51
2 ej.

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

SISTEMA DIGITAL DE ADQUISICION DE DATOS

T E S I S

QUE PARA OBTENER EL TITULO DE:

INGENIERO MECANICO ELECTRICO

P R E S E N T A :

JORGE ALBERTO ESTRADA CASTILLO

DIRECTOR DE TESIS: M. EN I. YI TAN LI.

MEXICO, D. F. 1992

TESIS CON FALLA DE ORIGEN

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

DERECHOS RESERVADOS © PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis está protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Indice

Introducción iv

Capítulo I
Sistemas Operativos

Sistemas Operativos y Device Drivers 1
Sistemas Operativos 1
Conceptos y definiciones 2
Sistemas de Control para Computadoras 7
Desarrollo Histórico de MSDOS 9
Estructura de MSDOS 13
Interrupciones 26

Capítulo II
Device Drivers

Concepto 29
Controladores, Adaptadores e Interfaces 30
Dispositivos tipo Caracter y de Bloque 31
Device Drivers Standards de DOS 32
Control de Dispositivos a Través de los Servicios de DOS 34
Traducción de Servicios de DOS a comandos para el Device Driver 35
Reemplazo de los Device Drivers de DOS 37
Formación de la Cadena de Drivers 38
Estructura de la Cadena de Drivers 38
Estructura de un Device Driver 41
Comunicación del Kernel con el Device Driver 54
Llamadas al Driver desde DOS 56
Llamadas a Device Drivers de Bloque y de Caracter 58
Comandos para el Device Driver 59

Capítulo III

SAD

Sistema de Adquisición de Datos

Características 67

Principio de Operación 67

Proceso de Recepción de Datos 68

Proceso de Recuperación de Datos 71

Capítulo IV

SAD

Device Driver

para el Sistema de Adquisición de Datos

Estructura y Operación del Driver SAD 73

Area de Directivas 74

Device Header 76

Area para Almacenamiento de Variables Locales 77

Sección de Estrategia 81

Sección de Interrupciones 81

Area para definición de Procedimientos Locales 84

El Interrupt Handler del Sistema de Adquisición 84

Sección de Comandos 93

Inicialización 93

Area deshechable 95

Configuración del Vector de Interrupciones 96

Lectura de Parámetros para Velocidad de Recepción 97

Configuración del PIC 82C59A-2 101

Inicialización de los UARTs 102

Comando 3 IOCTL_Input 103

Transferencia de datos de memoria Extendida a Principal 104

Comando 4 Input 106

Comando 12 IOCTL_Output 107

Comando 13 Open 109

Comando 14 Close 109

Sección para Actualización de Status 110

Sección para Restauración de Registros del CPU 111

Puesta en Operación del Sistema SAD 112

Capítulo V

Consideraciones para el Diseño de la Tarjeta de Interface del Sistema de Adquisición de Datos SAD

Consideraciones de Diseño de la Tarjeta de Interface 117
Decodificación y Direccionamiento de puertos de I/O 124
Decodificación de Direccionamiento Fijo 124
Decodificación de Direccionamiento por Switch 125
Decodificación de Direccionamiento con Memoria PROM 127
Decodificación de Direccionamiento Indirecto 128
Decocificación de Direccionamiento por Mapeo de Memoria 129
Técnicas de Transferencia de Datos 131
Selección de UART'S para la Tarjeta de Interface 134
Selección del Controlador de Interrupciones del Sistema SAD 135

Capítulo VI

Implementación de la Tarjeta de Interface del Sistema SAD

Operación del Controlador de Interrupciones 82C59A-2 137
Modos de Operación del Controlador de Interrupciones 143
Modo de Fully Nested 143
Modo de Rotación de Prioridades 144
Detección por Flanco ó Nivel de la Solicitud de Interrupción 144
Modo de Máscara Especial 145
Modo de Sondeo 145
Modo Especial de Fully Nested 145
Modo Buffered 146
Programación del Controlador de Interrupciones 146
Comandos de Inicialización del Controlador de Interrupciones 146
Comandos de Operación del Controlador de Interrupciones 149
Inicialización y Operación del UART SCC2698B 153
Modo de Fuera de Tiempo 161
Modo de Despertar 162
Descripción de los Registros del UART SCC2698B 162
Diseño de la Tarjeta de Interface: Sistema SAD 169
Programación de Inicialización del Controlador de Interrupciones 172

Programación de Inicialización del UART SCC2698B 174

Conclusiones 179

Apéndices
Diagramas de Tiempos Teóricos 182
Formato de Recepción de Datos 189
Rutinas de Interface del Sistema SAD 190
Diagramas de Tiempos de Operación 198

Bibliografía 207

"Y sin embargo se mueve..."
y con sus movimientos
manifiesta al hombre,
la fuerza brutal en su
interior contenida...

La Tierra.

Galileo Galilei

"Los mensajeros que galoparon hasta la capital China de Loyang, descubrieron con asombro que las noticias de un devastador terremoto ocurrido 600 Km. al noroeste de la capital, llegaron antes que ellos. Aunque nadie en esa ciudad percibió movimiento alguno, un ingenioso artefacto, con un mecanismo sensible al movimiento, detectó el sismo e indicó su dirección." La instrumentación sísmica había comenzado. (138 A.D.).

Time Life

Los terremotos desde la creación, han sido uno de los fenómenos naturales que mayor incertidumbre y desolación han causado en el espíritu del hombre.

Su estudio sistemático es reciente, pero el avance tecnológico en el campo de la electrónica y la computación, ha ayudado a los especialistas a comprender las causas que los originan y a identificar las regiones donde se producen.

Actualmente, los esfuerzos en todo el mundo, están encaminados hacia la obtención de datos que, en calidad y cantidad suficiente, permitan desarrollar teorías capaces de modelar con mayor exactitud el comportamiento de zonas de alto riesgo sísmico. El Sistema de Adquisición de Datos presentado en este trabajo se suma a este esfuerzo.

El lograr el anterior objetivo, contribuirá sin duda a reducir los enormes estragos que los sismos causan entre amplios sectores de la población.

Introducción.

México, a consecuencia de estar situado en una zona de intensa actividad sísmica, ha logrado un importante desarrollo en cuanto a recursos humanos e infraestructura se refiere, para el estudio de este fenómeno natural.

La identificación de zonas sísmicas y la evaluación del riesgo sísmico correspondiente, requiere de la implementación de redes de monitoreo sísmico que permitan detectar y registrar los movimientos telúricos de las regiones de interés.

Una *red sísmica,* se compone generalmente de dos sistemas: el de *instrumentación de campo y el de adquisición y procesamiento de datos.* Con el primero, se detectan los movimientos del suelo, con el segundo se realiza el registro y el análisis de los datos que le son transmitidos por el sistema de instrumentación.

Como parte del proceso de modernización llevado a cabo por el Servicio Sismológico Nacional, en combinación con el Departamento de Sismología del Instituto de Geofísica de la U.N.A.M., se han realizado importantes aportaciones tecnológicas en el área de instrumentación sísmica y en los sistemas de adquisición. Uno de estos desarrollos, culminó con la construcción de un sistema completo de instrumentación de campo: la estación sismológica digital GEOS-3. (Tan Yi, 1989)

Si bien, con la estación GEOS-3, se dió un paso muy importante hacia el desarrollo de una red sísmica completamente digital, aún faltaba por desarrollar un sistema que pudiese adquirir y procesar los datos transmitidos por esta clase de estaciones.

Así nace el proyecto ***SAD*** (Sistema de Adquisición de Datos), con el objetivo de lograr que los datos transmitidos por estaciones GEOS-3, puedan ser captados, graficados y procesados con un programa de detección en tiempo real, con la ayuda de una computadora.

El sistema ***SAD***, puede recibir hasta 16 canales con información digital proveniente de las estaciones GEOS-3. Para lograr esto, fué necesario desarrollar una tarjeta que posibilitara la adquisición de datos y un device driver que se encargara de organizarlos y permitiera al mismo tiempo, el fácil acceso a los mismos desde un programa de aplicación.

En el primer capítulo de este trabajo, se presentará un panorama general de la función que desempeñan los sistemas operativos, dentro del esquema de operación de una computadora. Se tratarán las diversas clases de sistemas operativos que existen en función de su grado de sofisticación. Se explicará la importancia que tiene la implementación de programas de aplicación, independientes del hardware de la máquina. Así mismo, se describirá en detalle la forma de operación del sistema operativo MSDOS.

El segundo capítulo, versará sobre la estructura y principio de operación de los *device drivers*. Se explicará para qué sirven y cómo se usan. Se diferenciará entre los *drivers residentes* y los *drivers instalables*, y se explicará el mecanismo mediante el cuál, los drivers instalables reemplazan a los residentes y el porqué. Se analizarán cada una de las secciones que conforman la columna vertebral de un device driver y la función que cada una realiza. Para concluir este tema, se expondrá la manera como el driver se comunica con el sistema operativo y se hablará de los comandos que se usan para tal fin.

En el tercer capítulo, se presentará una visión global del esquema de operación del *Sistema SAD*, incluyendo los procesos de recepción, almacenamiento y recuperación de datos. Se presentarán los componentes que lo integran y se expondrán sus características fundamentales.

En el cuarto capítulo, se analizarán en profundidad cada una de las secciones que conforman el *device driver del sistema SAD*. Aquí, se efectuará la descripción de cada una de sus componentes y la función que realizan será justificada. En este capítulo, se presentan varios ejemplos, donde se reseña el uso de diversas rutinas de interface, que posibilitan la interacción entre un programa de aplicación y el driver SAD.

El quinto capítulo, inicia con las consideraciones de diseño para el desarrollo de tarjetas de interface. En la primera parte, se analizan las opciones disponibles para el acoplamiento de interfaces en la arquitectura de la computadora. Así, se presentan alternativas para la transferencia de datos entre la tarjeta de interface y la computadora, y se discuten las diversas técnicas de decodificación y control disponibles. En la segunda parte, se da inicio a la selección de los dispositivos, que permitirán efectuar la recepción de los 16 canales de datos propuestos.

El sexto capítulo, detalla la operación de la tarjeta de Adquisición de Datos Digitales **SAD**. La descripción hace enfásis en dos partes fundamentales de la misma: los dispositivos de recepción (UARTs) y el controlador de interrupciones (PIC). Se habla de su funcionamiento en conjunto y de la interacción que realizan con la computadora.

El apéndice A, muestra algunos diagramas de tiempo que ilustran la operación de los dos circuitos más importantes de la tarjeta SAD.

El apéndice B, presenta los dos formatos que utilizan actualmente las estaciones sismológicas *GEOS-3*, para la transmisión de una o tres componentes de datos.

El apéndice C, aborda la forma de programación y uso de las rutinas de interface.

El apéndice D, muestra algunos diagramas de tiempo donde se puede apreciar la forma como el CPU realiza la atención de los canales de recepción de la tarjeta SAD.

Sistemas Operativos

Sistemas Operativos y Device Drivers.

Un device driver, es un programa que forma parte del sistema operativo de una máquina, su función es brindar al desarrollador de aplicaciones una interface de alto nivel para el acceso a dispositivos periféricos. El concepto de device driver está estrechamente ligado a la estructura y forma de operación de un sistema operativo, para comprender su naturaleza debemos entender el entorno donde realiza su función.

Sistemas Operativos.

Un sistema operativo, es un programa que se encarga de llevar a cabo la administración de los recursos de la máquina, que provee al usuario de un mecanismo de comunicación que permite la interpretación de comandos y que controla la ejecución de programas.

En su función como administrador de recursos, el sistema operativo determina la asignación de los mismos en base a prioridades.

Como procesador de comandos, interpreta las instrucciones del usuario y da los pasos necesarios para efectuar su ejecución.

Como controlador, el sistema operativo debe evitar que errores en la programación de aplicaciones, alteren la realización de tareas independientes que se ejecutan simultáneamente en la computadora. Si un programa falla por cualquier causa, el sistema operativo debe recuperarse y retomar de nueva cuenta el control de la máquina.

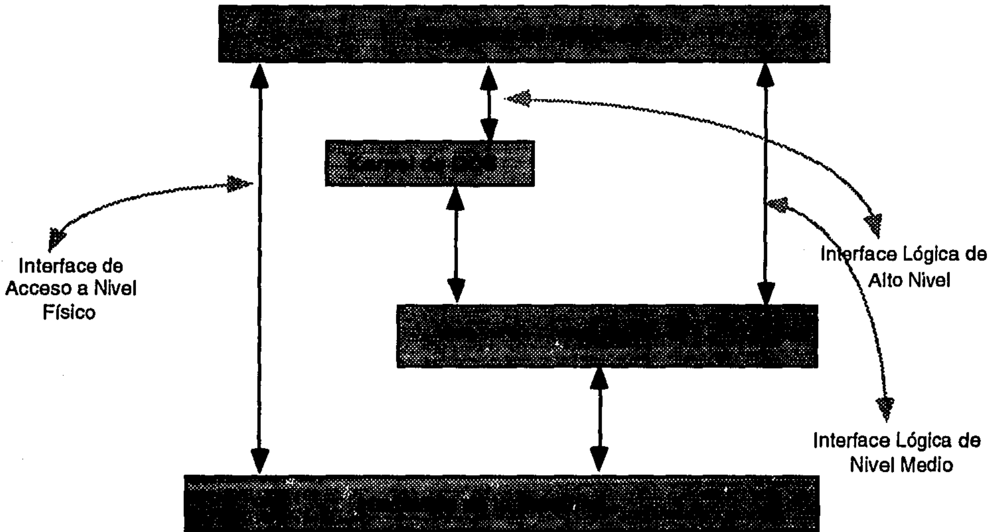
Dentro de la diversidad de sistemas operativos que existen, se pueden identificar al menos tres distintos niveles de sofisticación en los mismos, que serán tratados más adelante.

Todos los sistemas de control para computadoras, tienen ciertas características en común, y por esa razón, todos ellos son referidos como sistemas operativos. Estos toman el control del hardware al encender la máquina, realizan las tareas de configuración y ponen a disposición del usuario un interpretador de comandos y una interface para el desarrollo de aplicaciones.

Antes de proceder con la descripción de los diferentes niveles de programación existentes para el control de las computadoras, es conveniente precisar el significado de algunos de los términos utilizados.

Conceptos y Definiciones

Una *aplicación o programa de aplicación* consiste de una serie de instrucciones dirigidas al CPU, que al ser ejecutadas producen una función útil para el usuario de la computadora.



El presente esquema nos muestra los tres diferentes niveles de Acceso en los que puede operar un programa de aplicación bajo MSDOS

Un *programa de interface para desarrollo de aplicaciones* (API) comprende de una serie de librerías que proporciona el sistema operativo al desarrollador de aplicaciones, para el acceso a dispositivos periféricos conectados a una máquina, como pueden ser teclados, displays, impresoras y discos. El API (Aplication Program Interface) que se suministra al usuario puede variar en complejidad. Así, cuando el programador debe involucrarse con las peculiaridades del hardware para lograr el acceso y control del mismo, se dice que el API con el que trabaja es a nivel físico. En cambio cuando el

programador no necesita profundizar en el funcionamiento del hardware para lograr su control o acceso, el API utilizado es a nivel lógico. La tendencia de los sistemas operativos actuales es ofrecer al programador un API a nivel lógico, para que el tiempo empleado en el desarrollo de aplicaciones sea menor.

En un sistema operativo *multiusuario*, pueden trabajar varias personas al mismo tiempo utilizando los servicios de un sólo procesador. La mayor parte de los sistemas operativos para minicomputadoras y equipos grandes son sistemas de este tipo. Un sistema multiusuario es considerado también un sistema *multiproceso*, lo que significa que un sólo usuario puede estar ejecutando en forma simultánea varios procesos a la vez sobre un sólo CPU.

Un sistema operativo *multitarea*, es aquél que brinda al usuario el entorno necesario para trabajar con varias tareas a la vez, con la peculiaridad que sólo una de ellas está en ejecución, quedando las demás en estado de espera. Por ejemplo, una persona está con tres libros abiertos sobre su escritorio, los tres libros quedan a su alcance y disponibles, pero únicamente puede realizar la lectura de sólo uno a la vez. Un sistema multiproceso puede considerarse como un sistema multitarea, pero un sistema multitarea no es un sistema multiproceso, es un sistema con capacidad para la ejecución de un sólo proceso cada vez, por lo tanto es capaz de soportar únicamente a un usuario.

Si el sistema operativo maneja código reentrante, las rutinas disponibles en memoria pueden ser accesadas y compartidas por dos o más programas simultáneamente. Los sistemas operativos multiusuario sobre todo, permiten el uso de esta facilidad. Cuando se utiliza el código reentrante, las variables intermedias de la rutina compartida se almacenan en un stack, lo que evita la modificación o destrucción de las mismas, al ser esa rutina utilizada por otros programas en forma simultánea.

La memoria de la computadora, desde el punto de vista de la aplicación puede ser de tipo real o virtual. Si la memoria es real, los programas de aplicación pueden accesar directamente las localidades de memoria y modificar la información en estas contenida. Cuando la memoria es virtual, el sistema operativo provee al usuario de un bloque imaginario de memoria, el cuál es mapeado posteriormente a memoria real. El acceso a las localidades de memoria, siempre es

realizado a través del sistema operativo y da por resultado que la dirección lógica que se maneja en un programa no corresponde a la misma dirección física que le asigna el sistema operativo. A menudo los sistemas que trabajan bajo este esquema, pueden soportar programas con tamaños mayores a la capacidad física disponible de memoria RAM, mediante su modularización automática, de esta forma, al ser un programa invocado para su ejecución, sólo se aloja en memoria un módulo del mismo, dejándose la parte restante en un área de swaping o de intercambio en el disco, en espera de ser cargada cuando la ejecución de su código sea requerida.

Modos de Operación del Microprocesador i80286.

El microprocesador intel 80286, puede operar en modo real y en modo protegido. La diferencia básica entre uno y otro modo tiene relación con el método de direccionamiento que se usa en cada caso, para el acceso de memoria. (Intel, 1987).

Modo de Direccionamiento Real.

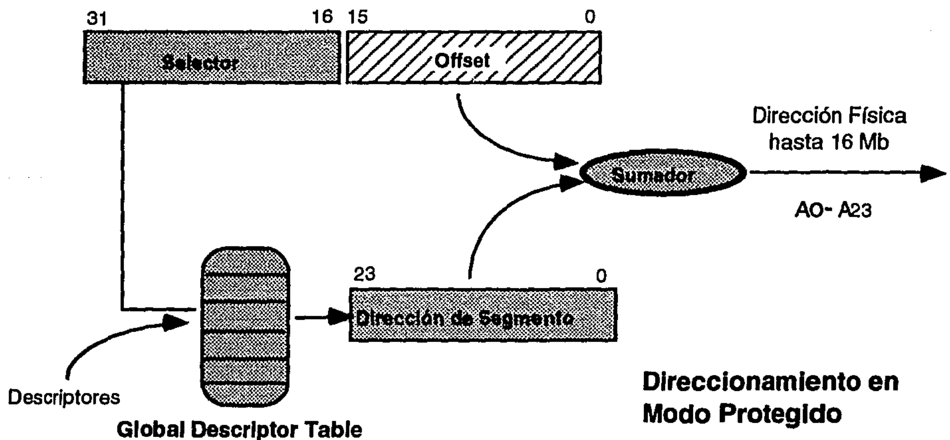
Este tipo de direccionamiento, tiene su origen en la arquitectura y forma de operación de los microprocesadores i8086 e i8088. Aunque a partir del procesador i80286 se cuenta ya con una arquitectura más avanzada para el direccionamiento de memoria, Intel puso mucho cuidado en mantener la compatibilidad de este procesador con su predecesores. De ésta forma, el i80286 puede operar en modo real.

El esquema de direccionamiento en modo real, permite el acceso hasta de 1,048,576 bytes de memoria RAM, a través de las salidas A_0 - A_{19} del CPU. En este caso, la problemática que existe en el direccionamiento de memoria, tiene relación con el número de bits que puede manejar un registro del procesador i80286, que son 16, y el número de líneas necesarias para accesar 1 Mb de memoria, que son 20. Para poder manejar esas líneas, se optó por la utilización de dos registros: uno de offset y otro de segmento. El registro de segmento controla los valores de las salidas A_4-A_{19}. El registro de offset maneja los valores de salida A_0-A_{15}. Así, para accesar cualquier localidad de memoria dentro del modo de operación real, es necesario especificar los valores de segmento y offset que el CPU sumará internamente para generar una salida adecuada en las líneas de direccionamiento A_0-A_{19}.

Modo de Direccionamiento Protegido.

Este modo de operación es introducido a partir de los microprocesadores i80286. Estos incorporan en su arquitectura 4 líneas adicionales para el acceso de memoria, lo que da un total de 24 líneas y una capacidad de direccionamiento hasta de 16 Mb de RAM. El tamaño de los registros del CPU sin embargo, continúa siendo de 16 bits, por lo que para poder controlar las 24 líneas de salida del CPU se utiliza un método llamado modo de direccionamiento protegido.

El esquema de operación del modo protegido involucra la definición de dos registros, uno que funciona como selector y otro que funciona como offset. Con el registro de selección, el programador accesa una tabla de 8192 elementos, conocida como Global Descriptor Table, en la cuál son definidas unas estructuras llamadas descriptores. Cada descritor contiene un valor de 24 bits a través del cuál es posible lograr el acceso a cualquier localidad de memoria dentro de los 16 Mb de la máquina. Puesto que sólo es posible definir a través de los descriptores 8192 localidades de memoria, es necesario utilizar el registro de offset para lograr el acceso a localidades no definidas directamente en la tabla de descriptores. El CPU, cuando opera en modo protegido, suma automáticamente el valor definido en el descriptor seleccionado con el valor del registro de offset indicado por el programa.



Direccionamiento en Modo Protegido

Conmutación de Modos en i80286.

El microprocesador i80286 inicia siempre su operación en modo real; para conmutar a modo protegido, es necesario encender el bit PE (Protection Enable) de la palabra de status de la máquina (MSW), mediante la instrucción LMSW.

El sistema operativo MSDOS fué diseñado para operar exclusivamente dentro del modo real del CPU, por lo que los programas que funcionan bajo su entorno, no disponen de una función de DOS para lograr el acceso a localidades situadas arriba de 1 Mb de RAM. A la memoria situada después del primer Mb de RAM, se le denomina ***memoria extendida***.

El BIOS *(Basic Input Ouput System)* proporciona sin embargo, una función que permite a los programas de aplicación el acceso restringido a memoria extendida para el intercambio de datos. Cuando se utiliza la función 87h en conjunto con la interrupción 15h del BIOS, el procesador conmuta su operación de modo real a modo protegido y realiza la transferencia de datos entre la memoria principal y la memoria extendida de acuerdo a las direcciones fuente y destino definidas en una tabla que se configura antes de que la instrucción int 15h función 87h sea ejecutada. Al finalizar la operación de transferencia, la rutina del BIOS regresa el CPU al modo de operación real. (Duncan, 1988).

Un problema que se presenta en la conmutación de modos es que los tiempos utilizados para cambiar la operación de modo real a modo protegido y viceversa son significativos, por lo que el uso de la función 87h del BIOS, no debe hacerse indiscriminadamente para evitar que el sistema se degrade. Los tiempos de conmutación varían según la marca de la máquina en que se trabaje. En general las máquinas que usan procesador i80386 ofrecen tiempos de conmutación bastante menores a los que se encuentran en las máquinas con procesador i80286. La máquina utilizada en el desarrollo del driver ***SAD***, fué una ACER 915v, con procesador 80286 a 16 Mhz, con un tiempo de conmutación de modo real a protegido de 160 microsegundos, y de modo protegido a real de 251 microsegundos. En total 411 microsegundòs son consumidos en la conmutación de modos. Esta condición obliga a que las transferencias de datos entre memoria extendida y memoria principal deban realizarce en bloques extensos, y no de byte en byte, pues de hacerlo así, el tiempo de conmutación ocuparía gran parte del ancho de banda disponible del microprocesador, no haciendo viable la implementación del sistema.

Niveles de Sofisticación en los Sistemas de Control para Computadoras.

Nivel Cero:

Los sistemas de control para computadoras de nivel Cero, denominados también ROM Monitor, están diseñados para llevar a cabo la ejecución de diversas tareas de arranque dentro de la computadora cuando ésta es encendida. Esas tareas abarcan desde la configuración del hardware de la máquina hasta la puesta en ejecución de un programa único que trabajará en algunos casos durante todo el ciclo de operación de la máquina (Vgr. un sistema de control industrial). Los sistemas de control de nivel Cero son almacenados en chips de ROM e instalados como parte del hardware para formar el firmware de la máquina. Puesto que este tipo de sistemas están diseñados para trabajar con un sólo programa, el API se maneja a nivel físico, debido a que el programador de la aplicación debe conocer en detalle el funcionamiento del hardware del sistema para lograr su control y acceso. De acuerdo con esta definición, el ROM-BIOS (Read Only Memory, Basic Input Output System), que se encuentra en las computadoras IBM PC y compatibles, es un sistema de control nivel cero, ya que arranca la computadora configurando el hardware asociado y pone en ejecución un único y pequeño programa llamado loader. (Townsend, 1989).

Nivel Uno:

Un sistema de control de nivel Uno, es más sofisticado que el discutido anteriormente. Este tipo de sistemas generalmente son capaces de ejecutar programas variados, pero sólo uno a la vez. El código no es reentrante pero la interface que ofrecen para el desarrollo de aplicaciones API es de tipo lógico en su generalidad. MSDOS cae dentro de esta clasificación, ya que maneja los recursos de la máquina permitiendo la ejecución de un sólo programa cada vez y provee al desarrollador de aplicaciones de una interface lógica para operaciones de I/O sobre dispositivos. Los sistemas de control de nivel 1 son frecuentemente llamados sistemas operativos, pero desde un punto de vista más estricto, no debieran recibir esta clasificación. Cuando un programa es ejecutado en este nivel, los recursos de la máquina pasan casi en su totalidad bajo el control de la aplicación. MSDOS por ejemplo, sólo interviene intensamente durante el proceso de carga en memoria del programa de aplicación y durante las llamadas al sistema de archivos. En las demás operaciones, la

aplicación puede realizar directamente accesos a memoria y a dispositivos sin la intervención del sistema operativo. Esa pérdida de control sobre algunos de los recursos de la máquina durante la ejecución de programas de aplicación, hace que algunas personas no consideren a MSDOS como un sistema operativo real.

Nivel DOS:

Los sistemas de control de nivel 2, son en el estricto sentido de la palabra, los únicos sistemas operativos reales. Estos siempre retienen el control de los recursos de la computadora, administrando la ejecución de procesos, la asignación de memoria y el uso de periféricos. Esta clase de sistemas es un orden de magnitud mayor en tamaño y complejidad que los sistemas operativos de nivel 1. Un sistema de control de Nivel 2, presenta una completa interface lógica para aplicaciones, manejo de código reentrante y uso de memoria virtual. Ejemplos de este tipo los tenemos en los sistemas operativos VMS, Xenix, Unix y OS/2. Bajo esta definición tal vez sea más claro entender el porqué MSDOS como un sistema de control de nivel 1, no es un sistema operativo verdadero, puesto que no retiene el control de la máquina durante la ejecución de programas, no soporta multitareas, no hace previsiones para el manejo de código reentrante y no maneja memoria virtual.

Desarrollo Histórico de MSDOS.

En sólo un década MSDOS ha evolucionado hasta convertirse en un sofisticado sistema operativo para computadoras personales basadas en la familia de microprocesadores Intel 8086. El progenitor de MSDOS fué un sistema operativo llamado 86-DOS, el cuál fué escrito por Tim Paterson para la compañía Seattle Computer Products a mediados de 1980. En aquellos tiempos, el sistema operativo CP/M-80 de la Digital Research fué el más comúnmente usado en microcomputadoras basadas en los microprocesadores Intel 8080 y Zilog Z-80. Una amplia gama de aplicaciones fueron desarrolladas para correr en tal ambiente, como procesadores de palabras y manejadores de bases de datos.

En ese tiempo, comenzaron a surgir microprocesadores más poderosos, ahora de 16 bits, los cuáles para poder ser introducidos en el mercado con buenas posibilidades de aceptación deberían ser capaces de ejecutar las cientos de aplicaciones ya existentes, desarrolladas para el i8080 o el Z-80.

Para facilitar el translado de esas aplicaciones basadas en micros de 8 bits bajo CP/M-80 hacia la nueva generación de micros de 16 bits, fué diseñado el sistema operativo 86-DOS, bajo la filosofía de proporcionar las mismas funciones y estilo de operación que CP/M-80. Esto dió por resultado, que los programas existentes que corrieran bajo CP/M-80 pudieran ser convertidos casi mecánicamente mediante el procesamiento de su código fuente, a través de un programa traductor especial que los dejaba listos para ejecutarse bajo 86-DOS con un procesador i8086.

Pero 86-DOS, fué vendido como sistema operativo propietario de Seattle Computer Products en una linea de microcomputadoras basadas en i8086 utilizando un bus llamado S-100, por lo que el impacto que tuvo éste en el mundo de las microcomputadoras fué mínimo. Otros competidores que hacían computadoras basadas en i8086 estaban renuentes a adoptar un sistema operativo de la competencia y esperaban con impaciencia la nueva versión de la Digital Research, el CP/M-86.

En Octubre de 1980, IBM convocó a las compañías desarrolladoras de software a que presentaran propuestas de un sistema operativo para la nueva línea de computadoras personales que

estaban diseñando. Microsoft no tenía sistema operativo alguno que ofrecer, pero acordó el pago de regalías a la Seattle Computer Products por el derecho de venta del sistema operativo de Paterson 86-DOS. En Julio de 1981, Microsoft compró todos los derechos del 86-DOS, haciendo alteraciones substanciales al mismo, renombrándolo MSDOS. Así cuando la primera IBM PC fué vendida en el otoño de 1981, IBM ofreció MSDOS como su sistema operativo primario. (PC-DOS 1.0). IBM también seleccionó a CP/M-86 de la Digital Research y P-system de Softech como sistemas operativos alternos para la PC. Sin embargo, ambos parecieron lentos en su operación respecto a PC-DOS y sufrieron la desventaja adicional de sus altos precios y falta de lenguajes de programación disponibles. IBM entonces determina dar su apoyo total a PC-DOS, y todos los productos por ellos desarrollados toman como plataforma de operación este sistema operativo, esto obliga a que los desarrolladores de software independientes (thirty-party) tomen a PC-DOS como base de desarrollo para sus aplicaciones, quedando CP/M-86 y P-system relegados por completo del mercado.

IBM fué el único constructor de computadoras OEM (Original Equipment Manufacturer), en incluir MSDOS en todos sus productos (PC-DOS 1.0).

MSDOS version 1.25 (equivalente a IBM PC-DOS 1.1) fue liberado en Junio de 1982 para corregir una serie de errores que existían en la versión inicial, para brindar soporte a la introducción de discos de doble densidad y para mejorar el diseño del kernel de DOS haciéndolo más independiente del hardware de la máquina. Esta versión es incluída ya por otros constructores de equipo como Texas Instruments, Columbia y COMPAQ, que lograron penetrar en el mercado de las computadoras rápidamente.

MSDOS versión 2.0 (PC-DOS v2.0) fué introducido en Marzo de 1983. Este en la práctica, fué completamente diferente en su filosofía de construcción a su antecesor, cuidándose siempre a pesar de ello, de mantener la compatibilidad con el mismo. Algunas de las innovaciones introducidas en este nuevo sistema operativo son listadas en el *cuadro 1.*

La versión 3.0 de MSDOS, aparece en Agosto de 1984 con la introducción de las máquinas AT, basadas en el i80286 de 32 bits. Esta nueva versión, representa otro cambio de importancia que implica el rediseño de parte del sistema operativo a fin de incluir facilidades adicionales como son el soporte para compartir archivos y

registros (file and record locking and sharing) lo cuál preparaba el camino para la introducción en gran escala de los sistemas de redes.

Otra versión importante es la 3.2, lanzada al mercado en 1986, la cuál generaliza la definición de los Device Drivers de tal forma que los nuevos dispositivos periféricos que estaban surgiendo en el mercado, pudieran ser incorporados con mayor facilidad a la arquitectura de las computadoras personales.

- Soporte para discos duros y flexibles de mayor capacidad.
- Muchas de las características de UNIX/XENIX fueron adoptadas, incluyendo la organización de archivos en estructuras jerárquicas, file handles, redirección de I/O, pipes y filtros.
- Impresión en background.
- **Device Drivers Instalables.**
- La introducción del archivo **CONFIG.SYS**, el cuál entre otras cosas, **controla la instalación de Device Drivers** adicionales y la definición del número de buffers que se usarán para el manejo de disco duro.
- La introducción del **device driver ANSI.SYS** que permite a los programas controlar la posición del cursor y las caracteristicas de despliegue en forma independiente del hardware.
- Soporte para el uso de memoria en forma dinámica, que permite la modificación y la liberación de memoria desde los programas de aplicación.
- Soporte para manejar interpretadores de comandos definidos por el usuario (shells).

Cuadro 1. Innovaciones Introducidas en la versión 2.0 de MSDOS

MSDOS versión 4.0 aparece en 1988, proporcionando un shell más rico en su interface con el usuario y soporte para manejar particiones mayores de 32 MB en los discos. En contraparte consume una cantidad considerable de memoria RAM respecto a la versión anterior. MSDOS versión 5.0 logra resolver el problema de consumo de memoria mediante el uso de utilerías que permiten la colocación en memoria expandida de programas residentes y device drivers.

Un nuevo sistema operativo, llamado *Microsoft Operating System* (MS OS/2) fué liberado en 1987, introduciendo nuevos conceptos para los desarrolladores de aplicaciones en el mundo de las computadoras personales. Importantes facilidades fueron incorporadas como: multitareas, código reentrante, mejor soporte para la conmutación entre los modos de operación del CPU (real - protegido) y un sistema de manejo de memoria virtual para aplicaciones con altos requerimientos de graficación. Aunque MS OS/2 es un nuevo producto no derivado de MSDOS, la interface de usuario y el sistema de archivos son compatibles con éste último, por lo que el primero tiene la habilidad para trabajar aplicaciones en modo real, (MSDOS) junto con aplicaciones en modo protegido. A pesar de estas facilidades, su penetración en el ambiente comercial no ha tenido la suficiente fuerza para desbancar a MSDOS. La razón de esto, puede estribar en que a diferencia de lo que sucedió con la migración exitosa de CP/M-80 a MSDOS, donde la tarea para el traslado de un programa de un sistema operativo a otro era trivial, la migración de DOS a OS/2 no es nada fácil, si existe el deseo de poder aprovechar las ventajas adicionales que brinda OS/2. El problema aquí, es que ambos sistemas no son compatibles a nivel del programa de interface para desarrollo de aplicaciones, lo cuál hace necesaria la reprogramación de las mismas, para lograr que se beneficien con las facilidades que les brinda el nuevo sistema. Esto implica un mayor manejo de recursos tanto humanos, como de tiempo y dinero que muchas empresas desarrolladoras de software no pueden asumir, por lo que las nuevas versiones de sus productos van dirigidas aún en exclusiva hacia el mercado de mayor base instalada de usuarios: DOS. De este modo, un usuario que trabaja bajo OS/2, usando aplicaciones orientadas a DOS, por no existir para OS/2 disponibles, en ningún momento verá la ventaja de hacer uso del mismo. Es por ello, que nuevas versiones de MSDOS han sido liberadas en paralelo a las de OS/2.

No es muy claro aún, que OS/2 se convierta en el sistema operativo preponderante del futuro. Unix en cambio, se vislumbra podría entrar en la competencia por este mercado. Lo que es seguro asumir es que MSDOS, aún con sus importantes limitaciones inherentes a su desarrollo histórico, pero con sus relativamente pequeños requerimientos de memoria, adaptabilidad para diversas configuraciones de hardware y enorme base de usuarios instalada, permanecerá en el ambiente de las computadoras personales algunos años más.

Estructura de MSDOS.

El sistema operativo MSDOS conforma un nivel superior de código, que se sitúa sobre la estructura del ROM-BIOS (Read Only Memory, Basic Input Output System). Aunque el ROM-BIOS es independiente del sistema operativo utilizado en una máquina, para efectos de la presente explicación, será considerado como un elemento más dentro de la organización de MSDOS.

El sistema operativo, proporciona una interface de usuario para el uso de la computadora, un sistema de administración de archivos y un programa de interface para desarrollo de aplicaciones (Aplication Program Interface API), con características Nivel 1. MSDOS está dividido en seis importantes niveles para su operación, cuya finalidad primordial en su conjunto, *es aislar la percepción que tiene el usuario del sistema, del hardware de la máquina.* Los niveles son:

- **ROM-BIOS.**
- **Loader**
- **BIOS (extensión)**
- **Kernel**
- **Interface de usuario**
- **Utilerías**

La extensión del BIOS (no confundir con el ROM-BIOS), el kernel y la interface de usuario, están contenidos en tres archivos que son en el orden respectivo: el IO.SYS, el MSDOS.SYS y el COMMAND.COM. El IO.SYS contiene extensiones y modificaciones de las rutinas básicas definidas por el ROM-BIOS y un loader inteligente. En el MSDOS.SYS ó KERNEL, se albergan un conjunto de rutinas que conforman lo que se denomina servicios de DOS y los programas de administración del sistema. El COMMAND.COM contiene la interface de usuario que provee el prompt y comandos intrínsecos. IO.SYS y MSDOS.SYS deben ser los primeros archivos dentro de la estructura del disco de arranque, ya que el loader del ROM-BIOS está exclusivamente diseñado para realizar la búsqueda del archivo IO.SYS en el primer sector del disco. A causa de ello, muchas veces cuando se necesita convertir un disco flexible que ya contiene información en disco de arranque, la operación no puede llevarse a cabo, pues otros archivos de datos ya han ocupado el primer sector del mismo, por lo que sólo moviéndolos y colocando en ese lugar los archivos IO.SYS y MSDOS.SYS será posible que el disco de arranque funcione como tal.

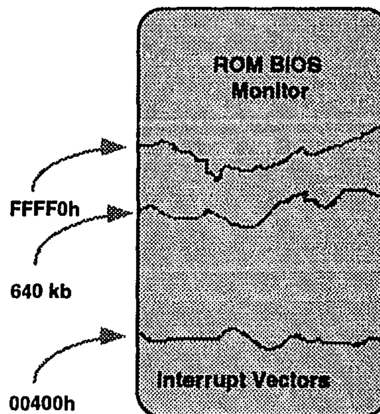
EL papel del ROM-BIOS y su Interacción con MSDOS.

El módulo de ROM-BIOS (Basic Input Output System) es suministrado por el fabricante de la computadora como parte del hardware de la máquina. Este se compone de una serie de programas para el control del hardware almacenados en ROM (Read Only Memory). El ROM-BIOS ó Monitor es independiente del sistema operativo utilizado en la máquina; así por ejemplo, una computadora puede trabajar a partir de un mismo ROM-BIOS con el sistema operativo Xenix o con el MSDOS. La función primaria del ROM-BIOS es la de inicializar el sistema y ejecutar un pequeño programa de carga llamado Loader.

Cuando la computadora es encendida, el ROM-BIOS lleva a cabo la ejecución de la siguiente secuencia de instrucciones:

- **Las rutinas de POST (Power On Self Test) son ejecutadas.**
- **Parte de la inicialización del sistema es llevada a cabo.**
- **El Loader trata de leer el primer sector ya sea del disco flexible o del disco duro.**

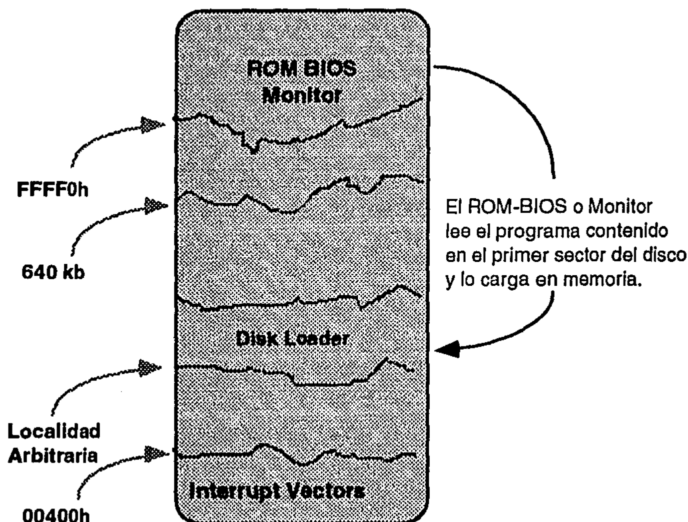
En el primer paso, el POST determina cuáles periféricos están instalados y realiza pruebas sencillas de funcionamiento en los mismos, que le permiten determinar si existen problemas de hardware.



Estado de la memoria principal, un instante después de que la computadora ha sido encendida. La ejecución del código asociado al ROM-BIOS inicia en la localidad FFFF0h.

Durante el proceso de inicialización, se instala la tabla de interrupciones, se configura el hardware y se verifica si ROM-BIOS adicionales han sido instalados (Las tarjetas EGA, VGA, discos duros, tarjetas de red y otros dispositivos cuentan con ROM-BIOS de extensión).

Finalmente, el loader es ejecutado. Este es una rutina cuya finalidad es realizar la lectura de otro programa también de carga aún más sofisticado, que debe de estar almacenado en el primer sector del disco flexible o del duro. Analicemos como se lleva a cabo el proceso:



El ROM BIOS o Monitor sube a memoria el Disk Loader contenido en el primer sector del disco El control del CPU será transferido a continuación a éste último.

El ROM-BIOS o MONITOR trata de leer el primer sector del drive A, si esto falla, entonces el MONITOR trata de encontrar el primer sector del disco duro. El programa Monitor no necesita saber mucho acerca del formato del disco duro, este simplemente irá al inicio del disco e intentará leer un sector de información, mismo que copiará a memoria. Una vez que el MONITOR ha copiado el código del primer sector de un disco (flexible ó duro) a memoria, iniciará la ejecución del mismo. Esta sección de código, corresponde al segundo loader de instalación.

El segundo loader se encargará de subir a memoria un tercer programa llamado sysinit quién se encargará de alojar la parte restante del sistema operativo. Es durante esta secuencia de carga donde es posible insertar las instrucciones necesarias para que en vez de que el segundo loader suba a memoria el programa sysinit, instale en su lugar software especializado para el diagnóstico de la máquina.

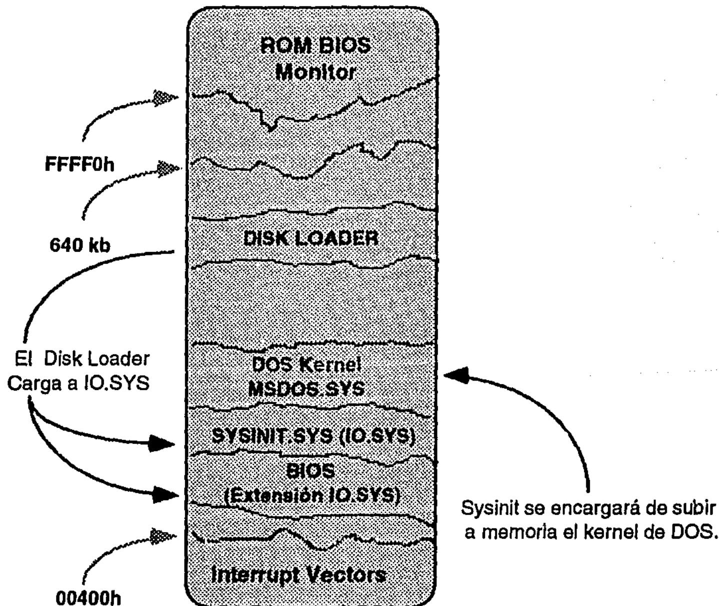
En algunos casos, el papel del ROM-BIOS pasa a segundo término una vez que el sistema operativo es cargado en memoria, ya que toda la comunicación entre las aplicaciones y los recursos de la máquina se lleva a cabo forzosamente a través del mismo. En el caso de MSDOS, el ROM-BIOS puede ser directamente accesado por programas de aplicación. Si usamos las rutinas del ROM-BIOS para el acceso a periféricos tendremos mayor velocidad en la ejecución de la tarea, si usamos los servicios de MSDOS habrá una menor velocidad de ejecución, pero el programador deberá escribir menos código para la aplicación. Al ser los servicios de MSDOS más sofisticados, el programador no tiene necesidad de codificar tantas rutinas para el control y la organización de datos, como cuando utiliza las rutinas del ROM-BIOS que son más primitivas.

Adicionalmente, si lo que se busca es procurar que las aplicaciones sean lo más independientes posibles del hardware de la máquina, diremos que una aplicación diseñada utilizando los servicios de MSDOS, se ejecutará sin problemas en cualquier computadora capaz de correr este sistema operativo. Si la aplicación usa procedimientos no definidos bajo MSDOS, como el acceso directo a las rutinas del ROM-BIOS, al ser el BIOS dependiente del hardware, la aplicación tendrá problemas de ejecución en aquellas máquinas que no sean BIOS compatibles con las máquinas donde se desarrolló la aplicación. Esta clase de problemas con las aplicaciones que usan el ROM-BIOS para aumentar su velocidad de ejecución, tienen su exponente mas claro en las utilerías que se usan para el manejo del sistema de archivos en las computadoras. En general estas utilerías tienden a hacer uso de rutinas del ROM-BIOS, y trabajan bien en ambientes de un usuario, pero simplemente no funcionan cuando ese usuario, intenta accesar los discos pertenecientes a un servidor de archivos dentro de una red. El sistema operativo de una red, en general, es montado como un nivel superior dentro de la estructura de MSDOS y funciona bajo este esquema, brindando provisiones adicionales para compartir recursos entre distintas máquinas. El ROM-BIOS no hace provisiones para el acceso de recursos disponibles bajo un ambiente de RED, es por ello que los programas diseñados en base al ROM-BIOS, no pueden operar bajo esta clase de ambientes.

El Loader de MSDOS.

Este programa es instalado en memoria por el loader del ROM-BIOS al realizar la lectura del primer sector del disco. El loader de MSDOS, es pequeño al igual que el del ROM-BIOS, pero un poco más inteligente, su función es leer el primer archivo del disco, el cuál contiene a su vez un tercer loader, más sofisticado aún, comúnmente conocido como SYSINIT.

SYSINIT se encarga de subir a memoria el resto del sistema operativo. Así, estamos hablando de una carga en tres etapas, en la primera sólo se puede leer el primer sector del disco de arranque, con el loader del ROM-BIOS, en la segunda ya se puede leer el primer archivo del disco de arranque, con el loader contenido en el primer sector del disco, en la tercera es posible terminar de subir a memoria y configurar totalmente el sistema operativo, con el programa SYSINIT.



El Loader contenido en el primer sector del disco (dentro de IO.SYS) cargará en memoria al programa SYSINIT.

La Extensión del BIOS.

El archivo IO.SYS, que se sube a memoria durante la segunda etapa de carga del sistema, está constituído de la siguiente manera:

- **Rutinas y Devices drivers para substitución de módulos del ROM-BIOS.**
- **El loader SYSINIT.**

Las nuevas rutinas aquí definidas, pueden usar componentes del ROM-BIOS o extenderlo. La ventaja de estas rutinas respecto a las anteriores es que prácticamente no están limitadas en cuanto a la longitud de su código, por lo que pueden ser más inteligentes y flexibles para adaptarse a los cambios del sistema. Podemos decir, que las rutinas que se incluyen en la extensión del BIOS forman parte de un mecanismo muy apropiado para hacer cambios, correcciones o adiciones a las operaciones básicas de I/O definidas en el ROM-BIOS. Con esto se elimina la necesidad de reemplazar los chips de ROM cuando surge alguna variación en la arquitectura de la máquina no considerada por el ROM-BIOS.

