CONCEPCION DE UN BIOS ORIENTADO A SISTEMAS DE TIEMPO REAL

José C. Lizzoli
H. Fabián Torres
Armando E. De Giusti

Laboratorio de Investigación y Desarrollo en Informática. (L.I.D.I.)
Departamento de Informática. Facultad de Ciencias Exactas.
Universidad Nacional de La Plata. (U.N.L.P.)
50 y 115 - 1er. Piso, Tel. (021) 21-4633 - (1900) La Plata - Argentina

RESUMEN

Se presenta el análisis y desarrollo de un BIOS para microcomputadora, orientado a aplicaciones de ingeniería en el área de Sistemas de Tiempo Real.

Las características más destacadas del desarrollo son:

** extensiones para el control directo de hardware no standard.

** la implementación en lenguaje de alto nivel.

** transparencia respecto del sistema operativo.

Asimismo se detalla la utilización de este BIOS en una aplicación de control de un subsistema de adquisición de datos cardiovasculares en tiempo real, a fin de investigar los problemas biológicos asociados.

ABSTRACT

This paper presents the analysis and development of an oriented BIOS for real time applications, specially data acquisition control in engineering problems.

This work means:

** Extended primitives to control "non-standard" hardware in pc-like architectures.

** High level language BIOS implementation.

** User transparency, because he sees an extended operating system with no special restriction.

An application which permits to use a microcomputer for controlling biological experiences in the cardiac output in animals submitted to hypertension is also detailed.

INTRODUCCION

Los sistemas dedicados de tiempo real constituyen una de las áreas de mayor investigación y desarrollo en la última década, tanto en hardware como en software [1], [2] y [3].

Una rápida caracterización de los sistemas dedicados de tiempo real (SDTR) admite los siguientes elementos dignos de mención:

** Se requiere una especificación rigurosa, especialmente en cuanto a los problemas que dependen del tiempo y de la ocurrencia de eventos asincrónicos, externos al software en si. [4] y [5].

** En general la puesta a punto de un sistema de este tipo es dificultosa y requiere integrar elementos de hardware no siempre fáciles de modelizar. La verificación o test funcional no podrá ser exhaustiva, lo cual nos hace insistir en la importancia de la especificación [6].

** Las ayudas al diseño (ambientes de programación, lenguajes de especificación y simulación, sistemas de desarrollo) son en muchos casos experimentales, complejos y de alto costo. Incluso en muchos casos las herramientas utilizadas en la concepción del hardware son independientes y mutuamente excluyentes con las empleadas en el desarrollo del software [7] y [8].

Estos elementos nos conducen naturalmente a identificar algunos objetivos en el desarrollo de sistemas de tiempo real, objetivos que no siempre son complementarios:

1- El objetivo tradicional que ha sido obtener una alta eficiencia en la respuesta temporal: en general un sistema de control, o de adquisición de datos o de procesamiento de señales requiere un software que minimice el tiempo de respuesta. Este objetivo tiende a soluciones de hardware y software que pongan énfasis en los detalles de implementación que optimicen el número y tipo de instrucciones de máquina a ejecutar.

2- Al mismo tiempo, a mayor complejidad del sistema aparece un requerimiento más importante de lograr una especificación precisa que permita probar corrección o al menos realizar una verificación que aisle los errores detectables en la puesta a punto. Este objetivo pone el énfasis en la descripción formal del sistema y sus relaciones, y muy especialmente en la posibilidad de derivar alguna forma de código ejecutable a partir de una especificación validada [9].

3- Por ultimo, la misma complejidad de los sistemas de tiempo real hace necesario

trabajar con la suficiente abstracción, encapsulando funciones y definiendo tipos de datos y operaciones válidas sobre ellos, de modo de favorecer la documentación y mantenimiento del sistema de software [10].

Si consideramos estructuralmente un SDTR podemos mencionar, tal como en la [11], tres "capas" o niveles tanto de hardware como de software:

A- La capa mas "externa" esta constituida por la interfaz entre el fenómeno de tiempo real tratado y el sistema digital que lo procesa. En ella habitualmente existe algún tipo de conversión de señal (por ej. A/D) y las soluciones de hardware y software son especificas según las características del fenómeno que se esta atendiendo y procesando.

B- La capa de "comunicación" es la que permite resolver la recolección y trasmisión de los datos al procesador y de este al subsistema externo. Habitualmente es una capa "crítica" desde el punto de vista de los tiempos involucrados y dadas las velocidades muy diferentes de los distintos procesos que pueden estar en curso, es clásico que dicha comunicación se base en interrupciones asincrónicas.

C- Por último la capa más "interna" esta constituída por el procesamiento puro de la información ya convertida a digital. Habitualmente el software desarrollado en esta capa no se diferencia de cualquier sistema de aplicación, salvo en condiciones de contexto como la memoria disponible, el hecho de que la versión definitiva deba residir en ROM y por lo tanto requerir la generación de un módulo ejecutable que sea reubicable a partir de una dirección especifica, el tiempo máximo permitido entre dos acciones, etc.

