) {
               if (rp->p_alarm != 0) {

                    /* SI ES UN PROCESO DE USUARIO, SE LE ENVÍA UNA SEÑAL. SI ES UNA TAREA, SE LLAMA A LA FUNCIÓN PREVIAMENTE ESPECIFICADA */
                    if (rp->p_alarm <= realtime) {
                         proc_nr = proc_number(rp);
                         if (watch_dog[proc_nr+NR_TASKS]) {
                              watchdog_proc= proc_nr;
                              (*watch_dog[proc_nr+NR_TASKS])();
                         }
                         else
                              cause_sig(proc_nr, SIGALRM);
                         rp->p_alarm = 0;
                    }
```

```
                    /* SE DETERMINA CUÁL ES LA PRÓXIMA ALARMA*/
                    if (rp->p_alarm != 0 && rp->p_alarm < next_alarm)
                         next_alarm = rp->p_alarm;
               }
          }
     }

      /* SI UN PROCESO DE USUARIO HA ESTADO EJECUTÁNDOSE DEMASIADO TIEMPO, SE ESCOGE OTRO*/
      if (--sched_ticks == 0) {
             if (bill_ptr == prev_ptr) lock_sched();
          sched_ticks = SCHED_RATE;
          prev_ptr = bill_ptr;
     }
}
```

5.3.2.- ALGORITMO:



5.4.- DO_GETUPTIME:

```
PRIVATE void do_getuptime()
{
/* SE OBTIENE EL Nº DE TICKS DESDE EL ARRANQUE */

     mc.NEW_TIME = realtime;
          /* Tiempo desde el arranque actual */
}
```

5.5.- GET_UPTIME:

```
PUBLIC clock_t get_uptime()
{
/*OBTIENE Y RETORNA EL Nº DE TICKS DESDE EL ARRANQUE. ESTA FUNCIÓN HA SIDO DISEÑADA PARA QUE PUEDA SER
LLAMADA DIRECTAMENTE POR OTRAS TAREAS, EVITANDO ASÍ LA SOBRECARGA DE MENSAJES */

     clock_t uptime;
     lock();
     uptime = realtime + pending_ticks;
     unlock();
     return(uptime);
}
```

5.6.- DO_GET_TIME:

```
PRIVATE void do_get_time()
{
/* OBTIENE Y RETORNA EL RELOJ ACTUAL EN SEGUNDOS */

     mc.NEW_TIME = boot_time + realtime/HZ;
          /* Tiempo real actual */
}
```

5.7.- DO_SET_TIME:

```
PRIVATE void do_set_time(m_ptr)
message *m_ptr;
{
/* ACTUALIZA EL TIEMPO REAL. SÓLO PUEDE SER LLAMADA POR EL SUPERUSUARIO */

     boot_time = m_ptr->NEW_TIME - realtime/HZ;
}
```

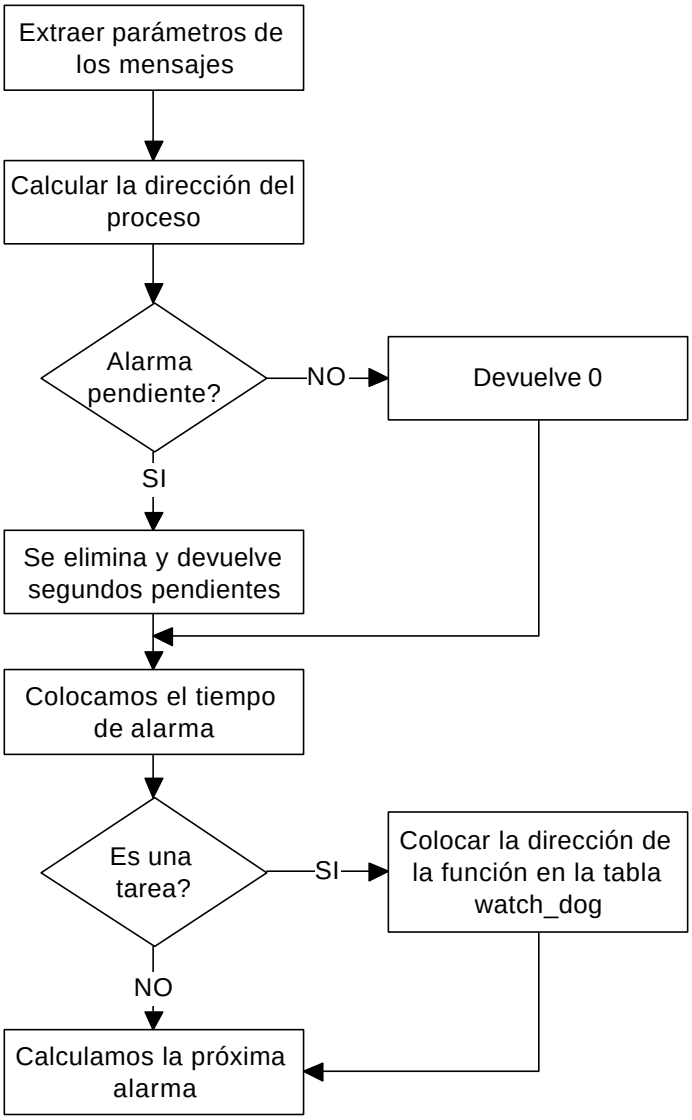
5.8.- DO_SETALARM:

5.8.1.- CÓDIGO EN C:

```
PRIVATE void do_setalarm(m_ptr)
message *m_ptr;
{
/* UN PROCESO PIDE UNA SEÑAL DE ALARMA O UNA TAREA PIDE QUE SE INVOQUE A UNA FUNCIÓN VIGILANTE DESPÚES DE UN
CIERTO TIEMPO*/

     register struct proc *rp;
     int proc_nr;
     long delta_ticks;
     watchdog_t function;
```

```
        /* SE EXTRAEN LOS DATOS DEL MENSAJE */
        proc_nr = m_ptr->CLOCK_PROC_NR;
                /*Proc. a interrumpir despues*/
        delta_ticks = m_ptr->DELTA_TICKS;
                /* Cuantos ticks esperar */
        function = (watchdog_t) m_ptr->FUNC_TO_CALL;
                /* Funcion a llamar (solo para las tareas) */

        /* SE ACTUALIZA EL VALOR DE LA ALARMA DE LA TAREA O DEL PROCESO */
        rp = proc_addr(proc_nr);
        mc.SECONDS_LEFT = (rp->p_alarm == 0 ? 0 : (rp->p_alarm - realtime)/HZ );
        if (!istaskp(rp)) function= 0;
                /* Marcamos los procesos de usuario */
        common_setalarm(proc_nr, delta_ticks, function);
}
```

5.8.1.- ALGORITMO:



5.9.- DO_SETSYN_ALRM:

```
PRIVATE void do_setsyn_alrm(m_ptr)
message *m_ptr;
{
/* UN PROCESO PIDE UNA ALARMA SÍNCRONA */

     register struct proc *rp;
     int proc_nr;
     long delta_ticks;

     /* SE EXTRAEN LOS DATOS DEL MENSAJE */
     proc_nr = m_ptr->CLOCK_PROC_NR;
          /* Proceso que interrumpir despues */
     delta_ticks = m_ptr->DELTA_TICKS;
          /* Cuantos tics esperar */

     /* SE ACTUALIZA EL VALOR DE LA ALARMA DEL PROCESO */
     rp = proc_addr(proc_nr);
     mc.SECONDS_LEFT = (rp->p_alarm == 0 ? 0 : (rp->p_alarm - realtime)/HZ );
     common_setalarm(proc_nr, delta_ticks, cause_alarm);
}
```

5.10.- COMMON_SETALARM:

```
PRIVATE void common_setalarm(proc_nr, delta_ticks, function)
int proc_nr;
     /* Que proceso pide la alarma */
long delta_ticks;
     /* Intervalo de tiempo deseado, en ticks */
watchdog_t function;
     /* Función a llamar */
{
/* ACABA CON EL TRABAJO DE LAS FUNCIONES DO_SET_ALARM Y DO_SETSYN_ALRM. ALMACENA LA FUNCIÓN REQUERIDA Y
ACTUALIZA LA NUEVA PRÓXIMA ALARMA */

     register struct proc *rp;

     rp = proc_addr(proc_nr);
     rp->p_alarm = (delta_ticks == 0 ? 0 : realtime + delta_ticks);
     watch_dog[proc_nr+NR_TASKS] = function;

     /* SE ACTUALIZA LA PRÓXIMA ALARMA */
     next_alarm = LONG_MAX;
     for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++)
          if(rp->p_alarm != 0 && rp->p_alarm < next_alarm)
               next_alarm=rp->p_alarm;
}
```

5.11.- CAUSE_ALARM:

```
PRIVATE void cause_alarm()
{
/* RUTINA QUE SE LLAMA CUANDO EXPIRA UNA ALARMA SÍNCRONA. EL NÚMERO DEL PROCESO ES UNA VARIABLE GLOBAL
(HACK) */

     message mess;

     syn_table[watchdog_proc + NR_TASKS]= TRUE;
     if (!syn_al_alive) send (SYN_ALRM_TASK, &mess);
}
```

5.12.- SYN_ALRM_TASK:

```
PUBLIC void syn_alrm_task()
{
/* TAREA QUE SE ENCARGA DE CONTROLAR LAS ALARMAS SÍNCRONAS */

     message mess;
     int work_done;
     int *al_ptr;
     int i;

     syn_al_alive= TRUE;
     for (i= 0, al_ptr= syn_table; i<NR_TASKS+NR_PROCS; i++, al_ptr++)
          *al_ptr= FALSE;

     while (TRUE) {
          work_done= TRUE;
          for (i= 0, al_ptr= syn_table; i<NR_TASKS+NR_PROCS; i++, al_ptr++)
               f (*al_ptr) {
                    al_ptr= FALSE;
                    mess.m_type= CLOCK_INT;
                    send (i-NR_TASKS, &mess);
                    work_done= FALSE;
               }
          if (work_done) {
               syn_al_alive= FALSE;
               receive (CLOCK, &mess);
               syn_al_alive= TRUE;
          }
     }
}
```

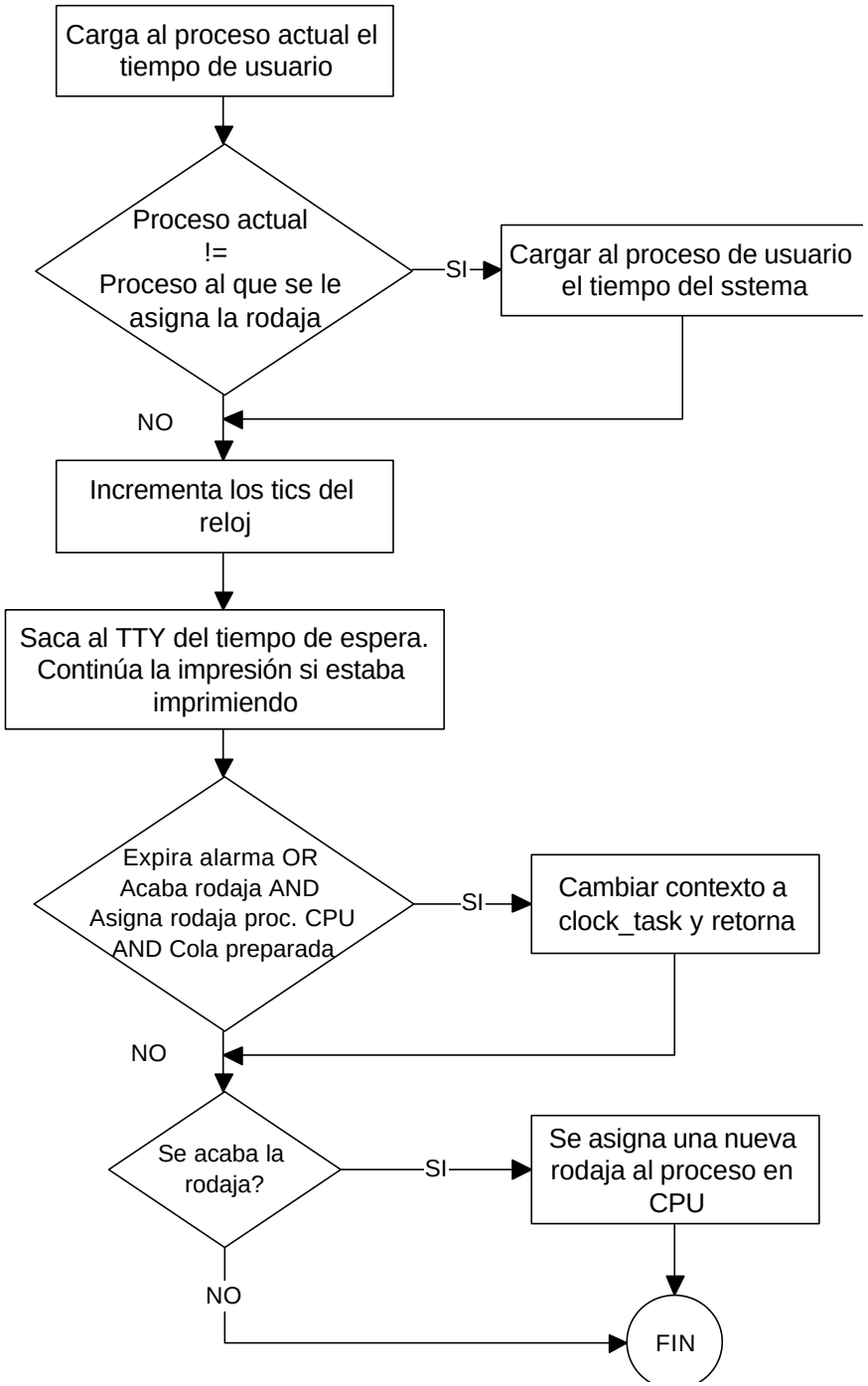
5.13.- CLOCK_HANDLER:

5.13.1.- CÓDIGO EN C:

```
PRIVATE int clock_handler(irq)
int irq;
{
/* SE LLAMA EN CADA INTERRUPCIÓN PROVOCADA POR UN TICK DE RELOJ. SE ENCARGA DE CONTABILIZAR LOS TICKS
TRANSCURRIDOS Y DE LLEVAR LA CONTABILIDAD DE LOS TIEMPOS DE USUARIO Y DE SISTEMA. APARTE TAMBIÉN LLAMA A
TTY Y PRINTER SI ES NECESARIO, Y AL PLANIFICADOR EN CASO DE AGOTARSE LA RODAJA DE CPU ACTUAL */

     register struct proc *rp;
     register unsigned ticks;
     clock_t now;

     /* ACUSAR RECIBO DE LA INTERRUPCION DE RELOJ PS/2 */

     if (ps_mca) {
          out_byte(PORT_B, in_byte(PORT_B) | CLOCK_ACK_BIT);
     }

     /* ACTUALIZAR TIEMPOS DE USUARIO Y SISTEMA CONTABILIZADOS. COBRARA EL TIEMPO DE USUARIO AL PROCESO
     ACTUAL. SI EL PROCESO ACTUAL NO ES EL FACTURABLE (P.E. SI ES UNA TAREA), COBRAR AL PROCESO
     FACTURABLE EL TIEMPO DEL SISTEMA */
     if (k_reenter != 0)
          rp = proc_addr(HARDWARE);
     else
          rp = proc_ptr;
     ticks = lost_ticks + 1;
     lost_ticks = 0;
     rp->user_time += ticks;
```

```
     if (rp != bill_ptr && rp != proc_addr(IDLE)) bill_ptr->sys_time += ticks;

     pending_ticks += ticks;
     now = realtime + pending_ticks;
     if (tty_timeout <= now) tty_wakeup(now);
          /* Quiza despertar TTY */

     if (  next_alarm <= now || sched_ticks == 1 && bill_ptr == prev_ptr &&
          rdy_head[USER_Q] != NIL_PROC) {
          interrupt(CLOCK);
          return 1;
               /* Rehabilitar interrupciones */
     }

     /* SI BILL_PTR == PREV_PTR, NO HAY USUARIOS LISTOS */
     if (--sched_ticks == 0) {
          sched_ticks = SCHED_RATE;
               /* Restablecer el cuanto */
          prev_ptr = bill_ptr;
               /* Nuevo proceso previo */
     }
     return 1;
}
```

5.13.2.- ALGORITMO:



5.14.- INIT_CLOCK:

```
PRIVATE void init_clock()
{
/* INICIALIZAMOS EL CANAL 0 DEL TEMPORIZADOR 8253A, POR E.J. A 60 HZ */

     out_byte(TIMER_MODE, SQUARE_WAVE);
          /* Ejecucion continua del temp. */
     out_byte(TIMER0, TIMER_COUNT);
          /* Cargar el byte bajo del temp. */
     out_byte(TIMER0, TIMER_COUNT >> 8);
          /* Cargar el byte alto del temp. */
     put_irq_handler(CLOCK_IRQ, clock_handler);
          /* Fijar el manejador del interrups. */
     enable_irq(CLOCK_IRQ);
          /* Listo para las interrup. de reloj */
}
```

5.15.- CLOCK_STOP:

```
PUBLIC void clock_stop()
{
/* RESTABLECEMOS EL RELOJ A LA FRECUENCIA DE LA BIOS (USADO PARA EL REARRANQUE) */

     out_byte(TIMER_MODE, 0x36);
     out_byte(TIMER0, 0);
     out_byte(TIMER0, 0);
}
```

5.16.- MILLI_DELAY:

```
PUBLIC void milli_delay(millisec)
unsigned millisec;
{
/* SE UTILIZA PARA IMPLEMENTAR RETARDOS (EN MILISEGUNDOS) */

     struct milli_state ms;

     milli_start(&ms);
     while (milli_elapsed(&ms) < millisec) {}
}
```

5.17.- MILLI_START:

```
PUBLIC void milli_start(msp)
struct milli_state *msp;
{
/* PREPARA LA LLAMADA A MILLI_ELAPSED */

     msp->prev_count = 0;
     msp->accum_count = 0;
}
```

5.18.- MILLI_ELAPSED:

```
PUBLIC unsigned milli_elapsed(msp)
struct milli_state *msp;
{
/* RETORNA EL Nº DE MILISEGUNDOS TRANSCURRIDOS DESDE LA LLAMADA A MILLI_START */

     unsigned count;

     out_byte(TIMER_MODE, LATCH_COUNT);
     count = in_byte(TIMER0);
     count |= in_byte(TIMER0) << 8;

     msp->accum_count += count <= msp->prev_count ? (msp->prev_count - count) : 1;

     msp->prev_count = count;

     return msp->accum_count / (TIMER_FREQ / 1000);
}
```

6.- CUESTIONES DEL MANEJADOR DE RELOJ

6.1.- PRINCIPALES FUNCIONES DEL MANEJADOR DE RELOJ:

- Conservar la hora y la fecha.
- Contabilidad del tiempo de ejecución de los procesos.
- Contabilidad del tiempo de uso de la CPU.
- Implementación de alarmas (síncronas y asíncronas).
- Provisión de cronómetros guardianes (funciones *watch_dog*).
- Monitorización y estadísticas.

6.2.- MENSAJES QUE RECIBE EL MANEJADOR DE RELOJ:

- HARD_INT: Ha expirado una alarma o se ha acabado la rodaja de un proceso.
- GET_UPTIME: Un proceso desea conocer el nº de ticks transcurridos desde el arranque.
- GET_TIME: Un proceso desea conocer la hora actual.
- SET_TIME: El super-usuario desea modificar la hora actual del sistema.
- SET_ALARM: Un proceso o tarea desea una alarma al cabo de un cierto tiempo.
- SET_SYNC_AL: Un proceso desea una alarma síncrona al cabo de un cierto tiempo.

6.3.- ¿CUÁL ES LA SECUENCIA DE PASOS QUE OCURRE DESDE QUE UNA TAREA INDICA AL RELOJ QUE EJECUTE UNA FUNCIÓN EN UN TIEMPO DETERMINADO HASTA QUE ÉSTA SE EJECUTA?:

Cuando un proceso solicita al Sistema Operativo una alarma, manda un mensaje a CLOCK_TASK. Éste, al ver que el mensaje es del tipo SET_ALARM, llama al procedimiento DO_SET_ALARM.

El procedimiento DO_SET_ALARM, calcula la dirección del proceso, a partir de los parámetros del mensaje. Con esta dirección va a la tabla de procesos y comprueba si ese proceso ya tenia una alarma activada. De ser así se elimina y se retorna los segundos que faltaban. De no ser así retorna un cero. En los dos casos se coloca la nueva alarma.

En el caso de ser una tarea, se guarda la dirección de la función que se llamará al explorar la alarma en la tabla WatchDog[].

Luego en cada tick de reloj, CLOCK_HANDLER, determina si ha expirado la alarma más próxima. De ser así, genera una interrupción del tipo HARD_INT. Al recibir CLOCK_TASK esta interrupción llama al procedimiento DO_CLOCKTICK. Este último testea, para todos los procesos de la tabla de procesos, si su alarma ha expirado.

Si su alarma expiró envía una señal de respuesta; si se trata de una tarea lo que hace es ejecutar una función que se encuentra en la tabla Watchdog[].

6.4.- ¿POR QUÉ ES PRECISO EL USO DE REGIONES CRÍTICAS EN EL MANEJADOR DE DISPOSITIVO DE RELOJ CLOCK_TASK?:

Las regiones críticas se utilizan para garantizar el acceso exclusivo a la variable *pending_ticks* en la actualización de *realtime* mientras ésta es utilizada por el proceso CLOCK_TASK, o sea, bloquear el acceso a la variable *pending_ticks*.

6.5.- ¿QUÉ FUNCIÓN REALIZA EL VECTOR WATCH_DOG?:

Almacena las direcciones de las funciones a realizar cuando la alarma de una tarea expira.