Si nuevos dispositivos se introducen como equipo standard de la computadora, los programas para su control y acceso pueden ser incluídos dentro del archivo IO.SYS. Si esta operación se realiza, el usuario de la computadora no tendrá necesidad de definir un device driver en el archivo CONFIG.SYS que se encargue del manejo del nuevo dispositivo, pues el fabricante, previamente, se habrá tomado la molestia de incluirlo en el archivo IO.SYS. El hacer inclusiones de los device drivers para dispositivos periféricos dentro de la estructura del archivo IO.SYS, tiene también sus desventajas; la principal es que el dispositivo no podrá ser utilizado como periférico en una computadora de otra marca, a menos que el fabricante del dispositivo proporcione un device driver adicional en disco que pueda instalarse en el archivo config.sys de otra máquina. La decisión final se reduce a una cuestión de políticas por parte del fabricante: si el dispositivo que suministra como standard en la configuración de la máquina no desea que sea utilizado en máquinas de otra marca, incluye el device driver en el IO.SYS (casos así eran frecuentes en las máquinas HP). Si su pretención en cambio es ofrecer una arquitectura abierta, proporcionará un archivo que contenga un device driver instalable para que el nuevo dispositivo pueda utilizarse también con otras clases de máquinas. La utilización de arquitecturas abiertas, es lo que se ha impuesto en el mercado, esto evita en un alto grado los problemas de compatibilidad entre computadoras. El uso del archivo IO.SYS para

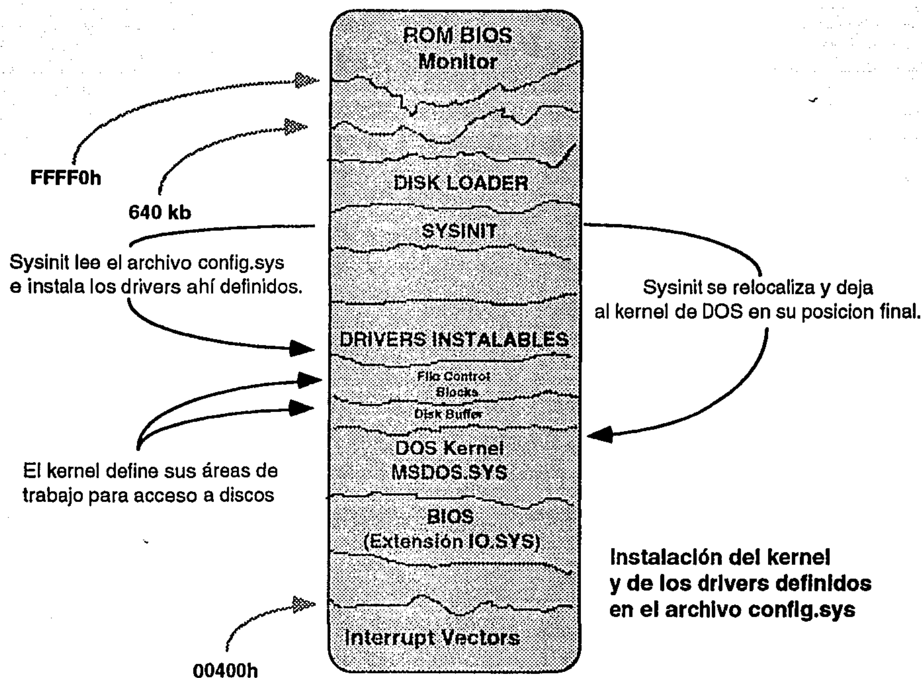
subsanar problemas graves en el ROM-BIOS o para introducir devices drivers en él, ocasiona pérdidas considerables de compatibilidad con otras máquinas, comenzando por el hecho de que un disco de arranque creado en una máquina no podrá ser utilizado por otra que necesita de un IO.SYS especial.

Cuando el programa SYSINIT se invoca a través de la Extensión del BIOS (acondicionada por el fabricante de la máquina), SYSINIT determina primeramente la cantidad de memoria presente en el sistema y se relocaliza en la parte alta de la misma. A continuación mueve el kernel de DOS de la localidad de memoria donde había sido dejado originalmente, hacia su posición final, ocupando parte de las localidades donde se alojaba SYSINIT en un principio.

Ya finalizado el proceso de relocalización del kernel, SYSINIT pasa de inmediato el control al código de inicialización presente en el archivo MSDOS.SYS. Cuando esto sucede, el kernel de DOS acondiciona sus áreas de trabajo y tablas internas, configura los vectores de interrupción (20h al 2fh) y hace un barrido sobre la cadena de device drivers, llamando a las funciones de configuración de cada uno. A través de estas funciones se determina el estado del equipo, se llevan a cabo las operaciones de inicialización necesarias en el mismo y se configuran los vectores de interrupción con las direcciones de los interrupt handlers encargados de dar servicio a las interrupciones generadas por el hardware asociado.

Como parte de la secuencia de inicialización, el kernel de DOS examina los Disk Parameter Blocks (DPB), que son creados por los device drivers encargados del manejo de los discos, determinando el tamaño máximo del sector ahí definido, que será usado por el sistema para el cálculo del tamaño de los buffers de acceso al disco. El control pasa nuevamente a SYSINIT.

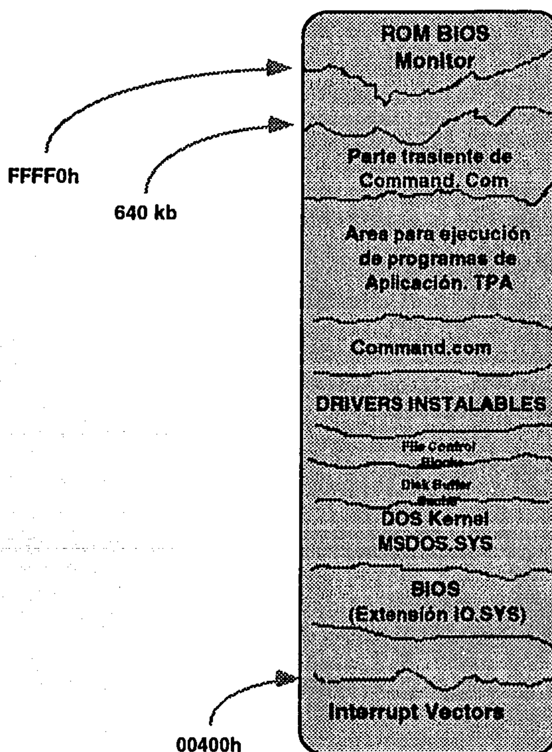
Una vez que el kernel ha sido configurado y los drivers residentes están disponibles, SYSINIT invoca los servicios de DOS para control de archivos y accesa el archivo CONFIG.SYS. Este archivo, que es opcional, puede contener una gran variedad de comandos que permiten al usuario adecuar a sus necesidades el entorno del sistema. En él, se definen los device drivers para el control y acceso a dispositivos periféricos, el número de buffers deseados para operaciones con el disco, el número de archivos que pueden ser abiertos simultáneamente y el procesador de comandos que va a ser utilizado.



El programa SYSINIT cuando encuentra el archivo CONFIG.SYS, lo sube en memoria y procede a la interpretación línea por línea de cada uno de los comandos. Conforme cada comando se ejecuta, algunas áreas de memoria van siendo reservadas para los buffers encargados del acceso a disco y para el almacenamiento de estructuras, como los *file control blocks* que son utilizadas por el manejador del sistema de archivos. Cuando SYSINIT se encuentra con la definición de un device driver, intenta alojarlo en memoria y genera las instrucciones necesarias para que la rutina de configuración del driver sea ejecutada y para incluir el nombre del driver dentro de una cadena especial de acceso.

Hasta este momento, los únicos drivers en operación son los definidos dentro del archivo IO.SYS. Cada uno puede ser accesado por cualquier programa a través de un *file handle*. Los drivers definidos en el archivo config.sys aunque ya se encuentren configurados y alojados en memoria, aún no están en este momento disponibles para ninguna aplicación pues no tienen un file handle asociado que permita su acceso.

Una vez que los device drivers definidos en el archivo config.sys han sido instalados, SYSINIT deshecha todos los file handles que había utilizado para el acceso a los drivers definidos en el archivo IO.SYS. A continuación, SYSINIT comienza a inspeccionar la cadena de drivers asignando conforme la recorre, un file handle específico a cada driver. Los device drivers definidos en el archivo CONFIG.SYS se insertan siempre al inicio de la cadena, lo que hace que SYSINIT los encuentre primero que los definidos dentro del archivo IO.SYS. Al finalizar esta inspección, todos los drivers en la cadena tendrán asociado un file handle que permitirá su acceso desde cualquier programa de aplicación.



Así queda organizada la memoria al final del proceso de arranque del sistema. Al momento de cargar a memoria el procesador de comandos, Sysinit sale de la escena, y la máquina queda lista para ser utilizada por el usuario.

Finalmente, SYSINIT ejecutará la función EXEC de MSDOS para subir a memoria el interpretador de comandos o shell, que al momento de instalarse desplegará el prompt del sistema, en espera del primer comando del usuario. En este momento SYSINIT sale de la escena.

El Kernel.

El kernel de MSDOS es un shell que maneja la comunicación entre el BIOS y el programa de aplicación. El kernel proporciona a la aplicación una interface lógica, aislándola de las peculiaridades físicas de la máquina. El kernel está contenido en el archivo MSDOS.SYS y brinda los siguientes servicios:

a).- **Control de Procesos**
b).- **Administración de Memoria.**
c).- **API (Interface para Programas de Aplicación)**
d).- **Sistema de Control de Archivos.**

a).- Control de Procesos.

El control de procesos tiene una relación intrínseca con la ejecución de programas, ya que realiza la carga del programa en memoria, prepara su entorno de ejecución y pasa el control del CPU al mismo. Cuando el programa termina, el control de procesos recupera de nueva cuenta el control de la máquina. En un sistema operativo nivel 2, el control de procesos involucra también la intercomunicación entre tareas, MSDOS aunque no es propiamente un sistema multitareas, soporta cierto nivel de intercomunicación entre las mismas a través del uso de pipes.

b).- Administración de Memoria.

La administración de memoria que se realiza través de MSDOS, se considera por su simplicidad como nivel 1, la memoria se maneja en todo momento bajo el esquema de memoria real, por lo que a toda localidad que se hace referencia dentro del programa debe corresponder una localidad física. MSDOS divide la memoria para su control en bloques de 16 bytes. Microsoft llama a cada uno de esos bloques *Arena Entry.* Todo programa que se ejecuta bajo MSDOS es alojado en un área de memoria llamada Trasient Program Area (TPA). Para la mayoría de los programas el manejo de esta área no implica más que la elección del modelo apropiado para la aplicación. En ocasiones sin embargo, el programador debe obtener o liberar espacio en memoria, cuando esto sucede MSDOS debe verificar en su lista de Area Headers, si existen Arena Entries disponibles o no para otorgarlos a la aplicación.

c).- Interface para Programas de Aplicación.

El kernel proporciona una interface (API) específica para que el programador pueda comunicarse con el hardware de la máquina. Esta interface comúnmente llamada DOS, es independiente del hardware para todos los programas que corren bajo MSDOS. El API para usuarios de MSDOS es una estructura de tres niveles, que comprende el nivel físico el nivel de BIOS y por último el de MSDOS. Para escribir programas es posible hacer uso de cualquiera de los tres niveles, pero cada elección conlleva pérdidas en portabilidad contra ganacias en la rapidez de ejecución o viceversa.

En el nivel de interface físico o directo, el programador maneja puertos y localidades físicas de memoria. A este nivel, la programación no soporta operaciones de redireccionamiento, multitareas o utilización de redes. Este nivel se utiliza principalmente en aplicaciones especiales donde la velocidad de acceso a los dispositivos es crítica. Por ejemplo en los sistemas de manejo de gráficos.

En el siguiente nivel se encuentra la interface con el BIOS (ver esquema pág. 2). Al igual que el nivel físico, este tipo de interface no soporta operaciones de multitareas o redes. Algunas de sus características hacen aparecer al BIOS como una interface lógica, pero no por ello pierde su dependencia del hardware. Las ventajas que ofrece el uso del mismo en cuanto a velocidad tienen el costo de pérdida en la portabilidad de la aplicación.

En el nivel superior se encuentra la interface de MSDOS, la cuál es el standard definido por Microsoft. Aunque es más lenta que los otros métodos de acceso, proporciona mayor compatibilidad y control sobre los recursos de la máquina. Este es el nivel de interface donde se debe trabajar a menos de que necesidades específicas hagan necesaria la utilización de niveles mas bajos para el acceso a dispositivos desde el programa de aplicación. Examinando los servicios disponibles de MSDOS, se pueden apreciar dos cosas:

Primero, que algunos de los servicios de MSDOS parecen duplicados a los servicios ofrecidos por el BIOS. La manera como estos son utilizados puede ser ligeramente diferente pero la función es idéntica. Ya mencionamos el costo que implica usar una u otra interface, aún en el caso de servicios que hagan funciones similares.

Segundo, la mayoría de los servicios de MSDOS tienen una estructura más compleja en su funcionamiento que los del BIOS, por lo que al invocarlos permiten hacer más con una sola instrucción, ahorrando tiempo de programación.

Los programas de aplicación, llaman tanto a los servicios de DOS como a los del BIOS a través de un conjunto de interrupciones, cuyos vectores asociados se localizan en la *tabla de interrupciones* definida en la parte baja de la memoria RAM.

d).- Sistema de Control de Archivos.

Esta es la más extensa sección del kernel y se encarga del manejo de todas las operaciones de I/O que se ejecutan sobre archivos. A menos que por alguna razón particular sea necesario el acceso físico a algún sector del disco, los programas de aplicación usan generalmente los servicios de DOS para realizar este tipo de operaciones. La utilización de otros métodos de acceso como el directo o físico y mediante los servicios del BIOS, impone limitaciones importantes a las aplicaciones que los usan, pues pierden transportabilidad para su ejecución bajo otros ambientes como en el caso de redes de computadoras, donde no pueden utilizarse.

Interface de Usuario.

Esta define la forma de comunicación entre el usuario y el kernel del sistema. La interface de usuario de MSDOS viene definida dentro del programa COMMAND.COM. El programa command.com puede ser substituído por otro interpretador de comandos a elección del usuario. La manera de indicarle a DOS el uso de otra interface de usuario es a través el comando Shell en el archivo CONFIG.SYS. El command.com consta de tres secciones:

La sección *residente*, presente siempre en memoria. En esta se encuentran las funciones básicas del sistema operativo, necesarias para realizar la ejecución de una aplicación.

La sección *trasiente*, que contiene los comandos intrínsecos del sistema, como dir, copy, erase, etc. Esta se carga en el área que se encuentra justamente abajo del límite de los 640 kb. Si la aplicación es muy grande, el programa puede ocupar esta área mientras efectúa su ejecución. Al terminar, la porción residente del command.com cargará nuevamente la parte trasiente del mismo.

La sección de inicialización, opera únicamente durante el arranque del sistema, contiene el código del archivo autoexec.bat, al terminar su ejecución las localidades de memoria ocupadas por el mismo son liberadas.

Cuando el usuario requiere la ejecución de un comando, el COMMAND.COM trata de encontrar un programa intrínseco con el nombre sumistrado. Si la búsqueda en el directorio de trabajo falla, usa el *path* para localizar en los subdirectorios ahí definidos el comando en cuestión. Cuando el COMMAND.COM encuentra un archivo con extensión .COM ó .EXE. invoca la función EXEC del kernel, la cuál carga el programa en memoria y le transfiere el control. Cuando el programa finaliza, el control de la máquina regresa al sistema operativo.

Utilerías.

Los programas de utilerías componen el conjunto de comandos extrínsecos de DOS, tales como el Format y el Diskcopy.

Interrupciones.

Las interrupciones juegan un papel muy importante en la programación de MSDOS, éstas son en esencia el mecanismo de control más poderoso del mismo. Se dice que la computadora es interrupt driven, pues todos los requerimientos que llegan a ella, son en forma de una interrupción. (Norton, 1985).

Cuando una interrupción ocurre, el control de la computadora es pasado a una rutina de servicio particular llamada *Interrupt Handler.* La manera de requerir la ejecución de una rutina de este tipo, es mediante la definición de la localidad de memoria donde inicia su código en los registros que controlan el flujo del programa.

El mecanismo que se utiliza bajo DOS para el acceso a cualquier localidad de memoria conlleva la definición de un segmento y un offset. Un *segmento* consiste de un conjunto de localidades contiguas de memoria, su tamaño máximo es de 64 Kb. La dirección de inicio de todo segmento se define a partir de localidades que son múltiplos de 16 (paragraphs boundary). Cuando se requiere accesar alguna rutina, se hace un primer acercamiento a través de la definición del segmento, pero muchas veces esta rutina cae en una localidad que no es múltiplo de 16, la manera de accesarla es sumando un offset a la definición de segmento. El *offset* contendrá el número de bytes adicionales que habrá que saltar a partir de la localidad definida por el segmento para quedar posicionados en el inicio exacto de la rutina.

El interrupt handler es llamado entonces, mediante la carga de su dirección de segmento en el Code Segment register y el offset necesario en el Instruction Pointer register, ambos registros son comúnmente referidos como CS:IP. Las direcciones de segmento y offset que definen las localidades de memoria donde se alojan los interrupts handlers son llamadas *vectores de interrupción.*

Los vectores de interrupción, se definen durante la primera etapa de inicialización del sistema, al ejecutarse el loader del ROM-BIOS. En esa etapa los vectores se quedan apuntando hacia los interrupts handlers del propio ROM-BIOS. Posteriormente los valores de algunos de los vectores de la tabla, son substituidos en el proceso de inicialización por los drivers instalables o por los programas residentes. Los vectores de interrupción, se definen en una tabla de 256 elementos que se localiza dentro de un área reservada en memoria RAM, que comprende las localidades 0000 - 03FFh, cada vector ocupa 4

bytes, dos bytes se destinan a la definición del offset y dos bytes se destinan a la definición de segmento. Los vectores de interrupciones pueden cambiarse de forma que apunten hacia otro interrupt handler simplemente localizando su posición en la tabla y cambiando su valor.

Clasificación de las Interrupciones.

Interrupciones en el Microprocesador.

Existen cuatro interrupciones (0,1,3,4) que son generadas dentro del microprocesador, y una adicional (int 2), llamada no mascarable que es activada al ocurrir una situación de excepción (un error) en el funcionamiento de alguno de los dispositivos externos conectados al CPU.

Interrupciones Generadas por Dispositivos Periféricos.

Hasta 15 diferentes líneas de interrupción pueden ser manejadas en el caso de una computadora AT (una XT maneja sólo 8 como máximo). La manera de lograr esto, es mediante el uso de dos chips (82C59A Programable Interrupt Controller) que trabajan en coordinación con el microprocesador a través de la línea de interrupciones mascarable.

El micro, sólo tiene disponible un pin para recibir interrupciones mascarables, a ese pin INTR está conectado uno de los dos PICs, que se denomina PIC ***maestro.*** Una de las 8 líneas de entrada del PIC maestro, está destinada a recibir la señal de interrupción generada por el segundo PIC, el PIC ***secundario.*** Así, los dos PICs están interconectados en cascada, lo cuál en términos generales significa que ambos se coordinan para la recepción de interrupciones. El PIC maestro puede manejar sólo 7 dispositivos por estar una de sus líneas de entrada destinada al manejo del PIC secundario, el PIC secundario puede manejar hasta 8 líneas, lo que dan las 15 interrupciones arriba mencionadas.

Interrupciones Generadas por Software.

En esta clase de interrupciones, se incorporan aquellas generadas por programas de aplicación, destinadas al requerimiento de servicios de DOS y del ROM-BIOS.

Interrupciones de DOS.

Las interrupciones que caen dentro de esta clasificación, son las que generan los programas de aplicación, con la finalidad de requerir la ejecución de alguno de los servicios ofrecidos por DOS. Una de las interrupciones más importantes de DOS es el int 33 (hex 21h). Esta interrupción, proporciona al programador acceso a un amplio número de rutinas, llamadas funciones o servicios de DOS. Estas funciones ejercen un control más sofisticado sobre las operaciones de I/O que las rutinas del BIOS, especialmente en lo que se refiere a operaciones que involucran el manejo de archivos y de dispositivos a través de device drivers. Todos los procesos standards del disco como inicialización, lectura y escritura de datos, operaciones de borrado de archivos y búsqueda en directorios, se incluyen dentro de las funciones de DOS, éstas proporcionan la base para muchos otros programas de alto nivel como Format, Copy y Dir. Los programas de aplicación, a través de rutinas especiales llamadas *Interfaces*, pueden hacer uso de servicios de DOS no contemplados en el lenguaje de programación donde se realiza el desarrollo. Una interface, es una rutina escrita en un lenguaje de programación (Vgr. ensamblador) distinto al que sirve de base para el desarrollo de una aplicación. La mayor parte de los lenguajes de programación de alto nivel, por más limitados que sean, hacen previsiones para poder incluir *interfaces* en los mismos y dar la posibilidad al programador de crear funciones que permitan ejecutar operaciones de I/O que se adapten a sus necesidades.

La tabla que contiene a los vectores de interrupción se encuentra localizada en la parte más baja de memoria RAM. La primera localidad de memoria contiene el vector de interrupciones número 0 y así sucesivamente. Ya que cada vector ocupa cuatro bytes para su definición, la manera de encontrar la localidad de memoria donde se encuentra un vector, es multiplicando el número de interrupción asociada al vector por 4. Por ejemplo, el vector asociado a la interrupción número 5, que corresponde al servicio de PrintScreen, se encuentra en la localidad 20 de memoria.

Desde el punto de vista de programación, el interés que existe en los vectores de interrupción nace de la posibilidad de cambiar su contenido, de manera que estos apunten a un nuevo *interrupt handler*, definido por el usuario. Para lograr esto, el programador debe escribir una rutina que realice una función diferente a la standard del ROM-BIOS o de DOS, dejarla residente en memoria RAM y almacenar la dirección de la misma en la tabla de interrupciones.

Device Drivers

Device Drivers.

Uno de los objetivos perseguidos durante el desarrollo del tema sobre sistemas operativos, era dar realce a la importancia que tiene la generación de programas de aplicación independientes del hardware de la máquina. La utilización en los programas de aplicación de una interface a nivel lógico que permita el acceso transparente a dispositivos periféricos, asegura por un lado el rápido desarrollo de la aplicación y por el otro la transportabilidad de la misma bajo diferentes ambientes de ejecución. Esta clase de ideas dieron la pauta para la creación de sistemas operativos que ofrecieran interfaces para el desarrollo de aplicaciones cada vez más completas. Los device drivers desempeñan un papel de primera importancia en torno a este tema, pues a través de ellos el sistema operativo presenta una interface lógica para el acceso a dispositivos muy fácil de implementar dentro de un programa de aplicación.

Los device drivers, son módulos del sistema operativo encargados del control del hardware de la máquina. Estos, permiten aislar al kernel de las características específicas que presenten los dispositivos periféricos que hacen interface con el procesador central. La relación que existe entre el kernel y los drivers, es análoga a la que existe entre el sistema operativo y los programas de aplicación. Los programas de aplicación se comunican con el sistema operativo a través de una serie de reglas, que permiten la interacción ordenada entre uno y otro para el intercambio de información y la ejecución de procesos. Para todos es claro, que si un programa cumple con los lineamientos impuestos por el sistema operativo, el programa, cualesquiera que sea su finalidad se ejecutará en ese entorno sin mayores problemas. El sistema operativo permanece inalterable, los programas que corren en él, son tan disímbolos como la imaginación misma del hombre lo permite. Esto exactamente pasa con los device drivers, el sistema operativo siempre es el mismo, pero los dispositivos periféricos que trabajen bajo su entorno podrán ser tan variados como las necesidades y el desarrollo tecnológico lo permitan, lo requerido para que todo funcione, es que el programa de control del dispositivo esté diseñado para comportarse conforme las especificaciones impuestas por el sistema operativo.

Controladores, Adaptadores e Interfaces.

Los dispositivos periféricos, a nivel de hardware necesitan de una vía de acceso que les permita integrarse a la operación de una computadora. Afortunadamente, desde la aparición de las primeras computadoras personales, se introdujeron en el diseño de las mismas mecanismos que permitieron incorporar periféricos adicionales, que servían a propósitos específicos del usuario final. Los elementos proveídos para el acoplamiento de periféricos dentro de la arquitectura de la máquina, se denominan slots o ranuras de expansión. Los slots o ranuras de expansión cumplen dos funciones primordiales:

a).- Brindan el mecanismo de sujección en la computadora a la electrónica del dispositivo.

b).- Ponen a disposición de los dispositivos una serie de contactos eléctricos que dan acceso a lo que podríamos llamar el sistema nervioso del procesador central, el BUS de la máquina

¿Qué es el bus de una computadora?. Podemos definirlo como el conjunto de líneas diseñadas para llevar a cabo el intercambio de información entre el procesador central y el exterior. Dependiendo del contenido o finalidad de la información que circula a través de estas líneas, el bus será identificado como de datos, de control o de direcciones. *Las señales que circulan dentro del Bus reciben el nombre de señales de I/O. Las tarjetas que se insertan en los slots son conocidas con el nombre de controladores, adaptadores o interfaces.* La función genérica de estas tarjetas consiste en proporcionar una interface entre el hardware del dispositivo y la PC. Esta característica de diseño, que permite la inserción de tarjetas controladoras de dispositivos en los slots de la computadora para convertirlas en parte integral de la misma, recibe el nombre de *Arquitectura Abierta.*

El procesador central dispone de un banco de direcciones exclusivo para el acceso a dispositivos conectados a través de las ranuras de expansión, cada dirección asociada a este banco recibe el nombre de ***puerto de I/O.*** En general los dispositivos tienden a utilizar varios puertos para su operación. Por ejemplo, una impresora tiene asignado un puerto para la transferencia de datos a impresión, un ***puerto de status*** y otro de ***control.*** Cuando una PC transfiere datos hacia un dispositivo, la instrucción OUT es utilizada para seleccionar el puerto de I/O y el caracter a ser enviado. Cuando una PC ejecuta la

instrucción OUT, la dirección del puerto es colocada en el bus junto con el valor del dato. Por su parte el controlador del dispositivo está constantemente monitoreando las señales que circulan por el bus de direcciones. Cuando encuentra el valor de un puerto asignado a él, efectúa el acceso al bus de datos y toma el valor que en el bus exista. Los adaptadores realizan las funciones básicas de control de dispositivos y transferencia de datos entre la PC y los periféricos mediante el reconocimiento de las señales enviadas por el microprocesador a través del bus de la máquina.

Dispositivos tipo Caracter y de Bloque.

En el ambiente de las PC's, los dispositivos periféricos se dividen en dos clases: dispositivos tipo *caracter* y dispositivos de *bloque*, la distinción entre uno y otro está basada en la forma como se realiza la transferencia de datos entre la computadora y el periférico conectado a ella.

Los dispositivos tipo caracter, manejan la transferencia de datos sobre la base de un caracter por cada acceso. Por ejemplo modems y teclados.

Los dispositivos de bloque, manejan la transferencia de datos en grupos de caracteres por cada acceso. Ejemplos de estos dispositivos los podemos encontrar en los discos y lectoras de cintas. Los dispositivos de bloques generalmente son utilizados cuando se necesita lograr altas velocidades de transferencia. Si los discos por ejemplo, fueran definidos como dispositivos tipo caracter, la velocidad de transferencia estaría severamente limitada, en primer lugar, porque los tiempos de posicionamiento de la cabeza lectora, siempre son bastante considerables como para que se efectúe la transferencia de un sólo caracter por acceso.

Hay disponibles todo tipo de interfaces para control de dispositivos en las PC's. La variedad abarca desde aquellos encontrados en las tarjetas de funciones múltiples, como las de interface serie o paralelo, hasta las tarjetas de propósito específico, como las controladoras de lectores para discos ópticos. En cualquier caso el programador, para poder realizar la escritura de los device drivers debe tener un conocimiento profundo de la forma de operación de los dispositivos que se desean incorporar a la PC.

Device Drivers Standards de DOS.

DOS una vez instalado, permite a los programas de aplicación el control de un conjunto de dispositivos periféricos que forman parte de la configuración mínima necesaria para la operación de una PC: teclado, pantalla, discos, interfaces serial y paralelo. Cada uno de estos dispositivos tiene un nombre único asignado por DOS, con el cuál los programas de aplicación y los comandos de DOS realizan el acceso al dispositivo. (Lai, 1987).

CON:

El device driver CON: se encarga del control de dos de los elementos mas importantes para la comunicación entre el CPU y el usuario: el teclado y la pantalla. Una vez encendida la máquina, para la generalidad de los usuarios es perfectamente natural que toda operación que efectúen sobre el teclado, se vea reflejada de inmediato en la pantalla. Pero el proceso no es tan fácil de llevar a cabo como se ha logrado hacer que parezca. La realidad es que cada ocasión que una tecla se oprime, se genera una interrupción, la cuál es atendida a través de una rutina de servicio llamada interrupt handler. Es a partir de esta rutina como se efectúan los pasos necesarios para determinar qué tecla fué oprimida, guardándose el código correspondiente en un buffer. La rutina del ROM-BIOS que captura los códigos de las teclas oprimidas tiene asignado el interrupt *9h*. Cada tecla que se oprime está compuesta de dos bytes, uno contiene el código ASCII de la tecla y otro contiene el código de *Scan*, este último se utiliza para identificar cuando teclas especiales como F1, del, ret, son oprimidas. La longitud del buffer de almacenamiento es de 32 bytes generalmente, por lo que es posible guardar en él hasta 16 operaciones sobre el teclado. Este buffer, permite que las teclas oprimidas sean capturadas aún cuando algún programa está ocupado en ese momento procesando información no relacionada con el teclado, esta función es conocida con el nombre de *type ahead*. Para lograr el acceso al buffer de almacenamiento con un programa pueden hacerse dos cosas, una es invocar una rutina del BIOS a través de la interrupción *16h* y la otra es simplemente efectuar una lectura sobre el archivo CON:.

El despliegue de información en pantalla puede ser logrado a través del uso de la interrupción *10h* del BIOS.

Todas estas operaciones, que se realizan para el control del teclado y la pantalla, están construídas dentro del device driver CON:, el usuario nunca se entera que está generando una interrupción, o

nunca necesita de escribir un programa para desplegar los caracteres en pantalla, el driver operando silenciosamente, tras bambalinas, robándose tiempo del procesador, se encargará de efectuar las operaciones pertinentes. Esta es la finalidad del device driver: ***hacer que las cosas parezcan sencillas tanto para el usuario como para el programador de aplicaciones.***

AUX:

El device driver AUX: se encarga del manejo de la interface serial RS-232C de la computadora. Este driver permite el control simultáneo hasta de dos interfaces seriales conocidas bajo el nombre lógico de COM1 y COM2. Este es un ejemplo de como un sólo device driver logra el control de más de un dispositivo.

PRN:

A través del device driver PRN: los sistemas bajo DOS tienen acceso hasta a tres interfaces paralelas destinadas al control de impresoras. Los nombres lógicos asignados a esas interfaces son LPT1, LPT2 y LPT3.

NUL:

Este es un device driver muy peculiar de DOS. Cuando se escribe algo en él no sucede absolutamente nada, los datos son tirados a una especie de basurero sin fondo. Este dispositivo es usado a menudo en aquellos casos donde las aplicaciones despliegan una gran cantidad de datos en pantalla y el usuario en turno no tiene el menor interés en los mismos. Cuando se desea evitar el despliegue y hacer que el programa se ejecute un poco mas rápido, puede ser redireccionada la salida del programa temporalmente hacia NUL.

CLOCK$:

Este es otro device driver especial de DOS. A través de él se construye el mecanismo para llevar la hora del sistema. La forma de hacerlo es la siguiente: existe en la computadora un timer conocido como 8253-5, el cuál se encarga de generar interrupciones a un ritmo de 18.2 veces por segundo ó 18.2 ticks. El vector de interrupción asociado se identifica bajo el número 08h. Cada vez que se genera una interrupción, entra en acción el interrupt handler correspondiente, el cuál incrementa un contador. El valor del contador puede oscilar en el rango que va desde 0, correspondiente a la media noche del día que corre, hasta el 1573040, que correspondería al último segundo del día. Así, cuando el usuario desea conocer la hora del sistema, el device

driver entra en operación y calcula a partir del valor del contador, la hora, el minuto y los segundos del sistema, entregándole al usuario la información en un formato inteligible. Si el usuario por el contrario desea accesar el valor en ticks del sistema y tomarse la molestia de hacer sus propios cálculos, puede usar la interrupción 1AH, que permite el acceso al área de almacenamiento del contador.

Device Drivers para el acceso a Discos.

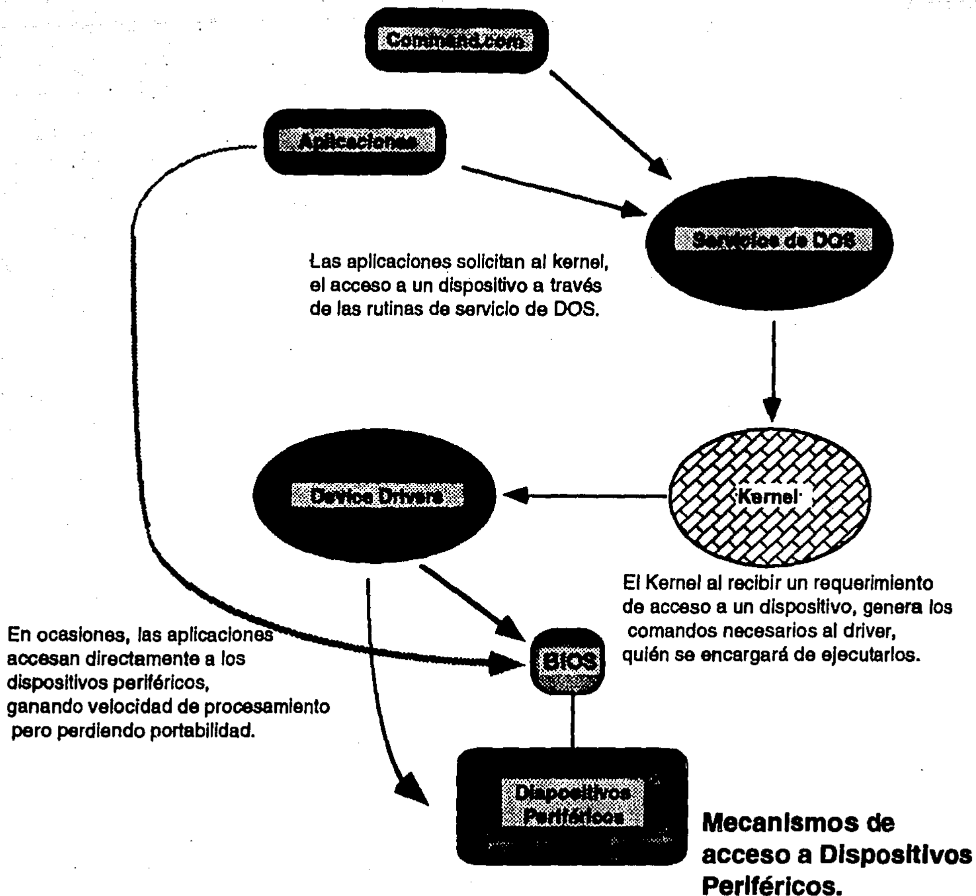
Los discos standard encontrados en las PC hoy en día pueden ser discos blandos o duros. En este caso DOS no reserva nombres específicos para su acceso como pasa con los dispositivos descritos anteriormente, asigna las letras del alfabeto como medio de identificación de cada drive. Para el caso de dispositivos de almacenamiento no standards tales como discos ópticos, RAM disks y otros, los device drivers encargados de su control son consistentes con la nomenclatura que impone DOS a pesar de utilizar técnicas distintas para el almacenamiento y recuperación de la información.

Control de Dispositivos a Través de los Servicios de DOS.

Los servicios de DOS consisten de una serie de funciones disponibles bajo la interrupción 21h. Estas funciones están destinadas al procesamiento de operaciones de entrada/salida que involucran el acceso a archivos y a dispositivos periféricos.

Para accesar un dispositivo usando DOS, los programas necesitan indicar primeramente el nombre lógico asociado al mismo. DOS requiere que el nombre del device driver que controla al dispositivo sea especificado al momento de invocar el servicio Open (0Fh). Cuando este servicio es requerido, DOS responde a la aplicación con un número que queda asociado permanentemente al driver: el **File Handle.** Este número deberá ser utilizado por la aplicación en todas las operaciones subsecuentes que efectúe sobre el driver.

Cuando se solicita un servicio de DOS, el kernel lo traduce en una serie de comandos para el driver. La traducción de los mismos, se realiza de acuerdo a un conjunto de reglas uniformes para todos los dispositivos. El resultado es un conjunto de comandos que el kernel de DOS le pedirá al device driver que se ejecuten. Cada comando, es ejecutado por una o varias rutinas específicas residentes en memoria. Este conjunto de rutinas constituyen el device driver del dispositivo.



Mecanismos de acceso a Dispositivos Periféricos.

Traducción de Servicios de DOS a comandos para el Device Driver.

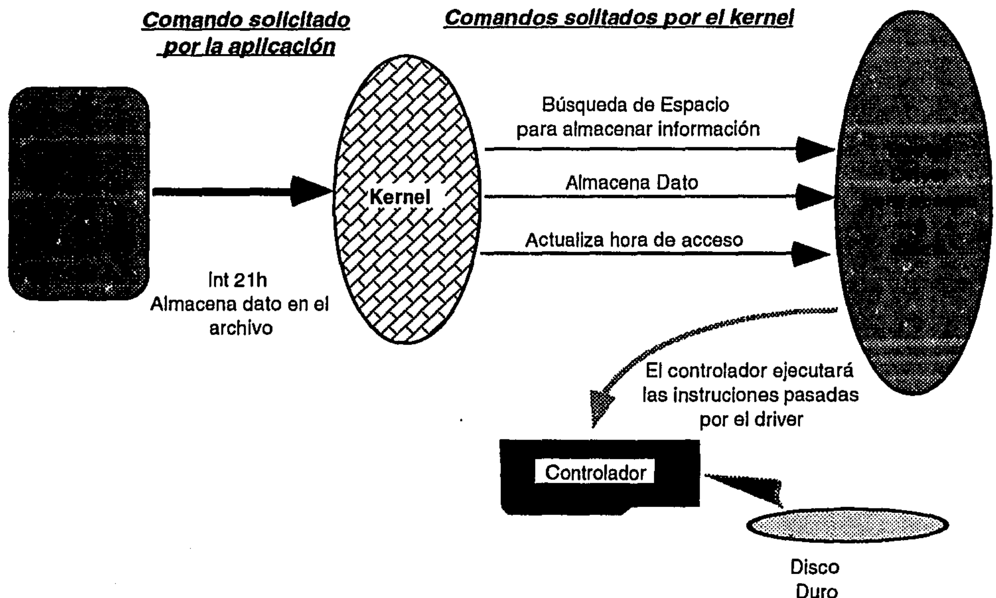
Los device drivers están diseñados para manejar comandos simples provenientes del *kernel de DOS*. Los *servicios de DOS* más comúnmente usados por las aplicaciones para el acceso a dispositivos son el Read y el Write. Estos servicios de DOS son relativamente complejos y pueden no ser traducibles por el kernel en un sólo comando para el device driver. Con un sólo requerimiento de lectura o escritura generado por la aplicación, el kernel puede necesitar generar varios comandos al device driver apropiado para lograr satisfacer el servicio de DOS.

Analicemos el caso de un programa con el que se escriben datos en el disco duro, para lo cuál se requiere la intervención del servicio Write de DOS. Al recibir la petición de escritura, el kernel la divide en varios comandos para el driver:

a).- **El primer comando, se genera para que el driver encuentre un lugar dentro del disco donde exista espacio disponible para el nuevo dato, para lo cual el driver tendrá que leer el File Allocation Table.**

b).- **Si hay espacio disponible, DOS requerirá la escritura del nuevo dato en el archivo, a través del comando Write dirigido al driver. El comando write que DOS genera al driver, es de naturaleza distinta al comando Write que la aplicación genera a DOS.**

c).- **Finalmente el kernel actualizará la información que indica la hora del último acceso al archivo, generando para ello otro comando al driver para que este efectúe esa tarea.**



El presente diagrama nos muestra como el kernel traduce una solicitud de escritura a disco generada por una aplicación en tres diferentes comandos que el driver debe ejecutar para satisfacer así el requerimiento solicitado por la aplicación.

Reemplazo de los Device Drivers de DOS.

Para que un device driver funcione correctamente dentro de DOS, debe ser escrito de acuerdo a las especificaciones de diseño de Microsoft. Las reglas que Microsoft especifica para el diseño de drivers, permiten que todos los drivers presenten una interface uniforme al kernel de DOS.

Si los device drivers no tuvieran una interface uniforme y bien definida con el kernel, la integración de nuevos dispositivos a la computadora sería complicada. El constructor tendría que suministrar una versión modificada de DOS para que el dispositivo pudiese ser utilizado. Esto crearía un sinnúmero de problemas, comenzando por el hecho que las nuevas versiones de DOS liberadas, no podrían ser utilizadas hasta que el constructor del periférico las modificara para seguir haciendo uso del dispositivo. Ya que cada constructor utilizaría un mecanismo diferente para integrarse a DOS, resultaría que versiones de DOS con el mismo número de liberación, pero provenientes de diferentes fabricantes de dispositivos, serían incompatibles.

Algunas veces, el usuario pudiera toparse con que el device driver que suministra DOS como el standard, no satisface sus necesidades específicas. Previendo esta situación, DOS cuenta con un mecanismo para el reemplazo de los device drivers standards por drivers instalables del usuario. Un device driver de uso muy difundido que se utiliza en situaciones de esta clase es el ANSI.SYS. Cuando este driver es definido en el archivo CONFIG.SYS, el kernel lo instala reemplazando al driver standard de DOS: CON:.

¿Qué hace ANSI.SYS?

Hace algunos años la American National Standards Institute (ANSI), diseñó una serie de secuencias de escape, que podrían ser usadas para ejecutar funciones específicas independientemente del tipo de video o teclado utilizado. Una secuencia de escape consiste en un grupo de caracteres precedidos por el caracter ASCII 10h. Tales secuencias incluyeron el manejo de color en los caracteres y en el fondo de la pantalla, uso de video inverso y asignación de códigos especiales al teclado. Estas secuencias de escape han otorgado una gran portabilidad a muchas aplicaciones, porque permiten a los desarrolladores de software la creación de programas que requieren un control complejo sobre el teclado y el monitor sin que por ello deban hacerlos dependientes de un tipo de máquina en especial.

A partir de la explicación anterior, podemos entender que los drivers no sólo son construídos con objeto de integrar a una computadora nuevos dispositivos. Los device drivers pueden ser desarrollados con el objeto de mejorar o adecuar la operación de los drivers standards del sistema operativo.

Formación de la cadena de Drivers.

El kernel, durante la etapa de arranque del sistema, va formando una lista ligada con los nombres de los device drivers que va instalando. Esta lista incluye tanto a los device drivers residentes o standards de DOS como a los device drivers instalables del usuario.

Antes de usar un driver, se requiere que la aplicación solicite la ejecución del servicio de DOS Open. Este servicio especifica la inclusión de una cadena de caracteres tipo ASCII que contiene el nombre del archivo o dispositivo a accesar. El kernel, al recibir el requerimiento, compara el nombre pasado por la aplicación con la lista de drivers instalados. Si el nombre es encontrado, llama a las rutinas pertenecientes al comando Open del driver. Si la operación que se realiza a través de las rutinas del comando Open no genera errores, el kernel regresará al programa de aplicación el número de un *file handle*, el cuál quedará asociado al device driver mientras la aplicación no genere el comando de DOS Close. Todas las operaciones sucesivas que tengan lugar entre el device driver y la aplicación, utilizarán el número de *file handle* obtenido durante la operación de DOS Open, nunca más será necesario hacer referencia al nombre lógico del device driver para accesarlo.

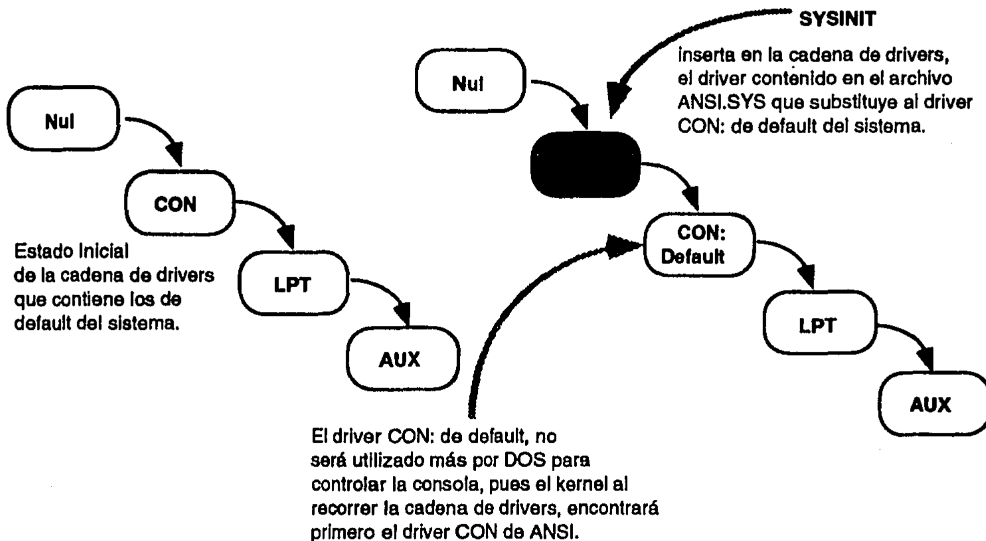
Estructura de la Cadena de Drivers de DOS.

A la cabeza de la lista ligada de DOS, se encuentra el device driver NUL. Este contiene dentro de su estructura un apuntador hacia la localidad de memoria donde se encuentra el siguiente driver. Cada driver a su vez, apuntará a la dirección donde se localiza el driver subsecuente. El apuntador del último driver contendrá el valor de -1, señalando el fin de la lista.

Para el reemplazo de los device drivers residentes por drivers instalables, DOS se vale de un mecanismo simple pero muy efectivo, que será descrito a continuación:

La lista ligada de drivers que DOS forma, inicialmente comprende en exclusiva a los drivers residentes o standards del sistema, cuyo orden de aparición es: NUL:, CON:, LPT:, AUX:, etc.

En el momento de leer el archivo CONFIG.SYS, SYSINIT verifica si existe algún driver ahí definido para su instalación. En caso afirmativo, DOS inserta en su cadena de drivers el device driver instalable, justamente después de NUL:. Para el caso del driver ANSI.SYS por ejemplo, la lista quedaría: NUL:, CON: (del archivo ANSI.SYS), CON:, LPT:, AUX:, etc.



En esta figura se observa el proceso de inserción del driver instalable ANSI.SYS dentro de la cadena de drivers del sistema operativo. Puesto que la cadena de drivers es analizada siempre a partir de NUL:, los requerimientos de acceso que realice el kernel a la consola siempre serán atendidos por el driver definido en el archivo ANSI.SYS.

Cada vez que se genera un requerimiento de acceso a un driver, DOS verifica su existencia en la lista, realizando para ello una búsqueda ordenada del nombre proveído por la función DOS Open en la cadena de drivers que inicia a partir de NUL:. En caso de existir en la cadena dos drivers con el mismo nombre, el kernel seleccionará para realizar el trabajo a uno de ellos únicamente; el que esté más cerca en la cadena de NUL:. Ya que los drivers instalables se insertan siempre después de NUL:, es fácil de entender ahora, como es que pueden reemplazar a los drivers residentes con el mismo nombre.

En la figura, se puede observar que el archivo que contiene el driver que maneja el monitor y teclado se llama *ANSYS.SYS*, pero el nombre del device driver debe ser *CON:* pues ese es el nombre genérico para el acceso a la consola. Si el driver definido en el archivo *ANSI.SYS* debe modificar al driver standard de DOS que se llama CON:, el nombre del driver dentro del archivo *ANSI.SYS* debe ser *CON:*, para poderlo reemplazar.

La lista device drivers que construye DOS, es conocida con el nombre de *cadena de drivers*. La cadena de drivers es una lista ligada, donde cada elemento de la misma, contiene la dirección donde se aloja el driver siguiente. Esto es: cada uno de los elementos o drivers de la lista, pueden ser accesados únicamente a través de los elementos o drivers anteriores. Este mecanismo evita que los drivers necesiten estar almacenados en localidades contiguas de memoria y forza al kernel a inspeccionar la lista secuencialmente. Cuando DOS inserta un nuevo driver en la cadena, lo que realmente sucede es que el apuntador de NUL:, que hasta ese momento tenía la dirección donde se encuentra el driver CON:, es cambiado a la dirección donde se encuentra el nuevo driver que se inserta, este a su vez, dirigirá su apuntador hacia la localidad de memoria donde se aloja CON:. Para accesar los drivers, todo lo que necesita DOS es un apuntador hacia su primer elemento NUL:, a partir de él, el kernel podrá encontrar el resto de los device drivers, navegando en la lista con ayuda de los apuntadores.

Hasta ahora se ha mencionado la existencia de drivers cuya finalidad es o mejorar el funcionamiento de los drivers residentes de DOS o proporcionar el mecanismo para la integración de nuevos dispositivos al ambiente de DOS. Existen además otra clase de drivers, que simplemente no controlan dispositivo alguno, su finalidad es simular el funcionamiento de un dispositivo. Un importante exponente de drivers de esta clase lo encontramos en el RAM DISK. Las aplicaciones que frecuentemente realizan accesos a disco (disk bound), se ven beneficiadas con el uso de este driver, por el hecho que las velocidades de acceso que se manejan para memoria RAM, son notablemente superiores respecto a las velocidades logradas en el caso de los discos. El resultado es que la aplicación se ejecuta con mayor rapidez.

Estructura de un Device Driver.