Estas tres capas de software constituyen un problema importante por su costo relativo, dadas las dificultades de obtener una especificación rigurosa; de lograr generalizaciones que no dependan del fenómeno externo a tratar y también por la "distancia" que separa la especificación abstracta del sistema de su implementación de detalle sobre un equipo real. De hecho el área de software de tiempo real constituye una de las de mayor inversión en investigación y desarrollo en Informática.

En este contexto, la utilización de hardware basado en arquitecturas "PC Like" para el desarrollo de sistemas dedicados de tiempo real constituye una alternativa muy utilizada por dos ventajas relativas:

** En la capa de aplicación se dispone de herramientas (lenguajes, ambientes) de desarrollo de software muy completas.

** El costo y el tiempo de desarrollo de hardware se reducen sensiblemente.

A su vez, partir de una arquitectura orientada a una microcomputadora crea limitaciones dadas fundamentalmente por las capas de software de bajo nivel (BIOS) que parten de la suposición de trabajar con una arquitectura limitada a un conjunto de periféricos "standard" y que no está pensado para su fácil adecuación a variantes de hardware (en general está codificado en Assembler del microprocesador correspondiente).

En este trabajo se exponen los resultados de una tarea de investigación y desarrollo que se inicia en el estudio, especificación e implementación de un BIOS para PC y posteriormente continua en extensiones de este LIDI-BIOS y del Sistema Operativo convencional de una PC, a fin de obtener un sistema dedicado a la aplicación, con alta eficiencia en el manejo de la interfaz con el fenómeno real a controlar y con todas las ventajas del desarrollo de software de procesamiento, utilizando recursos de alto nivel.

DE UNA ARQUITECTURA TIPO PC A UN SISTEMA DEDICADO

La computadora PC de IBM sus clones o compatibles pueden ser vistos como un modelo de capas de hardware y software:

** La capa de hardware.

** La capa de BIOS.

** La capa del Sistema Operativo.

** Las aplicaciones.

Veamos cada capa en detalle(Fig. 1):

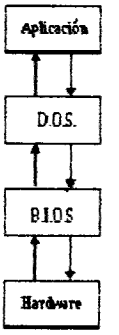


Fig. 1

** HARDWARE: Responsable de ejecutar las tareas. En esta capa reside la arquitectura física de la microcomputadora, que comprende el sistema central de procesamiento y los periféricos estandard.

** BIOS: la sigla significa Sistema Básico de Entrada y Salida. Su función es manejar el hardware, y lo hace de dos formas distintas, por un lado atendiende sus necesidades, y por otro, "encapsula" las funciones de los periféricos de modo de independizar su control de las capas superiores.

** SISTEMA OPERATIVO: la capa de más alto nivel provista, que para el usuario, universaliza el uso del hardware, esto es brinda una visión de dispositivo de cada uno de los componentes.

** APLICACION: En esta categoría se encuentran los programas del usuario.

En este último nivel se encuentran disponibles compiladores de lenguajes de alto nivel y otras herramientas para el desarrollo de aplicaciones.

Sin embargo, en el caso de ampliar el hardware se debe construir una aplicación que lo controle. Las alternativas son varias, la primera es que la misma aplicación controle el hardware adicional. Esto no es conveniente debido a que no hay primitivas en los lenguajes para hacerlo y porque se aparta de la arquitectura PC.

Una segunda alternativa de solución es construir las capas faltantes, de ese modo se extiende primero el BIOS, y sobre éste se construye un **device driver**, que representa la extensión del SO. Por último la aplicación solo tendrá que utilizar las primitivas del lenguaje para manejo de archivos para realizar el control del nuevo dispositivo.

CARACTERISTICAS GENERALES DEL BIOS

El BIOS es un conjunto de rutinas destinadas a controlar el hardware de una microcomputadora tipo PC.

Estas rutinas residen en ROM y constituyen la capa de software más relacionada con la implementación física de la arquitectura de la microcomputadora involucrada.

De este modo los programas de mayor nivel (incluyendo al mismo sistema operativo) "ven" una máquina "abstracta" relativamente independiente de su implementación física, brindando una mayor flexibilidad e independencia y asegurando la portabilidad de estos programas a modelos de arquitectura "compatible" que no han sido implementados con idéntico hardware y/o BIOS.

Un rápido análisis de las rutinas del BIOS nos permite apreciar:

** Atención de RESET.

** Atención de NMI (interrupción no-enmascarable).

** Operación de carga del Sistema Operativo mínimo.

** Atención de Servicios (video, teclado, diskettes, impresora, puerta serie).

** Manejo del reloj de tiempo real.

** Atención de otras interrupciones.

La Fig. 2 nos muestra una máquina de estados del funcionamiento del BIOS en un equipo convencional y la Fig. 3 expresa simbólicamente el "programa" que ejecuta este conjunto de rutinas. En la Fig. 4 podemos ver el diagrama de ejecución del servicio de manejo de diskettes.