Dentro de la estructura de un device driver se pueden identificar varias secciones, cada una con funciones específicas de operación. En los siguientes párrafos se hará una descripción detallada de las mismas.

- **Area de Directivas para el Ensamblador.**
- **Device Header.**
- **Area para Almacenamiento de Variables Locales.**
- **Sección de Procedimientos Locales.**
- **Sección de Estrategia.**
- **Sección de Interrupción.**

Area de Directivas para el Ensamblador.

Las directivas, son instrucciones especiales para el ensamblador, que no causan la generación de código ejecutable. Tales instrucciones, indican la manera particular como las instrucciones del programa fuente deben ser traducidas a código objeto. A continuación, serán comentadas algunas de las más utilizadas en la programación del driver SAD.

Constantes.

Los valores de uso frecuente dentro de un programa, es posible asociarlos con identificadores, que por no modificar su valor dentro del mismo reciben el nombre de constantes. El manejo de constantes en lugar de números es una facilidad muy apreciada por el programador, la forma de definirlos se ejemplifica así:

```
max_buf    =    900        ;identificador = valor
```

La directiva ***.286P*** le indica al ensamblador el tipo de máquina para la que debe generar el código ejecutable. Conforme el procesador utilizado aumenta su capacidad de procesamiento, el set de instrucciones disponibles se enriquece para aprovechar las nuevas capacidades introducidas. Esta directiva indica, que se desea utilizar el set de instrucciones para un procesador 80286.

Definición de un Segmento.

Un segmento, define un área contigua de memoria cuya extensión máxima es de 64 kb. El ensamblador calculará la extensión total de un segmento al momento de traducir el programa fuente a código objeto. La manera de indicar al ensamblador que un conjunto de datos o de

instrucciones forma parte de un segmento se ejemplifica mediante el siguiente listado:

```
.286P
  abs0    segment at 0
          org 0dh*4
          adq_int label word
  abs0    ends
```

En el listado anterior se define que el segmento *abs0* inicia en la localidad 0 de memoria. La terminación de un segmento se indica mediante la declaración del nombre asociado al mismo, seguido de la etiqueta *ends.*

```
cseg      segment para public 'code'
adq       proc far
assume    cs:cseg,es:cseg,ds:cseg
.
.
Begin:
.
.
cseg      ends
end       Begin
```

El código anterior corresponde a la definición de un segmento llamado **cseg.** La directiva *segment* define el inicio del segmento y la directiva *ends* el final del mismo. La directiva ***PARA*** señala al ensamblador, que debe realizar la alineación en memoria del segmento en base a un paragraph boundary. Un *paragraph boundary* corresponde a a una dirección de memoria múltiplo de 16.

La directiva *public* señala que el código perteneciente al segmento *cseg* podrá ser referenciado externamente por otro programa.

La directiva *code* indica al ensamblador que el segmento ahí definido contendrá código ejecutable.

La segunda línea del listado define, mediante la directiva *proc,* un procedimiento llamado *adq,* que marca el inicio del procedimiento principal. El parámetro *far,* señala que al poder quedar el procedimiento alojado en cualquier área de memoria, es necesario generar un *far call* para lograr su acceso desde otro programa. Los *far calls* se utilizan cuando no es posible asumir que la rutina referida se

encuentre definida dentro del mismo segmento donde está el procedimiento que la llama.

En la tercera línea encontramos la directiva *assume*, la cuál indica que los registros *CS, ES y DS*, hacen referencia a un mismo segmento, en este caso al segmento *cseg*. Normalmente cada uno de los tres registros tiene diferentes bloques de memoria asignados. En este caso, los tres comparten y hacen referencia a un mismo bloque, por lo que las direcciones asociadas a estos registros serán relativas al inicio de *cseg*.

La etiqueta *Begin* marca el inicio del programa principal. Begin no es una palabra reservada, lo que la hace en este caso especial es el hecho que está asociada a la directiva *end*, definida al final del programa. Cualquier etiqueta que se asocie con la directiva *end*, señala al ensamblador el inicio del programa principal.

Estructuras de Datos.

Dentro de la sección de directivas del driver se acostumbra la definición de estructuras de datos, las cuáles permiten el fácil acceso a información que el kernel de DOS pasa al Driver, a través de un paquete especial de datos llamado ***Request Header***. El uso de estructuras hace que el ensamblador calcule los offsets de cada uno de los elementos almacenados en un paquete de datos, liberando al programador de esta tarea. Así para accesar la información contenida en el Request Header, el programador no tiene que calcular un offset, sino hacer referencia tan sólo al nombre lógico asignado a la dirección de memoria donde se encuentra la información. Las estructuras definidas dentro de un programa fuente no ocupan memoria en el programa ejecutable, sólo ayudan al programador a realizar el acceso de localidades de memoria a través de mnemónicos.

DOS ha definido 25 diferentes comandos que pueden ser ejecutados por un device driver. Por cada comando, existen datos específicos que deben ser intercambiados entre DOS y el Driver, por lo cuál, es entendible la existencia de diferentes Request Headers que son construídos por el kernel de acuerdo al comando a ejecutarse. Así el Request Header de escritura a una impresora debe ser diferente en alguna de sus partes al Request Header utilizado para obtener su estado de operación. Para poder manejar con facilidad los diferentes tipos de Request Headers que DOS puede enviarle al driver, se definen

al inicio del mismo las estructuras correspondientes a los Request Headers de cada uno de los comandos que el driver debe ejecutar.

Todo Request Header que utilizan DOS y el Driver para su comunicación está estructurado por dos componentes: Una componente fija, común para todos los Request Headers y otra variable. A continuación se muestra un ejemplo de la forma como se define la estructura de datos utilizada para el acceso al Request Header, que es construído por el kernel para la ejecución del comando de Inicialización de un driver.

```
rh          struc         ; Request Header parte fija
rh_len      db      ?     ; Longitud total del Request Header
rh_unit     db      ?     ; Número de unidad (aplicable para
                          ; dispositivos de bloque solamente)
rh_cmd      db      ?     ; Comando a ejecutarse
rh_status   dw      ?     ; Palabra de Status para indicar a DOS
                          ; el resultado de la operación
rh_res1     dd      ?     ; campo reservado para DOS
rh_res2     dd      ?     ; campo reservado para DOS
rh          ends
```

Estructura de Datos del Request Header Sección Fija.

Esta es la parte fija que todo Request Header tiene, independientemente del comando a ejecutarse. En el primer renglón se define una estructura a través de la directiva *struc*, a esa estructura se le ha asignado el nombre lógico *rh* (Request Header).

El siguiente renglón define al campo que contiene la longitud total del Request Header. La finalidad de éste es indicar al driver el tamaño total del Request Header con que deberá trabajar. Recuérdese que a la estructura fija definida, aún falta adicionarle la parte variable, cuya longitud en bytes dependerá del comando a ejecutarse. La abreviatura *db* indica que *rh_len* es un campo que ocupa en el Request Header un byte.

El campo asociado con *rh_unit* se refiere al número de unidad de disco a quién va dirigido el Request Header, para los casos en que el device driver controla varias unidades simultáneamente.

El campo de *rh_cmd*, identifica la localidad donde se aloja el número de comando que el driver debe ejecutar.

El campo *rh_status*, es utilizado para señalar la localidad de memoria donde el driver debe colocar el código que indicará a DOS si la operación solicitada fué ejecutada con éxito o no. Este campo ocupa dos bytes dentro del Request Header. Todo driver al finalizar su ejecución y antes de devolver el control a DOS, deberá actualizar el campo de status. En relación a este hay varias condiciones que deben tenerse presentes. El bit de DONE debe ser encendido en el momento que el driver termine la ejecución de un comando. Ciertos comandos como Input Status, Output Status, Removable Media y Output til busy, encenderán el bit de Busy. El bit de Error deberá ser encendido si el driver determina que un error en la operación ha ocurrido, en este caso el campo de error deberá ser actualizado con el código de error que será pasado a DOS.

Estructura del Campo de Status del Request Header

Bit	Nombre del Bit	Descripción
15	Error	Si está encendido un error ha ocurrido.
8	Done	Deberá ser encendido al terminar la ejecución de un comando del driver.
9	Busy	Para prevenir operaciones sobre un dispositivo que está ocupado ejecutando una operación.
0-7	Código de Error	Código que indica a DOS el tipo de error ocurrido en la ejecución del comando.

Códigos de Error.

No.	Descripción
0h	Write Protect violation
1h	Unknown unit
2h	Drive not ready
3h	Unknown command
4h	CRC error
5h	Bad drive request structure lenght
6h	Seek error
7h	Unknown media
8h	Sector no found
9h	Printer out of paper
Ah	Write fault
Bh	Read Fault
Ch	General Failure.

Es posible observar de la tabla anterior, que los códigos de error que maneja DOS son limitados en cuanto a su significado, y por ello tal vez no sean los más adecuados para la descripción de los errores generados dentro de un device driver no standard. Es por ello, que muchas veces se deben utilizar otros comandos del driver como IOCTL Input y IOCTL Output para dar información más completa a la aplicación de lo que está sucediendo en el driver.

Los dos campos siguientes: *rh_res1 y rh_res2*, han sido reservados por DOS para su uso particular, por lo que no deben ser alterados. La directiva *dd* indica que el campo ocupa cuatro bytes.

Procedamos ahora a realizar el análisis de la estructura del Request Header correspondiente al comando de Inicialización.

```
rh0  struc                            ; Inicialización
rh0_rh        db size rh dup(?)      ; Región Fija del Request Header
rh0_nunits    db    ?                ; Número de unidades a ser controladas
                                     ; por el driver (dispositivos de bloque)
rh_brk_ofs    dw    ?                ; Offset for break address
rh_brk_seg    dw    ?                ; Segment for break address
rh_bpb_tbp    dw    ?                ; Offset del apuntador al arreglo de
                                     ; Bios Parameters Blocks.
rh_bpb_tbs    dw    ?                ; Segmento del apuntador al arreglo de
                                     ; Bios Parameters Blocks
rh_drv_ltr    db    ?                ; Primer drive disponible.
rh0  ends
```

Estructura de Datos para el Acceso al Request Header de Inicialización. Incluye la sección Fija y la Variable.

En el primer renglón aparece la definición de la estructura, que ahora lleva el nombre lógico de *rh0*, para indicar que con ella se accesa el Request Header correspondiente al comando 0 de inicializacion.

El primer campo de la estructura corresponde a la etiqueta *rh0_rh*, que engloba a través de la directiva *size rh* la definición de la sección fija del Request Header. Para poder accesar correctamente la estructura variable del Request Header, es necesario saber el número de bytes de que se compone la sección fija y saltarlos (13 bytes). Si es del interés del programador accesar algún elemento de la parte fija del Request Header durante la Inicialización del driver, debe usar la primera estructura de datos definida anteriormente bajo el nombre *rh*.

El segundo *campo rh0_nunits*, se usa solamente para drivers asociados a dispositivos de bloque. Es a través de este campo como DOS puede obtener información referente al número de drives que controla el driver. Si se quisiera accesar este campo sin el uso de estructuras, el programador tendría que considerar el offset de la parte fija, que son 13 bytes y el offset de la parte variable 1 byte, y hacer referencia al offset +14 a partir del inicio del Request Header, en lugar de simplemente invocar *rh0_nunits*.

Los campos correspondientes a *rh0_bkk_ofs* y *rh0_brk_seg*, sirven para indicarle a DOS la última localidad de memoria RAM ocupada por el driver.

A través de los campos *rh0_bpb_tbo* y *rh0_bpb_tbs*, DOS almacena la dirección de memoria donde se encuentra alojada la línea de definición del driver, tomada del archivo CONFIG.SYS. El driver por su parte, al término de la ejecución del comando de inicialización si es el caso, indica a DOS a través de estos campos la dirección donde se encuentra el arreglo de los BPBs (BIOS Parameters Blocks) asociados a cada uno de los drives controlados por el driver.

En el caso de que el driver controle un dispositivo de bloque, el último campo del Request Header sirve para indicarle al driver el nombre del primer drive disponible bajo DOS.

Con el propósito de facilitar el manejo de los Request Headers utilizados en el intercambio de información entre el driver y DOS, pueden ser definidas las 24 estructuras restantes, de forma similar a la anterior.

Para finalizar, es pertinente hacer incapié en dos cosas:

- **El Request Header, es construído por DOS en el momento de generar un comando para el driver.**

- **Las estructuras de datos definidas dentro de la sección de directivas, constituyen tan sólo un medio para hacer que el código del driver sea más comprensible y fácil de manejar para el programador.**

El Device Header.

Esta es la primera sección de código perteneciente al driver. El Device Header ocupa siempre los primeros 18 bytes del driver bajo la siguiente estructura:

```
next_device  dd   -1       ;apuntador al siguiente dispositivo
attribute    dw   8000h    ;características del dispositivo
strategy     dw   strat    ;apuntador a la rutina de estrategia
interrupt    dw   intr     ;apuntador a la rutina de interrupción
dev_name     db   'ADQ    ' ;nombre del dispositivo
```

next_device:

Es posible que en un mismo archivo se definan varios device drivers. El campo de *next_device*, indica a DOS si existen en el archivo sobre el que realiza la lectura, otros drivers que también deban ser instalados. Si ese es el caso, en este campo se debe indicar la dirección donde comienza el siguiente device driver a instalar. Cuando en el archivo sólo exista un driver, a este campo se le asignará el valor -1, indicando a DOS que es el último driver a instalar. Por ejemplo, en el archivo IO.SYS, están contenidos todos los drivers de default que proporciona DOS: CON:, AUX:, PRN:, etc., en cambio en el archivo ANSI.SYS, sólo se encuentra el driver que sustituye al de default CON:, para éste ultimo caso el valor del campo *next device* es igualado a -1.

atributte:

Este campo, describe a DOS el tipo de dispositivo que el device driver está controlando, así como las clases de comandos que el driver tiene implementados. En la siguiente tabla se proporcionan los diferentes significados de cada bit del campo.

Valores del Campo de Atributo.

Bit	Valor	Significado
15	0	El dispositivo es orientado al manejo de bloque
	1	El dispositivo es de tipo caracter
14	0	I/O control no es soportado
	1	I/O control es soportado
13	0	Dispositivo de Bloque que maneja formato IBM
	1	Dispositivo de Bloque que no maneja el formato IBM
	1	Output until Busy. (para dispositivos tipo caracter)
12	0	Reservado
11	0	Open/Close/Removable media no es soportado
	1	Open/Close/Removable media es soportado
10	0	Reservado
9	0	Reservado
8	0	Reservado
7	0	Reservado
6	0	Get/Set logical device. (para dispositivos de bloque)
5	0	Reservado
4	1	Si el driver realiza la función de CON:
3	1	Si el driver realiza la función de CLOCK:
2	1	Si el driver realiza la función de NUL:
1	1	Si el driver es el dispositivo de salida standard
0	0	Si el driver es el dispositivo de entrada standard (dispositivos tipo caracter)
	1	El driver soporta generic I/O Control. (dispositivos orientados al manejo de bloques)

Puede notarse de la tabla, que algunos bits sirven para indicar si ciertos comandos están disponibles dentro del device driver. Básicamente a través de esos bits es posible determinar si el driver soporta los comandos de extensión introducidos en la versión 3.2 de DOS.

El bit 15 indica a DOS, si el device driver controla un dispositivo que maneja la transferencia de información en forma de caracter o de bloque. Este bit es crucial porque según su valor, los bits 13 y 0 adquieren diferente significado.

El bit 14 es usado para indicarle a DOS, si el driver soporta operaciones de control de entrada/salida. Este tipo de operaciones son usadas para la obtención y actualización de los parámetros que controlan la operación de un dispositivo.

El bit 13, tiene varios significados, dependiendo del tipo de driver definido con el bit 15. Si el dispositivo es de tipo caracter y el bit está encendido, el driver debe soportar el comando *output til busy*. Si el driver es orientado a transferencias de bloque, el encendido o apagado de este bit indicará a DOS la manera como debe obtener información del disco. Si el disco no está en un formato IBM compatible, quiere decir que el FAT (File Allocation Table) no comienza en el segundo sector del disco, por lo que DOS debe usar el BPB (Bios Parameter Block) para determinar el formato del disco, para encontrar el FAT y el área de datos del usuario. Si el formato es IBM compatible, DOS usa el Media Descriptor Type del FAT para determinar el tipo de disco que está manejando. Existen ciertas inconsistencias respecto al mecanismo que debe ser utilizado en la identificación del tipo de medio. Los discos actuales vienen con diferentes capacidades y diferentes características físicas, como número de cilindros, pistas, sectores y cabezas. El problema que existe con la utilización del Media Descritor Type que se obtiene de leer el FAT, es que DOS sólo asignó tres bits para identificar todos los tipos de discos que pudieran existir disponibles para el funcionamiento de una PC, lo cuál por la diversidad de productos de almacenamiento que existen hoy en dia, no es ni remotamente suficiente esa área para su identificación, por lo se ha optado por utilizar el BPB, como el medio mas adecuado para la caracterización de un disco. Así el encendido o apagado de este bit tiene relación sobre la forma como es caracterizado un disco por DOS, si a través del FAT o a través del BPB.

El bit 11 señala si el driver soporta los comandos *Device Open, Device Close y Removable Media.* Los tres comandos pueden ser implementados en drivers orientados al manejo de bloques, como controladores de discos, pero sólo los dos primeros son válidos para drivers orientados al control de dispositivos tipo caracter como teclados, pantallas o impresoras.

El bit 6, indica si el driver soporta la ejecución de los comandos *Get Logical device y Set logical device* introducidos a partir de la versión 3.2 de DOS.

El bit 4, tiene significado sólo en el caso de los device drivers encargados del manejo de la consola. Si el bit es encendido, el driver debe hacer que la interrupción 29h quede asociada a una rutina de desplegado de caracteres muy rápida. Normalmente DOS, inspecciona cada caracter proveniente del teclado en busca de la secuencia CTRL-C, este modo de operación se denomina ***cooke mode***. Cuando no se desea que DOS intervenga y verifique cada caracter que se oprime en el teclado se enciende el bit 4 y se introduce otra rutina manejadora del interrupt 29h. Este modo de operación, sin inspección de DOS se conoce como ***raw mode***.

Si el bit 3, es encendido, el device driver tiene asociado el control del reloj del sistema, por lo que reemplaza el driver *clock$:* suministrado por DOS como standard.

El bit 2, indica que el driver está asociado al dispositivo NUL. Este driver no puede ser reemplazado por ningún driver instalable.

Si el bit 1 es encendido, el driver actual reemplazará al driver de salida de default. Si este es el caso, el bit 0 también deberá ser encendido, pues la consola controla tanto al dispositivo de entrada como al de salida.

El bit 0, para el caso de drivers orientados al manejo de caracteres, debe ser encendido si el bit 1 es encendido, para indicar que el driver sustituye al de default de entrada proporcionado por DOS. Para el caso de drivers asociados al manejo de bloques, el encendido de este bit indica a DOS que el driver tiene implementado el comando Generic I/O Control, introducido a partir de la versión 3.2.

device name:

Para el caso de drivers orientados al manejo de dispositivos tipo caracter, este campo contendrá el nombre lógico del dispositivo. Cuando un driver *instalable*, tiene que reemplazar a un driver *residente* de DOS, el nombre asignado al instalable deberá ser exactamente el mismo que el asignado al driver residente. El nombre de un driver queda reservado para el acceso al dispositivo por él controlado, por lo que ningún otro programa o archivo podrá ser utilizado bajo el mismo nombre. En el caso de los drivers orientados al manejo de bloques, en este campo, no se especifica nombre alguno, pero el primer byte del campo es utilizado para señalar el número de dispositivos que el drive controla.

Area para Almacenamiento de Variables Locales.

En esta sección se definen todas las variables utilizadas en la operación del driver. Por ejemplo, es en esta sección donde se almacenan a través de dos variables, el offset y el segmento que definen la dirección donde se encuentra el Request Header.

Sección de Procedimientos Locales.

En esta sección se definen las rutinas auxiliares para el procesamiento de cada uno de los comandos del driver. Muchas de ellas son utilizadas indistintamente por varios comandos. Dentro de esta sección es posible enmascarar rutinas que no son invocadas para el desarrollo de ninguno de los comandos que ejecuta el driver.

Muchos de los dispositivos conectados a la computadora, realizan las operaciones de intercambio de información con el CPU a través de la generación de interrupciones. Cuando un dispositivo genera una interrupción y el CPU es notificado, DOS instruye al CPU para que obtenga el número de la línea que generó la interrupción. En base a esta información, el CPU va a la tabla de interrupciones y lee la dirección de la rutina, encargada de atenderla: *el Interrupt Handler*: enseguida DOS pasa el control de la máquina a tal rutina. ***En general, cuando no existe un device driver asociado al dispositivo, el interrupt handler se almacena en memoria en forma de un programa residente. Cuando existe un device driver de por medio, el interrupt handler puede insertarse como una rutina más en el área de procedimientos locales y el vector de interrupción configurarse con la dirección de inicio de esa rutina.***

Lo que se gana con esto, es que todo el código necesario para el control y comunicación con un dispositivo queda integrado dentro del device driver, por lo que el programador y el usuario pueden despreocuparse del mantenimiento e instalación de un programa residente que haga las funciones de interrupt handler del dispositivo.

Sección de Estrategia.

Esta sección, es la encargada de salvar la dirección del Request Header generado por DOS. Cuando DOS requiere la atención de un driver construye un Request Header con toda la información necesaria para que el driver ejecute un comando. Una vez construído el Request Header, su dirección de memoria se almacena en los registros BX:ES. La sección de estrategia, debe contar con todo el código necesario para leer la información de esos dos registros y almacenarla en alguna variable local del driver.

Sección de Interrupción.

Una vez que DOS ha llamado a la sección de estrategia, pasará el control a la sección de interrupción. En esta sección se realizan varias acciones las cuales se describen a continuación:

a).- **Se salva el estado actual del sistema, guardando en el stack los valores de los registros que serán modificados durante la operación del driver.**

b).- **Se efectúa el acceso al Request Header, para identificar el comando que DOS desea que el driver ejecute. Para lograr el acceso al mismo, se utiliza la dirección del Request Header que había sido previamente guardada en una variable local durante la ejecución de la sección de Estrategia.**

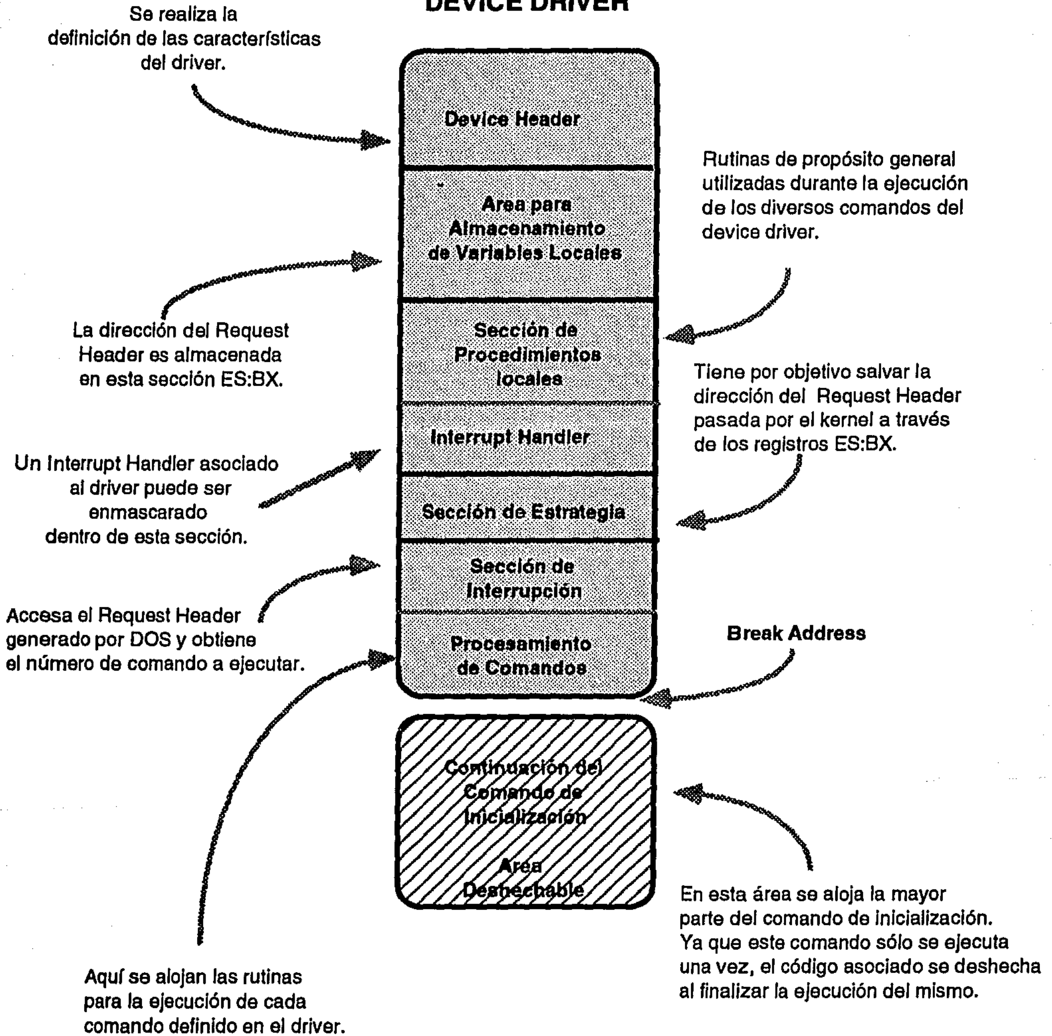
c).- **Ya obtenido el número de comando a ejecutar, el control se pasa a la rutina encargada de efectuar esa operación. (Sección de Procesamiento de Comandos)**

d).- **Una vez ejecutado el comando requerido, la rutina debe actualizar el campo de status del Request Header, para indicar el resultado de la operación.**

e).- **El entorno del sistema se regresa al mismo estado en que se encontraba antes de comenzar la ejecución de esta sección, mediante la restauración de valores de los diferentes registros del CPU que habían sido previamente guardados en el stack.**

f).- **El control del CPU vuelve nuevamente al kernel de DOS.**

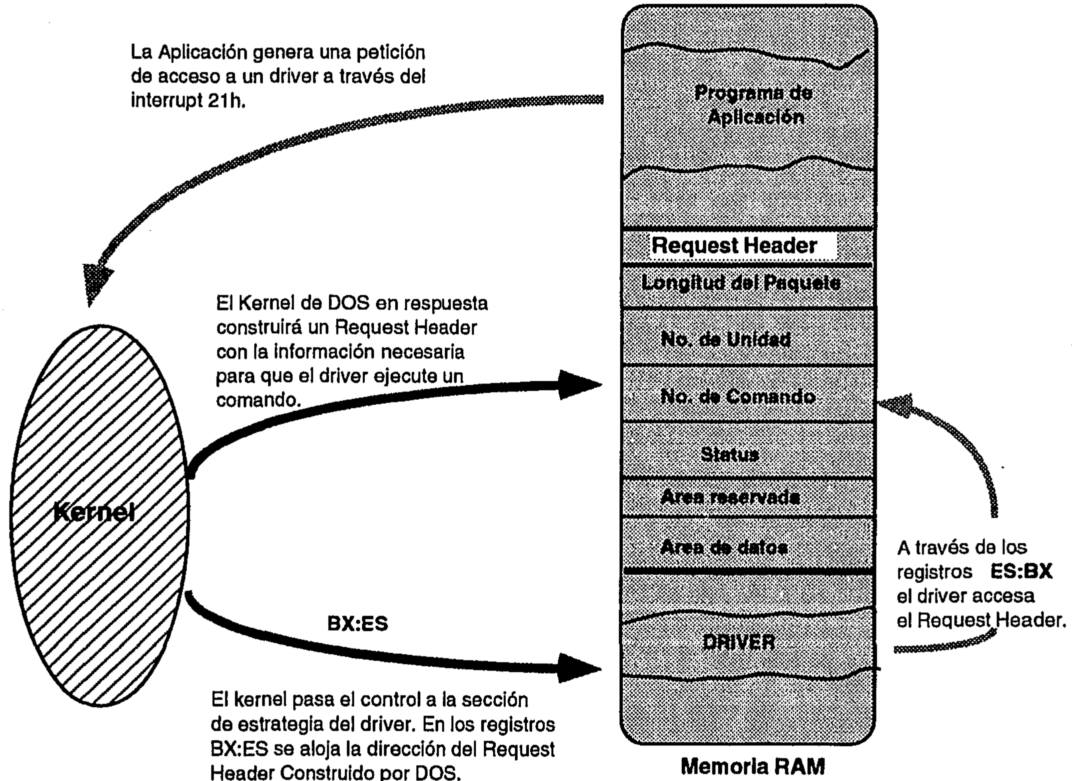
ESTRUCTURA DE UN DEVICE DRIVER



La presente figura muestra los elementos que conforman la Estructura de un Device Driver.

Comunicación del Kernel con el Device Driver

El siguiente esquema, nos muestra la forma como el driver y DOS trabajan en conjunto. En el dibujo podemos apreciar que cuando DOS hace una llamada al driver pasa la dirección de un paquete de datos al mismo. Ese paquete de datos recibe el nombre de *Request Header* y contiene información en base a la cuál el driver realizará la ejecución de un comando de DOS.



El Request Header.

El Request Header, citado ya en páginas anteriores, es una estructura que se utiliza para realizar la comunicación entre DOS y el Driver; a través

de ésta, el device driver determina las operaciones que debe realizar y obtiene las direcciones de memoria donde se encuentran los datos involucrados. Por su parte DOS, una vez notificado que el comando ha sido ejecutado, realiza una revisión sobre esa estructura para determinar a través del campo de status, si la ejecución del comando fué correctamente realizada por el driver o no. Cuando DOS requiere escribir un caracter a un puerto serial, por ejemplo, necesita indicarle al driver correspondiente que ejecute el comando Write y pasarle al mismo tiempo el caracter a escribir. La manera de hacer esto es a través de un Request Header, en la estructura del mismo irán contenidos el comando a ejecutar y los datos requeridos. Al terminar la ejecución del comando, el driver actualizará el campo de status del Request Header para reflejar el resultado de la operación. DOS, al recobrar el control de la máquina revisará el campo de status, para decidir las operaciones subsecuentes a realizar.

Estructura del Request Header

Campo	Longitud (en Bytes)	Descripción
1	1	Longitud en bytes del Request Header.(El valor varía de acuerdo al comando a ejecutarse)
2	1	Código de unidad del dispositivo.
3	1	Código del comando a ejecutarse
4	2	Campo de Status
5	8	Reservada para DOS
6	n	La longitud de este campo varía de acuerdo al comando a ser ejecutado

En la tabla precedente, se puede observar que el Request Header es un paquete de datos de longitud variable. Dentro de este paquete, la longitud total del Request Header se define dentro del primer campo.

El segundo campo del Request Header, contiene el código de unidad del dispositivo. Este, normalmente se usa cuando más de un dispositivo está conectado a un controlador. Con el mismo, se indica al driver a cuál dispositivo de todos los que controla está destinada la operación. Por ejemplo el controlador del lector de floppy disk, usualmente controla dos drives. El drive A:, que podría ser la unidad 0 y el drive B:, la unidad 1. DOS a través de este campo, señala al driver si la operación se debe aplicar sobre el drive A: ó sobre el drive B:.

El tercer campo, indica el número del comando a ejecutarse, que el driver usa para saber qué clase de operación debe efectuar.

El cuarto, funciona como un indicador de status, y es el medio a través del cuál, el driver le informa a DOS el resultado de una operación.

El quinto campo está reservado para DOS y no está documentado.

El sexto, corresponde al área de datos. Esta varía en longitud dependiendo del número de comando que DOS haya definido en el tercer campo.

DOS, automáticamente construye un Request Header en el momento que un programa genera una petición que involucra el acceso a un driver. El paquete de datos reside en el espacio de memoria reservado a DOS y es construído en base a la información que proporciona la aplicación. La dirección donde se encuentra el Request Header se entrega al device driver en el momento que DOS pasa el control al mismo. Esta dirección la guarda el driver en la sección de almacenamiento de variables locales para ser usada posteriormente. Ya que el Request Header puede estar en cualquier parte de los 640 Kb. de memoria de la máquina, son necesarios un segmento y un offset para especificar por completo su dirección, DOS pasa la dirección del Request Header al driver a través de los registros **ES y BX** (segmento y offset respectivamente).

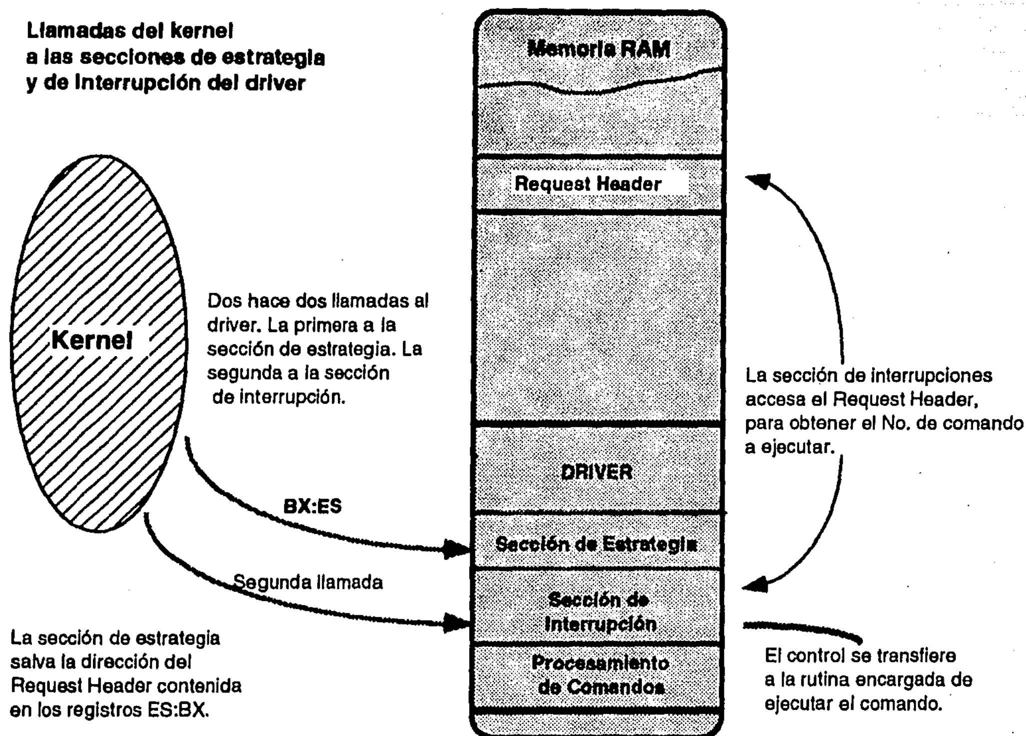
Llamadas al driver desde DOS.

Se pudiera pensar que cada comando que DOS pasa al driver involucra una sóla llamada al mismo, pero no es así.

Cada vez que DOS pide al driver que procese un comando, llama al driver dos veces. En la primera llamada, DOS transfiere el control a la sección de Estrategia definida en el driver. En la segunda, DOS deja el control de la máquina a la sección de Interrupción.

La sección de Estrategia tiene como misión primordial el salvar la dirección de Request Header entregada por DOS y contenida en los registros **ES:BX**.

La sección de Interrupción tiene por objetivo la ejecución del comando señalado dentro del Request Header. Es a partir de esta sección donde se transfiere el control a rutinas específicas que hacen posible la ejecución de un comando.



La razón por la que DOS hace dos llamadas al driver para la ejecución de cada comando la podremos encontrar en la siguiente explicación:

En un ambiente multiproceso, pudiera darse el caso, que requerimientos a distintos drivers fueran solicitados por varios programas de aplicación en forma simultánea. El sistema operativo tendría que decidir en ese momento que petición atender primero. Si el esquema de comunicación con el driver involucrara sólo una llamada por comando, el sistema operativo atendería siempre primero al requerimiento generador de la primera petición. Esta estrategia de atención estaría bastante limitada por el hecho que una petición de mayor importancia tendría que esperar a que se atendiera una de importancia menor, por la sóla razón de que la petición de menor importancia se hubiera generado tan sólo unos microsegundos antes.

A fin de dar mayor flexibilidad en la atención de los device drivers e introducir prioridades, se adopta el esquema de dos llamadas al driver por cada comando. En la primera llamada, el sistema operativo pasa al driver la dirección de Request Header, el driver operando en la sección de Estrategia salva esta dirección y regresa de inmediato el control a DOS. DOS forma una lista con las peticiones generadas en una ventana de tiempo, con esa lista, analiza qué device driver tiene la mayor prioridad y a esa petición atiende primero. El device driver de mayor prioridad es el que se encuentra mas cercano al inicio de la cadena, es decir, el definido a continuación de NUL:.

Bajo este esquema, el sistema operativo notifica primero al driver que hay trabajo por hacer llamando a la sección de Estrategia, después pide al driver de mayor prioridad que inicie la ejecución del trabajo, llamando a la sección de Interrupciones del mismo. Esta forma de operación fué pensada previendo que algún día DOS podría convertirse en un sistema operativo multiproceso y la introducción de este esquema de atención, facilitaría mucho las cosas al momento de realizar la migración. MSDOS aún no se ha convertido en un sistema multiproceso y tal vez nunca llegue a serlo, pero esta estrategia para la asignación de prioridades y atención a los drivers funciona dentro de DOS desde la versión 2.0. Como dato interesante, Microsoft en su sistema operativo OS/2 que es multitareas, introdujo una estrategia de llamado a los drivers muy similar a la manejada por MSDOS. (Mastrianni, 1991).

Llamadas a Device Drivers de Bloque y de Caracter.

Dentro del conjunto de comandos que DOS tiene a su disposición para el acceso a drivers, existen algunos que son específicos para drivers que controlan dispositivos de Bloque y otros apropiados solamente para drivers que controlan dispositivos tipo caracter. Por ejemplo el comando Media Check, sólo tiene significado cuando es dirigido a un dispositivo de bloque, veamos porqué. En el caso de los discos flexibles, el usuario de la máquina tiene libertad para cambiarlos en el momento que desee. Si el cambio de discos se realiza cuando aún existen datos en el buffer pendientes de escribirse, tales datos serán perdidos inadvertidamente para el usuario, y lo què es peor, tal vez serán escritos en el nuevo disco introducido dañando la información ahí contenida. Para evitar esta situación DOS, antes de realizar una escritura o lectura al disco genera este comando para tratar de asegurarse que el disco no ha sido sustituído por otro. DOS necesita saber a menudo que tipo de dispositivo maneja el driver, para así generar los comandos apropiados según el caso.

Comandos para el Device Driver.

Ya se ha descrito, que la manera para lograr el acceso a un driver desde un programa de aplicación es a través de peticiones de *servicio de DOS*. Cada requerimiento que llega al kernel de DOS proveniente de las aplicaciones, se traduce en una serie de comandos específicos que el driver debe entender para ejecutar. Esos comandos son comunes a todos los drivers, más como ya se mencionó, existen algunos que son específicos para un tipo de dispositivos solamente. En la tabla siguiente se presentará la lista de todos los comandos posibles que puede mandar a ejecutar el kernel a un driver, indicándose aquellos que sean aplicables sólo para dispositivos de bloque ó para dispositivos tipo caracter y la versión de sistema operativo donde fueron introducidos.

Tabla de Comandos de un Device Driver.

No.	Dispositivo	Version	Descripción
0	CB	2.0	Inicialización
1	XB	2.0	Media Check
2	XB	2.0	Get BIOS Parameter Block
3	CB	2.0	IOCTL Input
4	CB	2.0	Input
5	CX	2.0	Nondestructive Input
6	CX	2.0	Input Status
7	CX	2.0	Input Flush
8	CB	2.0	Output
9	CB	2.0	Output with Verify
10	CX	2.0	Output Status
11	CX	2.0	Output Flush
12	CB	2.0	IOCTL Output
13	CB	3.0	Device Open
14	CB	3.0	Device Close
15	XB	3.0	Removable Media
16	CX	3.0	Output Until Busy
17	XX	X.X	No definido
18	XX	X.X	No definido
19	CB	3.2	Generic IOCTL
20	XX	X.X	No definido
21	XX	X.X	No definido
22	XX	X.X	No definido
23	CB	3.2	Get Logical Device
24	CB	3.2	Set Logical Device

X.- No aplicable
B.- Aplicable para dispositivos de Bloque
C.- Aplicable para dispositivos tipo Caracter

Comando 0: Inicialización.

Una vez que el driver ha sido cargado en memoria, DOS automáticamente genera la petición para que se ejecute el comando 0 de Inicialización. Cuando éste es recibido por el driver, comienza a ejecutarse una secuencia de inicialización para el dispositivo que incluye:

- **Desplegado de mensajes en la pantalla anunciando la instalación del driver.**
- **Inicialización de variables locales.**
- **Ejecución de instruciones para la configuracón del dispositivo.**

Las llamadas a servicios de DOS (*int. 21h* servicios *01h a 0ch y 30h)* pueden ser solicitadas por el driver solamente durante la ejecución de este comando. Para el caso de los demás comandos, el driver no puede utilizar los servicios de DOS disponibles bajo la interrupción *21h*; si el driver genera una petición de servicio de esta clase durante la ejecución de cualquier otro comando, el sistema operativo se caerá.

Durante la ejecución del comando de Inicialización, el driver aún no se considera parte del sistema operativo, pues apenas se está realizando su instalación, desde el punto de vista del kernel, el driver es en ese momento un programa de aplicación común, por ello puede generar la petición de servicios de DOS agrupados bajo la interrupción 21h.

El comando de Inicialización, debe regresar al kernel la dirección de memoria donde termina el driver ***(Break Address).*** Con esta información DOS conoce exactamente a partir de que localidad de memoria puede disponer para la instalación de otros drivers, para construcción de estructuras de control o para la ejecución de aplicaciones.

En el caso de los drivers asociados a dispositivos de bloque, el comando de inicialización debe regresar el número de unidades manejadas por el driver y un arreglo de apuntadores a los **BPBs (Bios Parameters Blocks)**. MSDOS usa el número de unidades controladas por el driver en la asignación de identificadores de dispositivo. Por ejemplo, si el último drive definido fuese el D: y el driver que se estuviese inicializando fuera a controlar cuatro dispositivos, MSDOS asignaría al driver las letras E,F,G y H para el acceso a cada uno de ellos. En el Device Header, el driver ofrece información acerca del número de unidades que va a controlar, sin embargo DOS no hace uso de esa información.

El *BPB* es una estructura que existe en todo disco formateado. Esta contiene información relativa a las características específicas del disco, como son el número de sectores por pista, el número de cabezas y el número de bytes por sector. Por cada drive en uso, el driver debe alojar en memoria una copia del BPB del disco para que DOS esté enterado de las características físicas de cada unidad. El número de elementos existentes en el arreglo de apuntadores a los BPBs corresponderá al número de unidades controladas por el driver. Si el driver maneja 2 unidades, el arreglo de apuntadores tendrá dos direcciones, cada una de ellas apuntando a un BPB correspondiente a cada unidad de disco. Si varias unidades tienen las mismas características en su estructura, es posible crear solamente un BPB, a fin de ahorrar memoria. Durante la inicialización del sistema, DOS recorre todos los BPBs a fin de localizar el tamaño de sector más grande definido, y en base a él realizar la asignación de memoria para los buffers de acceso a disco.

Una técnica de programación que se utiliza para el ahorro de memoria, es la colocación de las rutinas de inicialización al final del driver. Cuando la inicialización termina, las rutinas asociadas a este comando no se usan más, por lo que pueden ser deshechadas. Así la dirección de finalización del driver ***(break address)*** se da a partir de la ***localidad donde comienza la primera línea de código de las rutinas de inicialización.*** DOS asume con ello, que la memoria donde se almacenan estas rutinas queda disponible.

Si durante la ejecución del comando de inicialización se determina que el dispositivo no se encuentra disponible o tiene alguna falla, es posible abortar la instalación del driver a fin de no ocupar innecesariamente memoria. Para hacer esto, el driver colocará un cero en el campo correspondiente al número de unidades del Request Header y definirá el *break address* del mismo en la dirección *CS:0000h.* Para el caso de los driver asociados a dispositivos tipo caracter, además de lo anterior, se debe igualar con cero el bit 15 de la palabra de atributo perteneciente al *Device Header* del driver.

Comando 1: Media Check.

Este comando, se aplica sólo a drivers que controlan dispositivos de bloque. Los drivers que controlan dispositivos tipo caracter, sólo deben responder a él asignando un 1 al bit de done en la bandera de status del Request Header. *Media Check* es utilizado para determinar si un disco ha sido removido. DOS realiza esta verificación antes de leer o escribir a un disco para asegurarse que este no ha sido cambiado después del último

acceso. Para el caso de los RAM disk y discos duros, este comando debe reportar siempre que el disco no ha sido cambiado. Los códigos de status que el driver entrega a DOS (IBM, 1987) pueden ser:

- -1 : El medio de almacenamiento ha sido cambiado.
- 0: No se puede determinar si el medio fué cambiado.
- 1 : El Medio no fué cambiado.

Comando 2: Get BPB (Bios Parameter Block).

Este comando se aplica sólo a dispositivos de Bloque. El kernel lo invoca para obtener la dirección de memoria donde se encuentra el BPB del disco a accesar. *Get BPB* es generado cuando el comando Media Check indica al kernel que el disco ha sido cambiado, o cuando no existe certeza si fué cambiado o no. La información existente en el BPB, permite a DOS determinar las direcciones donde se alojan el FAT, el File Directory, así como otros parámetros importantes de acceso al disco.

Comando 3: IOCTL Input.

I/O Control Input es un comando que permite el intercambio flexible de información entre el driver y la aplicación. En principio, está diseñado para que a través de él, el driver envíe información al programa de aplicación, relativa a los valores asociados a los parámetros de control que usa para su operación. Ya que los parámetros de control varían de un driver a otro, DOS no realiza la verificación de los mismos, pero da lugar a que el programa de aplicación pueda obtenerlos y realice su interpretación. El número y el orden de los parámetros queda a elección del diseñador del driver, el cuál deberá dar información pertinente al desarrollador de aplicaciones acerca del formato y significado de cada uno de ellos.

Comando 4: Input.

Este comando instruye al driver para que sea realizada una lectura de datos sobre el dispositivo periférico. Los datos leídos se almacenan en un buffer, que es señalado por el kernel del sistema. Si un error se detecta durante la lectura al dispositivo, el driver debe encender el bit de error en el campo de status del Request Header y reportar el número de bytes o sectores que pudieron ser almacenados en el buffer antes de que el error ocurriera.

Comando 5: Nondestructive Input.

Este comando se utiliza solamente en dispositivos tipo caracter. Su función es similar a la realizada por el comando Input, con la variante que el caracter leído (si lo hay), no se remueve del buffer interno del dispositivo. DOS usa esta función para verificar si existen o no caracteres disponibles para lectura. Si hay caracteres esperando ser leídos, la aplicación simplemente generará un Input para obtenerlos. No todos los dispositivos tienen buffer interno de datos, en caso de que no exista el buffer, el driver debe realizar la lectura de un caracter sobre el dispositivo. El caracter leído es pasado a DOS y al mismo tiempo almacenado dentro de una variable local del driver, para dejarlo disponible para el próximo comando Input a ejecutarse. Con esta acción se cumple el requerimiento de lectura no destructiva.

Comando 6: Input Status.

El uso del comando Input Status, es válido sólo en el caso de dispositivos tipo caracter. Este comando permite a DOS verificar si el buffer asociado a un dispositivo de entrada tiene caracteres en espera de ser leídos o no. También permite verificar el estado de operación del dispositivo. Si el dispositivo no cuenta con un buffer asociado o bien existen caracteres almacenados en él, el bit de Busy del campo de status del Request Header deberá ser apagado a fin que DOS genere de inmediato un comando de lectura. Si el dispositivo no está listo o no existen caracteres en el buffer, ninguna operación de lectura debe ser ejecutada, por lo que el bit de Busy deberá ser encendido.

Comando 7: Input Flush.

Este comando, permite deshechar la información almacenada en el buffer interno del driver. Supóngase que un programa pregunta al usuario si desea borrar todos los archivos del disco. Si no se hace uso de este comando para limpiar la información contenida en el buffer, pudiera darse el caso que el caracter almacenado mediante la función de *type ahead* del teclado, coincidiera con la letra esperada por la aplicación para proceder a borrar todos los archivos. La utilización de este comando es recomendable para deshacerse de todos los posibles caracteres extraños antes de realizar la lectura de datos críticos de un dispositivo, en este caso del teclado.

Comando 8: Output.

Esta función transfiere datos desde un buffer señalado por el Request Header, hacia el dispositivo. Si un error ocurre durante la escritura, el bit de error del campo de status del Request Header es encendido, y el número de bytes o sectores que lograron transferirse al dispositivo reportado.

Comando 9: Output with Verify.

Este comando es similar al anterior, pero tiene una función adicional: cuando el switch de verificación se enciende en el nivel de comandos de DOS, el driver después de cada operación de escritura debe leer el dato que escribe. Esta es una característica útil cuando se necesita tener certeza que datos críticos han sido escritos correctamente. La aplicación de este comando no tiene ningún significado en impresoras o pantallas, su utilización se enfoca a dispositivos de almacenamiento magnético como discos duros.

Comando 10: Output Status Command.

Cuando DOS necesita escribir a un dispositivo, éste es el primer comando que se envía al device driver. De la información que éste regresa, dependerá si DOS genera el comando Output inmediatamente o espera a que el dispositivo se encuentre listo. Este comando es válido sólo para dispositivos tipo caracter y se utiliza para obtener el estado de operación de dispositivos de salida como impresoras. Dispositivos de este tipo, cuentan con buffers que pueden almacenar datos en espera de ser procesados. Mediante este comando, es posible indicarle al dispositivo que verifique si su buffer está lleno o no, o bien verificar si el dispositivo está en condiciones de atender una operación de salida, al terminar su verificación, el dispositivo entregará al driver el resultado de la misma, el cual, dependiendo de esa información, encenderá o apagará el bit de Busy de la palabra de status del Request Header.

Comando 11: Output Flush.

Este comando instruye al driver para que envíe una señal al dispositivo informándole que todos los datos presentes en él deben ser deshechados. En una impresora tendríamos el ejemplo más claro. Las impresoras tienen un buffer interno en donde se van almacenando los caracteres a imprimir. Cuando el usuario desea abortar una impresión, puede pedir al driver se ejecute este comando, para que los datos presentes en el buffer sean deshechados y la impresión detenida.

Comando 12: IOCTL Output.

El presente comando, está diseñado para enviar al driver parámetros de control, que le permitan modificar su operación. Al igual que ocurre con el IOCTL Input, el formato de transferencia de datos entre el driver y la aplicación definido por el programador del driver.

Comando 13: Device Open.

Este comando fué introducido en la version 3.0 de MSDOS y puede ser utilizado siempre que el bit 11 de la palabra de atributo del device header esté encendido.

En los dispositivos de bloques, es usado por el driver para actualizar el contador de archivos abiertos, o para inicializar el contenido de buffers y variables locales.

En los dispositivos tipo caracter, es usado para inicializar contadores y demás variables de operación del driver. Hay que hacer notar que los file handles predefinidos para los drivers CON, AUX y PRN siempre permanecen abiertos.

Comando 14: Device Close.

Este, es utilizado a partir de la versión 3.0 de MSDOS y posteriores, siempre y cuando el bit 11 de la palabra de atributo del Device Header esté encendido. Device Close trabaja en coordinación con el comando Open.

Comando 15: Removable Media.

Este comando se usa en dispositivos de bloque. Con él se pregunta al device driver si el dispositivo asociado maneja medios de almacenamiento removibles. Esta información puede ser de utilidad a DOS para saber a que drivers es necesario enviarles el comando Media Check.

Comando 16: Output Til Busy.

Esta función es válida solamente en dispositivos tipo caracter y es usada cuando se desea enviar un conjunto de caracteres al dispositivo en lugar de un sólo caracter cada vez. En el caso de las impresoras, por ejemplo, puede utilizarse para enviar toda la información al buffer de la misma, con esto el programa de aplicación una vez que ha transferido todos los caracteres al buffer queda liberado para hacer otra cosa, mientras la impresora continúa efectuando paralelamente su tarea. El driver deberá notificar a la aplicación el momento en que el buffer se llene, indicando a la aplicación el número de caracteres transferidos.

Comando 19: Generic IOCTL.

Este comando es soportado solamente en la versión 3.2 de DOS en adelante y es válido sólo en los casos que el bit 0 de la palabra de atributo del device header esté encendido. Este es accesible a las aplicaciones a través de la interrupción 21h, servicio 44h, subfunción 0Dh. El propósito del mismo es proveer una interface de control standard para dispositivos orientados al manejo de bloques. A partir de la versión 3.2, MSDOS hace una definición más clara de la interface necesaria para el control de dispositivos de bloques. Los servicios que pueden ser solicitados con este comando, si el driver los soporta incluyen la configuración del driver para el manejo de formatos de disco no standard, lectura y escritura de pistas completas de datos, inicialización del disco y verificacion de pistas. Esta clase de servicios eran proveídos anteriormente a través de utilerías, con este comando se abrió el camino, para que el driver mismo suministrara este tipo de facilidades.

Comando 23: Get Logical Device.

A partir de la versión 3.2, MSDOS permitió al usuario especificar nombres lógicos múltiples a un mismo drive. Este comando pregunta al device driver que otra letra, diferente a la de default ha sido utilizada para accesar un mismo drive. Por ejemplo, la segunda unidad de floppy, normalmente accesada a través de la letra B:, podría ser accesada con la letra lógica E: previa asignación mediante el comando set logical device.

Comando 24: Set Logical Device.

Esta función informa al driver, que el próximo identificador lógico de drive disponible será utilizado como letra alterna para referenciar a un dispositivo de bloque asociado al driver.

SAD
Sistema de Adquisición de Datos

SAD.

Sistema de Adquisición de Datos.

El sistema de Adquisición de Datos SAD, ha sido diseñado con el objeto de posibilitar la recepción de hasta 16 señales digitales de datos. Para la operación de este sistema son necesarios los siguientes elementos:

a).- Una computadora IBM compatible, con procesador 80286 operando a 16 Mhz, disco duro, coprocesador matemático y un Mb de memoria RAM como mínimo.

b).- Una tarjeta para adquisición de datos **SAD**, que va instalada en una de las ranuras de expansión de la computadora.

c).- El device driver **SAD**, encargado de brindar al programa de aplicación una interface lógica de alto nivel para el control de la tarjeta de adquisición y el acceso a los datos recibidos.

Características del Sistema SAD.

1).- Posibilita la recepción hasta de 16 canales de datos digitales en tiempo real.

2).- Capacidad para el manejo de varias velocidades de recepción por canal, configurables por el usuario. Actualmente implementada la recepción a 1200 y 2100 bps.

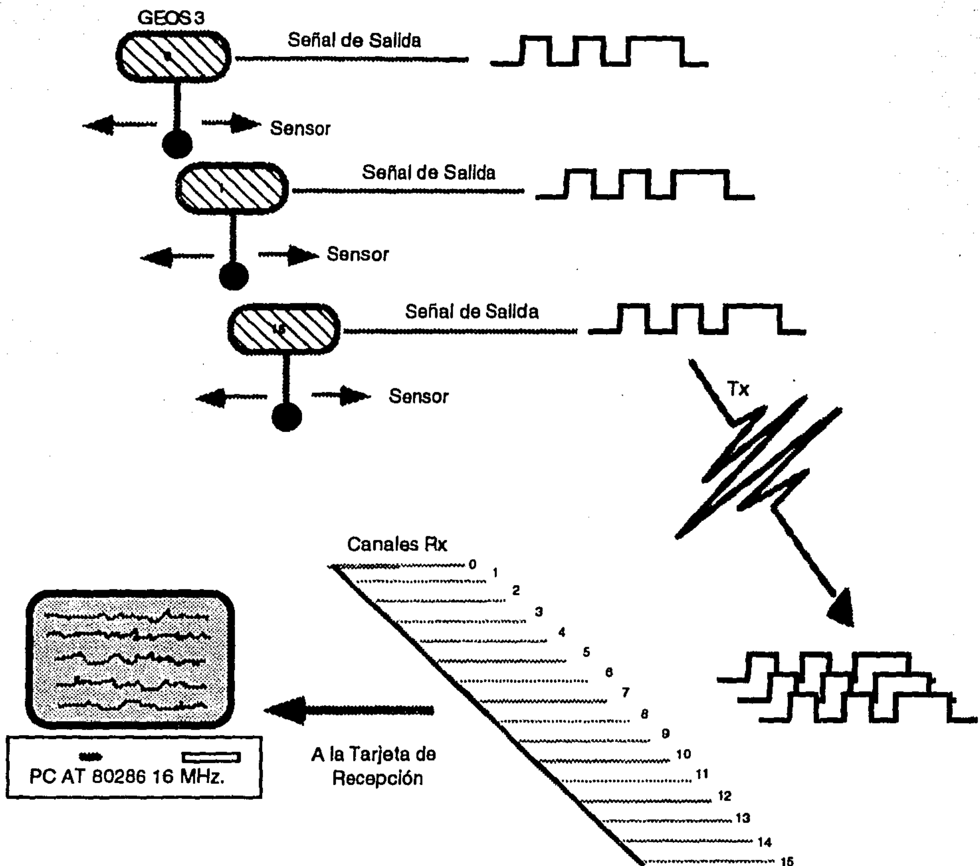
3).- Efectúa la recepción de datos con rapidez suficiente para permitir al procesador la ejecución en tiempo real de tareas relacionadas con la graficación y el análisis de los datos adquiridos.

Principio de Operación del Sistema SAD.

La operación del sistema SAD, contempla dos procesos primordiales:

a).- Recepción y Almacenamiento de Datos.

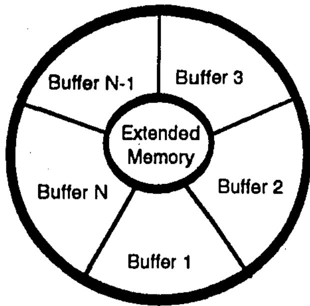
b).- Recuperación de Datos.



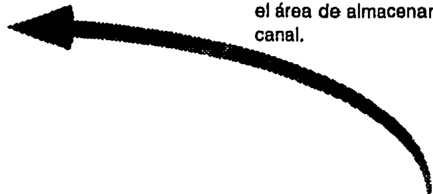
Sistema de Adquisición de Datos Digitales

El proceso de Recepción de Datos.

La recepción digital de datos es llevada a cabo por el *Sistema de Adquisición* mediante el uso de dos UARTs 2698B. (ver pág. 69). Cada UART (Universal Asynchronous Receiver Transmitter) tiene capacidad para la recepción simultánea hasta de 8 canales de datos, por lo que es factible la recepción de 16 canales a través de **SAD**. Cada canal dispone de un pequeño buffer dentro del UART, identificado bajo el

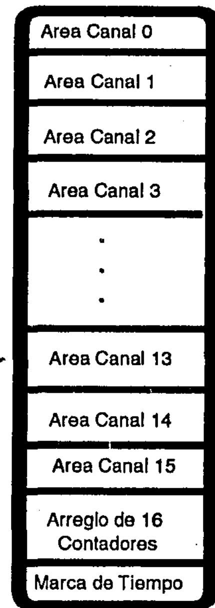


Sistema Circular de Buffers



El *Buffer Primario* es almacenado en el *Sistema Circular de Buffers,* al llenarse el área de almacenamiento de un canal.

Buffer Primario



Interrupt Handler

CPU i80286

INTR

PIC MAESTRO PC

IRQ

PIC Externo

Tarjeta de Adquisición

Los datos provenientes de los UARTs son almacenados en el buffer Primario.

UART 1

UART 2

0 7 8 15

Canales de Entrada

PROCESO DE RECEPCION DE DATOS

nombre de RHR (Receive Holding Register), donde es posible almacenar hasta tres bytes con la información recibida. Los UARTs están programados de tal forma, que al momento de que uno de los 16 buffers de recepción se llena, se genera una interrupción.

Las interrupciones generadas por los UARTs, son captadas mediante un PIC 82C59A-2 (Programable Interrupt Controller) perteneciente a la tarjeta de adquisición. Este, se conecta a su vez a una de las líneas de entrada del PIC interno de la computadora. El PIC de la tarjeta de adquisición, generará una interrupción y al mismo tiempo analizará qué canales de los UARTs han generado interrupciones, con objeto de determinar cuál de ellos tiene la mayor prioridad.

El PIC de la computadora, evaluará a su vez las interrupciones que los diferentes dispositivos hayan generado, con objeto de identificar la de mayor prioridad.

El microprocesador al recibir la interrupción, preguntará al PIC interno que línea de interrupción debe ser atendida. EL PIC interno le dará un número de línea al CPU, con el cuál obtendrá si es el caso, la dirección del Interrupt Handler del sistema SAD.

El Interrupt Handler del sistema de adquisición, cuenta con todo el código necesario para detectar cuál de los 16 RHR de la tarjeta se ha llenado. Ya efectuada esa detección, procede a realizar el vaciado correspondiente. Los datos leídos del RHR, se almacenan en un buffer de mayor capacidad, llamado ***buffer primario***, residente dentro del área de los 640kb de memoria RAM de la computadora. El interrupt handler, debe detectar el momento en el que alguna de las áreas del *buffer primario* se llene, para enviar todos los datos almacenados en él, a un ***sistema circular de buffers*** situado en memoria extendida. El *buffer primario* queda entonces disponible para el almacenamiento de nueva información procedente de los RHR de los UARTs.

En el *sistema circular* de buffers podrán almacenarse tantos *buffers primarios* como *memoria extendida* exista disponible. La gran ventaja de usar este sistema de almacenamiento, es que el área de memoria principal (0-640K) se deja prácticamente liberada para el uso de aplicaciones, al ocuparse sólo el área correspondiente a un *buffer primario.* El uso de memoria extendida para almacenamiento de información es altamente recomendable en estos casos, ya que no existen limitaciones de importancia en cuanto a la cantidad de memoria a utilizar.

En el momento que los datos quedan almacenados dentro del sistema circular de buffers, quedan disponibles para ser accesados por cualquier aplicación.

El Proceso de Recuperación de Datos.

Ahora describiremos la manera como la aplicación obtiene los datos recibidos por la tarjeta de adquisición.

Cuando la aplicación necesite obtener la información alojada dentro del *sistema circular de buffers*, solicitará al kernel que efectúe una lectura sobre el archivo SAD, proporcionando para ello el *file handle* obtenido previamente con la función DOS Open.

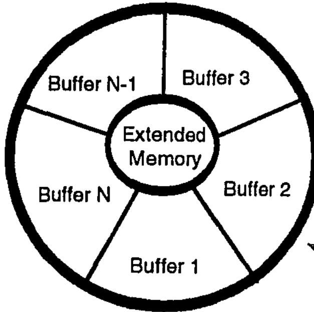
En primer término, el kernel transferirá el control del CPU a la sección de *estrategia* de *SAD*, en donde será salvada la dirección del Request Header.

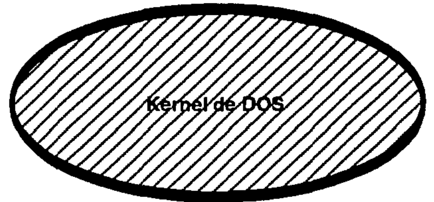
Cuando el control del micro se transfiere a la sección de interrupción de *SAD*, son efectuadas todas las tareas necesarias para la localización y transferencia de un bloque de datos del *sistema circular de buffers* en memoria extendida al área de datos de la aplicación en memoria principal. Nótese que el usuario de la aplicación sólo debe solicitar una lectura a un archivo, todo el proceso para la organización y recuperación de los datos es llevado a cabo por el device driver. Una vez que el driver ha terminado de transferir la información al área de datos de la aplicación, actualiza el campo de status del Request Header para reflejar el resultado de la operación, y regresa el control del CPU al kernel de DOS.

Proceso de Recuperación de Datos

Int 21h
Requerimiento de Lectura de Datos.

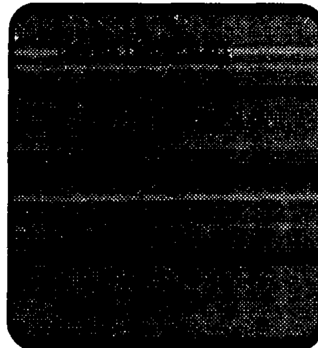
El contenido de un buffer es pasado al segmento de datos de la aplicación.





Sistema Circular de Buffers

Un Comando de IOCTL_INPUT es generado por el kernel para el driver



El driver determina que buffer del Sistema Circular está disponible para la aplicación.

SAD
Device Driver para el Sistema de Adquisición de Datos

SAD
Device Driver para el Sistema de Adquisición de Datos.

Durante el desarrollo de los temas precedentes, fueron introducidos los conceptos necesarios para el entendimiento de la operación de un Device Driver. Se habló de su origen, de su necesidad e importancia, de la interacción con su entorno y de su estructura interna. Ahora, ya sentadas las bases, es posible proceder a realizar la descripción de la estructura y modo de funcionamiento del device driver del Sistema de Adquisición de Datos: ***SAD***.

Estructura y Operación del Driver SAD.

La estructura del driver para el Sistema de Adquisición de Datos ***SAD***, se compone de las siguientes secciones:

- **Area de Directivas.**
- **Device Header.**
- **Area para Almacenamiento de Variables Locales.**
- **Sección de Estrategia.**
- **Sección de Interrupciones.**
    - **Sección para preservación de Registros del CPU.**
    - **Tabla de Selección de Comandos.**
    - **Area de Procedimientos locales.**
        - **Interrupt Handler para Recepción de Datos ADQ_IN**
        - **Outpt**
        - **Putblk**
        - **Getblk**
    - **Sección de Comandos.**
        - **Inicialización**
        - **IOCTL_Input**
        - **IOCTL_output**
    - **Sección para Actualización de Status.**
    - **Sección para Restauración de Registros del CPU.**
    - **Continuación de la Sección de Inicialización.**
    - **Area deshechable.**

Area de Directivas.

En esta área se define al ensamblador la manera como debe convertir en código ejecutable, el código fuente agrupado bajo el archivo SAD1.ASM. En el siguiente listado, se podrá apreciar la definición de algunas directivas importantes utilizadas. Por razones de espacio, sólo las partes más sobresalientes del código del driver serán listadas. La aparición de la siguiente secuencia de puntos . . . . indicará la presencia de instrucciones no listadas.

```
;
;********************************************************
;                    SAD Device Driver
;         Sistema Digital de Adquisición de Datos
;                      JAEC & JAPS
;                  Instituto de Geofísica
;                  28-Feb-92  ver  1.0
;********************************************************
;

;********************************************************
;
;                  Asambler  Directives
;********************************************************
;

;--------------------------------------------------------
Pic_Mask      =       21h           ; PC Pic Mask Reg Address
Pic_Eoi       =       20h           ; PC Pic EOI
Buffer_S      =      14440          ; 14400+32+8 = 16chls * 900 + counter + time

              . . . . . . . . . . . . . . . . . . . . .
.286P

       abs0      segment at 0
                 org        0dh*4
                 adq_int label word
       abs0      ends

       cseg      segment para public 'code'
       adq       proc       far
       assume    cs:cseg,es:cseg,ds:cseg
```

En el listado anterior, se puede apreciar la definición de los segmentos *abs0 y cseg.* El primero se usa durante la configuración del vector de interrupciones que atenderá al sistema de adquisición. La instrucción *org*, indica que la localidad de memoria a la que se hace referencia a través de la etiqueta *adq_int* es la 42 (0dh*4 = 13*4 =42). Con el segmento *cseg*, se define toda el área de memoria ocupada por el driver *sad.*

Definición de Estructuras.

Las estructuras son utilizadas en los device drivers para hacer referencia a localidades de memoria asociadas a los diferentes Request Headers generados por DOS.

Para el caso de este driver, sólo se han definido tres estructuras que corresponden a los tres comandos soportados por el mismo: *Inicialización*, *ioctl_input* é *ioctl_output*. El significado y uso de cada uno de los campos de las mismas, será analizado durante el desarrollo del tema sobre ejecución de comandos del driver. El siguiente listado muestra la forma como han sido declaradas dentro del driver *SAD*.

```
; Structures Definition Area
rh0   struc                             ; request header for initialization (command 0)
rh0_rh        db   size rh dup(?)       ; fixed request header portion
rh0_nunits    db   ?                    ; number of units (device header only)
rh0_brk_ofs   dw   ?                    ; offset address for break
rh0_brk_seg   dw   ?                    ; segment address for break
rh0_bpb_tbo   dw   ?                    ; offset segment of pointer to BPB
rh0_bpb_tbs   dw   ?                    ; segment address of pointer to BPB
rh0_drv_ltr   db   ?                    ; first available drive (Dos 3+)(block only)
rh0   ends

rh3   struc                             ; request header for IOCTL Input (command 3)
rh3_rh        db   size rh dup(?)       ; fixed request header portion
rh3_media     db   ?                    ; media descriptor from DPB
rh3_buf_ofs   dw   ?                    ; offset address of data transfer area
rh3_buf_seg   dw   ?                    ; segment address of data transfer area
rh3_count     dw   ?                    ; transfer count (sector for block)
                                        ; (bytes for caracter)
rh3_start     dw   ?                    ; start sector number (block)
rh3   ends

rh12  struc                             ;request header for IOCTL output (command 12)
rh12_rh       db   size rh dup(?)       fixed request header portion
rh12_media    db   ?                    ;media descriptor from DPB
rh12_buf_ofs  dw   ?                    ;offset address of data transfer area
rh12_buf_seg  dw   ?                    ;segment address of data transfer area
rh12_count    dw   ?                    ;transfer count (sector for block)
                                        ;(bytes for caracter)
rh12_start    dw   ?                    ;start sector number (block)
rh12  ends

;*************************************************
;                   Main Procedure Code
;*************************************************

begin:                                  ;code area.
```

Device Header.

```
;*****************************************************
;              Device Header Required by DOS
;*****************************************************
next_dev     dd    - 1              ; no others devices following
attribute    dw    0c000h           ; character device,IOCTL support
strategy     dw    dev_strategy     ; strategy routine address
interrupt    dw    dev_interrupt    ; interrupt routine address
dev_name     db    'SAD     '       ; console driver's name,8 bytes maxs
```

A través del device Header, el driver proporciona al kernel de DOS información de primera importancia sobre sus características específicas. A fin de que esta información esté disponible en todo momento para DOS independientemente de las peculiaridades del driver, Microsoft especifica que la primera sección de código presente en un driver deber ser ocupada por el Device Header. De esta forma DOS tiene la seguridad de que la información por él requerida, estará siempre contenida en los primeros 18 bytes del driver. Cinco campos componen un Device Header, estudiemos su función:

next_dev:

En el archivo SAD1.SYS, sólo está definido el device driver SAD, por lo que al campo next_dev, se asigna el valor de -1, para indicar a DOS que en el archivo SAD1.SYS sólo existe un driver para instalar.

attribute:

El valor *0c000h* asociado a *attribute* indica a DOS, que el driver es de tipo caracter y que puede manejar comandos de IOCTL. Ver tabla para definición de este campo pág. 48.

strategy:

El valor definido en *strategy* es un *apuntador a la sección de estrategia* de SAD, en este caso contenida bajo la rutina *dev_strategy*. DOS lo inspecciona para encontrar la dirección en memoria donde inicia el código ejecutable del driver.

interrupt:

La rutina de interrupción de SAD, inicia a partir de la localidad de memoria identificada con la etiqueta *dev_interrupt*.

La variable *interrupt* es un apuntador a la etiqueta *dev_interrupt*, por lo tanto, también *señala el inicio de la rutina de interrupción*. Este campo lo usa DOS para localizar el inicio del código de la sección de interrupción.

dev_name:

SAD es el nombre de acceso del device driver para el sistema de adquisición de datos digitales. Nótese que el nombre del driver es independiente del archivo donde se almacena, en este caso SAD1.SYS.

Area para Almacenamiento de Variables Locales.

Es aquí donde se definen todas las variables necesarias para el almacenamiento de datos y control de operación del driver. No existen restricciones en cuanto al espacio que puede ser ocupado por esta sección, pero considerando que en general el área de los 640kb de RAM es muy valiosa en una PC, es absolutamente necesario evitar excesos.

```
;*************************************************
;                    Device Driver Work Space
;*************************************************
offsou      dw    ?      ; offset source
segsou      dw    ?      ; segment source
offdes      dw    ?      ; offset destination
segdes      dw    ?      ; segment destination
rh_ofs      dw    ?      ; offset address of request header
rh_seg      dw    ?      ; segment address of the request header
factor      dw    14440  ; frame array size
rx          dw    ?      ; reception frame
tx          dw    ?      ; transmition frame
crx         dw    ?      ; reception cycle
ctx         dw    ?      ; transmition cycle
```

Las cuatro primeras variables definidas en esta sección, sirven para almacenar las direcciones de offset y segmento de las localidades de memoria involucradas en la transferencia de datos de memoria principal a memoria extendida y viceversa. Estas variables son utilizadas por las rutinas Putblk y Getblk.

La dirección de memoria donde se encuentra el Request Header construído por DOS, es salvada en las variables *rh_ofs y rh_seg* durante la ejecución de la sección de estrategia.

El ***buffer primario*** *del driver ocupa* ***14440*** *bytes*, la variable *factor* es utilizada para calcular la dirección de memoria extendida donde será transferido el mismo.

Rx es un contador que controla en qué buffer de memoria extendida deben ser alojados los datos a ser transferidos desde el *buffer primario.*

Tx es un contador que controla, qué buffer de datos en memoria extendida, deberá ser pasado a la aplicación cuando esta lo requiera.

Los contadores *crx y ctx* trabajan en conjunto con cx y tx. Cuando alguno de éstos últimos alcanzan su valor máximo, crx ó ctx se incrementan en una unidad según el caso.

```
full        dw      ?               ; largest frame available
bmdt        db      30h dup(0)      ; block move descriptor table
control     db      2 dup(0)        ; control speed
frame_l     db      8 dup(7)        ; length frame default 7
frame_c     db      16 dup(0)       ; Frame counter
chl         dw      16 dup(0)       ; Offset of channel
irq         db      ?               ; Interrupt level generated
limite      dw      ?               ; Max buffer size by channel
stack_ptr   dw      ?               ; old stack pointer
stack_seg   dw      ?               ; old stack segment
            dw      200h  dup(?)    ; 256*2 = 512 bytes for new stack
newstack    label   word            ;top of new stack
```

La variable *full* contiene el número máximo de buffers que pueden ser almacenados en memoria extendida.

Bmdt define un área de memoria donde se alojan los valores de los parámetros necesarios para efectuar la transferencia de datos entre memoria real y extendida. Usado en las rutinas Putblk y Getblk.

La variable *control* es usada durante el proceso de inicialización. Cada bit representa dos canales de recepción contiguos del UART. Cuando un bit es encendido indica que dos canales trabajarán a una velocidad de 2100 bps. Cuando es apagado indicará que dos canales asociados a un bloque del UART trabajarán a 1200 bps.

ESTA TESIS NO DEBE
SALIR DE LA BIBLIOTECA

La variable *frame_l* indica el número de bytes de que se compone un paquete completo de datos. En el caso de los canales que se reciben a 2100 bps, cada bloque de datos se compone de 7 bytes, en el caso de canales que funcionan a 1200 bps, el bloque de datos se compone de tres bytes. (ver **apéndice B**).

Con el contador *frame_c* se detecta el arribo de un bloque o trama completa de información a un canal. Para los canales que reciben a 2100 bps el tamaño de la trama es de 7 bytes, para los que reciben a 1200 es de 3 bytes. Con cada byte de *frame_c* , se lleva el conteo del número de elementos de una trama, que han arribado a cada canal. Cuando una trama se completa el valor del contador se iguala a cero.

Chl representa a una tabla que contiene apuntadores a cada área de datos asignada por canal dentro del *buffer primario.*

El buffer primario se divide en 16 áreas, cada una destinada a almacenar información proveniente de un canal.

Irq salva el número de bloque del UART que generó la interrupción. Existen 8 bloques definidos, cada bloque controla dos canales de recepción.

Con *limite* se maneja el número máximo de datos por canal que pueden ser almacenados dentro del buffer primario.

A través de las variables *stack_ptr y stack_seg,* se almacena la dirección de inicio del stack del sistema cuando es substituído por un nuevo stack definido dentro del driver.

La etiqueta *newstack* apunta al inicio de stack definido en el driver. Es necesaria la utilización de un stack alterno para evitar que el stack del sistema operativo se desborde debido a que su capacidad de almacenamiento es reducida. En este caso el stack del driver consta de 1024 bytes.

Los primeros tres arreglos del listado siguiente, deben permanecer en ese orden dentro del driver **SAD**. Estos almacenan la información en un formato especial que será esperado por el programa de aplicación al momento de realizar una lectura sobre el driver.

```
Buffer   db   Max_Buf*16 dup (0)  ; #m * Frame * time *#Channels
Count    dw   16 dup(0)           ; Counter data Rx/channel
Time     db   8 dup(0)            ; Buffer time mark
; the above three variables MUST BE defined in this order.
```

A fin de lograr mayor eficiencia en la transferencia de datos de memoria extendida a memoria principal, la transferencia de los mismos debe realizarse en bloques, por ello se agrupan estos arreglos dentro de áreas contiguas de memoria, para formar un sólo bloque que será trasmitido primero del *buffer primario al sistema circular de buffers y luego del sistema circular de buffers a la aplicación.* La variable **Buffer** define 16 áreas de memoria contiguas, cada área está destinada a almacenar los datos correspondientes a cada canal. **Max_Buf** define el máximo número de bytes por canal que pueden ser almacenados en el buffer primario.

La variable *Count* lleva el conteo del número de bytes por canal recibidos en el buffer primario.

La variable *Time* queda reservada para alojar la marca de tiempo asociada al buffer primario. En la actualidad no se usa.

```
Brinca LABEL WORD
        dw      Block1A     ; Control Address and Operation UART1 Block A
        dw      Block1B     ; Control Address and Operation UART1 Block B
        dw      Block1C     ; Control Address and Operation UART1 Block C
        dw      Block1D     ; Control Address and Operation UART1 Block D
        dw      Block2A     ; Control Address and Operation UART2 Block A
        dw      Block2B     ; Control Address and Operation UART2 Block B
        dw      Block2C     ; Control Address and Operation UART2 Block C
        dw      Block2D     ; Control Address and Operation UART2 Block D
```

Cuando el CPU recibe la interrupción de la tarjeta de adquisición, el interrupt handler correspondiente entra en operación. Este investigará que línea del UART generó la interrupción y pasará el control a una rutina específica para la atención de la misma. La tabla de arriba contiene apuntadores a direcciones de memoria donde se encuentran esas rutinas de atención. Es pertinente recordar que de cada UART salen 4 líneas de interrupción, que corresponden a igual número de bloques. Cada bloque ejerce el control sobre 2 canales de recepción contiguos. De esta forma el apuntador al bloque **Block1A**, hace referencia a la localidad de memoria donde comienza el código de atención para los canales de recepción **0 y 1 del UART A.**

Sección de Estrategia.

```
;*****************************************************
;
;                 the strategy procedure
;*****************************************************
;

dev_strategy:                     ;save the Request Header Address (ES:BX)
          mov cs:rh_seg,es        ;save the segment address
          mov  cs:rh_ofs,bx       ;save the offset address
          ret
```

A través de esta sección, la dirección de offset y de segmento del *Request Header* construído por el kernel, se almacena en las variables *rh_ofs* y *rh_seg*. El control se transfiere a DOS una vez realizado lo anterior.

Sección de Interrupciones.

```
;*****************************************************
;
;            the interrupt procedure
;*****************************************************
;

; dev interrupt - 2nd call from DOS
```

La sección de interrupciones, independientemente del comando a ejecutar, debe efectuar las siguientes tareas dentro de su primera etapa de operación :

1).- **Cambiar el stack del sistema operativo por el stack definido dentro del driver *SAD*.**

2).- **Salvar los valores de los diferentes registros del procesador, que serán usados en esta sección.**

3).- **Recuperar la dirección donde se aloja el Request Header construido por DOS.**

4).- **Obtener el número de comando a ejecutar señalado en el Request Header.**

5).- **Pasar el control del CPU a la rutina de atención específica de ese comando.**

A través de los siguientes listados, será expuesta la forma como se llevó a cabo la codificación de las tareas arriba señaladas.

Cambio de Stack

La etiqueta *dev_interrupt* marca el inicio de la sección de interrupción. DOS localiza la dirección de esta rutina con la ayuda del apuntador *interrupt* definido en el *Device Header.*

```
dev_interrupt:
        cli                         ;interrupts back on for keyboard,etc
        mov     cs:stack_ptr,sp
        mov     cs:stack_seg,ss
        mov     bx,cs
        mov     ss,bx
        lea     sp,cs:newstop
        sti
```

La primera operación que se hace en la sección de interrupciones es substituir el stack de default del sistema por el stack del driver. Esta operación se realiza para evitar que las múltiples operaciones de *push* que se efectúan durante la ejecución de los comandos del driver causen el desbordamiento del stack del sistema.

Preservación de Registros

```
        cld                     ;clears direction flag to process strings l-r
        push    ds
        push    es
        pusha                   ;pushes ax,cx,dx,bx,sp,bp,si,di
        mov     ax,cs           ;initializes ds=cs=cseg
        mov     ds,ax
```

Los valores de todos los registros que se utilizan durante la operación del driver son salvados para preservar el estado del sistema. Si esto no se hiciera así, el sistema se caería.

El valor del registro *DS* se iguala con *CS* para asegurar que las direcciones de memoria que usen *DS* como referencia, utilicen también la dirección de segmento asociada a ***cseg***.

Recuperación del Request Header.

```
        mov     ax,cs:rh_seg        ;restore ES
        mov     es,ax               ;change from ES=CSEG to ES=RH
        mov     bx,cs:rh_ofs        ;restore offset of request header
```

Las tres instrucciones arriba listadas recuperan la dirección donde se almacena el Request Header y la transfieren a los registros ***ES:BX***. Recuérdese que ésta dirección había sido salvada previamente durante la ejecución de la sección de estrategia.

Obtención del Número de Comando.

```
;jump to appropiate routine to process command

        mov   al,es:[bx].rh_cmd      ; get request header command.
        rol   al,1                   ; mult by two
        lea   di,cs:cmdtab           ; function (command) table address
        xor   ah,ah                  ; clear hi order
        add   di,ax                  ; add the index to start of table
        jmp   word ptr[di]           ; jump indirect

;Command table
cmdtab label byte
        dw    initialization         ;  initialization
        dw    media_check            ; media check (block only)
        dw    get_bpb                ; build bpb
        dw    ioctl_input            ; ioctl in
        dw    input                  ; input
        dw    nd_input               ; non destructive input no wait
        dw    input_status           ; input status
        dw    input_flush            ; input flush
        dw    output                 ;  output (write)
        dw    output_verify          ;  output write with verify
        dw    output_status          ; output status
        dw    output_flush           ; output flush
        dw    ioctl_out              ; ioctl output
        dw    open                   ; device open
        dw    close                  ; device close
        dw    removable              ; removable media
        dw    output_busy            ;output til busy
```

A través del código anterior, se accesa el Request Header y se obtiene el comando que desea DOS ejecute el Driver. El valor del comando se almacena en ***AL***. La manera como se obtiene el número de comando a ejecutar, es a través de la parte fija de la estructura del Request Header que fué declarada en la sección de directivas.

Transferencia del Control a la rutina de Ejecución del Comando.

Existe para cada comando definido, una rutina especializada en darle servicio. La técnica para saltar a la rutina de atención adecuada en base al número de comando se muestra en el listado anterior.

Cada elemento de la tabla arriba listada, es un apuntador a una rutina de atención. Cuando DOS pone en su campo de comandos un *4* por ejemplo, solicita que el comando *Input* sea ejecutado. El comando *4* en este caso, está asociado a una rutina de atención denominada **Input** dentro del driver.

Area para definición de Procedimientos Locales.

Aquí, se definen rutinas auxiliares para la ejecución de comandos. A través de esta sección es posible, enmascarar programas para la atención de interrupciones. Para el caso del Sistema *SAD,* se han definido varias rutinas auxiliares junto con el Interrupt Handler encargado de dar atención a las interrupciones generadas por el sistema de adquisición.

El Interrupt Handler del Sistema de Adquisición.

El ***Interrupt Handler*** del sistema de Adquisición, es la rutina encargada de dar atención a las interrupciones generadas por la tarjeta de adquisición.

Para lograr este objetivo, realiza las siguientes operaciones:

1).- Preserva los valores de los registros del procesador.

2).- Requiere al PIC de la tarjeta de adquisición el número de bloque generador de la interrupción.

3).- Determina cuál de los dos canales que controla el bloque generó la interrupción.

5).- Lee el buffer del canal que generó la interrupción (Rutina LEE).

6).- Almacena los datos leídos, en el área correspondiente a ese canal dentro del *buffer primario* (Rutina LEE).

7).- Aloja en el sistema *circular de buffers* el *buffer primario* en caso de que alguna de las 16 áreas de almacenamiento de datos se llene (Rutinas Outpt y Putblk).

8).- Notifica a los PICs de la computadora y de la tarjeta de adquisición que la interrupción ha sido atendida.

9).- Restaura los valores de los registros del CPU.

10).- Regresa el control del CPU a DOS.

Preservación de Registros del CPU

```
;**************************************************
;                    local procedures
;**************************************************

adq_In proc far
        cli
        pusha                   ; save ax,cx,dx,bx,sp,bp,si,di
```

El inicio de la rutina de interrupción se identifica con la etiqueta ***adq_In***. La tabla de interrupciones debe contener la dirección de memoria asociada con ***adq_In***.

Obtención de la línea generadora de la interrupción.

```
        mov     dx,Pic
        mov     al,0Ch          ; Poll command code
        out     dx,al           ; Write OCW3,Poll command to PIC
```

Las instrucciones anteriores se destinan al PIC de la tarjeta de Adquisición. La finalidad de las mismas, es indicarle al PIC que en la siguiente operación de lectura que se efectuará sobre él, deberá entregar en el bus de datos el número de línea generador de la interrupción.

```
        in      al,dx           ; Read Int level I0000LLLb from PIC
        test    al,80h          ; Is it an Int?
        jnz     Cont10          ;Yes
        jmp     regresa         ;No
```

Con la instrucción *in al,dx* se realiza una lectura al PIC, que en respuesta coloca en el bus de datos el número de línea que generó la interrupción.

Cuando ocurre una interrupción, el bit 8 del byte de datos que entrega el PIC debe quedar encendido, de no ser así se considera que la interrupción fué en falso y el control se regresa a DOS.

```
Cont10:
        and     al,07h          ; Clear int flag
        mov     cs:Irq,al       ; Save Int line
```

Si el bit 8 está encendido, se apaga, y el número de interrupción es salvado en la variable ***Irq***.

```
        xor     ah,ah                   ; clear ah
        mov     bp,ax
        mov     bh,cs:frame_l[bp]       ; get channel frame length
        shl     bp,1                    ; bp=(int level or al)*2
        jmp     cs:Brinca[bp]           ; Ejecuta la Subrutina de acuerdo al Bloque
```

Con el número de interrupción se salta a la rutina específica de atención, utilizándose para ello la tabla denominada ***Brinca*** definida en la sección de declaración de variables. Si la interrupción fuese ocasionada por cualquiera de los dos canales pertenecientes al primer bloque del UART 1, el salto se haría hacia la dirección indicada por el primer apuntador de *Brinca*, que es la dirección de memoria asociada a la etiqueta *Block1A.*

Determinación del Canal Generador de la Interrupción.

```
Block1A:
        mov     dx,Uart1 + SR_A         ; get IS_A address
        in      al,dx                   ; Read
        test    al,02h                  ; If ISR[2]=1 ->channel0 RHR is full
        jz      Canal1b                 ; If not then channel0 is no full
```

Las primeras tres instrucciones tienen como finalidad determinar si el primer canal asociado al *bloque 1A* fué el generador de la interrupción.

```
        mov     dx,Uart1 + RHR_a
        mov     di,0            ; Set di = Id channel
        call    Lee             ; Read 3 bytes RHR_A
```

En caso de que el primer canal del bloque haya sido el causante de la interrupción se llama a la rutina ***LEE***, a través de la cuál se hará el vaciado del buffer ***RHR*** del canal de recepción del UART. Los datos leídos son almacenados en el *buffer primario* del driver.

```
Canal1b:
        mov     dx,Uart1 + SR_B ; get IS_B address
        in      al,dx           ; Read
        test    al,02h          ; Is the Second channel full also?
        jnz     Contb           ; Envia EOI al Pic y regresa
        jmp     EOI             ; No the second channel is not full
```

Una vez vaciado el buffer de datos del primer canal del bloque, se aprovecha para verificar si el segundo canal que controla ese bloque también tiene su buffer lleno. De no ser así, el flujo del programa se dirige al área de salida del interrupt handler.

```
Contb:
        mov     dx,Uart1 + RHR_b        ; set ISR address
        mov     di,1                    ; Set di = Id channel
        call    Lee                     ; Read 3 bytes RHR_B
```

A esta sección puede llegarse de dos modos: uno indirecto, cuando inmediatamente después del vaciado del buffer del primer canal se detecta que el buffer del segundo canal del bloque también está lleno. El otro modo se da, cuando sólo el buffer del segundo canal está lleno. En ambos casos se procede al vaciado de datos del buffer del segundo canal con la rutina ***LEE***.

```
        jmp     EOI
```

Una vez realizado el vaciado del buffer, el control se pasa al área de salida del Interrupt Handler.

```
Block1B: ...  jmp  EOI
Block1C: ...  jmp  EOI
Block1D: ...  jmp  EOI
Block2A: ...  jmp  EOI
Block2B: ...  jmp  EOI
Block2D: ...  jmp  EOI
```

Operaciones similares a las realizadas sobre el *Block1A*, se efectúan sobre los otros 7 bloques del sistema:

a).- Determinación de qué canal del bloque generó la interrupción
b).- Vaciado del buffer *RHR_x* en el *buffer primario* del driver.

Notificación de que la interrupción ha sido atendida.

```
EOI:
        mov     dx,Pic                  ; get PIC address
        mov     al,11100000b            ; OCW2 Rotate Command specific EOI
        or      al,cs:Irq               ; add interrupt line
        out     dx,al                   ; data is sent to PIC
```

El listado anterior corresponde a las instrucciones necesarias para notificar al PIC de la tarjeta de Adquisición que la interrupción que generó ya ha sido atendida.

```
regresa:
        mov     al,20h                  ; Non specific EOI command
        out     Pic_Eoi,al              ; to 82C59A PC internal
```

Las dos instrucciones anteriores notifican al PIC maestro de la computadora que la interrupción por él generada ya ha sido atendida.

Restauración de Valores de los Registros del Sistema y Regreso del Control del Procesador a DOS.

```
        popa
        sti
        iret
```

Se reestablecen los valores que tenían los diferentes registros del sistema antes de entrar en operación el interrupt handler, se habilitan de nueva cuenta las interrupciones sobre el procesador y el control se transfiere del interrupt handler a DOS. La atención de la interrupción ha finalizado.

El listado siguiente muestra las partes más sobresalientes del código de la rutina *lee*.

```
;       Procedure LEE
;       Call with
;                       DI= Channel Number
;                       BH= Frame Length
;                       DX= RHR
;--------------------------------------
Lee     proc    near
        mov     cx,3                    ; I'm going to read the RHR 3 times
        mov     si,di                   ; si = di = channel ID
        shl     si,1                    ; si = 2(ID channel or di)
        mov     bp,cs:count[si]         ; get number of bytes received
        add     bp,cs:chl[si]           ; get channel offset = base offset+ bytes Rx
                                        ; bp = pointer to location to be writed
        mov     bl,cs:frame_c[di]       ; get frame counter flag
Read:
        in      al,dx                   ;Read UART RHR
        .
        .
        mov     cs:Buffer[di],al
        .
        .
        call    outpt
        .
        .
no_full: ret
lee     endp
```

El objetivo de la rutina *Lee* es vaciar el buffer de uno de los 16 canales de recepción y colocar los datos en el *buffer primario* del driver. Cada buffer de recepción del UART se llena con tres bytes de datos. De manera que esta rutina deberá hacer tres lecturas al UART cada vez que se invoca.

Lectura del Canal Generador de la Interrupción.

La lectura al UART se hace mediante la instrucción ***IN AL,DX.*** (ver listado anterior). En el registro *AL* se almacenará el dato proveniente del buffer del UART, en el Registro *DX* se indica la dirección del buffer donde se encuentra el dato a ser leído. Existen 16 direcciones posibles que puede tomar ***DX*** en esta rutina, cada una corresponde a uno de los 16 buffers de recepción de datos de los 2 UARTs del Sistema de Adquisición.

Almacenamiento de datos en el *buffer primario.*

Con la instrucción **MOV CS:Buffer[di],AL** ... el dato proveniente del buffer del UART que está en **AL**, es guardado en el *buffer primario* del driver. El registro de índice **DI**, selecciona el área dentro del *buffer primario* donde se alojará el dato. El *buffer primario* está subdividido en *16 áreas cada una destinada al almacenamiento de los datos provenientes de un canal.*

Cuando alguna de las 16 áreas del *buffer primario* se llena con datos provenientes de un canal, se genera una llamada a la rutina ***output***, la cuál *se encargará de calcular la dirección en memoria extendida donde los datos del buffer primario serán alojados.*

```
outpt proc    near
      .                 ;looking for one buffer available in extended
      .                 ;memory to save data
      call    putblk    ;data transfer from principal memory to extended
      .
      .
      ret
outpt endp
```

Dentro de la rutina output se realizan operaciones con las variables de control del *sistema circular de buffers,* a fin de determinar que dirección de memoria extendida está disponible para el almacenamiento de los datos del *buffer primario.*

Una vez determinada la dirección de memoria extendida, se llama a la rutina ***putblk*** la cuál se encarga del traslado de datos de memoria principal a memoria extendida.

```
;**************************************************************
;               Putblk
;       transfer block from principal  to extended memory.
;       call with
;               DX:AX =         destination linear 32 bit extended memory address
;               SEGSOU:OFFSOU = segment and offset source address
;               CX =            length in bytes
;               ES:SI =         block move descriptor table
;       returns
;               AH = 0 if transfer OK
;**************************************************************
putblk  proc    near
        cli                     ; interrupts back on for keyboard,etc
        mov     cs:stack_ptr,sp
        mov     cs:stack_seg,ss
        mov     bx,cs
        mov     ss,bx
        lea     sp,cs:newstack
        sti
```

La primera parte de putblk empieza con la deshabilitación de todas las interrupciones del CPU para facilitar el cambio de stack.

A fin de evitar el desbordamiento del stack del sistema, la dirección de éste es salvada en las variables ***STACK_PTR*** y ***STACK_SEG***, y su función substituída por otro stack presente en el área de memoria definida bajo la etiqueta newstack. El nuevo stack puede soportar hasta 1024 operaciones de push. Al finalizar, el pin de interrupciones mascarable del CPU es habilitado.

```
        .
        .
        mov     ah,87h          ; int 15h function 87h = block move
        int     15h             ; transfer the control to ROM BIOS
        . . .
        cli
        mov     ss,cs:stack_seg
        mov     sp,cs:stack_ptr
        sti
        ret                      ; back to caller
putblk endp
```

A través de la' interrupción ***15h función 87h,*** se efectúa la transferencia de datos de memoria principal (0-640 Kb) a memoria extendida (>> 1Mb).

En la sección final de la rutina putblk se restaura el stack original del sistema operativo. El control retorna a la rutina outpt.

```
;               Getblk
;       transfer block from extended memory
;       to principal memory.
;   call with
;       DX:AX =                 source linear 32 bit extended memory address
;       SEGDES:OFFDES =         segment and offset destination address
;       CX=                     length in bytes
;       ES:SI=                  block move descriptor table
;   return
;       AH=                     0 if transfer OK
;------------------------------------------------------------------------

getblk proc     near
       mov      cx,Buffer_S     ; bytes to move
       push     ds
       pop      es              ; es=ds
       lea      si,cs:bmdt      ; es:si
       .
       .
```

El procedimiento *getblk* funciona en forma muy similar a *putblk*, su objetivo es realizar la transferencia de la información de uno de los *buffers del sistema circular* en memoria extendida, a un buffer en memoria principal, que es definido dentro del área de datos de la aplicación.

Cuando la aplicación pide datos, el driver verifica a través de sus variables de control si existe al menos un buffer en memoria extendida con datos disponibles para ser leídos. Si es así, el driver llamará a esta rutina, la cuál pasará los datos directamente de memoria extendida al buffer del programa de aplicación en memoria principal.

La constante *Buffer_S*, contiene el número total de bytes a transferir. En este caso son *14440* bytes por bloque.

La etiqueta *bmdt*, representa una tabla donde se almacenan los parámetros que necesita la interrupción 15h para hacer la transferencia de datos. Esta tabla está compuesta por seis descriptores de 8 bytes que son utilizados por el CPU al trabajar en modo protegido. Cuatro descriptores situados en los offsets 00h-0fh y 20h-2fh son llenados por el ROM BIOS antes que el CPU realice la conmutación de modos. Los valores asociados a estos descriptores deben ser igualados a cero. La manera de utilizar los campos restantes de *bmdt* se muestra a continuación:

```
;       store length into descriptors
        mov     es:[si+10h],cx
        mov     es:[si+18h],cx

;       store access rigths bytes
        mov     byte ptr es:[si+15h],93h
        mov     byte ptr es:[si+1dh],93h

;       store source extended memory address
        mov     es:[si+12h],ax
        mov     es:[si+14h],dl
```

El registro **ES** señala el segmento donde se aloja la tabla ***bmdt***, que es el segmento definido bajo **CSEG. SI** contiene el offset donde inicia la etiqueta **bmdt**. A través del primer bloque de instrucciones se indica el número de bytes a ser transferidos.