Maquina de estados del BIOS

Estados

A: Inicializacion y testeo
B: Espera de interrupciones de la maquina BIOS
C: Atencion de una interrupcion de hardware
D: Atencion de una interrupcion de software

Condiciones

A: Conexion de la alimentacion o pulsacion de RESET
B: Invocacion de interrupcion
C: Invocacion de una interrupcion de software
D: Desconexion de la alimentacion

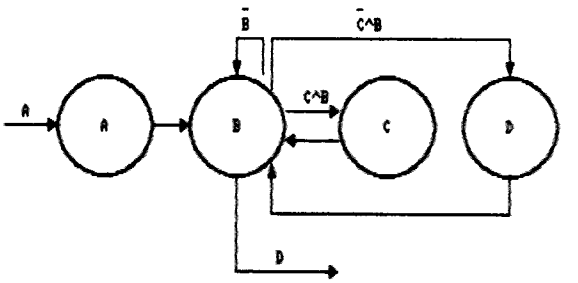


Fig. 2

Pseudocodigo

```
repeat
    repeat
    until Invocacion_de_Interrupcion_de_la_Maquina_BIOS;
    case Interrupcion of
        2          : NMI;
        5          : Prt_screen;
        8          : Timer_Int;
        9          : Kb_Int;
        A,B,C,D,F  : D_EOI;
        E          : Disk_Int;
        10h        : Video_IO;
        11h        : Equipment;
        12h        : Memory_Size_Det;
        13h        : Diskette_IO;
        14h        : RS232_IO;
        15h        : Cassette_IO;
        16h        : Keyboard_IO;
        17h        : Printer_IO;
        18h        : ROM_Basic;
        19h        : Boot_Strap;
        1Ah        : Time_of_Day;
        1Bh,1Ch    : Dummy_Return;
    end;
Until FALSE;
```

Fig. 3

Las rutinas de servicio que forman parte del BIOS "normal" son:

Int 5h -- Servicio de vuelco de la pantalla a la impresora.

Int 10h -- Servicio de video.

Int 11h -- Servicio de listado de configuración.

Int 12h -- Servicio de tamaño de la memoria.

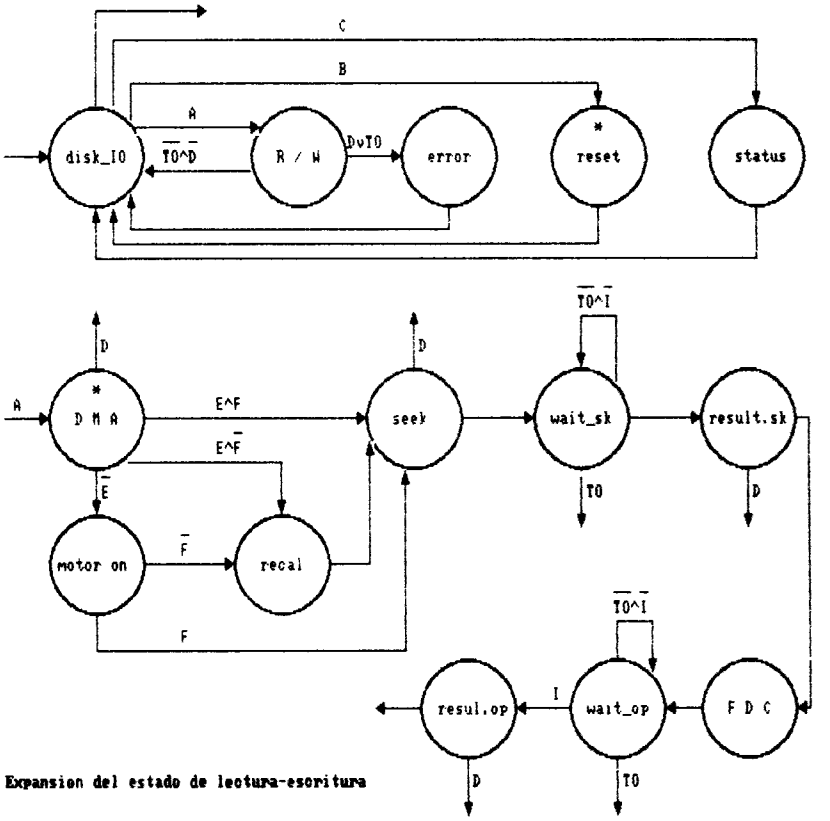
Estados

```
disk_IO   : estado de inicio y finalizacion de una
            operacion sobre diskettes.
reset     : reinicializacion de la unidad de diskettes
status    : obtencion del estado de la unidad
R/W       : operacion de lectura o escritura
error     : tratamiento de errores
DMA       : programacion del controlador DMA
motor_on  : arranque del motor
recal.    : recalibracion de la unidad de diskettes
seek      : comando de seleccion de una pista
wait_sk   : espera del tiempo de "seek"
result.sk : obtencion de resultados del comando de seek
FDC       : programacion del FDC para realizar la operacion de E/S
wait_op   : espera de la transmision de datos
resul.op  : obtencion dde resultados de la operacion
```

*** Indica estado no interrumpible**

Condiciones

```
A : solicitud de operacion de lectura o escritura
B : solicitud de operacion de reinicializacion
C : solicitud de operacion de informe de estado
D : error en alguna de las operaciones que conforman
    la lectura o escritura
E : motor en marcha
F : recalibracion no requerida
TO: error producido por exceso de espera
I : interrupcion producida por el controlador de diskettes
```



Expansion del estado de lectura-escritura

Int 13h -- Servicio de diskette.