En el segundo bloque, se define el valor del byte de derechos de acceso. A través de él se señala que el bloque de memoria extendida involucrado en la transferencia, se utilizará para almacenamiento de datos, con permisos de lectura y escritura.

La dirección fuente y destino que se usa durante la transferencia de datos en modo protegido, se especifica en forma lineal. Para este caso, el cálculo de la dirección lineal donde se encuentran los datos a ser transferidos se realiza en otra rutina, pero el resultado de esa operación queda almacenado en los registros ***dl:ax***. Vgr. 1 MB, en su representación lineal se define asi: ***01:0000 hex.***

```
;       convert destination segment and offset to linear address
        mov     ax,cs:segdes        ; segment * 16
        mov     dx,16
        mul     dx                  ; Product in DX:AX
        add     ax,cs:offdes        ; + offset = linear address
        adc     dx,0                ; if carry is set in (AX+OFFDES) then DX = DX + 1
```

Con las instrucciones del bloque anterior, se convierte la localidad de memoria de destino a su representación lineal. El resultado de la operación queda en los registros ***dl:ax***.

```
;       store destination addrèss
        mov     es:[si+1ah],ax
        mov     es:[si+1ch],dl
```

La dirección lineal del buffer de destino se guarda en dos campos de la tabla.

```
;       convert length to words
        shr     cx,1
;       int 15h function 87h = block move
        mov     ah,87h
        int     15h             ; transfer to ROM BIOS
        ret
getblk  endp
```

Ya definida la tabla con las direcciones fuente y destino, y con el número de bytes a transferir se invoca la interrupción *15h función 87*, del ROM-BIOS que permite la operación momentánea de la máquina en modo protegido y la transferencia de datos de memoria extendida a memoria principal.

La tranferencia de datos de memoria extendida a la aplicación ha finalizado. Para el caso de la rutina ***putblk***, se utilizan los mismos valores para el llenado de la tabla *bmdt* que los usados en la rutina ***getblk***.

Sección de Comandos.

Dentro de esta sección, y mediante la implementación de tres comandos, se llevan a cabo las operaciones de transferencia de datos y de control sobre la tarjeta de interface ***SAD***. La operación de tales comandos será descrita a continuación.

Comando 0:
Inicialization:

El comando 0 del driver ***SAD*** realiza las siguientes tareas durante su ejecución:

1).- Obtiene los parámetros de configuración del sistema definidos en el archivo config.sys.

2).- Configura el PIC de la tarjeta de adquisición.

3).- Configura la velocidad de recepción por canal, de cada UART de la tarjeta de adquisición.

4).- Entrega a DOS la dirección de finalización del driver (break address).

```
;*************************************************************
;                    dos command processing
;*************************************************************

;command 0  Initialization
inicialization:
        call    initial                     ; display installation message
                                            ; initialize PIC & UARTs
        mov     bx,cs:rh_ofs                ; restore bx to their original add
        mov     ax,cs:rh_seg
        mov     es,ax                       ; restore cs to their original add
        lea     ax,cs:initial               ; set break address at initial
        mov     es:[bx].rh0_brk_ofs,ax      ; store offset address
        mov     es:[bx].rh0_brk_seg,cs      ; store segment address
        jmp     done                        ; set done status and exit
```

El inicio del presente comando, está identificado por la etiqueta *inicialization*, a continuación de esa etiqueta hay una llamada a un procedimiento de nombre ***Initial***. Bajo este procedimiento están definidas las instrucciones necesarias para la configuración de los 2 UARTs y el PIC de la tarjeta de adquisición. En esa rutina se programa la velocidad de recepción de cada canal del UART y las condiciones bajo las cuáles generarán una interrupción. En el PIC se programan la forma de detección de la señal de interrupción proveniente del UART (si es de nivel o de flanco), y el modo de operación bajo el cuál trabajará el PIC para interactuar con el CPU.

Debido a que sólo es necesario se ejecute una vez el proceso de Inicialización, no es conveniente dejar el código asociado al mismo residente en memoria principal, pues es extenso. La recomendación es deshecharlo una vez utilizado para liberar así, la porción de memoria por él ocupada que de otra forma permanecería ociosa.

Durante la ejecución del procedimiento *initial* los valores de los registros *ES:BX* son alterados, por lo que no contienen ya la dirección donde se encuentra el *Request Header*. Las tres instrucciones de *mov* que siguen a la llamada a *initial* reestablecen los valores de los registros *ES:BX* para que apunten de nueva cuenta a la dirección donde se localiza el *Request Header.*

La instruccion *lea ax,cs:initial* ... tiene por objetivo, obtener el número de bytes a partir del inicio del segmento ***cseg***, que deben ser reservados por DOS para el driver ***SAD*** en memoria principal. Nótese que el offset que se obtiene, contempla todo el cuerpo del driver hasta el inicio del procedimiento *initial*. Así, el procedimiento *initial* no es

incluído en el conteo del número de bytes que son reservados para la operación del driver, por lo que el área donde se encuentra será considerada disponible por DOS y por lo tanto ocupada tarde o temprano por cualquier programa que necesite ejecutarse.

A través de las últimas tres instrucciones del listado anterior, se pasa al ***Request Header*** la información referente ***a donde comienza y donde termina el driver (El break address).*** El comienzo se define mediante el registro CS, que tiene la dirección asociada al segmento donde inicia el driver **cseg**. El final del driver se representa por el offset obtenido de la etiqueta *initial*, la cuál a su vez marca el comienzo del área deshechable del driver.

El salto a la etiqueta *done* se usa, para actualizar el registro de *status* del Request Header. Ya que la sección de status es común a todos los comandos implementados, se localiza en el área de salida del driver.

Continuación del Comando 0: Area deshechable.

```
;Continuación del comando 0 del driver.
;Area Deshechable.
;dos will overwrite this code after the initialization

initial   proc    near            ;display menssage on console
```

La rutina ***Initial*** es llamada durante la ejecución del comando ***0 de Inicialización*** del driver. Todo el código aquí definido será ejecutado una sóla vez dentro del ciclo de operación del driver ***SAD***.

En esta sección se efectúa todo el proceso de configuración del Sistema de Adquisición, que abarca desde la definición de valores iniciales para las variables de control de ***SAD***, hasta la configuración del PIC y los dos UARTs de la tarjeta de interface.

Asignación de Valores Iniciales a las Variables de Control.

A través de las instrucciones del siguiente listado, se asignan valores iniciales de las variables de control de ***SAD***.

```
;dos will overwrite this code after the initialization

initial proc    near                    ; display message on console
        mov     cs:rx,0
        mov     cs:tx,0
        mov     cs:crx,0
        mov     cs:ctx,0
        mov     cs:full,25
        mov     cs:limite,898           ; At least 1 chl of 3 Components
```

Configuración del Vector de Interrupciones.

```
        assume  es:abs0
        xor     ax,ax
        mov     es,ax
        lea     bx,cs:adq_in
        cli
        mov     es:adq_int,bx           ; set offset to int vector table
        mov     es:adq_int+2,cs         ; set code segment in vector address
        sti
```

El bloque de instrucciones que inicia con la declaración *assume*, tiene por objetivo configurar un vector de interrupciones para que apunte hacia la dirección de memoria donde se localiza el Interrupt Handler del sistema ***SAD***.

El Interrupt Handler está definido dentro del área de procedimientos locales del driver y es identificado bajo la etiqueta ***adq_In***.

Con la instrución *lea* se obtiene el offset del *Interrupt Handler* relativo al segmento *CSEG* donde se aloja el driver.

Con el bloque de instrucciones definido entre *cli y sti* se realiza la configuración del vector de interrupción de ***SAD***. La deshabilitación de las interrupciones antes de realizar la configuración, obedece a la necesidad de prevenir cualquier cambio sobre los valores de los registros que contienen la dirección de la rutina *adq_in*.

La etiqueta ***adq_Int***, (no confundir con adq_in) está asociada al vector de interrupciones ***0dh***. El offset del Interrupt Handler es pasado al vector a través del registro **BX**. La dirección del segmento donde se encuentra ***adq_In*** es la misma donde apunta el registro ***CS***, es decir la dirección asociada a la etiqueta ***CSEG***.

Los valores de ***CS:BX*** se almacenan en la dirección del vector ***0dh.***

Cuando se active la interrupción 0dh, DOS irá a la tabla de vectores de interrupción y obtendrá la dirección donde se encuentra el interrupt handler ***adq_In*** asociado al vector ***0dh.***

Lectura de Parámetros para Velocidad de Recepción.

```
        .
        .
        call    config
        lea     dx,cs:msg1          ; message special config
        jnz     default
        lea     dx,cs:msgd          ; message default config
default:
        mov     ah,9h               ; display
        int     21h                 ; dos call
```

La finalidad de la rutina *config*, es leer los parámetros de configuración del driver definidos en el archivo *CONFIG.SYS*. Es a través de esos parámetros como el usuario le indica al driver la velocidad de recepción con que deberá trabajar cada canal.

Mediante el anterior bloque de instrucciones, se manda el mensaje de identificación del driver ***SAD*** que aparece durante su instalación. Cuando los parámetros del archivo CONFIG.SYS son mal definidos, aparecerá el mensaje contenido a partir de la etiqueta *msgd*, indicándole al usuario que el driver asume que todos los canales de recepción serán configurados a la velocidad de default, que en este caso es 2100 bps. Cuando los parámetros han sido bien definidos en CONFIG.SYS el mensaje contenido bajo la etiqueta *msg1* será desplegado. El punto interesante de esto, es la utilización del servicio de ***DOS 09h***, para despliegue de información en pantalla.

```
        mov     al,cs:control
        test    al,0fh
        jnz     three
        mov     cs:limite,398       ; If all the chls are 1C
                                    ; I set limit to catch only 5 sec
```

El listado anterior tiene relación con el número máximo de bytes que pueden ser almacenados por cada canal dentro del *Buffer Primario.*

La pretención aquí es identificar si todos los canales de recepción definidos por usuario trabajarán a 1200 bps. En caso de que esto suceda, el número máximo de bytes a recibir por cada canal dentro del *Buffer Primario*, es ajustado a *398.* Con un sólo canal que trabaje a 2100 bps, obligará a que el número máximo de bytes a almacenar en el *buffer primario por canal* se ajuste a 898. Estos valores no son arbitrarios, obedecen a la necesidad de que el *buffer primario* tenga capacidad para almacenar *5 segundos de información por canal.* Cuando todos los canales trabajan a 1200 bps, en *5 segundos* llegarán sobre *398 bytes* con información. Cuando al menos 1 canal trabaja a *2100 bps*, arriban en 5 segundos sobre 898 bytes con información. A fin de simplificar las operaciones de almacenamiento, cuando la tarjeta recibe información a estas 2 velocidades, se define el tamaño máximo de almacenamiento en atención al máximo número de bytes que pueden recibirse en un lapso definido de tiempo por canal.

```
three:
        call    Init_Pic
        call    Init_Uart
        .

        .
        ret                         ;return to caller
initial endp
```

Mediante las llamadas a las rutinas *Init_PIC e Init_Uart*, el PIC y los 2 UARTs de la tarjeta de Adquisición son configurados y dejados listos para iniciar su operación.

Con el PIC y los dos UARTs configurados y habilitados para la recepción, la rutina ***Initial*** finaliza. El control del CPU es devuelto a la sección residente del ***comando 0*** que originó la llamada. En este momento toda el área de memoria ocupada por ***Initial*** podrá ser recuperada por el kernel de DOS.

```
token   proc    near
        lodsb                       ;get new char in al
        cmp     al,CR
        je      endt
        cmp     al,LF
        je      endt
endt:   ret
token   endp
```

El procedimiento *token,* pertenece al área deshechable del driver, definida por *initial. token* es llamado por la rutina *config.* Su

única función es detectar el fin de la línea donde el driver *SAD* es definido dentro del archivo CONFIG.SYS. El final de línea es detectado con la aparición de los caracteres *carry return y line feed.*

```
;*****************************************************
;
;                  Config
;
;   Returns:       Zero flag is set if default parameter will be use
;                  Zero Flag is not set, we will to use a special config
;*****************************************************
;
```

Desde la rutina *INITIAL* se llama al procedimiento *config.* Este tiene por objeto, leer los parámetros de configuración del driver que define el usuario en el archivo CONFIG.SYS.

La manera de definir las velocidades de recepción de cada uno de los canales de la tarjeta de adquisición se realiza así:

DEVICE = [PATH:] SAD1.SYS /S:UU

La palabra **DEVICE** en el archivo **CONFIG.SYS**, indica a DOS la existencia de un device driver que debe instalarse en memoria.

El parámeto **PATH** que es opcional, le indica al kernel la localización del archivo donde se encuentra el driver.

SAD1.SYS es el nombre del archivo que contiene al driver *SAD.*

A continuación aparece la llave /S:, que simplemente fué introducida para dar un toque de elegancia a la definición de las velocidades de recepción. Estos tres caracteres deben aparecer en la definición, de lo contrario el driver asumirá que la velocidad de recepción de todos los canales es 2100 bps.

Las letras **UU**, tienen que ver con los dos UARTs encargados de realizar la recepción de datos en la tarjeta de Adquisición. Cada UART puede recibir 8 canales de datos simultáneamente. Los canales de datos para su control son agrupados en *Bloques*. Un *Bloque* controla dos canales de un UART. Cuando se definen las velocidades de recepción, los dos canales que controla cada *Bloque* deben ser configurados a la misma velocidad (esto es una restricción de diseño del UART). Así son 8 canales por cada UART y 4 Bloques que los controlan.

El mecanismo para fijar la velocidad de recepción de cada *Bloque* se realiza de la siguiente manera:

Si se coloca un 1 en el arreglo de control del driver, el bloque correspondiente junto con los dos canales que controla, será configurado para trabajar a una velocidad de 2100 bps.

Si un 0 es colocado en el arreglo de control, el bloque asociado junto con los dos canales que controla será configurado para recibir a 1200 bps. Veamos un ejemplo:

UU =	UART2	+	UART1	
UU =	BloqueU2[4,3,2,1]	+	BloqueU1[4,3,2,1]	
UU =	{1011}	+	{1110}	
UU =	B	+	E	(en hexadecimal)
UU =	BE			

DEVICE = C:\SAD1.SYS S:/BE

Con el parámetro **BE** el usuario indica al driver, que los Bloques **4, 2 y 1 del UART 2** y los Bloques **4, 3 y 2 del UART 1** son configurados para recibir los datos a **2100 bps.**

El bloque **3 del UART 2** y el **Bloque 1 del UART 1** serán configurados para recibir datos a **1200 bps.**

La rutina CONFIG abajo listada, tiene por objeto leer esa información del archivo CONFIG.SYS, y guardarla en una variable llamada ***CONTROL***, que se usa en el momento de configurar la velocidad de recepción de los UARTs de la tarjeta de adquisición.

Durante el proceso de carga del sistema operativo, el programa **SYSINIT** copia a memoria todo el contenido del archivo CONFIG.SYS. Cuando DOS construye el Request Header para el comando de Inicialización, en los campos **rh0_bpb_tbo y rh0_bpb_tbs** señala la dirección de memoria donde se localiza la copia del archivo CONFIG.SYS. Esa dirección corresponde específicamente al área donde comienza la definición del driver (después de DEVICE =) sobre el cuál se ejecuta el comando de Inicialización.

Por lo tanto, *SAD* necesita obtener la dirección definida en los campos de **rh0_bpb_tbo y rh0_bpb_tbs** del Request Header, para poder inspeccionar los parámetros de instalación indicados por el usuario.

```
config  proc    near
        .
        mov     si,es:[bx].rh0_bpb_tbo      ; point after DEVICE =
        mov     ax,es:[bx].rh0_bpb_tbs      ; in CONFIG.SYS
        mov     ds,ax                       ; set DS value
        xor     al,al
ring:   call    token                       ; get token and check for CR or LF
        jz      exitc                       ; premature eoln
        cmp     al,'/'
        jne     ring
        call    token
        .
        mov     cs:control[di],al
        ret
config  endp
```

Configuración del PIC 82C59A-2.

El listado siguiente, muestra la manera como es programado el PIC de la tarjeta de adquisición. El PIC detectará las señales de interrupción provenientes de los dos UARTs, hasta que las mismas se estabilicen en un nivel de voltaje alto. No se usa la detección por flanco de la señal, para evitar problemas similares a los que ocurren en las computadoras personales, por el uso de esa forma de operación.

En este proceso de configuración, se define la forma como el PIC manejará la asignación de prioridades entre sus 8 líneas de entrada. La estrategia de operación programada es por rotación de prioridades, y consiste básicamente en la asignación del nivel de prioridad más bajo a la última línea de interrupción atendida.

```
Init_Pic  proc   near
          mov    dx,Pic            ; External Pic address
          mov    al,00011010b      ; ICW1 level triggerd, single & ICW4 no needed
          out    dx,al
          mov    al,00h   ; ICW2
          inc    dx
          out    dx,al
          mov    al,00000000b      ; OCW1 Reset Interrupt Mask Register
          out    dx,al
          dec    dx
          mov    al,11100000b      ; OCW2 Rotate on specific EOI IRQ0 = 0
          out    dx,al
          mov    al,00001100b      ; OCW3 Comando de Sondeo
          out    dx,al
          ret
Init_Pic  endp
```

Inicialización de los UART's.

El proceso de inicialización de los UART's del Sistema de Adquisición tiene dos objetivos primordiales:

1).- Realizar la programación de las velocidades de recepción a las que operará cada canal, de acuerdo a los parámetros de control definidos por el usuario.

2).- Programar las condiciones de operación que originarán que una interrrupción sea generada por el UART.

```
Init_Uart       proc      near
                .
                .
                ret
Init_Uart       endp
```

Una descripción detallada de como es llevada a cabo la programación de los mismos será dada posteriormente.

Area para Almacenamiento de Mensajes.

```
     .
msg1 db     07h,07h
     db     '    DATA ADQUISITION DRIVER',0dh,0ah
     db     '        Sismología UNAM',0dh,0ah
     db     '          Version  1.0',0dh,0ah
     db     '    Extended Memory Managment',0dh,0ah
     db     '             28/Feb/92',0dh,0ah
     db     ' Reception Speed Setup for 1 or 3 Component',0dh,0ah
     db       0dh,0ah,07h,07h,'$'

adq  endp                  ;end of console procedure
cseg ends                  ;end of cseg segment
     end    begin          ;end of program
```

En esta área se definen los textos de despliegue utilizados durante el proceso de inicialización del driver. La presente sección marca el fin del *área deshechable* del driver.

Comando 3: IOCTL_input:

El comando **IOCTL_Input** del driver ***SAD***, está diseñado para efectuar dos diferentes clases de operaciones, dependiendo del valor (1 ó 0) de un parámetro que la aplicación entrega a DOS:

1).- La tarea más importante para el comando IOCTL_input, consiste en realizar la transferencia de los datos recibidos por el Sistema de Adquisición al programa de aplicación.

2).- La tarea secundaria que puede efectuar consiste en proporcionar al programa de aplicación, información sobre los valores asociados a las variables de control del driver.

Las operaciones a realizar durante la ejecución de éste comando se detallan a continuación:

1).- Se inspecciona la variable *rh3_count* del request header, donde se almacena el valor del parámetro pasado por la aplicación, para determinar la clase de operación a realizar.

En caso que se trate de una operación de transferencia de datos de memoria extendida a memoria principal (par=1) ocurre lo siguiente:

2).- Se obtiene la dirección de destino en memoria principal.

3).- Se determina el número de Buffer del Sistema Circular, que va a ser transferido.

4).- Se calcula la dirección del Buffer a transferir de memoria extendida.

5).- Se realiza la transferencia de datos entre memoria extendida y memoria principal.

6).- Se actualiza del campo de status del Request Header.

7).- Se regresa el control del CPU a DOS.

En caso de que la aplicación requiera los valores de las variables de control del driver (par=0) las operaciones a realizar son:

2).- Se transfieren los valores de las variables de control al buffer de la aplicación.

3).- Se Actualiza el registro de status.

4).- Se regresa el control de la máquina a DOS.

```
;command 3
ioctl_input:
        cmp     es:[bx].rh3_count,1         ; Is an IOCTL status command?
        je      cmd_rx
        jmp     status
```

Lo primero que se hace en *ioctl_input*, es verificar el tipo de operación que desea la aplicación se ejecute. Si la aplicación necesita obtener datos del Sistema Circular de Buffers colocará en el registro CX un 1. La instrucción *cmp es:[bx].rh3_count,1* efectúa esa comprobación. En caso de que en el campo rh3*_count* del Request Header exista un 1, el driver realizará todas las operaciones necesarias para entregar a la aplicación un bloque completo de datos. (Ver interface rext).

Transferencia de Datos de Memoria Extendida a Memoria Principal.

```
cmd_rx:
        mov     ax,es:[bx].rh3_buf_seg   ; load segment destination address
        mov     cs:segdes,ax
        mov     ax,es:[bx].rh3_buf_ofs   ; load offset destination address
        mov     cs:offdes,ax
        mov     ax,cs:tx                 ; get buffer number to be read
        .                                ; Are there buffers with data?
        .                                ; if yes, then goto tx_ok
        .                                ; if no, then goto txbusy
txbusy: mov     es:[bx].rh3_count,0      ; there are not buffers availables
        .
        jmp     busy
tx_ok:  .
        mul     cs:factor                ; calculate linear address ax*factor
        add     dx,10h                   ; address=dx:ax
        call    getblk                   ;
        .
        jmp     done                     ; set done bit and exit
```

La dirección de destino en memoria principal, a donde serán transferidos los datos provenientes de un buffer en memoria extendida, se pasa de la aplicación al kernel a través de los registros **DS:DX**. DOS a su vez, coloca esa dirección en los campos **rh3_buf_seg y rh3_buf_ofs** del **Request Header**, donde son leídos por el driver mediante las instrucciones arriba listadas.

A través de la línea punteada, se representa todo el código de control, que debe ser ejecutado por el driver, para determinar si en el *Sistema Circular de Buffers*, existe un buffer con datos listos para ser transferidos. En caso de no existir datos disponibles para la aplicación, el driver notificará a DOS esa situación, quién a su vez indicará a la aplicación que la transferencia no pudo realizarse por no existir datos disponibles.

Para indicar a DOS que ningún byte pudo ser entregado del driver a la aplicación, en el campo de número de bytes a transmitir del Request Header se pone un cero. Un salto al área de status para encender el bit de busy es efectuado.

Si existe un buffer con datos disponibles en memoria extendida, se realiza el cálculo de la localidad de memoria donde inicia el mismo. Ya obtenidas las direcciones de memoria del buffer de destino y del buffer fuente, la transferencia puede llevarse a cabo, la rutina *getblk* se encargará de ello. Por fin, los datos recibidos a través de la tarjeta de adquisición se entregan ya organizados y listos para su procesamiento al programa de aplicación.

El control pasa al área de salida del driver, mediante el salto a la etiqueta *done*, donde se actualizarán los bits de status del Request Header, para indicarle a DOS que la operación fué completada con éxito.

Transferencia de Valores de las Variables de Control de SAD.

Si el valor asociado al campo rh3_Count del Request Header no es 1, esto indicará al driver que la aplicación le solicita información sobre los valores de las variables de control que usa para el manejo del Sistema Circular de Buffers. La variables de control del Sistema Circular de Buffers, permiten determinar si el driver está funcionando correctamente. El acceso a ésta información puede llegar ser importante durante el proceso de desarrollo de aplicaciones. Las variables de control para este sistema son: rx, tx, ctx, crx, y su función fué descrita en la sección de almacenamiento de variables locales.

Así, si el valor de rh3_Count es diferente de 1, los valores de las variables de control del driver serán pasados a la aplicación. De esto último trata el siguiente listado.

```
status:
        mov     di,es:[bx].rh3_buf_ofs          ; get buffer offset
        mov     ax,es:[bx].rh3_buf_seg          ; get buffer segment
        mov     es,ax                           ; change ES to RH segment
        cli

        mov     ax,cs:rx                        ; reception frame
        mov     es:[di],ax
        add     di,2
        mov     ax,cs:tx                        ; transmition frame
        mov     es:[di],ax
        add     di,2
        .
        .
        .
        sti
        jmp     done                            ; set done bit and exit
```

La dirección del buffer donde serán colocados los datos de control del driver es obtenida del Request Header. Las interrupciones sobre el CPU se inhabilitan a fin de que las variables de control del driver no sean alteradas a consecuencia de una interrupción en el momento de ejecutarse este comando. (Ver interface status).

Los valores de las variables de control ***CX, RX, CTX, CRX*** y otras más son alojadas dentro del buffer de destino, cuya dirección de inicio queda representada por los registros **ES:DI**. Una vez transferidos los valores de los registros de control, se rehabilitan las interrupciones y es efectuado un salto al área de salida del driver para actualización del campo de status en el Request Header.

Comando 4
Input:

```
;command 4
Input:
        jmp     done                    ; set done bit and exit
```

Bajo esta sección, de acuerdo a la definición de comandos del driver, se debió colocar todo el código necesario para efectuar la transferencia de datos del driver al programa de aplicación, y en un principio así se hizo, pero existieron problemas de operación entre el *driver, DOS y la aplicación*, que se resolvieron colocando las rutinas de transferencia bajo el comando *ioctl_input*.

Comando 12
IOCTL_output:

```
;command 12 ioctl_out
ioctl_out:
        mov     di,es:[bx].rh12_buf_ofs         ; get buffer offset
        mov     ax,es:[bx].rh12_buf_seg         ; get buffer segment
        mov     es,ax                           ; change ES to RH segment
        mov     ax,es:[di]
        cmp     ax,1110h
        je      sad_on
        cmp     ax,1111h
        je      sad_off
change:                                         ; change control parameters
        mov     ax,es:[di]
        mov     cs:rx,ax                        ; reception frame
        add     di,2
        mov     ax,es:[di]

        jmp     done
sad_on:                                         ; unmask INTR 80X86
        in      al,21h                          ; read 82C59A-2 IMR
        and     al,0dch                         ; unmask IR4,IR1,IR0 (video,kbd)
        out     21h,al
        int     0dh
        jmp     done
sad_off:                                        ; mask INTR 80X86
        in      al,21h                          ; read 8259A-2 IMR
        or      al,20h                          ; mask IR5
        out     21h,al


        jmp     done
```

1).- sad_on:
Si el primer campo del buffer de datos que manda la aplicación, contiene el valor 1110h, se habilita la detección de las interrupciones que genera la tarjeta de adquisición sobre el CPU. El sistema de adquisición comienza a trabajar.

2).- sad_off:
Si el valor del primer campo del buffer es 1111h, las interrupciones de la tarjeta de adquisición sobre el CPU son deshabilitadas. Se detiene la operación del sistema *SAD*.

3).- change:
Si ninguno de los valores anteriores están presentes en el primer campo del buffer, la aplicación indica que una operación de substitución de parámetros de control debe ser efectuada en el driver. Esta operación puede ser de gran utilidad durante la etapa de desarrollo de aplicaciones.

Cuando el flujo de control se transfiere a *sad_on*, durante el proceso de arranque del sistema, se habilita la línea de interrupción asociada a la tarjeta de adquisición. Para hacer esta operación, se accesa el PIC de la computadora, y se apaga el bit que inhabilita la línea de interrupción asociada al sistema *SAD*, mediante el comando **out 21h, AL...**

Los PICs en una computadora IBM PC compatible, se programan desde las rutinas del BIOS, para que reconozcan las interrupciones a través del flanco de subida de la señal.

Si un dispositivo genera una interrupción cuando la línea de detección del PIC a donde se conecta está deshabilitada, la interrupción no será detectada. Si el dispositivo mantiene el nivel de la señal en alto en espera de que se atienda su requerimiento, y la línea de entrada del PIC es habilitada, la interrupción *tampoco* será detectada, pues cuando la entrada del PIC fué habilitada el flanco de subida de la señal de interrupción ya había pasado.

Para prevenir ésta situación, es necesario hacer que el nivel de la señal de interrupción baje aunque sea unos instantes, para que el PIC de la computadora detecte el flanco de subida de la señal y acepte la interrupción.

Cuando se ejecuta la instrucción int 0dh. se forza a que el interrupt handler asociado al sistema *SAD* entre en operación. El interrupt handler intentará leer datos de uno de los buffers del UART de la tarjeta de adquisición, cualquiera que sea el resultado de la operación, el *interrupt handler* avisará al PIC de la tarjeta de adquisición que la interrupción que estaba generando ya ha sido atendida. Cuando esto sucede, el PIC de adquisición baja al menos un instante el nivel de la señal de interrupción, cuando el nivel de la línea vuelve a subir a consecuencia de otra interrupción, el PIC de la computadora previamente habilitado, podrá detectar el flanco de subida de esa señal y reconocer así todas las interrupciones subsecuentes. Podríamos llamar a este proceso sincronización de interrupciones. (Ver interface sad_on).

Cuando se desea que el PIC de la computadora no reconozca más interrupciones en una de sus líneas de entrada, se deshabilita esa línea, encendiendo para ello el bit correspondiente en el registro IMR.

Comando 13
Open:

```
;command 13    Open
open:
        jmp    done
```

Cuando una aplicación genera el comando *Open* sobre el driver *SAD*, DOS le entrega un file handle a la misma, con el cuál deberá realizar los accesos subsecuentes al sistema. El driver *SAD*, al recibir el requerimiento por parte de DOS para que se efectúe este comando, no hará operación alguna en este caso.

El bloque de instruciones agrupadas bajo la etiqueta ***sad_on*** del comando *ioctl_out.*, pudiera ser colocado dentro del comando Open. La decisión de no hacerlo así, se debió a que era de nuestro interés recordarle al programador de aplicaciones que la tarjeta de adquisición funcionaba en base a interrupciones y que había que mandar instrucciones al CPU para habilitar o deshabilitar las mismas. El haber puesto esas instruciones formando parte de los comandos *Open* para *sad_on* y *Close* para *sad_off* hubiera resultado en que esas operaciones sobre el PIC de la computadora hubiesen pasado desapercibidas.

Comando 14
Close:

```
;command 14    Close
close:
        jmp    done
```

Cuando la aplicación genera un ***DOS Close***, el kernel en respuesta libera el número de file handle con que identificaba el driver.

Sección para Actualización de Status.

```
;                           Error Bit   Area
;unknown:
        or      es:[bx].rh_status,8003h     ; set error bit and error code
        jmp     done                        ; command no aplicable, unknown

error:
or      es:[bx].rh_status,800bh             ; set error bit, error code
        mov     es:[bx].rh4_count,0         ; system  error, driver working
        jmp     done                        ; in bad state

;**********************************************
;                       common exit
;**********************************************

busy:
        or      es:[bx].rh_status,0200h     ; set busy bit
                                            ; data to available
done:
        or      es:[bx].rh_status,0100h     ; set done
```

Cuando se solicita al driver *SAD* que ejecute un comando inadecuado en relación a la función para la cuál está diseñado, *SAD* responde encendiendo el bit de *unknown* del campo de status del request header, para indicarle esa situación a DOS.

Cuando el flujo de ejecución, cae sobre la etiqueta *error*, los bits de status que se encienden, hacen que el kernel detenga por completo la ejecución del sistema, para indicarle al usuario que un grave error en la operación del driver ha ocurrido. Esta situación puede ser originada cuando la rutina de verificación de *SAD* situada en el comando ioctl_input, detecta valores anormales en las variables de control del driver.

El código asociado a la etiqueta busy hace la indicación de que no existen datos disponibles para ser transferidos.

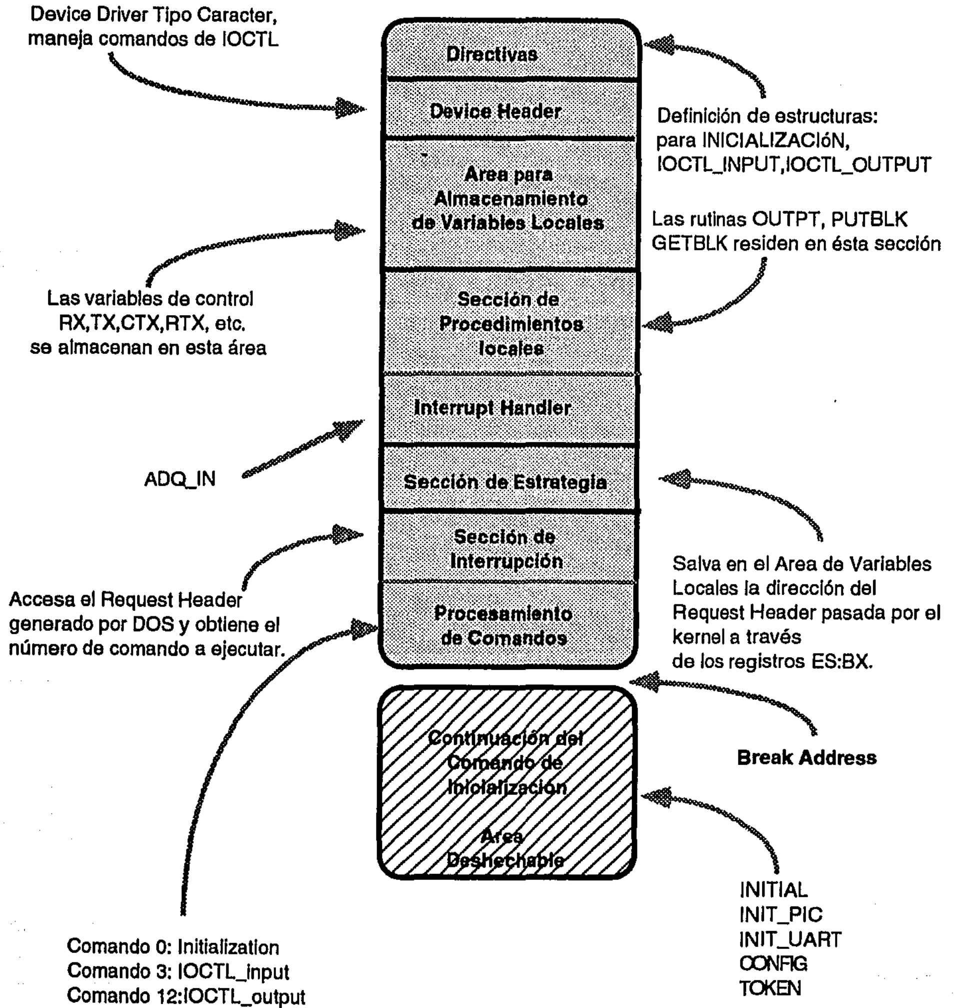
Independientemente del resultado de la operación, es una norma que el bit de done del campo de status sea siempre encendido al terminar el driver la ejecución de cualquier comando, así DOS estará seguro que la operación del driver se llevó a término conforme el protocolo establecido.

Sección para Restauración de Registros del CPU.

```
        popa                            ; pop  ax,cx,dx,bx,sp,bp,si,di
        pop     es
        pop     ds
        cli
        mov     ss,cs:stack_seg
        mov     sp,cs:stack_ptr
        sti
        ret                             ; return to dos
;****************************************************
;
;                  end of program
;****************************************************
;
initial         proc near               ; trasient area
        ......                          ; this area will be overwritted after
initial         endp                    ; its execution
```

Se restaura el stack original del sistema, así como los valores de los registros del CPU. El control se regresa a DOS, la operación del driver ha concluído.

La dirección de memoria donde se encuentra *ret*, es la que DOS considera como la última localidad asociada al driver. Después de esta instrucción, se define todo el código de inicialización de *SAD* que ejecuta la rutina *initial*. El área de memoria ocupada por *initial* es liberada después de su ejecución.


ESTRUCTURA DEL
DEVICE DRIVER
SAD
Device Driver Tipo Caracter,
maneja comandos de IOCTL
Directivas
Device Header
Definición de estructuras:
para INICIALIZACIóN,
IOCTL_INPUT,IOCTL_OUTPUT
Area para
Almacenamiento
de Variables Locales
Las rutinas OUTPT, PUTBLK
GETBLK residen en ésta sección
Las variables de control
RX,TX,CTX,RTX, etc.
se almacenan en esta área
Sección de
Procedimientos
locales
Interrupt Handler
ADQ_IN
Sección de Estrategia
Sección de
Interrupción
Salva en el Area de Variables
Locales la dirección del
Request Header pasada por el
kernel a través
de los registros ES:BX.
Accesa el Request Header
generado por DOS y obtiene el
número de comando a ejecutar.
Procesamiento
de Comandos
Continuación del
Comando de
Inicialización
Area
Deshechable
Break Address
INITIAL
INIT_PIC
INIT_UART
CONFIG
TOKEN
Comando 0: Initialization
Comando 3: IOCTL_input
Comando 12:IOCTL_output

Puesta en Operación del Sistema SAD.

El driver y la tarjeta del Sistema de Adquisición de Datos SAD, fueron diseñados para trabajar en combinación con un programa de aplicación. La tarjeta recibe los datos, el driver los organiza, almacena y los entrega previa solicitud, al programa de aplicación. Para el sistema SAD, el origen de los datos y uso que les dé el programa de aplicación será transparente, siempre y cuando éstos arriben con el formato adecuado y la aplicación los solicite conforme al protocolo establecido.

Desde el punto de vista del programa de aplicación, el Sistema SAD es también transparente, pues el programador de la aplicación no necesita profundizar en los detalles de operación de la tarjeta. Lo único que debe aprender, es a utilizar las rutinas de interface suministradas para comunicarse con el driver.

Las pruebas finales de funcionamiento de la tarjeta y el driver SAD, fueron realizadas con la ayuda de un programa de aplicación llamado *catch*. Este fué desarrollado en lenguaje C, utilizando el modelo mediano del compilador Quick C de Microsoft, versión 2.0.

Las rutinas de interface para comunicación con el driver, fueron adaptadas para trabajar dentro del esquema definido por el modelo mediano de compilación para la asignación de segmentos. Las rutinas de interface al igual que todo el device driver fueron desarrolladas en lenguaje ensamblador, utilizando Microsoft Macro Assembler versión 5.10.

El programa *catch* se comunica con el driver a través de las rutinas de interface, obtiene los datos y despliega en pantalla las 16 señales provenientes de igual número de canales. En este punto, es pertinente señalar que la técnica utilizada en el programa catch para realizar la graficación de las 16 señales, se vale de la utilización de las librerías standards disponibles en Quick C, por lo que el tiempo de graficación empleado, aunque es bueno para esta aplicación, puede ser mejorado en caso de ser necesario, mediante el uso de otras librerías, que accesen directamente las localidades de memoria asociadas al video de la máquina.

Ahora, serán comentadas las secciones del programa catch que ejemplifican el uso de las rutinas de interface para la comunicación con el driver SAD.

```
/* El programa catch, se comunica con el device driver del sistema SAD y grafica las
señales de cada uno de los canales de entrada. */

#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
#include <graph.h>

extern unsigned short sad_on(unsigned short fh);
extern unsigned short sad_off(unsigned short fh);
extern unsigned short rext(unsigned short *v2,unsigned short fh);

extern unsigned short status(unsigned short *v,unsigned short fh);
extern unsigned short wstatus(unsigned short *v,unsigned short fh);
```

Las cinco declaraciones de rutinas externas definidas al inicio del programa, corresponden a las rutinas de interface suministradas para la comunicación con el driver *SAD*.

A través de la rutina *sad_on*, se solicita al driver habilite el reconocimiento de las interrupciones generadas por la tarjeta de interface en la computadora. El parámetro *fh*, corresponde al número de *file handle* asignado al driver por el sistema operativo.

Con la rutina *sad_off*, se solicita al driver que inhabilite el reconocimiento de las interrupciones generadas por la tarjeta SAD.

Mediante la rutina *rext*, el driver entrega un buffer con *5 segundos de datos*, proveniente del *Sistema Circular de Buffers* que él mismo controla. El apuntador **v2*, deberá contener la dirección del primer elemento del arreglo de datos donde se recibirá la información. El driver recibirá esa dirección de inicio y transferirá todos los datos desde un buffer del *Sistema Circular*, hacia el arreglo señalado.

```
/* Boolean Constants */
#define TRUE        1
#define FALSE       0

/* Plotting Constants */
#define MAX_SIZE        7220
/* Global variable declarations */
```

El valor de la constante *MAX_SIZE*, indica el número de enteros que componen un buffer del sistema circular, en este caso 7220.

```
unsigned short int *v3;
unsigned short int *v;
unsigned short vector3[MAX_SIZE];
unsigned short vector[6];
```

Las variables *V* y *V3* se definen cómo apuntadores hacia arreglos de enteros.

El arreglo *vector3[MAX_SIZE]*, define al buffer donde serán recibidos los datos que el driver entregue al programa de aplicación. El número de elementos del arreglo *vector3*, corresponde al número de elementos que componen un buffer del Sistema Circular SAD.

El arreglo *vector[6]*, se usa con las rutinas *status* y *wstatus*, en las operaciones de lectura y escritura de parámetros de control del driver *SAD*.

```
void main(void)
{
        init_video_mode();
        fh=open("sad",O_BINARYIO_RDWR);     /* obtiene file handle */
        bad=sad_on(fh);                     /* habilita interrupciones */
        flag=TRUE;
        v3=&vector3[0];                 /* obtiene dirección de inicio vector3 */
        while (flag)
        {
                bad=1;
                while (bad)                 /* Inicio del loop */
                        bad=rext(v3,fh);    /* obtiene un buffer de datos */
        Plot_3_Canal();                     /* despliega 16 canales */
        }
        _setvideomode(_DEFAULTMODE);
        bad=sad_off(fh);                    /* deshabilita interrupciones */
        fh=close(fh);                       /* libera file handle */
        exit(0);
} /* end main */
```

El listado anterior muestra el procedimiento principal del programa *catch.*

A través de la rutina *OPEN*, el sistema operativo entrega el número de *file handle* asignado al driver *SAD*. Este número quedará almacenado en la variable *fh,* y será utilizado para realizar todos los accesos subsecuentes al driver. Puede notarse que el driver es abierto como si fuera un archivo.

Con la rutina *sad_on* y el número de *file handle*, se solicita al driver que habilite el reconocimiento de las interrupciones generadas por la tarjeta SAD en el PIC de la computadora. En este momento el sistema SAD entra totalmente en operación.

Se asigna la dirección del primer elemento del arreglo *vector3,* al apuntador *v3.*

Se introduce un *loop* donde se solicita un buffer de datos al driver. La rutina *rext* utiliza dos elementos para ello: el primero es la variable *V3*, la cuál contiene la dirección de inicio del arreglo *vector3* de recepción. El segundo elemento es el *file handle.* (Comando 3 IOCTL_Input del driver SAD).

Mientras no exista un buffer disponible en el *Sistema Circular*, el driver regresará a la rutina *rext* un **1**, y el flujo de instrucciones dentro del loop será ejecutado una y otra vez, hasta que el valor entregado por el driver a *rext* sea **0**. Cuando ésto suceda, el arreglo *vector3* habrá recibido 5 segundos de datos correspondientes a cada uno de los 16 canales de recepción de la interface*!!!.*

Con la rutina *sad_off* se solicita al driver, que sea deshabilitado el reconocimiento de las interrupciones generadas por la tarjeta de recepción, el sistema SAD deja en este momento de interrumpir la operación del CPU. Otras tareas no relacionadas con la adquisición de datos pueden realizarce en la computadora. (Comando 12 IOCTL_Output).

Con la función *close* se libera el número de file handle que había sido utilizado por la aplicación para la comunicación con el driver *SAD.*

El análisis de las rutinas *wstatus* y *status* se realizará a continuación. Ambas rutinas, se recomiendan para verificar que exista una correcta interacción entre el driver y la aplicación durante la etapa de desarrollo.

```
void main(void)
{
        . . .
        v=&vector[0];
        error=status(v,fh);              /*obtiene parámetros de control */
        if error==0
        {
                printf("rx : %u",&vector[0]);
                printf("tx : %u",&vector[1]);
                printf("crx: %u",&vector[2]);
                printf("ctx : %u",&vector[3]);
                printf("tpr : %u",&vector[4]);
                printf("control: %u",&vector[5]);
        }
        . . .
        printf("Cambio del parámetro rx: %u", vector[0]);
        i=scanf("%u",&vector[0]);
        error=wstatus(v,fh);             /*cambio de parámetros de control */
        . . .
        exit(0);
} /* end main */
```

Con la rutina de *status*, es posible obtener los valores de los parámetros que controlan la operación del driver. (Comando 3).

La dirección del apuntador *V* se asocia con el inicio del arreglo *vector* para el intercambio de parámetros. Ver definición de *vector* en el listado anterior.

Se llama a la rutina status, con el apuntador *V* y el número de *file handle*, para solicitar al driver que entregue los valores de sus parámetros de control.

Seis parámetros se utilizan para controlar la operación del driver: *rx, cx, ctx, crx, tail, counter.* La descripción de la función de cada uno se encuentra en el área de definición de variables del driver *SAD*. Cuando ha finalizado la ejecución de la rutina *status*, el arreglo *vector* contendrá los valores correspondientes a los parámetros de control en el orden ejemplificado en el listado anterior.

A través de la rutina *wstatus*, es posible modificar los valores de los parámetros que controlan la operación del driver. En el listado anterior se muestra el uso de la rutina wstatus para el cambio de valor del parámetro ***rx***. (Comando 12 IOCTL_OUPUT).

Consideraciones para el diseño de la Tarjeta de Interface del Sistema de Adquisición SAD

Tarjeta de Interface para el Sistema de Adquisición de Datos Digitales.

La tarjeta de interface del Sistema **SAD**, fue desarrollada para hacer posible la recepción simultánea en una computadora, de varios canales de datos provenientes de estaciones GEOS-3.

Uno de los mayores beneficios que se obtienen de incorporar la electrónica de recepción, dentro de la arquitectura de una PC, tiene relación con la enorme capacidad con que cuenta ésta para el tratamiento y análisis de datos. A través de una máquina, además de recibir los datos, es posible organizarlos, graficarlos y analizarlos todo en tiempo real. La importancia del presente desarrollo, radica en que abre la posibilidad para la operación de un sistema de detección sísmica en tiempo real, que utilice los datos captados por la tarjeta de recepción.

La tarjeta **SAD**, cuenta con dos UART's, cada uno con 8 canales para la recepción de datos, un controlador de interrupciones para la asignación de prioridades y solicitudes de servicio, un circuito para decodificación de direcciones y varios elementos para el acoplamiento de la tarjeta de interface con el bus de la computadora.

Consideraciones de Diseño en una Tarjeta de Interface para PC.

Una computadora, se comunica con el exterior a través de dispositivos periféricos. Todo dispositivo necesita de elementos que le permitan interactuar ó integrarse en la arquitectura de la máquina; esos elementos reciben el nombre de interfaces. No es raro que la electrónica de control del dispositivo, se confunda con la electrónica para hacer interface con la computadora, es por ello, que en general se acepta el nombre de interface para hacer referencia a toda la electrónica asociada a un dispositivo periférico. A continuación se listan algunos periféricos de uso común en las PC's.

- Tarjetas de gráficos.
- Discos Duros.
- Modems.
- Mouses.

- Teclados.
- Impresoras.
- Interfaces de Redes.

Para el desarrollo de una interface, varios elementos deben ser considerados, a fin de asegurar la correcta integración de la misma en la arquitectura de la máquina. A lo largo de los siguientes párrafos, serán abordadas las consideraciones más importantes.

Puertos de I/O.

Los periféricos utilizados en una computadora, son accesados y controlados a través de los puertos de I/O (entrada/salida). La arquitectura de los microprocesadores de la familia intel 80CX86, reserva 65536 puertos de I/O. Las computadoras personales XT sin embargo, por razones de diseño sólo manejan las líneas A_0-A_9 para el direccionamiento de puertos (en el modelo AT se pueden usar de la A_0-A_{15}). Así, sólo existen 1024 puertos disponibles para el direccionamiento de I/O. Los primeros 512 puertos, están reservados para periféricos integrados dentro la tarjeta principal del sistema (mother board), los restantes 512 son los únicos disponibles para el acceso a periféricos conectados a través de las ranuras de expansión. Con la línea A_9, se identifica si el dispositivo pertenece a la tarjeta principal de la computadora ó si es un dispositivo externo a ella. Cuando el nivel de la línea A_9 es alto (activo), el controlador del bus del sistema genera las señales adecuadas, para efectuar la interacción con una de las interfaces insertadas en los slots de la computadora. Cuando el nivel es bajo, el controlador del bus de la máquina, no realiza operación alguna sobre las tarjetas insertadas en las ranuras de expansión, pues el direccionamiento está dirigido hacia un periférico integrado en la tarjeta principal de la computadora. (Eggebrecht, 1990).