Int 14h -- Servicio de comunicaciones asincrónicas

Int 15h -- Servicio de cassette.

Int 16h -- Servicio de teclado.

Int 17h -- Servicio de impresora.

Int 18h -- Servicio de activación del ROM-BASIC.

Int 19h -- Servicio de carga y puesta en marcha del sistema.

Int 1Ah -- Servicio de hora y fecha.

Para utilizar cualquiera de estos servicios, es necesario cargar los parámetros requeridos en los registros del microprocesador y luego ejecutar la interrupción de software deseada.

Existe además un grupo de rutinas destinadas a controlar las interrupciones de hardware y previsiones para rutinas del usuario que deban ejecutarse a intervalos de tiempo determinados y la atención de 'break'.

La implementación de estas rutinas, realizada en Assembler 8088 o similar, es altamente eficiente tanto en código como en velocidad de procesamiento. Sin embargo la realización de cualquier modificación o extensión funcional de este conjunto de rutinas exige un profundo conocimiento de la arquitectura de la microcomputadora y del lenguaje Assembler del microprocesador correspondiente.

LA COMUNICACION CON EL SISTEMA OPERATIVO

El sistema operativo MS-DOS, a través de sus distintas versiones, ha incorporado varias características del UNIX, fundamentalmente la de tratar todos los periféricos como archivos; tendencia muy marcada en los sistemas modernos. Para lograr esto, el sistema operativo contiene por cada tipo de periférico un device driver, el cual consiste de un conjunto de rutinas que son invocadas cuando el sistema operativo necesita realizar alguna operación de entrada/salida sobre tal periférico.

En una primer aproximación, tales rutinas pueden ser clasificadas como sigue:

** OPEN/CLOSE : Utilizadas cuando el periférico en cuestión requiere alguna señal de inicialización y/o de final.

** READ/WRITE : Son las que realizan las operaciones de entrada/salida propiamente dichas.

** I/O CONTROL : Utilizada para modificar eventuales parámetros de operación del device driver.

además existen rutinas destinadas a verificar el estado del device driver y para vaciar sus buffers (sin fuesen utilizados).

El sistema operativo permite la construcción, por parte del usuario, de nuevos controladores de periféricos (devices drivers) destinados a controlar hardware adicional o emularlo (ej. discos virtuales). La carga de estos drivers es realizada por el mismo sistema, durante el "boot-strap" y su invocación es realizada automáticamente cuando un programa de aplicación requiere una operación de E/S sobre el periférico asociado.

Esto permite realizar un manejo homogéneo sobre todos los periféricos, ya que sobre estos es posible realizar solamente un dado conjunto de operaciones con comportamiento predeterminado.

Con esto se ve que las extensiones de funciones de sistema operativo requieren:

a) El desarrollo de devices drivers.

b) La implementación de las operaciones primitivas de BIOS que soporten al device driver.

EL DESARROLLO DEL LIDI-BIOS

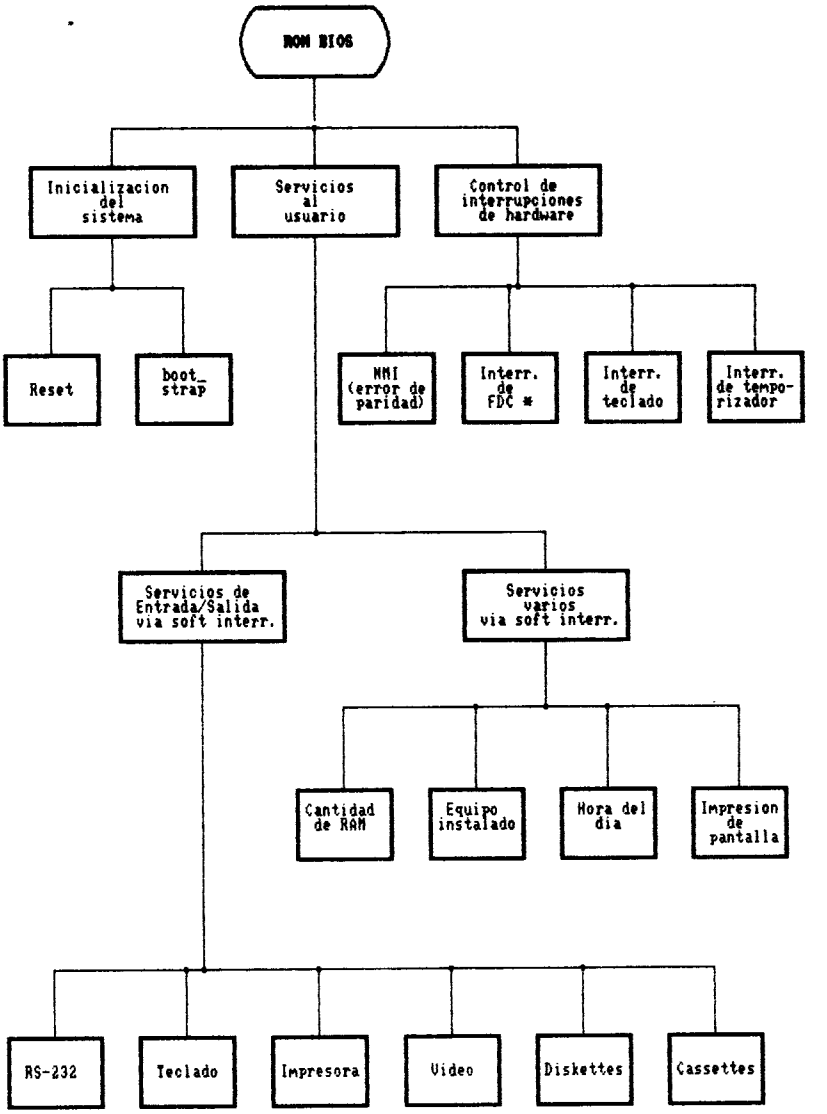
Del análisis del BIOS se pueden abstraer sus funciones de modo de obtener una estructura jerárquica tal como la que se muestra en la Fig. 5.