Control del Bus del Sistema.

La señal AEN (Address Enable), se usa para indicar al microprocesador y a otros dispositivos que utilizan el canal de I/O, que el bus del sistema no se encuentra disponible, debido a que operaciones de acceso directo a memoria (DMA) se están llevando a cabo.

Cuando esta señal es activa, el controlador del DMA queda a cargo de las líneas de: dirección, datos, de lectura y escritura del bus del sistema. Cuando la señal está en un nivel bajo, indica que el **CPU** tiene control total sobre el bus de la computadora.

Fuente de Alimentación.

Las características técnicas de la fuente de alimentación, deben ser consideradas en el diseño de la interface, pues ésta no debe de exceder los parámetros de diseño definidos. (IBM Technical Reference AT, 1984).

Los voltajes de salida generados por la fuente de alimentación, así como las tolerancias y cargas de corriente que soporta, son mostrados en la siguiente tabla:

Salida Nominal	Corriente [A] Minima	Máxima	Tolerancia de Regulación
+ 5 Vdc	7.0	19.8	+ 5% a -4%
- 5 Vdc	0.0	0.3	+10% a -8%
+12 Vdc	2.5	7.3	+ 5% a -4%
-12 Vdc	0.0	0.3	+10% a -9%

Dado que existe la posibilidad, de que una de estas salidas sea sobrecargada, la fuente de alimentación cuenta con circuitos que al detectar alguna condición de sobrecorriente, cortan la operación de la misma por un lapso de *20 milisegundos*, para evitar que se dañe.

La fuente de alimentación proporciona una señal de *power-good*, para indicar condiciones apropiadas de operación.

La señal de power-good, se habilita cuando las condiciones de voltaje en la entrada y en la salida de la fuente cumplen con los rangos de operación especificados. Si cualquiera de las dos condiciones es anormal; la señal de power-good tendrá un nivel bajo. Si la señal de AC en la entrada falla, causará que la señal de power-good llegue a un nivel bajo *1 milisegundo* antes, de que cualquier voltaje de salida de DC transpase sus límites de regulación.

La señal de power-good, se conserva en un nivel bajo cuando la fuente es encendida y no cambia hasta que todos los valores de salida de DC han alcanzado su nivel mínimo de operación. Power-good tiene un retraso de encendido que oscila entre los *100* y los *500 milisegundos.* En la siguiente tabla se indican los valores mínimos de operación de salida, que se requieren para que la señal de power-good se mantenga activa.

Nivel Normal (Vdc)	Nivel Mínimo (Vdc)
+ 5	+4.5
- 5	-3.75
+12	+10.8
-12	-10.4

Condiciones de Carga y Manejo de Corriente.

Cuando se diseña una tarjeta de interface, es necesario determinar si existe suficiente capacidad para el manejo de la corriente con que operan los circuitos que serán utilizados. En caso de no cumplirse el requerimiento anterior, deben ser integrados en la tarjeta, circuitos adicionales que resuelvan esta situación.

Para el caso de las señales de salida del bus de la máquina, se definen dos parámetros para el manejo de corriente: **IOL** representa la máxima corriente que puede absorver el manejador de salida (driver) en el nivel lógico bajo. **IOH** representa la máxima corriente que puede ser generada desde el manejador (driver) en el nivel lógico alto. A continuación se muestran las capacidades de los manejadores para las señales de salida del bus del sistema.

SEÑALES DEL BUS DE SALIDA	IOL(mA)	IOH(mA)
D_0-D_7	23.6	-14.96
A_{19}-A_{16}	7.2	-2.46
A_{14}-A_{15}	21.2	-2.51
A_{13}	23.2	-2.56
A_0-A_{12}	23.4	-2.56
IOR, IOW, MEMR, MEMW	23.8	-4.98
CLK	23.2	-14.96
AEN, DACK0	24.0	-15.00

SEÑALES DEL BUS DE SALIDA	IOL(mA)	IOH(mA)
DACK1	3.2	-0.20
DACK2, DACK3	2.8	-0.18
ALE	14.8	-0.94
RESET DRV, T/C	8.0	-0.40
OSC	5.0	-1.00

Las señales de entrada del bus del sistema, presentan una carga a las tarjetas conectadas en los slots de expansión. Esta carga, es la corriente de entrada de nivel bajo (**IIL**), que requiere el circuito manejador (driver) para establecer una señal lógica de nivel bajo. El driver a su vez, debe proporcionar la corriente (**IIH**) necesaria para el nivel alto, al bus de la tarjeta de expansión.

SEÑALES DEL BUS	IIL(mA)	IIH(mA)
D_0-D_7	-0.40	0.04
I/O CH CK	-0.40	0.02
I/O CH RDY	-0.40	0.02
IRQ_2-IRQ_7	-0.01	0.01
DRQ_1-DRQ_3	-0.01	0.01

La carga capacitiva presente en el bus del sistema, es otra variable que se debe considerar. Cada tarjeta que se inserta en una ranura de expansión, representa una carga capacitiva adicional a las señales del bus. Ya que las cargas capacitivas colocadas en paralelo se suman, la señal utilizada puede sufrir importante distorsión ó salirse de los tiempos máximos de operación. Cada carga, añade de *10* a *20 picofarads* de capacitancia a las señales del bus y para valores mayores de *200 picofarads* de carga capacitiva, las señales del bus se ven seriamente afectadas.

En general, no es necesario efectuar los cálculos de manejo de corriente y de carga capacitiva, para asegurar el buen comportamiento de las señales en el bus de expansión, si se siguen las siguientes recomendaciones:

1.- No conectar ninguna señal del bus del sistema directamente a un dispositivo con tecnología *NMOS LSI*, debido a que estos dispositivos tienen muy poca capacidad en el manejo de corriente y no toleran los disparos cortos negativos que pueden existir en el bus de la máquina.

2.- No conectar ninguna señal del bus del sistema a más de 2 cargas de dispositivos con tecnología *LS*.

3.- Desacoplar los manejadores del bus (drivers y transceivers), con un capacitor de aproximadamente 0.1 microfarad entre sus entradas de alimentación (+5 v) y tierra (GND).

4.- No utilizar las señales del bus del sistema a través de largas distancias en la tarjeta de interface, ya que esto incrementa la capacitancia de tal forma, que las señales en el bus pueden experimentar problemas serios de retraso y/ó distorsión.

Tiempos en el Bus del Sistema.

Uno de los puntos críticos en el diseño de las tarjetas de interface, tiene relación con los tiempos de respuesta de las señales en el bus del sistema. Los tiempos de conmutación de la señal, pueden salirse de su rango de operación debido a varias causas como: niveles incorrectos en los voltajes de alimentación, deficientes sistemas de tierra, altas cargas capacitivas y/ó de alimentación. **Las figuras 1, 2, 3** y **4** del **apéndice A**, muestran las relaciones entre tiempos mínimos y máximos, que deben conservar las señales en el bus de la máquina para su correcta operación.

Tiempos de Decodificación para el Direccionamiento de Puertos de I/O.

Dos importantes aspectos que no se deben soslayar en el diseño de una interface, tienen relación con los tiempos de decodificación que se deben de manejar.

Si la decodificación de un puerto de I/O se retrasa considerablemente, el puerto asociado no estará disponible en el momento que las señales de control de lectura (IOR) ó escritura (IOW) aparezcan dentro del bus. Si ésto sucede, los datos serán escritos ó leídos de un puerto equivocado. Para una computadora personal, el tiempo de retraso máximo permitido para decodificar y direccionar un puerto de I/O es de *92 nanosegundos*.

Si la señal de control de escritura se retrasa en forma considerable, y el circuito de decodificación de una interface responde con bastante rapidez, puede ocurrir un translape, ya que la interface queda habilitada cuando la señal de control de escritura, que había sido generada para otro dispositivo se encuentra aún activa; datos erróneos serán recibidos en esa interface. Para evitar estos problemas, se especifica un retraso máximo en la señal de IOW de *120 nanosegundos.*

Selección de Puertos de I/O.

Para incorporar una interface a la arquitectura de una máquina, una de las primeras tareas que se deben de efectuar, es buscar un bloque de puertos de I/O disponibles. Al existir sólo 512 puertos para dispositivos periféricos externos y considerando que la computadora utiliza gran parte de este espacio para interactuar con otros dispositivos, al final quedan muy pocos puertos factibles de ser utilizados en el diseño de nuevas interfaces. A continuación, se presenta una tabla con los puertos de I/O usados por los dispositivos periféricos más comunes en una computadora AT.

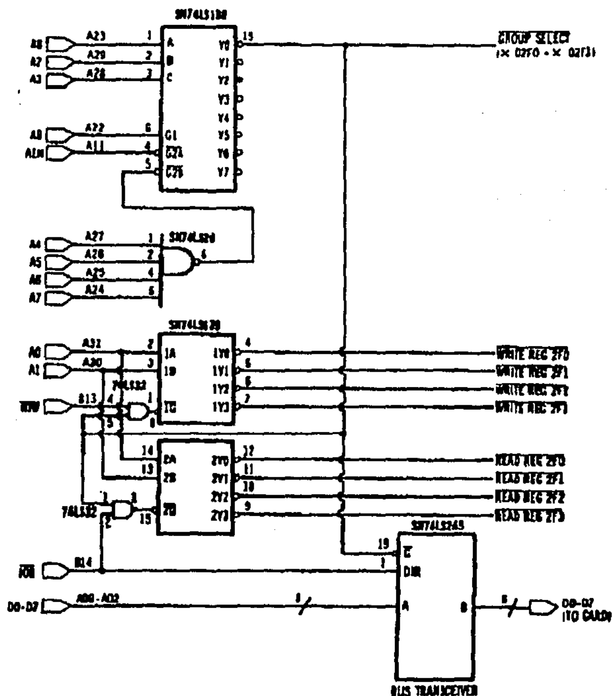
Rango en Hexadecimal	Dispositivo
000-01F	DMA controller 1, 8237A-5
020-03F	Interrupt controller 1, 82C59A, Master
040-05F	Timer, 8254.2
060-06F	8042 (Keyboard)
070-07F	Real-time clock, NMI (non-maskable interrupt) mask
080-09F	DMA page register, 74LS612
0A0-0BF	Interrupt controller 2, 82C59A
0C0-0DF	DMA controller 2, 8237A-5
0F0	Clear Math Coprocessor
0F1	Reset Math Coprocesor
0F8-0FF	Math Coprocessor
1F0-1F8	Fixed Disk
200-207	Game I/O
278-27F	Parallel Printer port 2
2F8-2FF	Serial Port 2
300-31F	Prototype Card
360-36F	Reserved
378-37F	Parallel Printer port 1
380-38F	SDLC, bisynchronous 2
3A0-3AF	Bisynchronous 1
3B0-3BF	Monochrome Display an Printer Adapter
3C0-3CF	Reserved
3D0-3DF	Color/Graphics Monitor Adapter
3F0-3F7	Diskette controller
3F8-3FF	Serial Port 1

Técnicas de Decodificación y Direccionamiento para Puertos de I/O.

En los siguientes párrafos, serán tratadas las técnicas más comunes de decodificación y direccionamiento de puertos de entrada/salida, utilizadas en el diseño de las tarjetas de interface.

Decodificación de Direccionamiento Fijo.

En el diagrama siguiente, se muestra un ejemplo de un circuito para decodificación de direccionamiento fijo.



Decodificación de Direccionamiento Fijo.

En el diagrama anterior, se observa que son 4 los puertos de I/O que pueden ser decodificados por este circuito (2F0h - 2F3h). Con las señales de control de lectura y escritura, se logra el acceso a dos buffers diferentes con un mismo puerto, un buffer para la lectura y un buffer para la escritura.

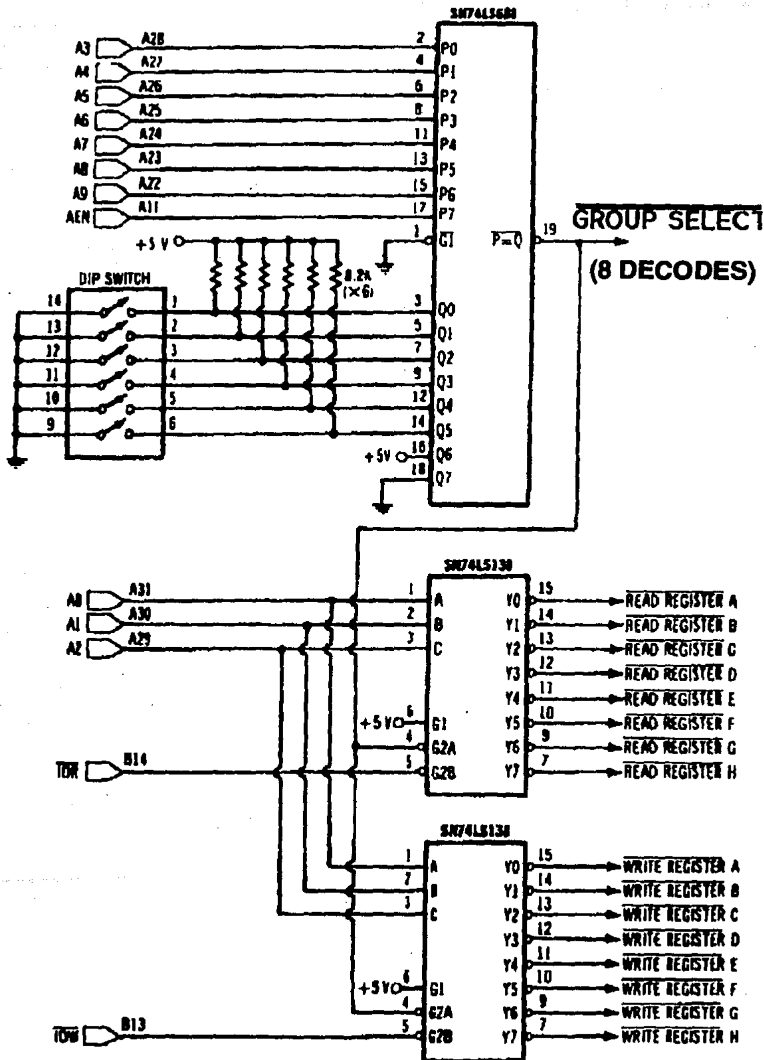
La señal de GROUP SELECT, puede ser utilizada para habilitar el manejador (transceiver) del bus de datos cuando cualquiera de los 4 registros decodificados es seleccionado. La línea A_9 de direcciones debe estar activa durante la decodificación, para indicar que el puerto direccionado se encuentra en el espacio de direcciones de dispositivos periféricos externos.

La señal de habilitación AEN, se usa para prevenir decodificaciones inválidas, durante los periodos de control del **DMA**.

Decodificación Seleccionable por Switch.

En la siguiente figura, se muestra un diseño en el cuál es posible mover, el bloque de direcciones de los 8 puertos elegidos dentro del espacio de direcciones disponibles de una PC, simplemente seleccionando un nuevo valor en un banco de switches tipo DIP.

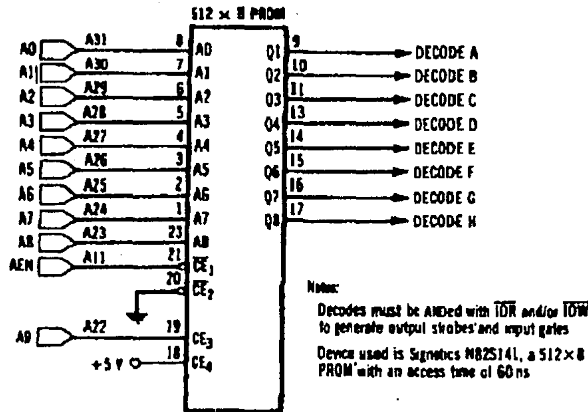
En este diseño, se utiliza un comparador binario octal 74LS688. De un lado del circuito comparador, se encuentran conectadas las lineas del bus de direcciones (de la A_3 hasta la A_9) y la señal AEN. En el otro lado, están conectadas las salidas de los switches tipo DIP. Cuando el valor seleccionado en los switches tipo DIP coincide con el valor del bus de direcciones, la salida del comparador se activà, y es entonces utilizada como la señal de control de GROUP SELECT. Dependiendo de la posición del switch es el valor asignado; si se encuentra abierto (OFF), éste tiene un valor de nivel lógico alto y si se encuentra cerrado (ON), éste tendrá un valor de nivel lógico bajo.



Decodificación de Direccionamiento seleccionable por Switch.

Direccionamiento con Memoria PROM.

Esta técnica se emplea, cuando es necesario decodificar múltiples puertos de direcciones con una sola tarjeta, la cual desempeña las funciones de varias interfaces, manteniendo el direccionamiento de puertos original. En la siguiente figura aparece un ejemplo del circuito empleado.



PROM I/O port address decoding

AT PROM ADDRESS

A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
1	1	0	0	0	0	0	1	0

PROGRAM DATA

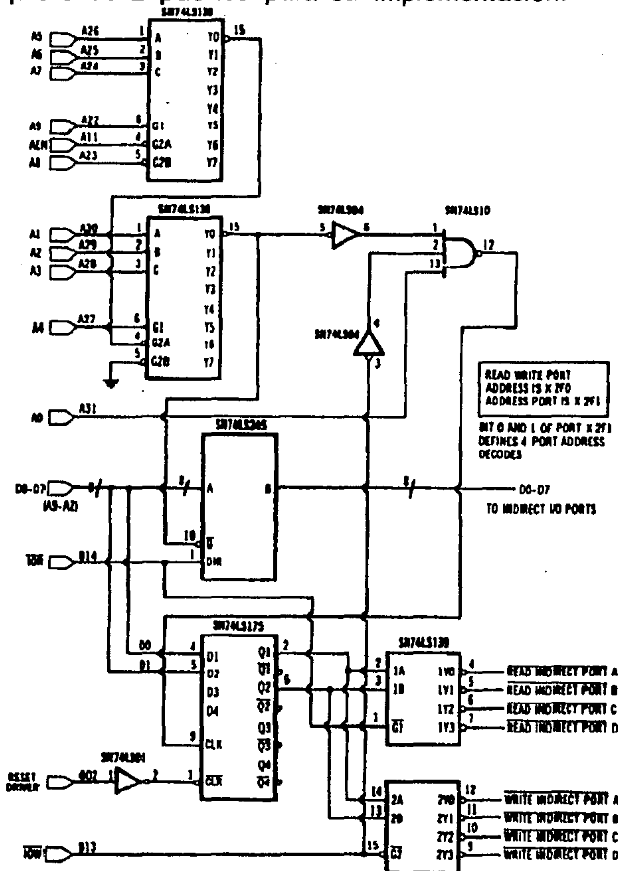
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	1

PROM decode programming example.

Las señales de dirección en la entrada de la memoria PROM (Programmable Read Only Memory), seleccionan una localidad donde se almacena un dato de salida único, el cual ha sido previamente grabado en la PROM en el momento de su manufactura. El dato de salida de la PROM es utilizado para direccionar el dispositivo elegido.

Direccionamiento Indirecto de Puertos.

En esta técnica, el contenido de un puerto es utilizado para direccionar otros. Si se usan puertos de 8 bits, es posible decodificar hasta 256 direcciones de puertos adicionales. Este método requiere de 2 puertos para su implementación.

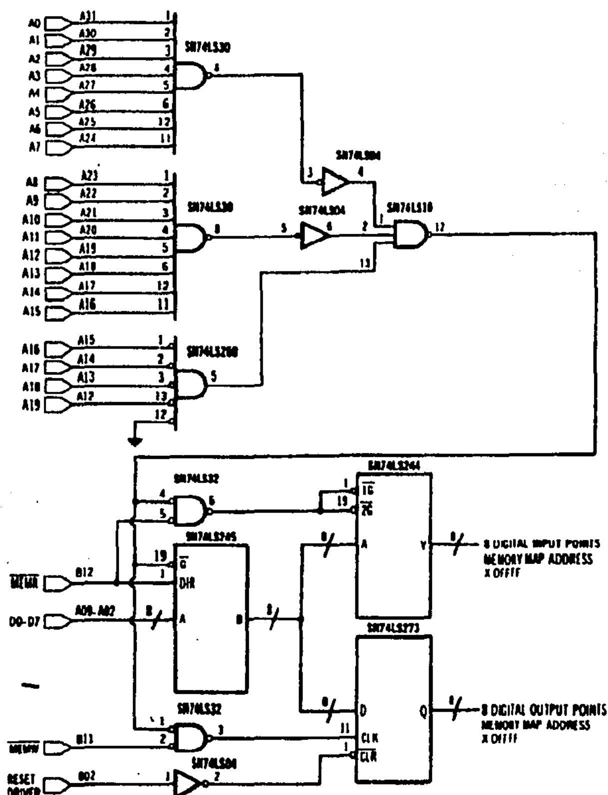


Decodificación de Direccionamiento Indirecto de Puertos.

El primer puerto es el de selección, la dirección que se escribe en él, sirve para habilitar el puerto indirecto que se desea accesar. El segundo puerto es el de datos. Cuando un dato se escribe en él, es al mismo tiempo escrito en el puerto habilitado previamente por el puerto de selección. Cuando el puerto de datos es leído, el dato obtenido proviene del puerto habilitado con el puerto de selección.

Decodificación de Puertos de Entrada/Salida por Mapeo de Memoria.

Esta técnica utiliza las direcciones de memoria como direcciones de los puertos de I/O. Con ésto se elimina en gran medida la restricción de los 512 puertos de I/O disponibles para dispositivos periféricos. Una ventaja adicional, es que al ser los puertos de I/O direccionados como localidades de memoria, el set completo de instrucciones del microprocesador queda disponible para manipular los datos asociados a esos puertos. Finalmente, se necesitan sólo 4 ciclos de reloj para trabajar con localidades de memoria, contra 5 ciclos de reloj que toma el acceso a puertos normales de I/O.



Decodificación de direccionamiento de Puertos por Mapeo de Memoria.

Ahora, ya estamos en codiciones de realizar un balance de las ventajas y desventajas que ofrecen cada una de las técnicas de decodificación disponibles. El hacer esto, nos ayudará a elegir el circuito de decodificación más apropiado a nuestras necesidades.

El único inconveniente en la DECODIFICACION DE DIRECCIONAMIENTO FIJO, es que el espacio disponible para el acceso a los puertos de entrada/salida no puede modificarse. Si dos interfaces utilizan un mismo puerto de I/O, con ésta técnica de direccionamiento, es realmente complicado ponerlas a trabajar al mismo tiempo en una computadora, pues al menos se requiere del corte y restauración de pistas en una de las tarjetas de interface.

En el caso de la DECODIFICACION CON MEMORIA PROM, la desventaja tiene relación con el alto costo de la misma y con la necesidad de que sus tiempos de acceso sean menores de 92 nanosegundos, para no tener problemas en los tiempos de decodificación.

Para la DECODIFICACION DE DIRECCIONAMIENTO INDIRECTO, la desventaja radica, en que es necesaria la utilización de dos instrucciones de lectura/escritura (OUT/IN-IN/OUT) para poder efectuar un solo comando de lectura y/ó escritura sobre un puerto de I/O, así, se pierde eficiencia en los accesos a cada puerto.

En el caso de la DECODIFICACION DE DIRECCIONAMIENTO POR MAPEO DE MEMORIA, la desventaja radica en que utiliza parte de la memoria principal de la computadora y es necesario utilizar el total de las líneas de direcciones existentes en el bus del sistema, para la decodificación de una dirección de puerto, lo cual incrementa en forma considerable el uso de circuitos decodificadores.

Del análisis anterior, es posible concluir, que una de las técnicas más simple y de mayor flexibilidad para el desarrollo de un prototipo de tarjeta de interface, es la técnica de DECODIFICACION SELECCIONABLE POR SWITCH.

Técnicas de Transferencia de Datos.

Una consideración importante en el diseño de interfaces, tiene relación directa con la forma como se lleva a cabo la transferencia de datos entre la tarjeta de interface y la computadora. El usar una técnica adecuada, es requisito indispensable para asegurar la viabilidad de un sistema.

Una de las técnicas más sencillas para transferencia de datos entre una PC y un dispositivo externo, involucra la utilización de las funciones de entrada/salida (INPUT/OUTPUT) del lenguaje de programación BASIC. El inconveniente de éste método, radica en que las velocidades máximas de transferencia, no llegan a sobrepasar los 210 bytes/seg. Cuando se utilizan instrucciones de entrada/salida IN/OUT en lenguaje ensamblador, se pueden obtener resultados en un orden superior a 400 veces la velocidad de transferencia que se alcanza utilizando instrucciones de entrada/salida en un lenguaje de alto nivel (PASCAL, FORTRAN, C, etc.).

En muchas aplicaciones que involucran el uso de dispositivos periféricos, se necesitan a menudo velocidades de transferencia memoria-interface mayores a las que son posibles de lograr con instrucciones simples de IN/OUT. Un ejemplo de este problema, tiene que ver con el funcionamiento de los controladores de disco flexible. La tasa de transferencia entre el CPU y el drive es suficientemente alta para hacer difícil que el microprocesador pueda además servir al mismo tiempo a otros dispositivos como el teclado y el mouse. Para resolver esta situación, una función especial llamada acceso directo a memoria (DMA), está disponible para el diseño de interfaces. Un controlador de DMA permite a una interface o adaptador leer o escribir datos a memoria sin la intervención del microprocesador. Con este método, se logran velocidades de 5 a 6 veces superiores a las alcanzadas utilizando las instrucciones de entrada/salida del lenguaje ensamblador. Las desventajas de este método, es que para utilizar el DMA, es necesario añadir circuitos independientes para crear la lógica de control necesaria para lectura/escritura en la tarjeta de interface y la programación de control es más compleja.

Otra técnica sugerida, en el caso de grandes requerimientos para la transferencia de datos, necesita la utilización de 2 puertos de alta velocidad. Mientras el microprocesador lee ó escribe a un puerto, la interface accesa el otro.

Los puertos son multiplexados por división de tiempo a través de un reloj. En una fase de reloj, un puerto se conecta al bus del sistema y el otro a la interface. En la siguiente fase de reloj, los puertos intercambian sus conexiones. La desventaja de este método, es que se debe de agregar memoria de alta velocidad al sistema para poder soportar los rápidos accesos del microprocesador y de la interface.

Finalmente, queda por analizar una técnica llamada buffer de ping-pong. En ella, se usan 2 buffers de memoria, cada uno con el espacio suficiente para almacenar el máximo número de datos requeridos para la operación de la interface. Los buffers pueden ser conectados al bus del sistema ó a la tarjeta de interface. Mientras un buffer recibe ó transmite datos hacia la interface, el otro estará siendo accesado por el microprocesador.

Los requerimientos para la transferencia de datos entre la tarjeta del sistema SAD y la computadora, indican valores máximos en los casos críticos de alrededor de 11.2 Kbytes/seg (considerando que las estaciones GEOS-3 transmiten a 7000 bauds). Esta condición, dió la pauta para decidir utilizar en nuestro proyecto, las operaciones de entrada/salida disponibles en el microprocesador, a través de las instrucciones IN/OUT del lenguaje de máquina. La velocidad máxima de transferencia, que puede alcanzarse con este método es de alrededor de 84 Kbytes/seg. Si en el peor de los casos se necesita operar a una velocidad de 11.2 Kbytes/seg, quedará disponible más del 85% del ancho de banda del CPU, para la ejecución de las demás operaciones y del programa de aplicación.

Técnicas para Atención a Dispositivos Periféricos.

En todo sistema de cómputo, el microprocesador necesita comunicarse con dispositivos de entrada/salida cómo: impresoras, monitores, etc. Desde el punto de vista del sistema, el procesador debe ocupar el mínimo tiempo posible en la atención de los periféricos, ya que el tiempo requerido en la ejecución de tareas de I/O, afecta directamente la cantidad de tiempo disponible para la ejecución de otros procesos. En otras palabras, el sistema debe ser diseñado de tal forma, que los servicios de I/O tengan un efecto muy pequeño en el desempeño total del sistema.

Existen dos métodos básicos para manejar los requerimientos de I/O: la verificación por sondeo y el servicio a interrupciones.

La verificación por sondeo, implica esencialmente que el procesador accese cada periférico y verifique en sus registros de status si el dispositivo requiere de atención. Existen varios problemas asociados a la implementación de esta técnica. La primera que se plantea, es que tan frecuentemente el procesador debe accesar cada periférico para verificar su status. Segundo, habrá veces que cuando el dispositivo sea accesado, no esté listo para entregar ó recibir información, lo cuál será una pérdida de tiempo para el procesador. Otras veces, un dispositivo que esté listo, tendrá que esperar hasta que el procesador termine la secuencia de verificación de todos los dispositivos, con resultados adversos para el desempeño del dispositivo.

Otros problemas surgen cuando ciertos periféricos son más importantes que otros. La única manera de asignarles prioridades, es accesando esos dispositivos con mayor frecuencia que los demás. Es fácil observar, que la verificación por sondeo para la atención de dispositivos periféricos puede ser ineficiente y en general se puede afirmar que este método de servicio, tiene un efecto adverso en la eficiencia del sistema, limitando las tareas que pueden ser desarrolladas por el procesador.

Una estrategia más utilizada en la mayoría de los sistemas, permite que el procesador atienda esencialmente al programa de aplicación y solamente detenga su ejecución cuando un dispositivo de I/O genere una señal de interrupción. El procesador al recibir la interrupción, abandonará momentáneamente la ejecución del programa principal, para ejecutar las instrucciones de la rutina de servicio asociada a esa interrupción. Una vez ejecutada la rutina de servicio, el procesador reasumirá la ejecución del programa principal, desde la instrucción donde lo había dejado. Con la estrategia de servicio a interrupciones, no se desperdicia el tiempo del procesador verificando el estado de los dispositivos y en cambio la eficiencia del sistema aumenta, permitiendo que más tareas y dispositivos sean manejados por el procesador.

Selección de Circuitos para la Tarjeta de Interface.

Ya puntualizados los aspectos más importantes que se deben de cuidar en el diseño de una tarjeta de interface, queda por elegir los circuitos que puedan proporcionar condiciones adecuadas de operación. Para hacer ésto, es necesario precisar con toda claridad los alcances de la interface a desarrollar. Para nuestro caso, la tarjeta tendría que recibir 16 canales de datos en forma asíncrona, a velocidades variables. Así necesitábamos de un circuito capaz de realizar esa tarea: un receptor de datos asíncronos programable, un UART (Universal Asinchronous Receiver Transmiter).

Selección del UART para la Tarjeta de Adquisición.

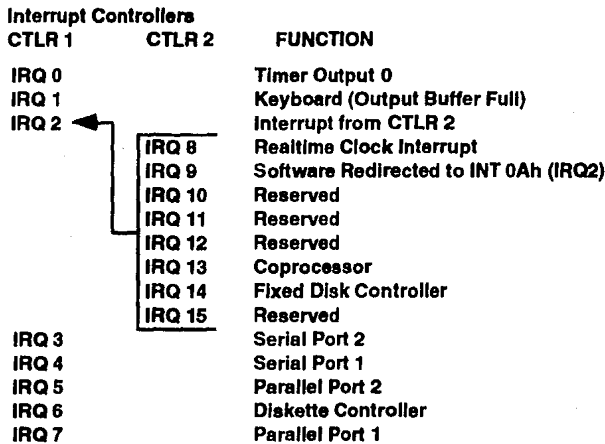
Existen en el mercado, diversas clases de UART's, el objetivo de nuestra investigación, era encontrar aquel que mejor balance proporcionara en aspectos relacionados con: bajo consumo de potencia, facilidad de programación, manejo de interrupciones, necesidades mínimas de puertos de I/O, etc.

Al investigar entre los diferentes UART's, se llegó a la conclusión de que el OCTAL *SCC2698B* de *SIGNETICS* era uno de los más balanceados. Otro, también seriamente considerado fue el CL-CD180 de CIRRUS LOGIC, este último, aunque más potente que el anterior en lo referente a control de la comunicación, buffer con mayor capacidad para la recepción por canal (8 bytes), timer por canal, manejo de buses para tecnología Intel ó Motorola y velocidad de reloj mayor (8 a 10 MHz), no fué elegido finalmente, por realizar el manejo codificado de interrupciones (sólo dispone de 3 lineas de solicitud de interrupción) y por requerir para su operación de 128 localidades de puertos de entrada/salida. El UART OCTAL SCC2698B únicamente necesita de 64 puertos de I/O y cuenta con 4 salidas de solicitud de interrupción (1 salida por cada 2 canales).

Ya que nuestrà intención es recibir 16 canales y el UART SCC2698B sólo cuenta con 8 canales, era necesario determinar si los 128 puertos que necesitarían dos UART's de este tipo, estarían disponibles dentro de la PC. Se detectaron dos bancos con 128 puertos, que podían considerarse disponibles para el desarrollo de la interface.

El contar con dos bancos de I/O aseguraba flexibilidad en la tarjeta y compatibilidad con el diseño de las tarjetas de interfaces comerciales.

Lo que no había disponible (ver siguiente figura), eran las 8 líneas de interrupción en el bus de la máquina, las cuáles se necesitaban para detectar el estado de saturación de los buffers de recepción de los UART's. Esto hacía necesaria la búsqueda de un dispositivo que por una parte, sólo empleara una de las líneas de interrupción del bus del sistema y por la otra pudiera manejar las ocho líneas de interrupción de los UART's. Se necesitaba de un circuito controlador de interrupciones programable.



Interrupt Controllers

CTLR 1	CTLR 2	FUNCTION
IRQ 0		Timer Output 0
IRQ 1		Keyboard (Output Buffer Full)
IRQ 2		Interrupt from CTLR 2
	IRQ 8	Realtime Clock Interrupt
	IRQ 9	Software Redirected to INT 0Ah (IRQ2)
	IRQ 10	Reserved
	IRQ 11	Reserved
	IRQ 12	Reserved
	IRQ 13	Coprocessor
	IRQ 14	Fixed Disk Controller
	IRQ 15	Reserved
IRQ 3		Serial Port 2
IRQ 4		Serial Port 1
IRQ 5		Parallel Port 2
IRQ 6		Diskette Controller
IRQ 7		Parallel Port 1

Selección del Controlador de Interrupciones Programables (PIC) de la Tarjeta de Interface.

En este caso, nuestra investigación consistió en determinar, si era factible la utilización de un PIC (Programable Interrupt Controller), similar al que se emplea en la tarjeta principal de las computadoras personales, para la expansión de las líneas de solicitudes de interrupción del CPU. Recuérdese que los microprocesadores de la familia intel 80CX86 disponen de sólo una línea de interrupción no mascarable.

En las computadoras AT, se utilizan dos PIC's (82C59A) en cascada, que expanden de 1 a 15 el número de líneas de interrupción disponibles. Aunque en el bus de la máquina no existen las líneas para conectar un tercer PIC en cascada desde una tarjeta de interface, determinamos que era adecuado el uso de un PIC 82C59A aún operando en forma independiente, por contar con las 8 líneas que los UARTS necesitaban para su operación y porque era posible mediante su programación, fijar los niveles de prioridad para la atención de cada línea de interrupción, de acuerdo a nuestras necesidades.

Así, los tres elementos fundamentales para la operación de la tarjeta de adquisición estaban determinados: los UART's para efectuar la recepción, el PIC para controlar y expandir las solicitudes de interrupciones y la técnica de decodificación seleccionable por switch para identificar los direccionamientos hacia la interface. Sin embargo, para lograr que estos tres elementos pudieran funcionar en conjunto y después interactuar con la computadora, habría de ser necesaria la introducción de elementos adicionales (buffers, transceivers, inversores, resistencias, etc.), que permitieran el correcto acoplamiento entre estos circuitos.

A partir de este punto, dos tareas restaban por efectuar: primero la familiarización con el principio de operación y programación del UART y del PIC y segundo, la programación de los mismos y su operación coordinada.

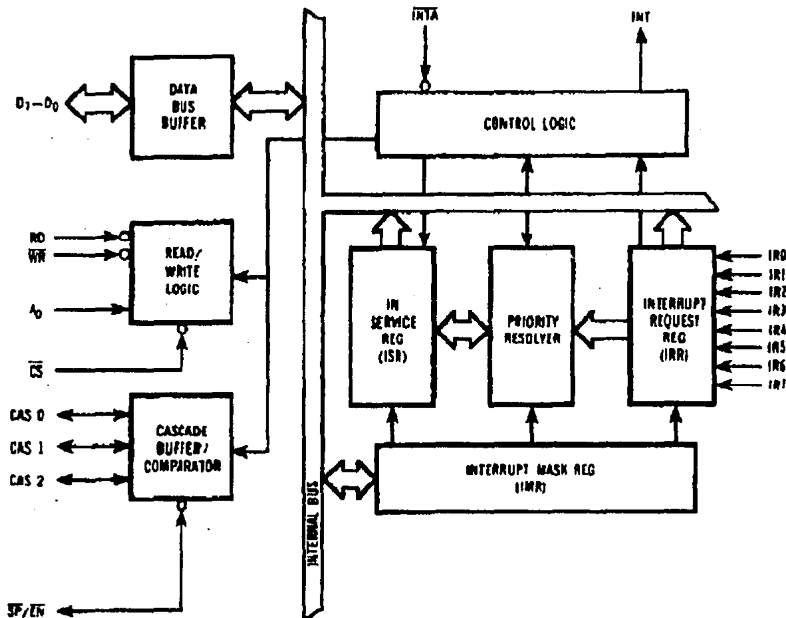
A continuación, se explicará en forma detallada, el principio de funcionamiento y programación del PIC y del UART. Finalmente se abordará la manera como fueron programados en nuestro sistema.

Implementación de la Tarjeta de Interface del Sistema SAD

El Controlador de Interrupciones del Sistema SAD.

El PIC 82C59A, es un dispositivo diseñado para relevar al microprocesador de las tareas de verificación de solicitud de servicio de dispositivos periféricos. Este se encarga de recibir las interrupciones generadas por los dispositivos, maneja 8 niveles de interrupciones, puede ser conectado en cascada con otros PICs y ser programado como un periférico de entrada/salida.

El PIC, funciona como el administrador general en un sistema de control de interrupciones. Este, acepta las solicitudes de interrupción de los equipos periféricos, analiza cuál tiene la mayor prioridad y envía una señal de interrupción al CPU. (Intel, 1989).



Block Diagram of an 82C59A Interrupt Controler
(Courtesy Intel Corporation)

Descripción de los Registros de Control y Operación del PIC.

La operación del PIC, se controla mediante la programación de una serie de registros internos de los que se hablará a continuación.

Registro de Solicitud de Interrupción y Registro de En-Servicio.

Las interrupciones en las lineas de entrada IR_{0-7} son manejadas por 2 registros en cascada, el registro de interrupciones solicitadas *(IRR Interrupt Request Register)* y el registro de en-servicio *(ISR In-Service Register)*. El IRR se usa para almacenar todos los niveles de interrupción que requieren atención ó servicio; el registro ISR se utiliza para guardar todos los niveles de interrupción que están siendo atendidos.

Analizador de Prioridad.

En este bloque se determina cuál de las solicitudes de interrupción tiene la mayor prioridad. Las líneas que requieren atención activan el bit correspondiente del IRR, mismo que se utiliza para saber que líneas solicitan atención. La línea de mayor prioridad es elegida. El bit correspondiente a la línea seleccionada es actualizado en el registro *ISR* durante el pulso de reconocimiento de interrupción *(INTA)* que genera el CPU.

Registro de Interrupciones Mascarables.

El registro de interrupciones mascarables *(IMR)*, cuenta con 8 bits, dependiendo del valor de cada bit, las líneas de solicitud de interrupción asociadas estarán ó no habilitadas.

Línea de Interrupción INT.

Esta linea se conecta directamente a la linea de entrada de interrupción del **CPU** ó de otro controlador de interrupciones programable.

Línea de Reconocimiento de la Interrupción INTA.

El pulso de reconocimiento de interrupción forzará al PIC, a liberar un vector de información hacia el buffer del bus de datos. El formato de estos datos, depende del modo de operación programado en el PIC.

Buffer del Bus de Datos.

El buffer del bus de datos es utilizado para hacer interface con el bus de datos de la computadora. Este cuenta con un circuito de tres estados (three state) y tiene capacidad de procesamiento de 8 bits. Las palabras de control e información de estado son transferidas a través de éste.

Lógica de Control para Lectura/Escritura.

La función de este bloque, es la de aceptar órdenes de inicialización y operación desde el CPU. En éste se encuentran los registros de inicialización *(ICW Initialization Command Word)* y de operación *(OCW Operation Command Word)*, los cuáles almacenan los diferentes formatos de control del dispositivo y permiten la transferencia de los estados del PIC hacia el Bus de datos.

Habilitación del Dispositivo. (CS)

Un nivel bajo en la entrada de esta línea habilita al dispositivo. Ninguna lectura ó escritura será permitida a menos que dicho dispositivo sea habilitado.

Escritura (WR).

Un nivel bajo en esta entrada, habilita al CPU para escribir órdenes de inicialización y control *(ICW y OCW)* en el PIC.

Lectura (RD).

Un nivel bajo en esta entrada, habilita al PIC para transferir cualquiera de los diferentes estados de operación. (*IRR, ISR, IMR* ó el nivel de interrupción hacia el bus de datos).

A_0.

Esta señal de entrada, es utilizada en conjunto con las señales de *WR y RD* para escribir órdenes en los registros de inicialización y control, así como para realizar las lecturas de los registros de estado del dispositivo. Esta línea, puede ser conectada directamente a una de las lineas de dirección del bus del sistema.

El Buffer/Comparador de Cascada.

La función que realiza este bloque es la guardar y comparar las distintas identificaciones de los PIC'S 82C59A utilizados en el sistema. Las 3 lineas de entrada/salida asociadas con el bloque *(CAS0, CAS1 y CAS2)* son *salidas* cuando el PIC es utilizado como *maestro* y son *entradas* cuando el *PIC* es utilizado como *esclavo*

Secuencia de Interrupciones.

La secuencia normal de eventos que ocurren durante una interrupción, depende del tipo de **CPU** utilizado. El PIC está diseñado para hacer interface con microprocesadores de la familia *MCS-80/85* y con los de la familia *MCS-CX86/88*. Para nuestro caso, nos avocaremos únicamente al estudio de la forma de operación del PIC con la familia *MCS-CX86/88.*

1.- Una ó más de las señales de solicitud de interrupción (**IR7-0**) alcanzan el valor de nivel alto, activando el bit correspondiente en el registro de solicitud de interrupción (**IRR**).

2.- El PIC evalúa la solicitud y envía un pulso de solicitud de interrupción (**INT**) al *CPU* si tal solicitud procede.

3.- El CPU recibe la solicitud de interrupción y responde con un pulso de reconocimiento de interrupción (**INTA**).

4.- El PIC al recibir el pulso de reconocimiento desde el CPU, analiza los niveles de prioridad programados en él, activa el bit correspondiente en el registro de en-servicio (**ISR**) y desactiva el bit en el registro de solicitud de interrupción (**IRR**). El PIC no controla el bus de datos durante este ciclo.

5.- El CPU inicia un segundo pulso de reconocimiento de interrupción y durante este pulso, el PIC transfiere un vector de **8 bits** hacia el bus de datos donde será leído por el CPU.

6.- Esto completa un ciclo de interrupción. En el modo de fin de interrupción automático (**AEOI**) el bit del registro en-servicio será desactivado al final del segundo pulso de reconocimiento de interrupción, de otra manera el bit estará activo hasta que el **CPU** le envíe una orden de fín de interrupción al final de la rutina de servicio de interrupción.

Configuración del Vector de Interrupciones del PIC.

La versatilidad del PIC, radica en su enorme flexibilidad para el direccionamiento de las rutinas de servicio de interrupción. El PIC entrega un vector de interrupción al CPU, a través del cual se indica al microprocesador qué rutina de servicio de interrupciones debe realizar la atención de la misma.

El primer pulso de reconocimiento de interrupción que envía el CPU al PIC, causa el congelamiento del estado de interrupciones para la resolución de prioridades. Durante el segundo pulso de reconocimiento de interrupción, el PIC coloca el valor del vector de interrupción en el bus de datos. La composición del vector de interrupción se detalla a continuación:

IR	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
7	T_7	T_6	T_5	T_4	T_3	1	1	1
6	T_7	T_6	T_5	T_4	T_3	1	1	0
5	T_7	T_6	T_5	T_4	T_3	1	0	1
4	T_7	T_6	T_5	T_4	T_3	1	0	0
3	T_7	T_6	T_5	T_4	T_3	0	1	1
2	T_7	T_6	T_5	T_4	T_3	0	1	0
1	T_7	T_6	T_5	T_4	T_3	0	0	1
0	T_7	T_6	T_5	T_4	T_3	0	0	0

Byte del Vector de Interrupción

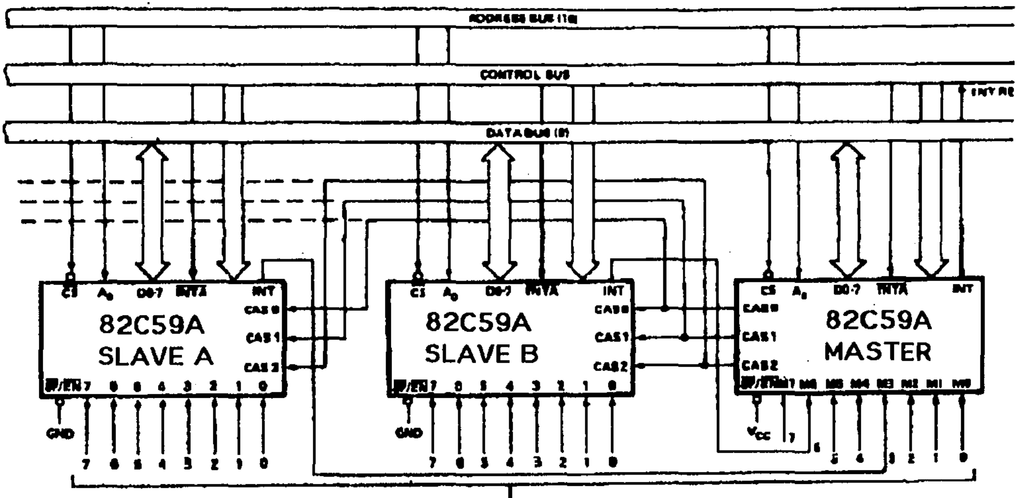
Los valores **T_7-T_3** son programados durante la inicialización del PIC, de acuerdo a los números disponibles para las rutinas de atención en el sistema. Los valores que corresponden a las líneas D_2-D_0, se insertan automáticamente. Si los valores de T_7-T_3 se programan con *ceros*, el primer valor del vector de interrupciones (IR=0) será 0 y el último 7. Si los valores de T_7-T_3 se programan con *unos*. El primer valor del vector de interrupción (IR=0) será 248 y el último (IR=7) 255. Con ésto, el programador tiene la flexibilidad de mover su banco de 8 rutinas de atención a un bloque disponible dentro de la tabla de interrupciones del sistema.

El PIC, proporciona **3** lineas con las cuáles es posible conectar en cascada hasta **8** PIC's adicionales, que expanden el número de interrupciones hasta un máximo de **64** líneas sin hardware adicional. Este método de expansión es llamado encadenamiento.

Modo de Encadenamiento.

Cuando un conjunto de PIC's se conectan en modo de encadenamiento, existe un PIC maestro que controla la operación hasta de 8 PIC's secundarios que operan como esclavos.

En la siguiente figura se muestra un diseño, el cual contiene 22 niveles de solicitudes de interrupción.



Encadenamiento de PIC'S. 22 Niveles de Interrupción

Observe que el PIC habilitado como maestro, está especificado por un nivel alto en el pin definido como SP/EN, mientras que este mismo pin mantiene un nivel bajo para los que funcionan como esclavos. Las señales de salida de interrupción (**INT**) de los dispositivos esclavos, están conectadas a las entradas de solicitud de interrupción (**IR**$_{0-7}$) del PIC maestro. Las líneas CAS0, CAS1 y CAS2 están conectadas en paralelo en los 3 dispositivos, funcionando como salidas si el PIC opera como maestro y como entradas si el PIC opera como esclavo. Con estas lineas se habilita a uno de los dispositivos esclavos, para que tome el control del bus del sistema. El dispositivo seleccionado como maestro debe recibir durante la inicialización cuáles de sus entradas de solicitud de interrupción están conectadas a las salidas de interrupción (**INT**) de los dispositivos esclavos. Los PIC's esclavos por su parte, deberán de asociarse a un número de identificación (0 hasta 7), el cuál corresponderá a la línea de solicitud de interrupción del PIC maestro.

Cada uno de los dispositivos (maestro y esclavos), deberán ser configurados, para dar a cada entrada de solicitud de interrupción un vector único de servicio.

Modos de Operación del PIC.

El PIC, dependiendo de las características de los dispositivos que deberá controlar, puede programarse para trabajar en los siguientes modos de operación:

a. Modo de fully nested.
b. Modo de rotación de prioridades.
c. Modo de máscara especial (Special Mask).
d. Modo de sondeo (Poll Mode).