Como se ve, tenemos tres grupos bien diferenciados que podemos describir como sigue:

** Las rutinas de inicialización del sistema son invocadas cuando se enciende la máquina y consisten en testeos de verificación del procesador y de memoria, inicialización de los circuitos integrados que lo requieran, inicialización de la tabla de vectores de interrupción, verificación de conexión de opcionales a la configuración básica y, si hay controlador de disco, carga del sistema operativo desde éste.

** Las rutinas de control de las interrupciones de hardware son las que se encargan de atender los requerimientos de comunicación con los subsistemas físicos de la microcomputadora y trabajan complementariamente con las rutinas de servicios al usuario. Una mención aparte hay que hacer para la rutina de atención de NMI, que se activa cuando se producen errores de paridad y provoca la detención de la máquina.

** Las rutinas de servicios al usuario se invocan por programa y atienden a la necesidad de utilizar el hardware disponible. El sistema operativo utiliza continuamente este mecanismo para "servirse" del BIOS.



* **FDC** : Floppy Disk Controller

El objetivo, entonces, fue diseñar e implementar un BIOS de modo que sea portable y a la vez más flexible.

Para lograr esto utilizamos un lenguaje de alto nivel, ya que nos permite lograr una fácil estructuración del algoritmo e incluso implementar de manera sencilla los árboles funcionales que describen las distintas partes del sistema.

El lenguaje de programación C nos permite establecer una relación casi directa con el hardware, ya que nos provee las siguientes herramientas:

** Aritmética de direcciones. (punteros)

** Manejo total de la memoria. En este sentido existe la posibilidad de solapar información en una misma posición de memoria, definición de estructuras a nivel de bit y otros mecanismos que son parte del lenguaje y que otorgan gran flexibilidad en las operaciones sobre memoria.

** Control de periféricos. Para esto contamos con operaciones que nos permiten programarlos por intermedio de sus puertas de datos y control de entrada/salida.

** Control de interrupciones. Se pueden enmascarar y desenmascarar interrupciones, de modo de asegurar la exclusión mutua de los procesos en ejecución.

** Control de llamadas a procedimientos de biblioteca. Podemos controlar la inclusión de procedimientos de bibliotecas, ya que éstas pueden involucrar invocaciones al sistema operativo, perdiendo además el control sobre el tamaño del código generado.

Contando con un lenguaje que posee estas características, vemos que es posible escribir un BIOS en un lenguaje de alto nivel (llamado LIDI-BIOS) de manera sencilla, legible y que al mismo tiempo sea razonablemente eficiente.

EL LIDI-BIOS EN EPROM

Anteriormente fue dicho que el BIOS, al residir en ROM, forma una capa de software transparente al usuario que aisla a los programas de aplicación del hardware. El LIDI-BIOS fue desarrollado teniendo en cuenta la posibilidad de implementarlo en alguna memoria de lectura solamente.

El problema que surge en forma inmediata al plantear el tema de la generación de código "romable", es la ubicación en memoria de las variables. El BIOS estandard utiliza una pequeña área de RAM de menos de 256 bytes reservada para su uso exclusivo, donde son almacenados ciertos datos que deben mantenerse entre llamadas sucesivas al BIOS, tales como direcciones de puertas de periféricos, parámetros corrientes de video, etc; las "variables" del BIOS son los propios registros del microprocesador.

Obviamente, este esquema no es aplicable a los lenguajes de alto nivel, dado que en este caso no se posee control directo sobre los registros y las variables siempre ocupan algún área en la memoria, mayor que los 256 bytes antes mencionados.

En la implementación del LIDI-BIOS también se planteó la existencia de alguna RAM para uso exclusivo, la cual podría estar fuera de los 640 K estandard o "quitándole" memoria al sistema haciendo que el BIOS reporte menos memoria que la real; en el trabajo citado en [11] puede encontrarse una discusión general del tema y también el detalle de la técnica de implementación utilizada en este caso.

Las variables utilizadas por LIDI-BIOS fueron definidas como estáticas; la imposibilidad de utilizar variables automáticas está dada por ser éstas definidas sobre el segmento de pila y es muy común que los programas de aplicación y comandos del sistema no provean de suficiente espacio de pila como para, además de salvar el estado de la máquina y realizar el apilado de los parámetros en el llamado a procedimientos, definir las variables locales. De haberse pretendido utilizar variables automáticas se debería haber provisto de un código de inicialización para cada módulo que armase una pila local, éste código debería haberse realizado en assembler lo cual hubiera desvirtuado uno de los objetivos centrales del trabajo.

EVALUACION DEL LIDI-BIOS

Todas las experiencias se realizaron teniendo en cuenta comparaciones de rendimiento entre las rutinas del BIOS original y el LIDI-BIOS. Se corrieron programas de prueba que ejecutaban llamadas al BIOS y utilizaban el temporizador de la máquina para determinar la cantidad de 'ticks' utilizados.

El testeo de la cuenta se realizó mediante programas escritos en assembler que involucraban únicamente las instrucciones de máquina indispensables para la generación de los lazos y la invocación a las rutinas.

1- Manejo de video

Se realizaron dos experiencias sobre la función de simulación de impresora (función 14, utilizada por el MS-DOS), que consistieron en:

A- Impresión de 25 líneas (una pantalla completa) comenzando en la esquina superior izquierda. No se consume tiempo en scrolling.
B- Impresión de 200 líneas a partir de la última fila (nótese el tiempo demorado en el scrolling).

2- Manejo de disco

Se realizaron dos experiencias:

A- Lectura secuencial de las dos caras de un disco.
B- Lectura de 720 sectores tomados al azar.

3- Manejo de impresora

Se realizaron dos experiencias:

A- Impresión de 7000 caracteres, menos que la capacidad del buffer de la impresora utilizada.
B- Impresión de 32000 caracteres, aproximadamente 4 veces la capacidad del buffer.

	1.A	1.B	2.A	2.B	3.A	3.B
Estandard	35	340	2833	3066	169	4025
LIDI-BIOS	46	563	2694	3049	169	4025
Rendimiento	0,76	0,60	1,05	1,01	1,00	1,00

Los tres dispositivos seleccionados para estas experiencias tienen características dispares que pueden justificar los diferentes resultados obtenidos.

El control de vídeo demanda un uso intensivo del procesador ya que la principal tarea consiste en el manejo de la RAM del adaptador. Este manejo se realiza por medio de punteros 'far' lo cual explica el menor rendimiento del programa en C.

La unidad de discos flexibles tiene piezas móviles con baja velocidad respecto del procesador, además la transferencia de información se realiza mediante DMA, lo cual no involucra uso del procesador. Debido a esto es que el programa en C puede competir con el assembler. Las diferencias a favor son debidas a que los algoritmos involucran lazos de espera optimizados por el compilador. En este punto debe hacerse notar la dificultad de los lenguajes de alto nivel para manejar retardos ya que dependen de cómo el compilador los implemente.

En el caso del control de la impresora no se notan diferencias por tratarse de un dispositivo lento. La sincronización del envío de información es realizada por medio de lazos de espera prolongados donde no se encuentra ninguna ventaja en los programas escritos en assembler.

APLICACION A UN SISTEMA DEDICADO

Estas ideas se utilizaron al diseñar y realizar una aplicación para realizar adquisición de datos en corazones de seres vivos en el Laboratorio de Investigaciones Cardiovasculares de la Facultad de Medicina de la Universidad Nacional de La Plata.

La aplicación debía utilizar un conversor analógico digital para obtener los datos, procesarlos calculando nuevos parámetros, graficarlos en pantalla y convertir los parámetros calculados a señal,analógica para graficarlos en registrador con papel (Fig.6).

Una solución del primer tipo consistiría en mediante alguna extensión del lenguaje de programación utilizado, desarrollar una aplicación que controle directamente el conversor (Fig.7a). Típicamente la digitalización y registro sobre una PC-AT permite un intervalo de muestreo de 2,5 mseg. en un canal por vez, utilizando el paquete de software PCLAB [13], [14] y [15]. Esto no cumplía con los requisitos y además se aleja de la arquitectura de capas del PC, por lo tanto se intentó una solución del segundo tipo (Fig.7b).