A continuación, serán mencionados los ambientes de operación recomendados para cada modo, con especial énfasis en el fully nested y en el modo de sondeo, que son los que mejor se adaptan a los requerimientos propios del sistema de adquisición.

Modo Fully Nested.

Este modo se activa, después de que los comandos de inicialización han sido generados. Al terminar la secuencia de inicialización, **IR_0** tiene la mayor prioridad e **IR_7** la menor.

Cuando una solicitud de interrupción es reconocida, se determina la prioridad de dicha solicitud y su vector de interrupción se habilita en el bus de datos. El bit correspondiente al registro de en-servicio (**ISR**) se activa. Este bit permanece activo bajo dos condiciones: hasta que el CPU emite un comando de fin de interrupción (**EOI**) ó en el borde de finalización (trailing edge) del último **INTA**, en el caso de que el bit de fin de interrupción automático (**AEOI**) esté activo. Mientras el bit del registro de en-servicio (ISR) está activo, todas las solicitudes de interrupción de *la misma ó menor prioridad son deshabilitadas.*

Fin de Interrupción (EOI).

Los bits del registro de en-servicio del PIC 82C59A pueden ser reinicializados automáticamente (cuando el bit **AEOI = 1** en el comando de inicialización número 4 **ICW4**), siguiendo el borde de finalización (trailing edge) del último pulso de reconocimiento de interrupción ó por una orden dirigida hacia el PIC antes de regresar de la rutina de servicio (comando de fin de interrupción). El comando de fin de interrupción, deberá de ser doble en el caso de sistemas en cascada, uno para el PIC maestro y otro para el PIC esclavo.

Hay 2 clases de comandos de fin de interrupción: específicos y no específicos. Un comando de fin de interrupción específico, puede ser recibido en el PIC por medio del comando de control número 2 (**EOI = 1, SL = 1, R = 0** y los bits **L2, L1** y **L0**) indicando el nivel binario del bit del registro de en-servicio a ser reinicializado. Para el caso del fin de interrupción no específico, el PIC automáticamente reinicializa el bit de en-servicio con mayor prioridad ó a los que estén activos (**EOI = 1, SL = 0** y **R = 0**). El modo de fin de interrupción automático (AEOI = 1), se considera como un comando de fin de interrupción no específico y solo es aplicable a PIC maestros.

Rotación de Prioridades.

Existen dos modos de operación de **rotación de prioridades: automática** (dispositivos con la misma prioridad) y **rotación específica.** En la rotación automática, si un dispositivo acaba de recibir servicio, se le asigna la menor prioridad y al siguiente la mayor prioridad, de tal manera que si un dispositivo solicita servicio, deberá de esperar en el peor de los casos hasta que los otros 7 hayan recibido servicio una vez, dicho modo es habilitado por el comando de control número 2 OCW2 (R = 1, SL = 0 y EOI = 0) y deshabilitado por el mismo comando (R = 0, SL = 0 y EOI = 0). Para la rotación específica, el programador cambiará la prioridad de las solicitudes dependiendo de su aplicación. Este modo se habilita con el comando OCW2 (R= 1, SL = 1. Mediante los bits L2, L1 y L0 se indica la línea de interrupción a la que se le asignará la menor prioridad).

Programación de la Detección por Flanco ó por Nivel en las líneas de Entrada de Solicitud de Interrupción.

Este modo es programado usando el bit 3 en ICW1. Si **LTIM = 0**, la solicitud de interrupción será detectada por un cambio de nivel de bajo a alto en las entradas **IR0-7**. La entrada **IR0-7** puede permanecer en un nivel alto sin generar otra interrupción.

Si **LTIM = 1** la solicitud de interrupción será detectada por un nivel alto en las entradas **IR0-7**. La solicitud de interrupción debe quitarse antes de que el comando de fin de interrupción se genere ó de que la linea de interrupción del CPU se habilite, para prevenir una repetición de la solicitud de interrupción innecesaria.

Modo de Máscara Especial.

Este modo se utiliza cuando se desea deshabilitar exclusivamente un nivel de interrupciones, de tal forma que interrupciones de mayor o menor prioridad al nivel inhabilitado, puedan seguir reconociéndose.

Modo de Sondeo (POLL MODE).

Cuando el PIC recibe el comando de control número 3 (OCW3) con el bit **P = 1**, entra a operar en modo de sondeo. A la siguiente operación de lectura que se realice sobre él, entregará en el bus de datos un byte con el número de línea de interrupción que solicita servicio.

La desventaja de este método, radica en que las interrupciones son congeladas desde el comando de escritura (**WR**), hasta el comando de lectura (**RD**), en la siguiente figura se muestra el contenido de dicho byte.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
I	---	---	---	---	W_2	W_1	W_0

W_2-W_0: Código binario de la entrada que requiere servicio (IRQx).

I: Igual a **1** si existe alguna solicitud de interrupción en sus entradas **IR_{0-7}**.

El modo de sondeo es utilizado, cuando se requiere expandir el número de entradas para solicitud de interrupciones y no es posible utilizar el modo de encadenamiento, ni el comando de reconocimiento de interrupción por parte del CPU.

Modo Especial de Fully Nested.

El modo especial de fully nested, se utiliza en diseños de grandes sistemas donde el modo de encadenamiento es usado y la prioridad de cada PIC esclavo debe ser conservada.

Modo Buffered.

Este modo es recomendado en los grandes sistemas, donde buffers manejadores son necesarios y el modo de encadenamiento es utilizado. En este modo, siempre que se habilita el bus de datos del PIC, la señal SP/EN se activa en nivel bajo. Esta señal puede utilizarse para habilitar un transceiver, que permita la transferencia de datos en la dirección requerida.

Programación de Inicialización y Control de Operación del PIC.

El controlador de interrupciones programables acepta dos tipos de comandos generados por el CPU que son:

1.- Comandos de inicialización (**ICW**). Cada controlador de interrupciones utilizado en el sistema, deberá llegar a un punto de arranque, mediante una secuencia de 2 a 4 bytes emitidos por el CPU.

2.- Comandos de operación (**OCW**). Estos hacen que el *PIC* opere en alguno de los diversos modos de operación antes mencionados. Los comandos de operación pueden ser escritos en el PIC en cualquier momento después de su inicialización.

Comandos de Inicialización.

Siempre que el CPU genera un comando con la líneas $\mathbf{A_0 = 0}$ y $\mathbf{D_4 = 1}$, el PIC lo interpreta como el comando de inicialización número 1 (ICW1). Este comando empieza la secuencia de inicialización, durante la cual ocurre automáticamente lo siguiente:

a. El circuito sensor de flancos es inicializado, lo cual significa que una entrada de solicitud de interrupción (**IR_0-IR_7**), deberá de tener una transición de nivel bajo a alto para generar una interrupción.

b. El registro de interrupciones mascarables (IMR) se configura habilitando todas las entradas de solicitud de interrupción (con todos sus bits igualados a cero).

c. Se asigna a la entrada de solicitud de interrupción número 7 (IR_7) la prioridad más baja (7).

d. Se asigna un 7 a la dirección del modo de dispositivo esclavo.

e. Se desactiva el modo de máscara especial (special mask) y el status de lectura se iguala al registro de solicitud de interrupciones.

f. Si el bit **IC4 = 0**, todas las funciones seleccionadas por el comando de inicialización número 4 (ICW4) son igualadas a cero.

Comandos de Inicialización 1 y 2.

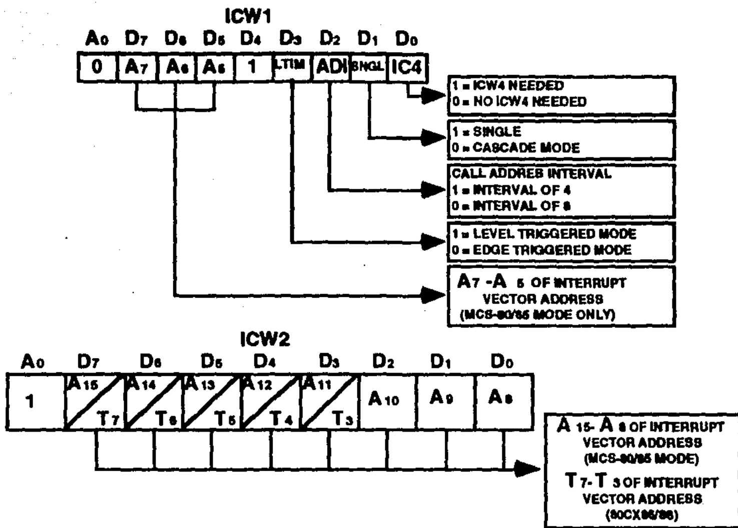
A_5-A_{15}: Dirección de inicio de rutinas de servicio. En un sistema *MCS-CX86/88*, los bits **A_{15}-A_{11}** son insertados en los **5** bits más significativos del byte de dirección y los **3** bits menos significativos son asignados por el dispositivo de acuerdo al nivel de interrupción activo. **A_{10}-A_5** son ignorados y el bit ADI (address Interval) no tiene efecto.

LTIM: Si este bit está activo, entonces el PIC operará en el modo de detección de solicitud de interrupción por nivel y el modo de detección por flanco de las entradas de interrupción será deshabilitado.

ADI: Intervalo de la dirección de llamada. Si **ADI = 1** el intervalo será de **4**. Si **ADI = 0** el intervalo será de **8**.

SNGL: Quiere decir que sólo existe un PIC en el sistema. Si **SNGL = 1** no se generará el comando de inicialización número 3.

IC4: Si el bit está activo, significa que el comando de inicialización número 4 deberá ser leído, si es desactivado, entonces no es necesario generar dicho comando.

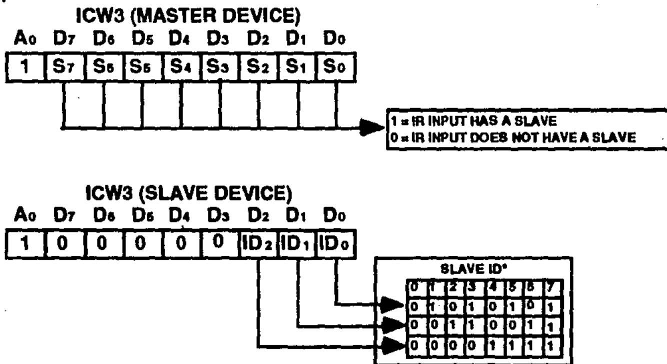


Comando de Inicialización 3.

Este comando, se usa cuando existe más de un PIC en el sistema y se utiliza el modo de encadenamiento. Las funciones de este comando son:

a. En el modo de **maestro** (cuando el pin **SP = 1** ó en el modo de **Buffered** cuando **M/S = 1** en **ICW4**) un "**1**" es puesto por cada dispositivo **esclavo** en el sistema.

b. En el modo de **esclavo** (**SP = 0**, ó si **BUF = 1** y **M/S = 0** con **ICW4**) los bits **2-0** asignan el número de identificación del dispositivo esclavo.



NOTE : SLAVE ID is equal to the corresponding master IR input

Comando de Inicialización 4.

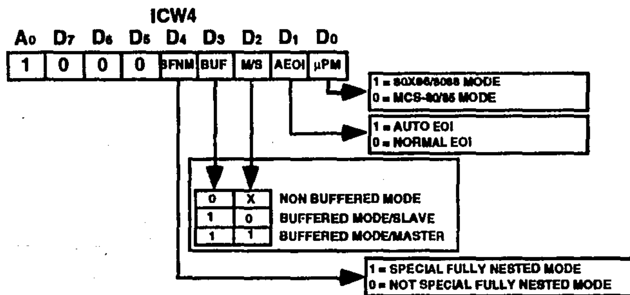
SFNM: Si **SFNM** = **1** se usa el modo especial de fully nested.

BUF: Si **BUF** = **1** el modo de buffered es programado. En el modo de buffered **SP/EN** llega a ser una habilitación de salida.

M/S: Si el modo de buffered es seleccionado: **M/S** = **1** significa que el PIC 82C59A se usa como maestro y si el **M/S** = **0**, estonces será como esclavo; pero si el bit **BUF** = **0**, entonces el bit **M/S** no tiene validez.

AEOI: Si **AEOI** = **1**, se programa el modo de fin de interrupción automática.

µPM: Modo de microprocesador. Si **µPM** = **0**, el PIC operará para un sistema **MCS-80/85** y si **µPM** = **1**, el PIC operará para un sistema **MCS-CX86/88**.

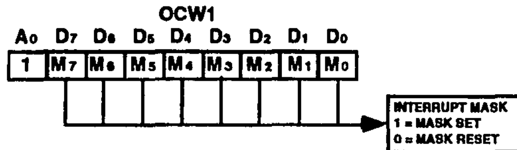


Comandos de Operación del PIC.

Ya programados los comandos de inicialización, el PIC queda listo para recibir las solicitudes de interrupción en sus líneas de entrada (**IR0-IR7**). Sin embargo, el PIC ofrece varias estrategias de asignación de prioridades, las cuáles pueden ser seleccionadas mediante la generación de comandos de operación al PIC (**OCW**).

Comando de Control de Operación 1.

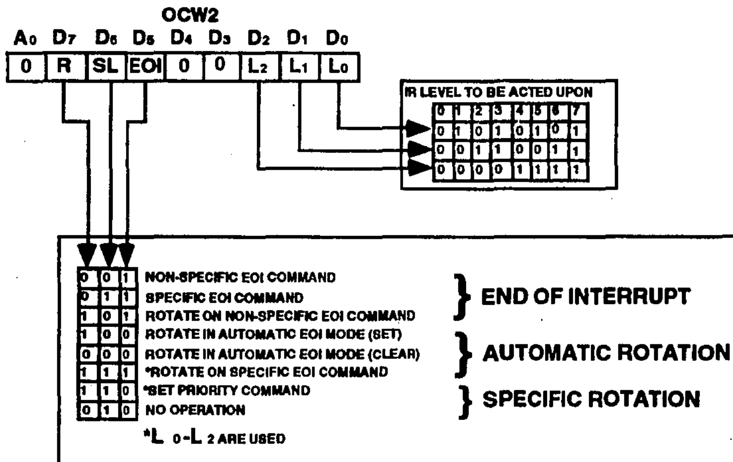
Este comando activa y desactiva los bits de máscara en el registro de interrupciones mascarables. Los bits **M_7-M_0** representan a los bits mascarables. Si **Mx = 1**, indica que el canal **x** está desactivado (deshabilitado) y si **Mx = 0** significa que el canal está activo (habilitado).



Comando de Control de Operación 2.

R, SL, EOI: Con estos tres bits se controlan los modos de rotación y fin de interrupción, así como las combinaciones de ambos.

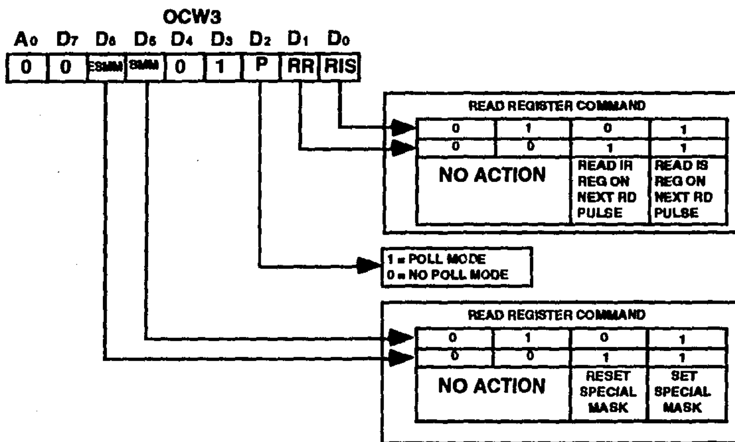
L_2, L_1, L_0: Con estos bits se determina el nivel de interrupción cuando el bit **SL** está activo.



Comando de Control de Operación 3.

ESMM: Habilita el modo de máscara especial. Cuando este bit es activo habilita el bit **SMM** para reinicializar el modo de máscara especial y cuando **ESMM = 0**, el bit **SMM** no tiene validez.

SMM: Modo de máscara especial. Si **ESMM = 1** y **SMM = 1** el PIC 82C59A iniciará el modo de máscara especial y si **ESMM = 1** y **SMM = 0** el PIC 82C59A volverá al modo de máscara normal.



En el diagrama de tiempos de la **figura 5**, que se anexa en el **apéndice A**, se pueden apreciar los rangos de operación del PIC 82C59A para los comàndos de lectura y escritura.

Uno de los problemas que se debían de resolver para la implementación de la tarjeta de adquisición de datos, tenía relación con la forma como el PIC 82C59A-2 debería de interactuar con los

PICs existentes en la tarjeta principal de la computadora. En primer lugar las líneas de conexión CAS0-CAS2 con que cuentan los PICs de la tarjeta principal, no forman parte del bus que llega a las ranuras de expansión del sistema, por lo que no existe la posibilidad de conectar un tercer PIC en cascada desde una interface. En segundo lugar la línea de INTA (Interrupt Acknowledge) tampoco llega al bus del sistema. Esta línea es necesaria para indicar al PIC que la interrupción por él generada ha sido reconocida por el CPU. Sin INTA no existe una forma directa de indicarle al PIC que su requerimiento ha sido reconocido. Por las causas anteriores, el modo de operación seleccionado tuvo que ser por sondeo. En él, el PIC genera la interrupción y el CPU responde con un comando de escritura (sondeo). Este comando ocasionará que el PIC entregue a la siguiente operación de lectura el número de interrupción que necesita servicio. El inconveniente de este método es que las detecciones de solicitud de interrupción son congeladas entre la emisión del comando de sondeo y del comando de lectura realizados por el CPU. Este problema nos obligó a utilizar la detección por nivel en lugar de la detección por flanco (transición de bajo a alto).

La programación de inicialización y de operación del PIC 82C59A-2 del controlador de interrupciones de la tarjeta de interface, quedó como a continuación se describe:

```
mov  dx,Pic       ; External PIC address
mov  al,1Ah       ; ICW1 Level triggerd, single & ICW4 no needed
out  dx,al
mov  al,00h       ; ICW2
inc  dx
out  dx,al
mov  al,00h       ; OCW1 Reset Interrupt Mask Register
out  dx,al
dec  dx
mov  al,E0h       ; OCW2 Rotate on specific EOI IRQ0 = 0
out  dx,al
```

Toda la anterior programación, será descrita en detalle mas adelante; en el capítulo de "Diseño de la Tarjeta de Recepción de Datos", por el momento solo queda como referencia.

Programación de Inicialización y Control de Operación del UART SCC2698B.

Entre las principales características técnicas con que cuenta el UART OCTAL **SCC2698B** de **SIGNETICS** están:

- 8 receptores/transmisores asíncronos con comunicación full duplex.

- Buffer de 4 registros de datos por cada canal de recepción.

- Formato de datos programable:

  - Tamaño de datos de 5 a 8 bits más paridad por caracter.
  - Par, non, sin paridad ó paridad forzada.
  - Tamaño de bit de parada programable de 1, 1.5 ó 2 con incrementos de 1/16 bit.

- Velocidad seleccionable para los receptores y transmisores:

  - 18 velocidades definidas de **50** hasta **38400** bauds.
  - Velocidades no comerciales hasta **115200** bauds.
  - Velocidades definidas por el usuario, derivadas de un counter/timer programable asociado a cada uno de los 4 bloques del UART.
  - Reloj externo de **1X** ó **16X**.

- Detección de **error** de: **paridad**, **formación** (framing) y de **desbordamiento** (overrun).

- Detección de bit de arranque falso.

- Detección y generación de pausa de línea.

- Modo de canal programable:

  - **Normal** (Full-duplex), **repetición automática**, **local loop back** y **remote loop back**.

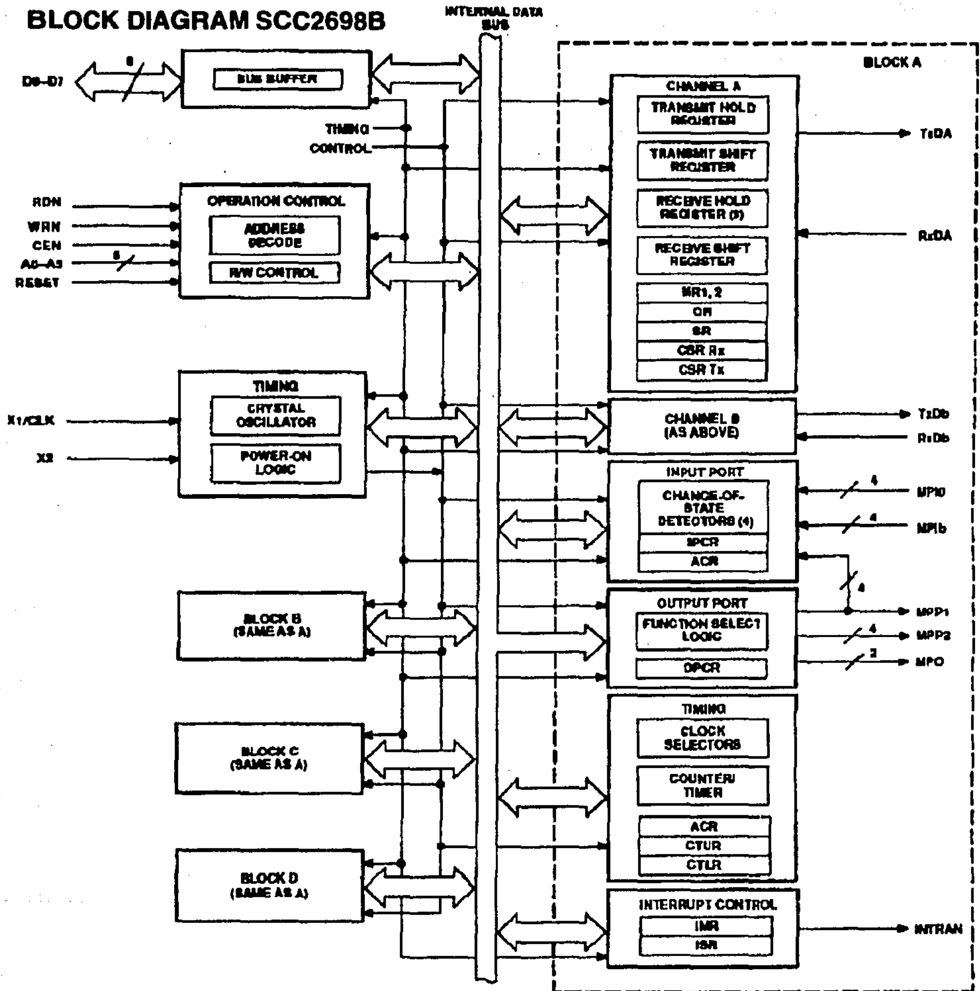
- 4 **counter/timers** multifunción de **16 bits** programables.

- 4 líneas de salida de interrupción (**INTRN**) con 8 condiciones de interrupción mascarable para cada salida.

- Bandera de estado para los receptores (**READY** ó **FFULL**) y transmisores (**READY**) en 16 líneas de salida de multifunción en el paquete de **PLCC** del dispositivo.

- Oscilador de cristal integrado.

- Compatible con lógica **TTL** y fuente de alimentación única de **+5** volts.



Como se observa en el diagrama de bloques, el UART OCTAL SCC2698B, consta de los siguientes elementos: un buffer para hacer interface con el bus de datos, control de interrupciones, control de operación, control de tiempos y 8 canales de transmisión y recepción. *Los 8 canales están divididos en 4 diferentes bloques, cada bloque es independiente uno del otro y contienen un par de receptores/transmisores*. La siguiente descripción del bloque **A** es aplicable a los demás bloques.

El Buffer y el Control de Operación.

El buffer del UART, sirve para hacer interface con el bus de datos del sistema. El buffer se controla a través del bloque de control de operación. Este último, contiene decodificadores de dirección y circuitos de lectura y escritura, que permiten la comunicación entre el CPU y el buffer interno de datos.

Control de Interrupciones.

Existe una sola salida de solicitud de interrupción por bloque (**INTRN**), la cual es válida en cualquiera de los siguientes eventos internos:

- Cuando un caracter está listo para ser transmitido por un canal.
- Cuando se recibe un caracter (**READY**) ó el buffer de un canal está lleno (**FFULL**).
- Por el cambio en la condición de pausa de un canal.
- Cuando el contador alcanza el valor especificado.
- Cambio en las entradas del puerto de multifunción (**MPI**).

Asociados con el sistema de interrupciones existen otros 2 registros, que son el registro de interrupciones mascarables (**IMR**) y el registro de estado de interrupción (**ISR**). El IMR determina las condiciones bajo las cuales el UART generará una interrupción. La lectura del registro ISR determina todas las condiciones activas de interrupción.

Circuitos de Tiempo.

Consisten de un oscilador de cristal, un generador de baudaje, un counter/timer programable de 16 bits asociado a cada block del UART y 2 selectores de reloj.

El oscilador de cristal tiene un rango de operación de **2** a **4 Mhz**, pero las velocidades definidas en sus tablas se basan en un cristal de **3.6864 Mhz**. La forma de conectar el cristal es a través de las entradas **X1/CLK** y **X2**. Si un reloj externo es utilizado en lugar del cristal, éste deberá ser conectado a la entrada X1/CLK. X2 quedará sin conexión como se muestra en la **figura 6** del **apéndice A**. El reloj sirve como referencia de tiempo para: el generador de baudaje, el counter/timer y otros circuitos internos.

El *generador de baudaje*, toma como base para su operación la señal del oscilador ó del reloj externo y es capaz de generar *18 velocidades* de datos comerciales, que van desde los **50** hasta los **38400 bauds**; 13 de éstos baudajes, están simultáneamente disponibles para ser usados por los receptores y transmisores, **8** son fijos y uno de los **2** conjuntos de **5**, puede ser seleccionado por programación del bit **ACR[7]**.

La salida del reloj del generador (**BRG**) es a razón de **16X** la velocidad seleccionada. El counter/timer puede ser utilizado como un timer, para producir un reloj de 16X que permita el uso de cualquier otra velocidad.

El **selector del reloj**, permite la selección independiente para el receptor y el transmisor de cualquiera de los baudajes especificados ó de un baudaje especial, generado por una señal externa de tiempo.

Existen **4 counters/timers** en el UART OCTAL, uno por cada bloque; la operación de éstos es programada por los bits del registro **ACR[6:4]**. Una de las **8** fuentes de tiempos puede ser usada como entrada de reloj para el counter/timer.

La salida del counter/timer está disponible a través del selector del reloj y puede ser programada por los bits del registro **OPCR[2:0]** para el canal **a** y los bits **OPCR[6:4]** para el canal **b**, las salidas de dichas señales estarán en los pines **MPOa** y **MPOb** respectivamente.

En el modo de **timer**, el counter/timer genera una señal cuadrada con un periodo, que es el doble del periodo del reloj definido en los registros de parte alta y baja (**CTUR** y **CTLR**) de los counter/timer. El bit de contador listo (**counter ready**) del registro de estado de interrupción, se activa una vez en cada ciclo de la señal cuadrada generada.

Si el valor de los registros de parte alta y baja de los counter/timers cambia, la mitad del periodo corriente no se verá afectada, pero a partir de ese momento, los periodos subsecuentes reflejarán los nuevos valores almacenados en esos registros. En éste modo, el counter/timer funcionará continuamente y no reconocerá el comando de parada (dicho comando solo reinicializa el bit del contador listo del registro ISR). Al recibir un comando de arranque, causa que el contador termine un ciclo é inicie uno nuevo utilizando los valores contenidos en los registros CTUR y CTLR.

En el modo de **contador**, el counter/timer cuenta en forma regresiva el número de pulsos almacenados en los registros CTUR y CTLR. La cuenta empieza a partir de la recepción del comando de arranque, una vez que se alcanza la cuenta especificada, el bit de contador listo del registro ISR se activa y continúa contando hasta que es detenido por el CPU.

Si el puerto de multifunción (MPO) se programa para operar como salida de los counter/timers, esta salida, permanece en un nivel alto hasta que se llega a la cuenta especificada. En este momento el nivel cambia a bajo. La salida regresa a nivel alto y el bit de contador listo es desactivado cuando el contador es detenido por el comando de parada. Si los nuevos valores de los registros CTUR y CTLR no han sido cargados, los valores previos son preservados y utilizados para el siguiente ciclo de conteo. En este modo, el valor corriente de los 8 bits de los registros CTUR y CTLR puede ser leído por el CPU. Sin embargo se recomienda, que el contador sea detenido para prevenir problemas potenciales que pueden ocurrir, si hay un acarreo desde los 8 bits menos significativos hacia los 8 bits más significativos, entre los tiempos en que se leen ambas mitades del contador.

El UART, cuenta con ocho canales receptores/transmisores asíncronos. La frecuencia de operación del receptor y del transmisor puede seleccionarse independientemente a través del generador de baudaje del counter/timer ó a través de una señal externa.

Los registros asociados con los canales de comunicación son: los registros de modo (**MR1** y **MR2**), el registro selector de reloj (**CSR**), el registro de comando (**CR**), el registro de estado (**SR**), el registro de retención de transmisión (**THR**) y el registro de retención de recepción (**RHR**).

En las tablas siguientes, se muestran las direcciones que debe generar el CPU en sus salidas, para lograr el acceso a cualquiera de los registros de control y operación asociados a cada uno de los 8 canales del UART.

CANALES a y b							
A5	**A4**	**A3**	**A2**	**A1**	**A0**	**READ (RDN = 0)**	**WRITE (WRN = 0)**
0	0	0	0	0	0	MR1a, MR2a	MR1a, MR2a
0	0	0	0	0	1	SRa	CSRa
0	0	0	0	1	0	Reservado*	CRa
0	0	0	0	1	1	RHRa	THRa
0	0	0	1	0	0	IPCRA	ACRa
0	0	0	1	0	1	ISRA	IMRA
0	0	0	1	1	0	CTUA	CTURA
0	0	0	1	1	1	CTLA	CTLRA
0	0	1	0	0	0	MR1b, MR2b	MR1b, MR2b
0	0	1	0	0	1	SRb	CSRb
0	0	1	0	1	0	Reservado*	CRb
0	0	1	0	1	1	RHRb	THRb
0	0	1	1	0	0	Reservado*	Reservado*
0	0	1	1	0	1	Input Port A	OPCRA
0	0	1	1	1	0	Star C/T A	Reservado*
0	0	1	1	1	1	Stop C/T A	Reservado*

CANALES c y d							
A5	A4	A3	A2	A1	A0	READ (RDN = 0)	WRITE (WRN = 0)
0	1	0	0	0	0	MR1c, MR2c	MR1c, MR2c
0	1	0	0	0	1	SRc	CSRc
0	1	0	0	1	0	Reservado*	CRc
0	1	0	0	1	1	RHRc	THRc
0	1	0	1	0	0	IPCRB	ACRc
0	1	0	1	0	1	ISRB	IMRB
0	1	0	1	1	0	CTUB	CTURB
0	1	0	1	1	1	CTLB	CTLRB
0	1	1	0	0	0	MR1d, MR2d	MR1d, MR2d
0	1	1	0	0	1	SRd	CSRd
0	1	1	0	1	0	Reservado*	CRd
0	1	1	0	1	1	RHRd	THRd
0	1	1	1	0	0	Reservado*	Reservado*
0	1	1	1	0	1	Input Port B	OPCRB
0	1	1	1	1	0	Star C/T B	Reservado*
0	1	1	1	1	1	Stop C/T B	Reservado*

CANALES e y f							
A5	A4	A3	A2	A1	A0	READ (RDN = 0)	WRITE (WRN = 0)
1	0	0	0	0	0	MR1e, MR2e	MR1e, MR2e
1	0	0	0	0	1	SRe	CSRe
1	0	0	0	1	0	Reservado*	CRe
1	0	0	0	1	1	RHRe	THRe
1	0	0	1	0	0	IPCRC	ACRe
1	0	0	1	0	1	ISRC	IMRC
1	0	0	1	1	0	CTUC	CTURC
1	0	0	1	1	1	CTLC	CTLRC
1	0	1	0	0	0	MR1f, MR2f	MR1f, MR2f
1	0	1	0	0	1	SRf	CSRf
1	0	1	0	1	0	Reservado*	CRf
1	0	1	0	1	1	RHRf	THRf
1	0	1	1	0	0	Reservado*	Reservado*
1	0	1	1	0	1	Input Port C	OPCRC
1	0	1	1	1	0	Star C/T C	Reservado*
1	0	1	1	1	1	Stop C/T C	Reservado*

CANALES g y h							
A5	A4	A3	A2	A1	A0	READ (RDN = 0)	WRITE (WRN = 0)
1	1	0	0	0	0	MR1g, MR2g	MR1g, MR2g
1	1	0	0	0	1	SRg	CSRg
1	1	0	0	1	0	Reservado*	CRg
1	1	0	0	1	1	RHRg	THRg
1	1	0	1	0	0	IPCRD	ACRg
1	1	0	1	0	1	ISRD	IMRD
1	1	0	1	1	0	CTUD	CTURD
1	1	0	1	1	1	CTLD	CTLRD
1	1	1	0	0	0	MR1h, MR2h	MR1h, MR2h
1	1	1	0	0	1	SRh	CSRh
1	1	1	0	1	0	Reservado*	CRh
1	1	1	0	1	1	RHRh	THRh
1	1	1	1	0	0	Reservado*	Reservado*
1	1	1	1	0	1	Input Port D	OPCRD
1	1	1	1	1	0	Star C/T D	Reservado*
1	1	1	1	1	1	Stop C/T D	Reservado*

Nota: Los registros reservados nunca deberán ser leídos durante la operación normal del dispositivo, ya que son reservados para diagnóstico interno del dispositivo.

ACR = Registro de control auxiliar
CR = Registro de comando
CSR = Registro selector del reloj
CTL = Parte baja counter/timer
CTLR = Registro parte baja counter/timer
CTU = Parte alta counter/timer
CTUR = Registro parte alta counter/timer
MR = Registro de modo
SR = Registro de estado
THR = Registro de retención de transmisión
RHR = Registro de retención de recepción
IPCR = Registro de cambios en puerto de entrada
ISR = Registro de estado de Interrupciones
IMR = Registro de Interrupciones mascarables

Transmisor.

El transmisor acepta datos en paralelo desde el CPU y los convierte en una sucesión de bits en serie para su salida en el pin **TxD**. El transmisor, automáticamente envía un bit de arranque seguido del número de bits de datos programado, un bit de paridad opcional y finalmente los bits de parada especificados. El bit menos significativo es enviado primero. Si después de la transmisión del bit de parada, no existe un caracter nuevo en el registro THR, el pin de salida TxD permanece en un nivel alto y se activa el bit de transmisor vacio **(TxEMT)** en el registro SR. Este bit se desactiva cuando se carga un nuevo caracter en el registro **THR.**

Receptor.

El receptor acepta datos en serie en el pin **RxD**, convierte la entrada serial a un formato en paralelo, checa por el bit de arranque, bit de parada, bit de paridad (si fué programado) ó condición de pausa (break condition) y almacena el caracter en el buffer del canal correspondiente. El receptor busca por una transición de nivel alto a bajo (marca a espacio) del bit de arranque en el pin de entrada RxD. Si se detecta una transición, el estado del pin RxD es muestreado otra vez cada ciclo del reloj de 16X, durante 7.5 ciclos (modo del reloj de 16X) ó en el siguiente flanco de subida del reloj (modo del reloj de 1X). Si RxD está en nivel alto, el bit de arranque es inválido y la búsqueda para un bit de arranque válido empieza de nuevo, si RxD es todavía bajo, entonces es asumido el bit de arranque y el receptor muestrea la entrada. Esto continúa en intervalos de un bit a la vez, en el centro teórico del bit, hasta que el número propio de bits de datos y paridad han sido recibidos y un bit de parada ha sido detectado. El dato posteriormente es transferido al registro **RHR** y se activa el bit **RxRDY** del registro SR. Si la longitud del caracter es menor a 8 bits, los bits más significativos no usados en el registro RHR son puestos a cero.

Los bits de error de: paridad, de formación (framing) y de desbordamiento (overrun), son actualizados en el registro SR en el momento que el caracter es recibido, antes de que el bit de estado RxRDY sea activado.

Si se detecta una condición de pausa (RxD tiene un nivel bajo por el caracter entero incluyendo el bit de parada), solo un caracter consistiendo de ceros será cargado hacia el buffer del FIFO y el estado de pausa recibido (break condition) se activará en el registro SR.

Modo Fuera de Tiempo.

Este modo se usa para notificar al CPU, que ha transcurrido ya el periódo de espera especificado por el usuario, sin que llegue un nuevo caracter. Cada vez que se recibe un caracter, el contador que controla el tiempo máximo de arribo es reinicializado. Si no se recibe un nuevo caracter antes que el contador alcance la cuenta final, el bit ISR[3] se activa y se genera una solicitud de interrupción.

Primero en Entrar al Receptor Primero en Salir del Receptor (FIFO).

El registro de retención de datos (RHR), consiste de un buffer de **FIFO (First Input First Ouptut)** con una capacidad de almacenamiento de **3 caracteres**. Los datos son cargados desde el registro de corrimiento de recepción hacia la primera posición disponible del buffer. El bit RxRDY del registro de estado (SR) se activa cuando uno ó más caracteres están disponibles para ser leídos y el bit de estado de FIFO lleno (**FFULL**) se activa, si las tres posiciones del buffer contienen datos. Cualquiera de estos bits, pueden ser elegidos para generar una solicitud de interrupción. Después del ciclo de lectura, los datos del buffer y sus bits de estado asociados son recorridos, dejando una posición vacía en el buffer del FIFO para poder recibir un nuevo caracter.

Si el buffer del FIFO está lleno y un nuevo caracter es recibido, dicho caracter es guardado en el registro de corrimiento de recepción, hasta que una posición del buffer esté disponible pero si un caracter adicional es recibido mientras este estado existe, entonces el contenido del buffer no será afectado, sin embargo el caracter que previamente había sido almacenado en el registro de corrimiento es perdido y se activa el bit de error de desbordamiento del registro SR.

Modo de Despertar (WAKE-UP).

Con este modo de operación, se proporciona un mecanismo de activación automática para los receptores de un sistema de comunicaciones con procesadores múltiples, a través del reconocimiento de las tramas de direccionamiento. En este modo de operación, una estación maestra transmite un caracter de dirección seguido por caracteres de datos con destino a una estación esclava. Las estaciones esclavas del sistema, cuyos receptores están normalmente desactivados, analizan los datos recibidos de la sucesión, a continuación, sólo la estación direccionada despertará al CPU correspondiente mediante la activación del bit RxRDY. El CPU comparará la dirección recibida con la dirección de la estación y habilitará al receptor si es el caso, para que reciba los caracteres de datos subsecuentes.

Pines de Entrada de Propósitos Múltiples.

Los pines de multipropósito, pueden ser programados como la entrada de alguna de las siguientes funciones del UART: GPI (General Purpose Input) y CTCLK (Counter/Timer External Clock Input). La función de los pines es seleccionada mediante la programación de los registros de control apropiados. Detectores de cambio de estado son proporcionados para cada pin de entrada de multipropósito en cada bloque del UART.

Pines de Salida de Propósitos Multiples.

Estos pueden ser programados, para desempeñar alguna de las siguientes funciones de salida: solicitud para envío de caracter (RTS), salida del counter/timer, salida del reloj del receptor ó del transmisor, salida de TxRDY y la salida de RxRDY/FFULL.

Descripción de los Registros del UART.

A continuación será detallado el formato de cada uno de los registros de control del UART SCC2698B y la manera apropiada de utilizarlos.

MR1 Registro de Modo 1.

El registro MR1, se accesa cuando el registro MR apunta hacia su dirección asociada. El apuntador es direccionado a **MR1** por reinicialización ó por medio de un comando de activación, aplicado a través del registro CR. Después de una operación de lectura ó escritura sobre el registro MR1, el apuntador quedará direccionado al registro **MR2**.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
RxRTS Control	RxINT Select	Error Mode	Parity Mode		Parity type	Bits per Character	
0 = No 1 = Yes	0 = RxRDY 1 = FFULL	0 = Char 1 = Block	00 = With Parity 01 = Force Parity 10 = No parity 11 = Special Mode		0 = Even 1 = Odd	00 = 5 01 = 6 10 = 7 11 = 8	

MR2 Registro de Modo 2.

Después de cualquier acceso a MR1, el apuntador MR direcciona a MR2, el acceso a MR2 no cambia el valor del apuntador.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Channel Mode		TxRTS Control	CTS Enable Tx	Stop bit Length*			
00 = Normal 01 = Auto-Echo 10 = Local Loop 11 = Remote Loop		0 = No 1 = Yes	0 = No 1 = Yes	0 = 0.563 1 = 0.625 2 = 0.688 3 = 0.750	4 = 0.813 5 = 0.875 6 = 0.938 7 = 1.000	8 = 1.563 9 = 1.625 A = 1.688 B = 1.750	C = 1.813 D = 1.875 E = 1.938 F = 2.000

* Añadir medio bit a los valores mostrados arriba de 0 a 7, si el canal es programado para 5 bits de datos por caracter.

CSR Registro de Selección de Reloj.

Cuando se utiliza un cristal ó un reloj externo de entrada de **3.6864 Mhz**, este registro selecciona el reloj del receptor y/ó del transmisor (**CSR[7:4]** son para el receptor y **CSR[3:0]** para el transmisor, como se muestra en la siguiente tabla).

CSR[7:4]	ACR[7] = 0	ACR[7] = 1
0000	50	75
0001	110	110
0010	134.5	38400
0011	200	150
0100	300	300
0101	600	600
0110	1200	1200
0111	1050	2000
1000	2400	2400
1001	4800	4800
1010	7200	1800
1011	9600	9600
1100	38400	19200
1101	Timer	Timer
1110	MP13-16X	MP13-16X
1111	MP13-1X	MP13-1X

Los receptores trabajan con un reloj de **16X** el periodo de la señal de reloj base del UART, a excepción de cuando se programa el registro **CSR[7:4]** = **1111**. Cuando **MPI3** se usa como entrada, **MPI3a** es para el canal **a** y **MPI3b** para el canal **b**. Para los transmisores, cuando se utiliza **MPPI** como entrada, **MPPIa** es para el canal **a** y **MPPIb** para el canal **b**.

CR Registro de Comando.

El valor codificado a través de los bits **CR[7:4]** del registro CR, puede ser utilizado para especificar uno de los siguientes comandos:

0000 Ninguna operación

0001 Reinicializa el apuntador del registro MR; ésto causará que el apuntador del registro MR direccione al registro MR1.

0010 Reinicializa el receptor. Funciona como si se aplicara una reinicialización del equipo, el receptor es deshabilitado y el apuntador del buffer del FIFO se direcciona hacia la primera localidad.

0011 Reinicializa el transmisor. Funciona como si se hubiera aplicado una reinicialización del equipo.

0100 Reinicializa el status de error. Inicializa los bits del registro de status SR[7:4], que corresponden a: pausa recibida (received break RB), error de paridad (PE), error de formación (framing FE) y error de desbordamiento (overrun OE).

0101 Reinicializa el bit de interrupción por cambio de pausa en el registro ISR.

0110 Pausa de arranque. Obliga a la salida de TxD a un nivel bajo (espacio). Si el transmisor está vacío, el arranque de la condición de pausa será retrasado máximo 2 veces del bit. Si el transmisor está activo, la pausa empieza cuando se completa la transmisión del caracter. El transmisor debe estar habilitado para iniciar la pausa de arranque.

0111 Pausa de parada. Obliga a la salida TxD a un nivel alto (marca) dentro de los dos bits siguientes. TxD permanecerá en alto durante el equivalente al tiempo de transmisión de un bit, antes que el siguiente caracter sea transmitido.

1000 RTSN activado. Causará que la salida de RTSN sea activada en lógica negativa (nivel bajo).

1001 RTSN desactivado. Causará que la salida de RTSN sea alto.

1010 Activa el modo de fuera de tiempo. El registro en este canal reinicializará el counter/timer cada vez que un caracter es transferido del registro de corrimiento al registro RHR.

1011 Reservado.

1100 Desactiva el modo de fuera de tiempo.

1101 Reservado

111x Reservado para pruebas de manufactura.

Los bits de **CR[3:0]** sirven para habilitar y/ó deshabilitar los transmisores y receptores.

CR (Registro de Comandos)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Comandos de Miscelanea				Disable Tx	Enable Tx	Disable Rx	Enable Rx
Ver comandos anteriores				0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes

SR Registro de Status del Canal.

En este registro, se activan los bits SR[7:4] cuando existen errores en la recepción. A través de los bits SR[3:0] se indica cuando: el buffer de datos se llena, el registro de transmisión queda vacío y cuando un dato en el transmisor está listo para ser transmitido.

SR (Registro de Estados por canal)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Rec´d Break	Framing Error	Parity Error	Overrun Error	TxEMT	TxRDY	FFULL	RxRDY
0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes

ACR Registro de control auxiliar.

Los bits de este registro sirven para seleccionar uno de los dos conjuntos de baudaje, que pueden ser utilizados por el receptor y el transmisor de cada canal. Así mismo, a través de ACR se determina el modo de operación del counter/timer, selecciona: la fuente del reloj y que bits del registro IPCR causan la activación del bit ISR[7]. A continuación se muestran las tablas con los conjuntos de velocidad, así como el modo de operación y selección de la fuente del reloj.

ACR (Registro de Control Auxiliar)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
BRG Set Select	Counter/Timer Mode and Source			Delta MPI1b INT	Delta MPI0b INT	Delta MPI1a INT	Delta MPI0a INT
0 = set 1 1 = set 0	Ver la siguiente tabla			0 = Off 1 = On	0 = Off 1 = On	0 = Off 1 = On	0 = Off 1 = On

ACR[6:4]	Modo	Fuente del Reloj
000	Contador	**MPI** pin
001	Contador	**MPI** pin dividido por **16**
010	Contador	**TxC-1X** reloj del transmisor
011	Contador	Cristal ó reloj externo **(X1/CLK)** dividido por **16**
100	Timer	**MPI** pin
101	Timer	**MPI** pin dividido por **16**
110	Timer	Cristal ó reloj externo **(X1/CLK)**
111	Timer	Cristal ó reloj externo **(X1/CLK)** dividido por **16**

IPCR Registro de cambio del puerto de entrada.

Los bits de este registro, son activados cuando sus correspondientes pines de entrada sufren un cambio de estado y se desactivan cuando son leídos por el **CPU**.

IPCR (Registro de cambios en el Puerto de Entrada)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Delta MPI1b	Delta MPI0b	Delta MPI1a	Delta MPI0a	MPI1b	MPI0b	MPI1a	MPI0a
0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = Low 1 = High	0 = Low 1 = High	0 = Low 1 = High	0 = Low 1 = High

ISR Registro de Status de Interrupción.

Este registro, proporciona los status de todas las fuentes de interrupción potenciales para cada bloque de control del UART. Estos bits son mascarables por el registro IMR, sin embargo pueden ser leídos por el CPU, sin afectar las salidas de solicitud de interrupción.

ISR (Registro de Estados de Interrupción por Bloque)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
MPI Port Change	Delta BREAKb	RxRDY/ FFULLb	TxRDYb	Counter Ready	Delta BREAKa	RxRDY/ FFULLa	TxRDYa
0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes

IMR Registro de interrupciones mascarables.

La programación del registro, selecciona cuales bits del registro de estado de interrupción (ISR), podrán generar una solicitud de interrupción en los pines de salida (INTRN). Si un bit del registro de estado de interrupción (ISR) está activo y su correspondiente bit del registro de interrupciones mascarables (IMR) también, entonces se activará en lógica negativa la salida correspondiente de INTRN.

IMR (Registro de Interrupciones Mascarables por Bloque)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
MPI Port Change INT	Delta BREAKb INT	RxRDY/ FFULLb INT	TxRDYb INT	Counter Ready INT	Delta BREAKa INT	RxRDY/ FFULLa INT	TxRDYa INT
0 = Off 1 = On	0 = Off 1 = On	0 = Off 1 = On	0 = Off 1 = On	0 = Off 1 = On	0 = Off 1 = On	0 = Off 1 = On	0 = Off 1 = On

CTUR y CTLR Registros de los counter/timers.

Los registros CTUR y CTLR contienen el valor a ser utilizado por los counter/timers, en cualquiera de sus modos de operación. En las siguientes fórmulas, se especifica la manera de calcular la velocidad de transmisión y/ó recepción para cada bloque de control del UART.