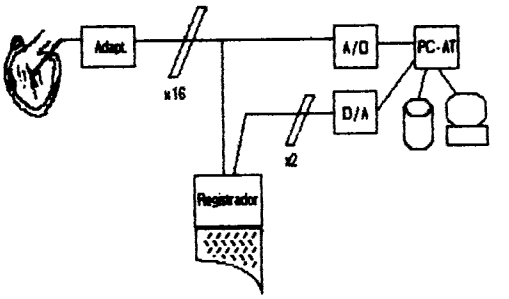


Fig. 6

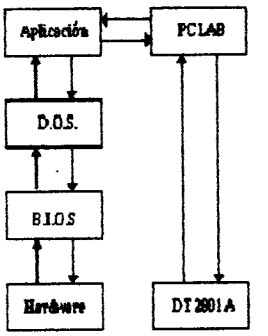


Fig. 7a

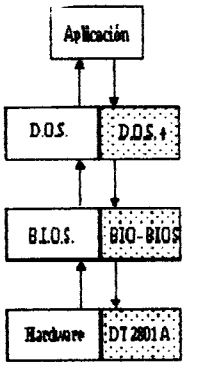


Fig. 7b

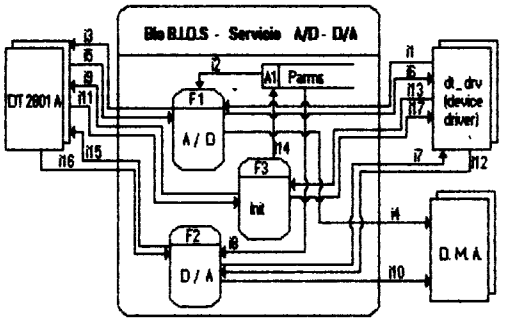
En primer término se realizó la extensión de la capa BIOS, partiendo de la tecnología desarrollada oportunamente en nuestro laboratorio [11], incorporando tres nuevas funciones a éste BIO-BIOS:

** INIT A/D: Se encarga de inicializar la placa conversora, y mantener la información de los parámetros de operación.

** READ A/D: Es la función encargada de la adquisición de datos analógicos.

** WRITE D/A: Se encarga de trasmitir los datos de los parámetros calculados al registrador.

En la Fig 8 se muestra un DFD de nivel uno de las extensiones realizadas.



Flujos de Datos:

i1) Solicitud de conversión A/D y parámetros dinámicos de operación.
i2) Parámetros de conversión A/D.
i3) Datos de programación del conversor A/D.
i4) Datos de programación del DMA.
i5) Resultado de la operación de conversión A/D.
i6) Información de terminación de la operación A/D.
i7) Solicitud de conversión D/A y parámetros dinámicos de operación.
i8) Parámetros de conversión D/A.
i9) Datos de programación del conversor D/A.
i10) Programación del controlador DMA.
i11) Resultado de la operación de conversión D/A.
i12) Información de terminación de la operación D/A.
i13) Solicitud de reinicialización del conversor y Cambio de parámetros estáticos.
i14) Parámetros estáticos.
i15) Parametros para reinicialización del conversor.
i16) Respuesta de la reinicialización.
i17) Información de terminación de la operación.

Fig. 8

En segundo lugar se realizo la correspondiente extensión en el sistema operativo, adoptando la forma de un **device driver** debido a que es el mecanismo provisto por la arquitectura y a que permite tratar al conversor como un archivo.

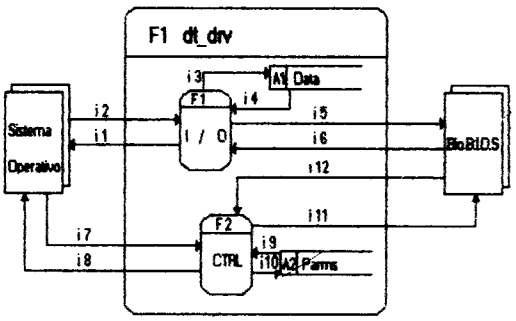
En la implementación se incorporaron tres tipos de funciones, dentro del patrón del **device driver**:

** CONTROL I/O: Es el conjunto de funciones que permiten fijar los parámetros de control del dispositivo, a fin de inicializar su modo de operación. En este caso se fijan primordialmente el número de canales a utilizar, la velocidad de muestreo y la ganancia, y se puede consultar el modo de operación vigente.

** READ/WRITE I/O: Estas funciones realizan las operaciones de adquisición y trasmisión de datos, así como el control del buffer de lectura/escritura.

** OPEN/CLOSE I/O: Respetando la idea de tratar el dispositivo como una unidad lógica más, estas funciones permiten "abrir" y "cerrar" el mismo.

La interacción entre el BIO-BIOS y el sistema operativo mediante estas funciones puede verse en el DFD de nivel uno del device driver mostrado en la Fig. 9



Flujos de Datos:

i1) Datos leídos / estado.
i2) Petición de apertura / cierre, lectura / escritura, flush, datos a escribir
i3) Datos a convertir a analógicos.
i4) Datos digitales ya convertidos.
i5) Petición de conversión de datos y parámetros dinámicos.
i6) Información de operación terminada.
i7) Solicitud de operación de control / Nuevos datos.
i8) Modo actual.
i9) Parámetros solicitados.
i10) Nuevos parámetros.
i11) Nuevos parámetros estáticos de operación.
i12) Operación terminada / Estado

Fig. 9

Un punto que merece especial consideración es la implementación de esta funciones en lenguaje C; el cual compite favorablemente con la tradicional programación en assembler [11].

Limitándonos a utilizar 64 Kb de memoria para el almacenamiento de los datos adquiridos, y teniendo en cuenta que cada muestra ocupa 2 bytes, podemos plantear la siguiente inecuación :

Frec. * Tiempo * Nro. canales * 2<=64 Kb (1)

Donde frecuencia se refiere a la velocidad de adquisición de muestras por canal. Teniendo en cuenta que la máxima frecuencia aceptada por el conversor, entre todos los canales, es de 27400 Hz. y los canales 16, es posible monitorear durante 1 seg., desde 1 canal a intervalos del orden 0,04 mseg. hasta 16 canales con 0,64 mseg. entre dato y dato.

Para los requisitos de la aplicación en cuestión es necesario adquirir datos durante períodos de 3 a 4 segundos utilizando hasta 6 canales por lo que, utilizando (1), podemos calcular el intervalo de muestreo, el cual resulta del orden de 0,8 mseg., cifra suficientemente buena y muy superior a las obtenidas sin las extensiones del BIO-BIOS y el device driver desarrollados.

CONCLUSIONES

Se ha presentado el desarrollo de un BIOS orientado a aplicaciones de tiempo real, así como alguna de las aplicaciones realizadas a partir del mismo.

La linea actual de trabajo es la de incorporar a nivel de BIOS y Sistema Operativo primitivas relacionadas con la aplicación especifica del sistema dedicado (por ejemplo adquisidores de datos distribuidos para procesamiento y control industrial en tiempo real). Esta linea de investigación y desarrollo busca obtener una mayor eficiencia y portabilidad en los sistemas de software para aplicaciones de ingenieria de tiempo real.

REFERENCIAS

[1] Alford, M.W. "A requirements Engineering methodology for Real Time processing requirements", IEEE Trans. Software Engineering. Jan 1977.

[2] Faulk, S. y Parnas, D. "On syncronization in hardware real-time systems", Comm. of the ACM. Mar 1985.

[3] Dahl, G. y Lahti, J. "An investigation of methods for production and verification of highly reliable software", IFAC workshop SafeComp. Stuttgart, May 1979.

[4] Gehani, N. y McGettrick, A. (Ed.) "Software specification techniques" Addison Wesley 1986.

[5] Balzer, R. y Goldman, N. "Principles of good software specification and their implications for specification language". Proc. of Specifications of reliable Software Conference. Cambridge Ap. 1979

[6] Quirk, W.J. (Ed.). "Verification and validation of Real Time software" Springer-Verlag 1985.

[7] Naiouf, M. y De Giusti, A. "Ambiente para la especificación de sistemas de Tiempo Real con Redes de Petri" Informe Técnico LIDI 89-3. 1989

[8] Azema, P., Berthoumieu, B. y Decitre "The design and validation by Petri nets of a mechanism for the invocation of remote servers" IEEE Trans. Computers. Jul. 1982.

[9] Nelson, R., Haibt, L. y Sheridan, P. "Casting Petri nets into programs" IEEE Trans. Software Engineering. Sept. 1983

[10] Hoare, C.A.R. "Communicating sequential processes" Prentice Hall 1985.

[11] Torres, H.F., Cardos, M., Lizzoli, J. y De Giusti, A. "Software para control de tiempo real: la concepción estructurada de un BIOS" Anales de la XIV Conferencia Latinoamericana de Informática. Buenos Aires 1988.

[12] Torres, H.F. y De Giusti A. "Código romable en lenguaje C", Informe Técnico LIDI 88-060110, 1988.

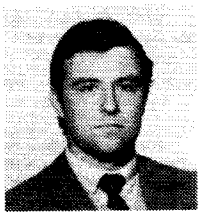
[13] Data Translation Inc. "User manual for DT2801 Series: Single Board analog and digital I/O system", 1985.

[14] Data Translation Inc. "Preliminary User Manual PCLAB" versión 2.00, 1985.

[15] Gelpi, R.J., Hittinger. L., Fujii, A.M., Crocker, V.M., Mirsky, I. y Vatner, S.F. "Sympathetic augmentation of cardiac function in developing hypertension in concious dogs", Am. J. Physiol. 255 (Heart Circ. Physiol. 24):H1525-H1534.

José C. Lizzoli es estudiante de la Licenciatura en Informática en la Facultad de Ciencias Exactas de la Univ. Nac. de La Plata (Argentina) y se desempeña como Becario en el Laboratorio de Investigación y Desarrollo en Informática (L.I.D.I.) en temas relacionados con Software de Base (BIOS/ Sistemas Opereativos) para sistemas dedicados de tiempo real.
Ha participado de diversos proyectos de investigación dentro del LIDI desde 1987.

H. Fabián Torres es estudiante de la Licenciatura en Informática de la Facultad de Ciencias Exactas de la Univ. Nacional de La Plata (Argentina) y se desempeña como Becario en el Laboratorio de Investigación y Desarrollo en Informática (L.I.D.I.) en temas relacionados con Lenguajes y Ambientes para el desarrollo de aplicaciones de Sistemas de Tiempo Real.
Ha participado en diversos proyectos de investigación dentro del L.I.D.I. desde 1987.



Armando E. De Giusti es investigador independiente del CONICET en el Departamento de Informática de la Facultad de Ciencias Exactas de la Univ. Nac. de La Plata (Argentina). Es Profesor Titular de Informática. Sus temas de investigación se relacionan con software para sistemas de tiempo real y aplicaciones de automatización industrial y de oficinas.
De Giusti egresó como Ingeniero Electrónico en la Facultad de Ingeniería de la U.N.L.P. y como Calculista Científico en la Facultad de Ciencias Exactas de la U.N.L.P. en 1973 y desde entonces se ha dedicado a la docencia e investigación universitaria registrando más de 50 publicaciones científicas. Dirige el Laboratorio de Investigación y Desarrollo en Informática (L.I.D.I.).