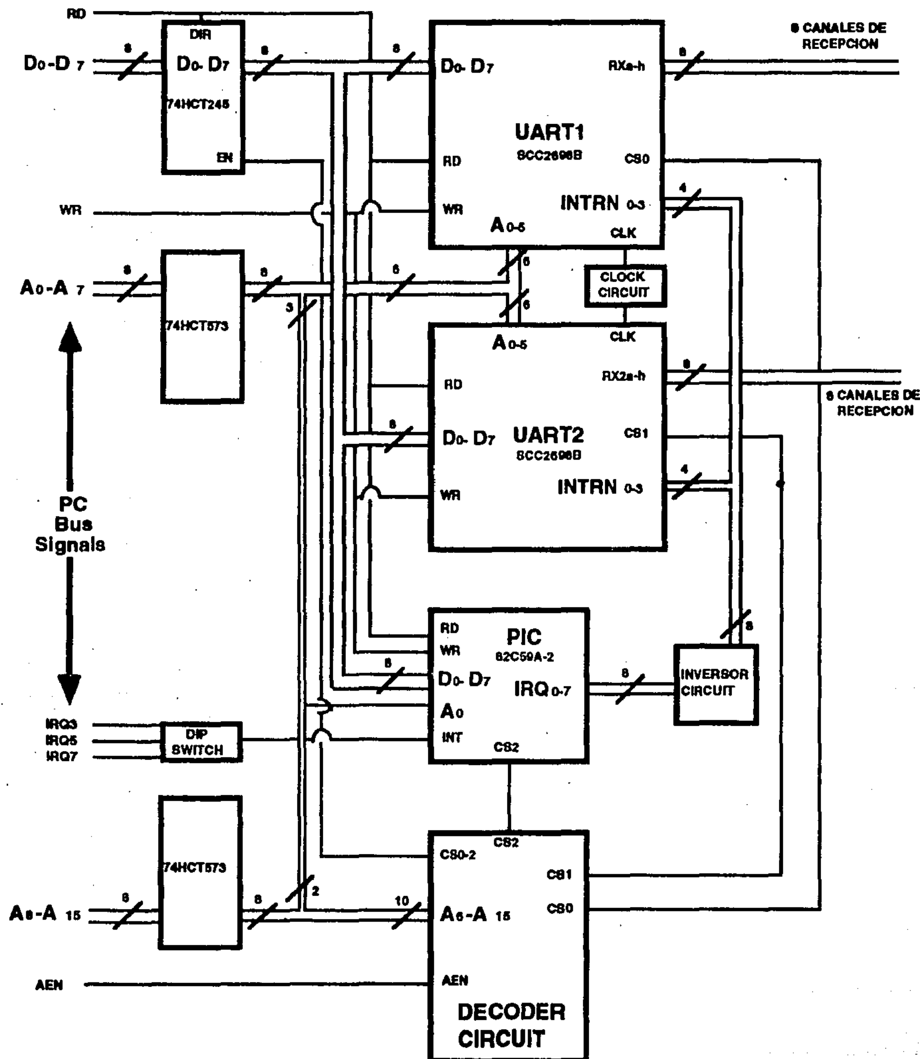
$$V_{BAUD} = \frac{\text{timer clock}}{\text{2 X valor de CTUR/CTLR}}$$ Si ACR[6:4] = 110

$$V_{BAUD} = \frac{\text{timer clock/16}}{\text{2 X valor de CTUR/CTLR}}$$ Si ACR[6:4] = 111

En las **figuras 7** y **8** que se incluyen en el **apéndice A**, se muestran los límites de tiempos mínimos y máximos para accesos de lectura y escritura de los dispositivos **SCC2698B**.

Diseño de la Tarjeta de Recepción de Datos SAD.

El diseño electrónico de la tarjeta para recepción de 16 canales de datos digitales, se ilustra en el siguiente diagrama de bloques.



Del lado izquierdo del diagrama anterior, se encuentran las señales eléctricas del bus del sistema de la computadora, las cuales son utilizadas para la operación de la interface.

El circuito decodificador, utiliza 10 líneas de dirección de la A_6 a la A_{15}, para direccionar y habilitar los puertos de I/O asociados al PIC y a los dos UARTs de la tarjeta.

En el circuito decodificador, existe un banco de selectores no ilustrado, a través del cuál, es posible cambiar el bloque de puertos de I/O asociados a la tarjeta, para evitar conflictos potenciales con otras interfaces.

El banco de selectores, se compone de 8 switches que determinan el bloque de direcciones donde operará la tarjeta. Las líneas de direccionamiento asociadas a cada switch van desde la línea A_8 hasta la A_{15}. De los 8 switches sólo 7 pueden ser utilizados, por estar uno de ellos conectado con la línea A_9, misma que siempre tiene un nivel alto de salida cuando se direcciona una interface externa. Así quedan 7 switches, con los cuales es posible direccionar 128 bloques diferentes de I/O (2**7=128).

A15	A14	A13	A12	A11	A10	A9	A8	A7-A6	A5-A0
0	0	0	0	0	0	1	0	X	X
0	0	0	0	0	0	1	1	X	X
0	0	0	0	0	1	1	0	X	X
0	0	0	0	0	1	1	1	X	X
0	0	0	0	1	0	1	0	X	X
0	0	0	0	1	0	1	1	X	X
0	0	0	0	1	1	1	0	X	X
0	0	0	0	1	1	1	1	X	X
0	0	0	1	0	0	1	0	X	X
0	0	0	1	0	0	1	1	X	X

Como se observa, cada bloque de direcciones, seleccionado a través de los switches, tiene una separación de 256 localidades de I/0 para 2 bloques contiguos y 512 hacia los 2 siguientes bloques, debido a que la linea A_9 siempre deberá estar en un nivel alto. El bloque de puertos más bajo disponible, para el direccionamiento de la interface comienza a partir de la localidad **0200h**.

Las líneas A_6 y A_7 entran a un multiplexor perteneciente al circuito de decodificación, con ellas se activa el chip select del PIC y/ó de los UART's (líneas **CS0**, **CS1** y **CS2**).

Las líneas A_0-A_5, direccionan hasta 64 puertos del dispositivo seleccionado con las líneas A_6 y A_7. Se recordará que cada UART necesita de 64 puertos de I/O para su configuración y operación.

La línea de solicitud de servicio de interrupción IRQx, que utilizará la interface se selecciona por medio de un banco de 3 switches, donde están conectadas las líneas **IRQ3**, **IRQ5** é **IRQ7**; ésto evita posibles conflictos con otras interfaces.

El circuito de reloj genera una señal cuadrada cuya frecuencia es de **3.6864 Mhz**. Esta sirve como base de tiempo para el **UART1** y el **UART2**.

Debido a que las líneas de interrupción del UART son de **pozo-abierto** (open-drain), es necesario agregar resistencias polarizadas a **+5** volts, en cada una de las líneas de solicitud de interrupción (INTRN).

Se puede apreciar un buffer inversor (**74HCT540**) entre el PIC y los UART's de la tarjeta de recepción. Esto se debe a que las señales de interrupción del UART, operan con lógica negativa y las entradas del PIC con lógica positiva.

El valor escogido de los puertos de entrada/salida para control y operación de la tarjeta de interface, fué el primer banco (**200h**), aunque puede ser utilizado cualquier otro bloque, que se encuentre a partir de la localidad **400h**, para evitar cualquier interferencia con otras posibles tarjetas de interface. Adicionalmente el número de interrupción activo en la computadora y en la tarjeta de interface es el **IRQ5**, quedando como opciones los números IRQ3 é IRQ7, para mayores detalles observar las tablas de las páginas 123 y 135.

A continuación, serán abordados los aspectos más importantes considerados para la programación del PIC y de los UART's de la tarjeta de adquisición, así como los modos de operación seleccionados.

Inicialización del PIC.

```
Init_Pic proc   near
         mov    dx,Pic              ; External Pic address
         mov    al,00011010b        ; ICW1 level triggerd, single & ICW4 no needed
         out    dx,al
```

Con la primera instrucción, se asigna al registro *dx*, la dirección de acceso al PIC.

Con la segunda instrucción, se define el contenido del comando ICW1 de inicialización del PIC. Con la tercera instrucción se envía al PIC el valor de ICW1.

A continuación, se justifica el valor asignado a cada uno de los bits del comando ICW1.

A_7-A_5 son ceros pues el **PIC** trabajará en modo de sondeo.

LTIM = 1 El PIC, realizará la detección de las señales de interrupción (**IR_0-IR_7**) por nivel. Cuando la detección de la señal se realiza por flanco, pueden existir problemas con la detección de la transición de nivel bajo a alto, si la interrupción ocurre en forma simultánea con la petición de un comando de sondeo. Esto se debe, a que las interrupciones son congeladas durante el intervalo de tiempo que transcurre entre una operación de escritura y la siguiente de lectura, que son necesarias para realizar la ejecución del comando de sondeo.

ADI = 0 El intervalo de direccionamiento, no es aplicable para la operación del PIC con un microprocesador de la familia **MCS-CX86/88**.

SNGL = 1 Sólo existe un PIC en la tarjeta de recepción. El funcionamiento de los PIC's de la computadora es independiente del PIC de la interface.

IC4 = 0 Se indica al PIC que inicialize todos los bits del comando ICW4 a ceros.

```
         mov    al,00h  ; ICW2
         inc    dx
         out    dx,al
```

Una vez recibido el comando ICW1, el PIC automáticamente queda en espera del comando ICW2 de inicialización. A través de este comando, se programa el número de interrupción, a partir del cuál serán identificadas cada una de las 8 líneas de interrupción del PIC. En este caso, se ha asignado el cero debido a que se utilizará el modo de sondeo. Así, la línea INTR0 queda identificada por un 0 y la línea INTR7 por un 7.

```
mov   al,00000000b      ; OCW1 Reset Interrupt Mask Register
out   dx,al
```

Con las intrucciones anteriores, se manda el comando de operación OCW1 al PIC. El valor del mismo, configura el registro IMR. Así, a través del comando OCW1 es posible habilitar ó deshabilitar las líneas de entrada de solicitud de interrupción del PIC. En este caso se habilitan todas las líneas.

```
dec   dx
mov   al,11100000b      ; OCW2 Rotate on specific EOI IRQ0 = 0
out   dx,al
```

El comando de operación OCW2, define los siguientes parámetros del PIC:

R =1, SL =1, EOI =1	Se habilita la rotación de prioridades y se configura al PIC para que mantenga la señal INTR en alto hasta que reciba un comando de fin de interrupción específico. Con los bits **L_2, L_1** y **L_0** se indica la línea de menor prioridad, en este caso IRQ0. La rotación de prioridades es muy útil, porque asegura iguales condiciones de atención a todas las líneas de entrada al PIC. Cada vez que una línea es atendida, se le asigna la menor prioridad. En el peor de los casos, una línea será atendida después de que las 7 líneas de entrada de solicitud de interrupción al PIC hayan sido atendidas.

```
        mov   al,00001100b        ; OCW3 Comando de Sondeo
        out   dx,al
        ret
Init_Pic endp
```

Con el comando de operación OCW3, se define que el **PIC** operará en el modo de **sondeo**.

INICIALIZACION DE LOS UART'S OCTALES SCC2698B.

A continuación, se detalla parte de la subrutina de inicialización de los UART's. Esta sección incluye la justificación de la forma de programación de un bloque de control de un UART. La programación de los bloques restantes es similar.

```
Init_Uart proc near
          mov   cx,0
Empieza:  mov   bx,cx
          shl   bx,1
          jmp   cs:salto[bx]
Blok_1A:  mov   dx,Uart1 + CR_a
          mov   al,00011010b        ; Disable Tx and Rx, pointer to MR1
          out   dx,al
          mov   al,01001010b        ; Reset Error Status
          out   dx,al
          mov   al,00101010b        ; Reset Receiver
          out   dx,al
```

Las cuatro primeras instrucciones, forman parte del flujo de control de la rutina de inicialización del UART. Con la instrucción *jmp cs:salto[bx]*, se llega al área de inicialización de cada bloque. Si es necesario inicializar el primer bloque del UART 1, el salto se hace a la etiqueta *Blok_1A*.

Las siguientes instrucciones, direccionan al registro de comando (CR), para que reinicialice: *el receptor, el registro de status (SR) y para que el apuntador quede direccionado hacia el registro de programación de modo (MR1) del canal **a** del UART 1.*

```
mov   dx,Uart1 + MR_a        ; Mode of programation
mov   al,01110111b           ; FFULL, Block mode, No Parity, 8 bits/C
out   dx,al                  ; MR1
mov   al,00000111b           ; Normal, Deactivation, No CTS, 1 Stop Bit
out   dx,al                  ; MR2
```

Las lineas anteriores direccionan al registro de modo **MR1** y posteriormente al registro **MR2.** A través de estos registros, se programan los siguientes parámetros en el receptor del canal a, UART 1:

MR1:
- Estado de interrupción activado si el FIFO se llena (FFULL).
- Determinación de errores por bloque (los 3 caracteres del FIFO).
- Bit de paridad suprimido.
- Tamaño de datos por caracter de 8 bits.

MR2:
- Modo de operación normal del receptor/transmisor.
- Longitud de parada de 1 bit.

```
mov   dx, Uart1 + CSR_a
mov   al, 11010000b          ; Timer Mode Reception
out   dx, al                 ; No transmition
```

Las líneas anteriores, habilitan al selector de reloj ***CSR*** del canal a UART 1, a definir la fuente de reloj a ser utilizada como referencia por el generador de baudaje.

El contenido del CSR, indica que la fuente del generador de baudaje de recepción será el **timer**. El timer deberá ser programado para operar a **2100** y **1200 bauds**. Estos baudajes, corresponden a las velocidades de trasmisión con que operan las estaciones GEOS-3 de tres componentes y de un componente respectivamente. Para el cálculo del valor que se utiliza en la configuración de los registros CTUR y CTLR, deberán emplearse las fórmulas descritas con anterioridad, dependiendo del valor de los bits ACR[6:4].

```
mov   dx, Uart1 + ACR_A      ; Auxiliary Control
mov   al, 11100000b          ; Timer source cristal or external clock
out   dx, al
```

Las líneas anteriores, direccionan al registro de control auxiliar (ACR) del canal a UART 1, para indicar que la señal de reloj del timer provendrá del cristal ó de un reloj externo conectado al pin X1/CLK.

```
          mov   dx,Uart1 + CTUR_A
          mov   al,00h               ; MSB of CTR
          out   dx,al
          inc   dx                   ; CTUR/CTLR = Clock Timer / (32 * Baud rate)
          mov   al,55                ; LSB of CTR Div = 55 ==> Baud ≈ 2100
          mov   cs:frame_l[0],07h    ; Frame size Block 1A 3 Componentes
          test  cs:control,01h       ; 1 = 2100        0 = 1200
          jnz   menor1
          mov   al,96                ; LSB of CTR Div = 96 ==> Baud = 1200
          mov   cs:frame_l[0],03h    ; Frame size Block 1A 1 Componente
menor1:   out   dx,al
```

Con el bloque anterior de instrucciones, se programa la velocidad de recepción para los dos canales controlados por el bloque A del UART 1. La programación se efectúa a través de los registros CTLR y CTUR. Si al registro CTLR se le asigna el valor de 55, la velocidad de recepción de los canales asociados al bloque quedará definida a 2100 bauds. Si al registro CTLR se le asigna el valor de 96, la velocidad de recepción del bloque será de 1200 bauds. La variable *control contiene 8 bits, cada bit representa uno de los 8 bloques de control de los 2 UART's de la interface. Cuando un bit de la variable control está activo, indica que la velocidad de recepción de los dos canales que controla el bloque, debe ser programada a 2100 bauds. Si es cero, la velocidad de recepción programada deberá ser de 1200 bauds.* Para mayores detalles de la configuración de la variable control, ver la rutina CONFIG definida dentro del área deshechable del driver.

La variable *Frame*, se utiliza para conocer el número de caracteres que se deben de recibir, después del byte de identificación (las estaciones GEOS-3 transmiten en cierto formato, descrito en el **apéndice B**). Su valor depende de la velocidad de recepción; 2 caracteres para la recepción a 1200 bauds y 6 caracteres para la recepción a 2100 bauds.

```
          mov   dx,Uart1 + ISR_A     ; Interrupt Mask Register and Status
          mov   al,00100010b         ; Enable INTRN FFULL Chl a and Chl b Block A
          out   dx,al
```

Las instrucciones anteriores, direccionan al registro de interrupciones mascarables (IMR) del bloque A UART 1 y fijan las condiciones que generan una señal de interrupción en la salida (**INTRN0**). En este caso la señal de interrupción se activa si cualquiera de los dos buffers de recepción de los canales **a** y/ó **b** se llena.

```
mov   dx,Uart1 + STAR_A
in    al,dx                      ; Start C/T Block A
jmp   todos
```

Este bloque de intrucciones, sirve para inicializar la operación del timer. Los receptores del bloque A (canales a y b) del UART 1, se encuentran listos para operar, únicamente falta habilitarlos.

La habilitación de cada canal se realiza direccionando cada registro de comando CR, de la siguiente forma:

```
mov   al,00001001b               ; Reset Tx and Set Rx
mov   dx,UART1 + CR_a            ; First Channel UART 1
out   dx,al
```

En este momento el canal de recepción a del Bloque A UART 1, ha sido habilitado para que reciba datos. Lo mismo se hace con los canales restantes. Es necesario aclarar, que si bien los canales de recepción a partir de este momento empiezan a recibir datos, el microprocesador de la computadora no los leerá, hasta que la línea de interrupción asociada a la tarjeta de interface, sea habilitada por el CPU para detectar las interrupciones generadas por el PIC de la interface.

```
todos:    inc   cx
          cmp   cx, 8
          je    termin
          jmp   Empieza
termin:   ret
Ini_Uart  endp
```

Las instrucciones anteriores controlan el número de veces que habrá de repetirse la operación de inicialización, una por cada bloque de cada UART.

Conclusiones.

Peter Norton, un prestigiado programador de computadoras de renombre mundial escribió: "El desarrollo de un device driver, es una de las tareas más sofisticadas e intrincadas en las que puede verse involucrado un programador. Escribir un device driver, es similar a escribir los programas del BIOS y del ROM-BIOS, que constituyen respectivamente el corazón de DOS y el corazón de la máquina."

Es innegable que hoy en día, las computadoras se encuentran al alcance de cualquier persona, pero también es innegable que a diez años de haber salido la primera computadora de las líneas de producción, aún existen vacios importantes en el conocimiento de esta tecnología. Conforme el tiempo transcurre, alguna información técnica va apareciendo en forma dispersa y el círculo de conocimientos se amplía un poco más. Sin embargo, durante la etapa de investigación para el desarrollo del sistema SAD, fué notoria la falta de fuentes de información para el diseño de device drivers. Muchas personas habían necesitado utilizarlos alguna vez, pero en general era claro que nadie sabía con certeza cuál era el papel que desempeñaban estos extraños programas en el funcionamiento de la máquina. Esta primera impresión, nos motivó para realizar una exposición detallada de nuestro trabajo, que pudiera ser aprovechada en el futuro por otros programadores interesados en el tema. Ahora, con toda la información y el conocimiento en la mano, es posible señalar, que si bien el desarrollo de device drivers requiere de un conocimiento especializado y la tarea puede ser ardua algunas veces, no por ello debe ser considerada al tal grado inaccesible o complicada como pudiera desprenderse de la afirmación realizada por Norton.

El dominio de las técnicas de diseño de interfaces y desarrollo de device drivers reviste real importancia. Todo dispositivo periférico a nivel comercial, requiere de estos dos elementos para integrarse en forma transparente a la operación de una máquina. El tardío desarrollo en nuestro país de la industria de la computación nos coloca en serias desventajas, en cuanto a las posibilidades de generar tecnología de punta para la fabricación de computadoras. Sin embargo, en el campo de los dispositivos periféricos, existen áreas donde se podría incursionar con reales posibilidades de éxito, incluso fuera del ámbito nacional.

"Writing a device driver is akin to writing the BIOS programs that are the heart of DOS and at the heart of the computer's built-in ROM-BIOS. It is among the most sophisticated and intricate programming that is ever undertaken". Peter Norton.

Los resultados iniciales, obtenidos en pruebas de funcionamiento del sistema SAD fueron alentadores. SAD ha trabajado bajo el control de un programa de pruebas durante decenas de horas, adquiriendo 16 canales de datos sin presentar problemas en su operación. Este hecho suguiere que la tarjeta de interface y el device driver, fueron desarrollados correctamente.

Con la interface, se incorporó un nuevo dispositivo en la arquitectura de la computadora. Con el device driver, se introdujo la programación de control y la organización de los datos dentro del esquema de operación del sistema operativo.

Las mejoras en el diseño del device driver, apuntan hacia la inclusión de una rutina que lea la información generada por un reloj especial, que irá conectado a la interface serial de la máquina. En base a esa información, será insertada una marca de tiempo en cada buffer de datos del Sistema Circular.

Se requiere así mismo, de la implementación de un programa, que auxilie al desarrollador de aplicaciones, en la configuración y reinicialización del driver de la tarjeta de adquisición.

En cuanto al diseño de la interface, se requiere la utilización de técnicas de acceso directo a memoria (DMA), que incrementen el ancho de banda disponible para la ejecución de programas de detección más complejos y/ó la adquisición de un mayor volumen de información digital, proveniente de más canales (posiblemente 32).

Otra mejora es la de tener un control completo sobre la comunicación, ya que la transmisión de los datos de las estaciones remotas GEOS-3 es "tonta" y continua, debido a que no se tiene retransmisión de datos en caso de algún error en la comunicación y/ó bloqueo del medio de transmisión.

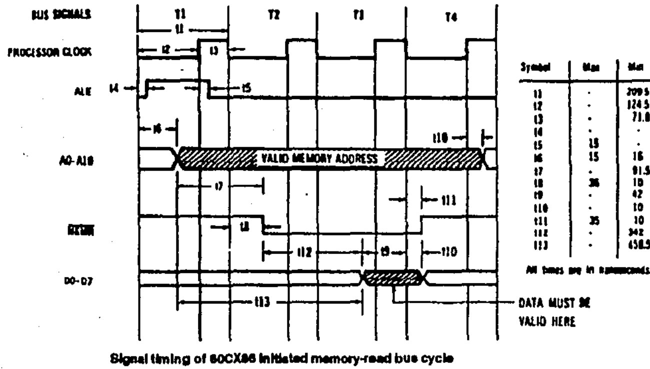
El campo de aplicaciones de SAD es extenso, el sistema no está limitado para recibir datos de estaciones sísmológicas GEOS-3: con adecuaciones mínimas, sería posible recibir datos de diversas clases de sensores digitales (inclinómetros, acelerómetros, etc.,).

En el futuro inmediato, son promisorias las perspectivas de SAD. Por ahora, se planea su instalación en cinco redes de monitoreo sísmico: tres en el estado de Guerrero, una en el estado de Jalisco y otra más en el Valle de México, lo cuál permitirá captar y procesar información valiosa de los fenómenos sísmicos que, por sus considerables repercusiones obligan a un estudio cada vez más profundo y sístemático. Si el sistema SAD contribuye en torno a este objetivo, nuestros esfuerzos quedarán grandemente recompensados.

Uno de los motivos que nos hicieron desarrollar este proyecto, era la necesidad de poner a prueba nuestra capacidad para resolver con la ayuda del conocimiento y el ingenio un problema real. Ahora, pensamos también que la difusión del conocimiento adquirido es importante y tenemos la esperanza de que nuestra contribución a través del presente trabajo, aunque pequeña resulte valiosa en ese sentido.

Apéndices

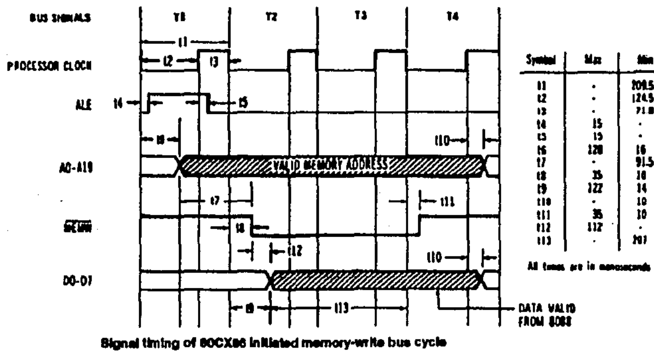
Apéndice A



Symbol	Max	Min
t1	-	209.5
t2	-	124.5
t3	-	71.0
t4	-	-
t5	15	-
t6	15	16
t7	-	91.5
t8	35	10
t9	-	42
t10	-	10
t11	35	10
t12	-	342
t13	-	458.5

Signal timing of 80CX86 initiated memory-read bus cycle

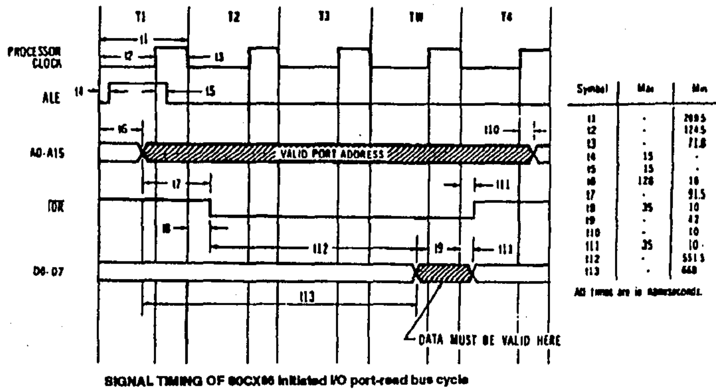
Figura 1



Symbol	Max	Min
t1	-	209.5
t2	-	124.5
t3	-	71.0
t4	15	-
t5	15	-
t6	120	16
t7	-	91.5
t8	35	10
t9	122	14
t10	-	10
t11	35	10
t12	112	-
t13	-	207

Signal timing of 80CX86 initiated memory-write bus cycle

Figura 2

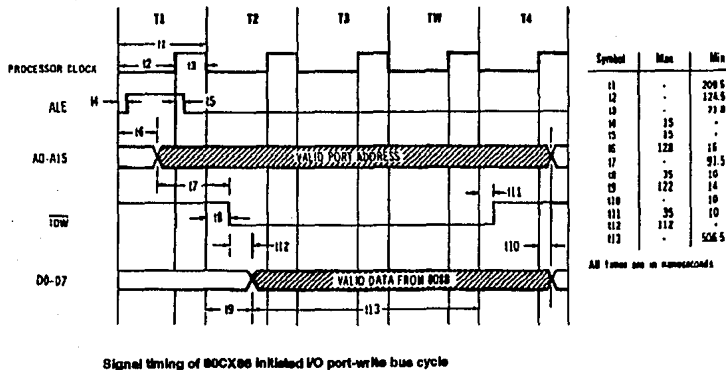


Symbol	Max	Min
t1	-	209.5
t2	-	124.5
t3	-	71.8
t4	15	-
t5	15	-
t6	128	16
t7	-	91.5
t8	35	10
t9	-	42
t10	-	10
t11	35	10
t12	-	551.3
t13	-	660

All times are in nanoseconds.

SIGNAL TIMING OF 80CX86 initiated I/O port-read bus cycle

Figura 3

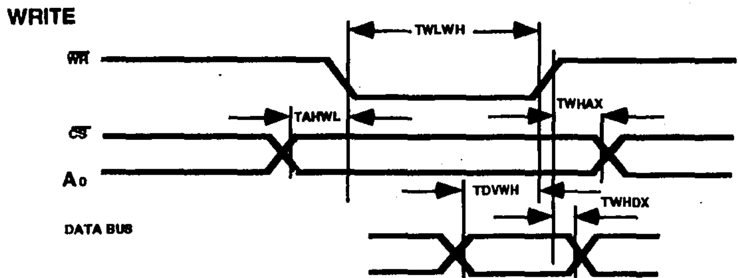


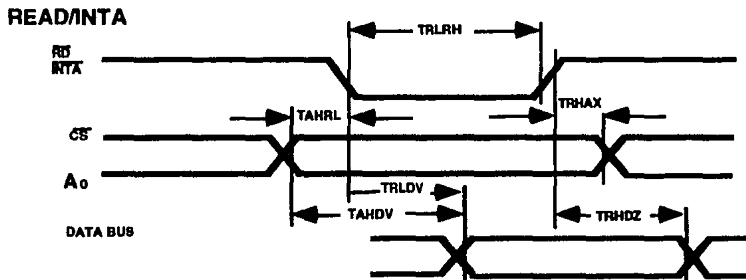
Symbol	Max	Min
t1	-	209.5
t2	-	124.5
t3	-	71.8
t4	15	-
t5	15	-
t6	128	16
t7	-	91.5
t8	35	10
t9	122	14
t10	-	10
t11	35	10
t12	112	-
t13	-	506.5

All times are in nanoseconds.

Signal timing of 80CX86 initiated I/O port-write bus cycle

Figura 4





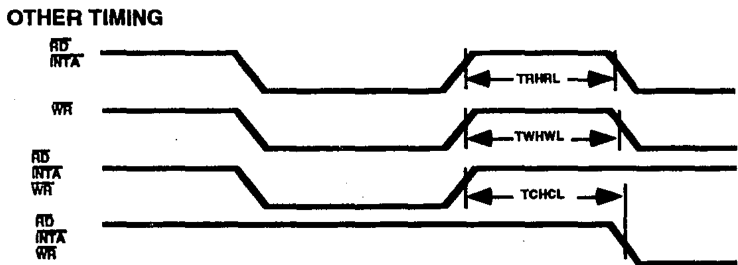
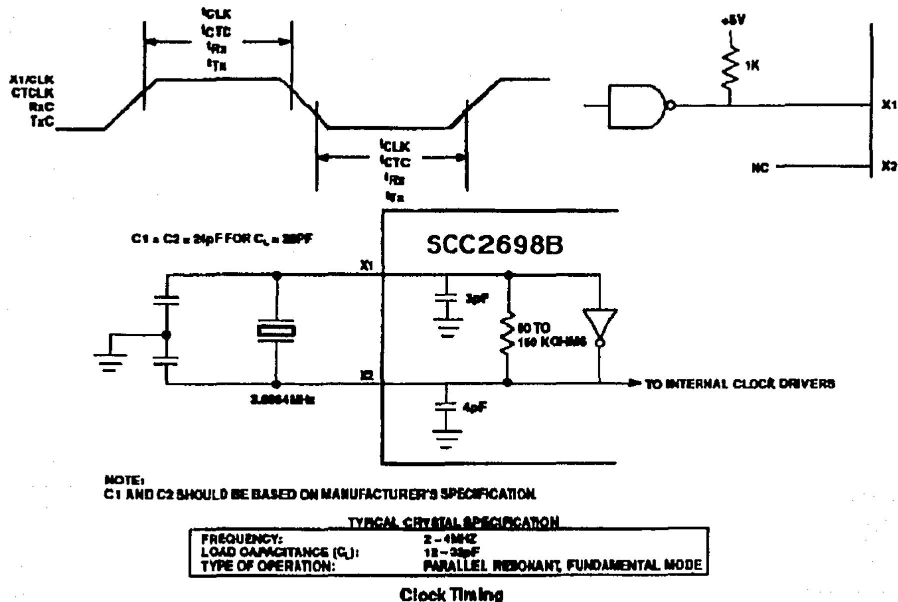


Figura 5

Symbol	Parameter	82C59A Min	82C59A Max	Units
TAHRL	**A0/CS** Setup to **RD/INTA**	10		ns
TRHAX	**A0/CS** Hold after **RD/INTA** †	5		ns
TRLRH	**RD/INTA** Pulse width	160		ns
TAHWL	**A0/CS** Setup to **WR**	0		ns
TWHAX	**A0/CS** Hold after **WR** †	0		ns
TWLWH	**WR** Pulse Width	190		ns
TDVWH	Data Setup to **WR** †	160		ns
TWHDX	Data Hold after **WR** †	0		ns
TRHRL	End of **RD** to next **RD**, End of **INTA** to next **INTA** within an **INTA** secuence only	160		ns
TWHWL	End of **WR** to next **WR**	190		ns
*TCHCL	End of command to next command (Not same command type)	400		ns
TRLDV	Data valid from **RD/INTA**		120	ns
TRHDZ	Data float after **RD/INTA** †	10	85	ns
TAHDV	Data valid from stable address		200	ns

* En el peor de los casos los tiempos para **TCHCL** en un sistema de microprocesador actual es típicamente mayor de **400 ns** (80C86 = 1 μs, 80C86-2 = 625 ns).

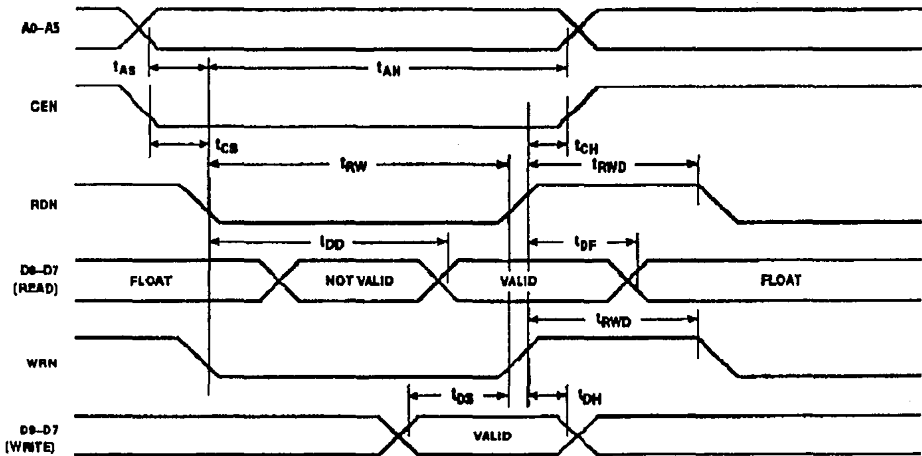


Clock Timing

Figura 6

En la siguiente tabla, se indican los límites de operación de los tiempos para el reloj del UART.

SYMBOL	PARAMETER	Min	Typ	Max	UNIT
t_{CLK}	**X1/CLK** high ol low time	120			ns
f_{CLK}	**X1/CLK** frecuency	2.0	3.6864	4.0	MHz
t_{CTC}	**Counter/timer** clock high or low time	120			ns
f_{CTC}	**Counter/timer** frecuency	0[11]		4.0	MHz
t_{RX}	RxC high or low time	200			ns
f_{RX}	RxC frecuency (**16X**) RxC frecuency (**1X**)	0[11] 0[11]		2.0 1.0	MHz MHz
t_{TX}	TxC high or low time	200			ns
f_{TX}	TxC frecuency (**16X**) TxC frecuency (**1X**)	0[11] 0[11]		2.0 1.0	MHz MHz

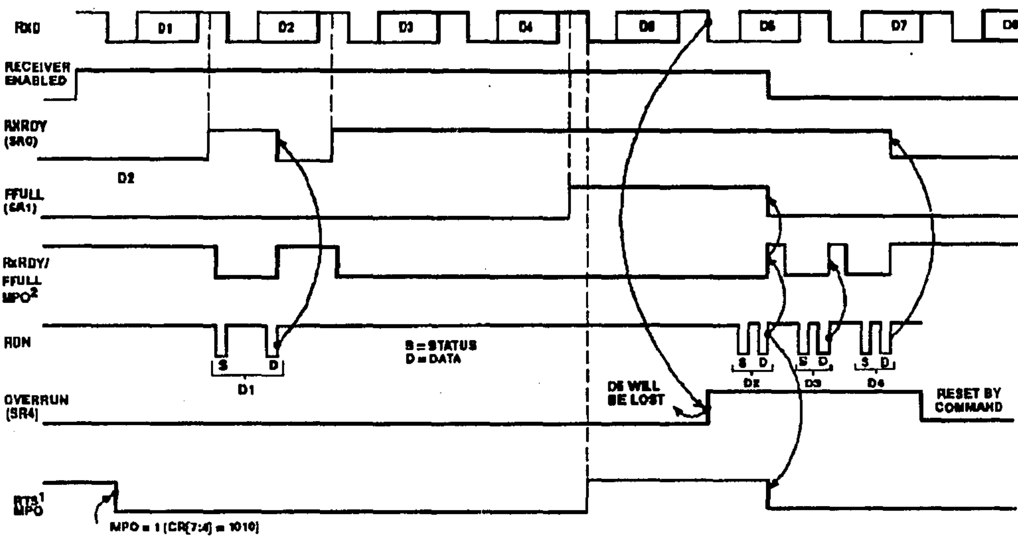


Bus timing

Figura 7

Symbol	Parameter	Min	Max	Unit
tCS	**CEN** setup time to **RDN**, **WRN** Low	0		ns
tCH	**CEN** hold time from **RDN**, **WRN** High	0		ns
tRW	**WRN**, **RDN** pulse width Low	225		ns
tDD	Data valid after **RDN** Low		200	ns
tDF	Data bus floating after **RDN** High		80	ns
tDS	Data setup time before **WRN** High	100		ns
tDH	Data hold time after **WRN** High	10		ns
tRWD	Time between reads and/or writes	100		ns

En el siguiente diagrama de tiempos, se muestra lo que sucede durante la recepción de datos, por cada caracter que recibe el UART OCTAL **SCC2698B** en uno de sus canales de recepción.



NOTES:
1. TIMING SHOWN FOR MR1[7] = 1.
2. SHOWN FOR OPCR [6X, 2:0] = 111 AND MR1[6] = 0.

Receiver timing

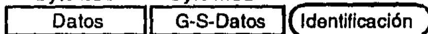
Figura 8

Apéndice B.

Formato de Recepción de Datos.

Actualmente, la recepción de datos puede realizarse a dos velocidades: 1200 y 2100 bps. Cuando en un canal se recibe a 1200 bps, los datos que envía una estación GEOS-3 provienen de un sismómetro que mide una de las tres componentes posibles de movimiento del suelo (Norte-Sur, Este-Oeste y la componente vertical Z, que mide los desplazamientos ascendentes descendentes). La secuencia de arribo de una trama de información, para una componente, a 1200 bps se presenta en la siguiente figura.

Secuencia de Arribo para una Componente.



Formato de Recepción.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	0	0	1

Byte de identificación de Trama.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
G2	G1	G0	D12	D11	D10	D9	D8

Primer byte de Datos MSB.

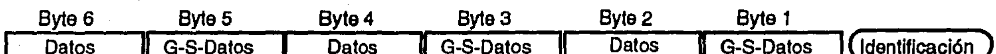
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
D7	D6	D5	D4	D3	D2	D1	D0

Segundo byte de Datos LSB.

Los bits **G_2, G_1** y **G_0** representan el valor codificado de ganancia del dato muestreado, el bit **D_{12}** representa el signo del dato muestreado y los bits **D_{11}-D_0**, representan a la magnitud del dato.

Cuando un canal de la tarjeta SAD recibe a 2100 bps, espera que la estación GEOS-3 tenga conectados tres sismómetros que determinen cada uno las tres componentes posibles de movimiento del suelo N-S, E-W y la componente vertical Z. El formato de recepción es similar al de 1 componente. La diferencia es que depués del byte de identificación siguen tres pares de bytes, cada uno con información de una de las tres componentes de movimiento.

Secuencia de Arribo Tres Componentes



Apéndice C.

Rutinas de Interface del Sistema SAD.

Es práctica común, el recurrir al uso de las rutinas de interface cuando el lenguaje de programación que sirve como plataforma de desarrollo de la aplicación, no brinda las funciones necesarias para el acceso y control de dispositivos. Las rutinas de interface proporcionan un mecanismo idóneo para utilizar funciones desarrolladas en lenguajes de programación alternos que permitan realizar el acceso y control a dispositivos. (Microsoft, 1987).

Para el caso del sistema SAD, fué necesaria la implementación de varias rutinas de interface, que permitieran la interacción entre el programa de aplicación y el driver. Con las rutinas de interface se lograron dos cosas: la primera, simplicidad para el uso de las mismas dentro del programa de aplicación. La segunda, eficiencia y flexibilidad en la comunicación con el driver. A continuación, serán comentados algunos aspectos importantes relacionados con la implementación de las rutinas de interface.

La aplicación, a través del stack del sistema, pasa a la rutina de interface los valores y direcciones de parámetros que requiere la misma para su operación. Los parámetros son colocados en el stack en orden inverso a cómo aparecen en el código fuente del lenguaje C. Esto es, el primer parámetro de una función será el último en ser almacenado en el stack, y el último será el primero en colocarse en el mismo. Los parámetros en C, por default son pasados por valor, con excepción de los vectores que son pasados por referencia.

La rutina de interface, debe hacer previsiones para obtener del stack los parámetros que le envía el programa de aplicación. En general se utiliza el registro BP del CPU como base para la obtención de los parámetros almacenados en el stack. En este último, también se almacena la dirección de retorno del programa que solicitó la ejecución de la rutina (return address). La dirección de retorno, ocupa 4 bytes del stack en el caso del modelo mediano de compilación. Las rutinas de interface, lo primero que hacen es salvar el valor del registro BP en el stack y asignarle un nuevo valor que permita el fácil acceso a los parámetros enviados por la aplicación. En los dos listados siguientes, podrá observarse que los parámetros de la interface rext(v,fh) se obtienen a través del registro *BP más un offset.*

Interface REXT.

```
;Interface REXT.ASM
;Para lenguaje C, modelo Mediano, flag=rext(v,fh);
.model medium
.code  rext
       public _rext

_rext  proc
       cld                  ; save machine state on entry
       push  bp             ; save base pointer of calling routine
       mov   bp,sp          ; set new reference base pointer
       push  ds
       push  es
       push  bx
       push  cx
       push  dx
       push  di
       push  si
       mov   ax,@data
       mov   ds,ax

       call  readd

exit:  pop   si             ; restore all registers
       pop   di
       pop   dx
       pop   cx
       pop   bx
       pop   es
       pop   ds
       pop   bp
       ret
_rext  endp
```

Arriba se muestra, el listado del programa principal, que es común a todas las interfaces desarrolladas para el sistema SAD.

Se observan tres bloques de instrucciones. El primero define el modelo de compilación, el nombre de la interface y el segmento donde ésta quedará alojada dentro del programa executable. Con el segundo bloque, se preservan los registros del CPU, que requiere la rutina de interface para realizar sus operaciones. En este bloque, se asigna al registro *ds* la dirección del segmento de datos @data que utiliza el programa .de aplicación para el almacenamiento de variables. Con la llamada a la rutina READD, se solicita al driver la transferencia de un buffer con datos del Sistema Circular en memoria extendida hacia el área de datos de la aplicación definida por @data. Ya ejecutada la rutina READD, se restauran los registros del CPU y el control se regresa al programa de aplicación.

```
;*******************************************
;       Readd
;       Input
;               AH = 44h        AH=02
;               BX = handle
;               CX = number of bytes to read
;               DS:DX = segment:offset buffer
;       Returns
;       If successful
;       Carry Flag = clear
;               AX = Bytes transferred
;       If unsuccessful
;               AX = error code
;       Carry Flag = set
;*******************************************
readd   proc    near
        mov     bx,[bp+8]       ; get file handle
        mov     dx,[bp+6]       ; get vector address vector3[0]
        mov     cx,1            ; CX = elements to move
        mov     ah,44h
        mov     al,02           ; function read
        int     21h             ; transfer to ms-dos
        jc      error4          ; jump if write failed, carry is set
        cmp     ax,cx           ; compare length of read
        jl      error4
        mov     ax,0            ; carry no set, read ok
        jmp     flagr
error4:
        mov     ax,1            ; set read error
flagr:  ret
readd   endp
        end
```

La rutina READD, perteneciente a la rutina de interface REXT, se encarga de realizar todos los preparativos para solicitar al driver SAD un buffer del Sistema Circular.

La primera operación que hace la rutina READD, es obtener los valores de los parámetros pasados a la rutina de interface por el programa de aplicación, a través de la función rext(v,fh) (*ver definición y uso en el programa catch.c*). Se obtiene así, la dirección del vector donde serán transferidos los datos del Sistema Circular y el número de file handle para acceso al driver.

La interrupción ***21h función 44h servicio 02h,*** se usa para indicarle a DOS, que una operación de lectura de datos de control sobre un dispositivo tipo caracter debe llevarse a cabo. El sistema operativo al recibir este requerimiento, pedirá al driver SAD se ejecute el ***comando 3 Ioctl_Input.***

El servicio *02h de la función 44h*, precisa de la siguiente información, que DOS utilizará en la construcción del Request Header.

AH	=	**44h**
AL	=	**02h**
BX	=	**File Handle.**
CX	=	**Número de bytes a leer.**
DS:DX	=	**dirección del buffer de intercambio.**

El número de file handle, es pasado a la interface como un parámetro de la función rext(v,**fh**).

El número de bytes a leer, en este caso 1, se almacena en el registro CX. Aquí es necesario detenernos, para hacer algunas consideraciones.

La rutina rext, se utiliza para transferir un buffer completo de datos de memoria extendida a memoria principal, la razón de indicar a la función 44h servicio 02, que sólo un byte será transferido del driver a la aplicación, se hace con el objeto de evitar que el sistema operativo intervenga innecesariamente en la transferencia de datos. Se recordará que el driver SAD, fué definido para manejar un dispositivo tipo caracter. Si el sistema operativo detecta que más de un byte debe ser transferido entre el dispositivo y la aplicación, puede intervenir y hacer que la transferencia se lleve a cabo de byte en byte y no en bloques de bytes, que es en este caso lo más deseable, por ser la transferencia más rápida. Para evitar esto, al sistema operativo se le indica que un sólo byte será transferido. El driver, al momento de recibir la solicitud de ejecución del comando ioctl_input, inspecciona el número de bytes que el sistema operativo le solicita transferir, si el número es igual con 1, el driver SAD accesa el área de datos de la aplicación y le transfiere todo un buffer de datos, sin que el sistema operativo se dé por enterado.

No debe existir confusión entre lo que hace el driver, la rutina de interface y el sistema operativo. La rutina de interface, solicita a DOS una operación de lectura a un archivo. El sistema operativo al recibir la solicitud, detecta que la lectura es hacia un driver, construye el request header y genera un comando para el driver SAD.

El driver al recibir la solicitud de ejecución del comando 3, verifica el número de bytes que le piden transferir, si el valor es 1, transfiere 14440 bytes a la aplicación, sin intervención del sistema operativo.

Con esto se logra que un driver definido como tipo caracter, pueda transferir buffers enteros de datos a la aplicación, en forma similar como lo haría un driver de bloque, pero sin sus limitaciones inherentes.

La dirección del buffer de intercambio, corresponde a la dirección de inicio del vector que recibirá los datos provenientes del Sistema Circular. Esa dirección se pasa al driver, con la llamada a la función rext(v,fh). La variable V es un apuntador al inicio del vector de intercambio. (ver definición programa catch.c).

Ya preparada, toda la información que requiere la función 44h servicio 02 se genera la interrupción 21h, para solicitar al sistema operativo la ejecución de un servicio de lectura.

El resultado de la solicitud de transferencia, será indicado por el driver al sistema operativo. Si la transferencia no pudo realizarse, el sistema operativo encenderá el bit de carry para señalar esta situación a la rutina de interface, quién a su vez entregará un 1 a la aplicación. En caso contrario, la bandera de carry permanecerá apagada al término de la ejecución de la función 44h, y la rutina de interface regresará un 0 al programa de aplicación, indicando con ello que la lectura pudo efectuarse correctamente.

Interface SAD_ON.

Parte de la estructura del cuerpo principal de la interface SAD_ON es mostrada a continuación. Puede apreciarse que la estructura es similar a la utilizada por la interface REXT: se definen directivas para el compilador, se preservan los valores de registros de máquina, se llama a la rutina *sadon*, que se encarga de solicitar al driver habilite el reconocimiento de las interrupciones generadas por la tarjeta SAD y se restauran finalmente los registros del CPU, regresándose el control del mismo al programa de aplicación. (Ver catch.c para recomendaciones de uso de esta rutina).

```
;Interface SAD_ON       Lenguaje C Modelo mediano. flag=sad_on(fh)
.model  medium.
.code   sad_on
        public _sad_on
_sad_on proc
;       save machine state on entry
        .
        call    sadon
        .
;       restore machine state
_sad_on endp
```

La rutina sadon, se encarga de realizar la petición al sistema operativo, para que se reconozcan las interrupciones generadas por la tarjeta SAD. Para hacer esto, se utiliza la función 44h servicio 03 de escritura. Esta solicitud es traducida por el sistema operativo en el comando de escritura número 12 IOCTL_OUT. El driver al recibir el comando 12, verifica el valor del primer elemento del área de datos. Si el valor es igual a 1110h, el driver no transfiere ningún byte, el driver accesa al PIC de la computadora y habilita la línea de interrupción asociada a la tarjeta de adquisición. (ver comando 12 driver SAD, etiqueta set_int).

```
;SADON          Input
;                       AL = 3  AH=44h
;                       BX = handle
;                       CX = number of bytes to write
;                       DS:DX = segment:offset buffer
;               Returns
;               If successful
;                       Carry Flag = clear
;               If unsuccessful
;                       AX = error code
;                       Carry Flag = set
;****************************************
sadon   proc    near
        mov     bx,[bp+6]       ; get file handle
        mov     dx,offset on    ; on it is defined in the data segment as: ON DW 1110h
        mov     cx,2            ; CX =bytes to move
        mov     al,3            ; write IOCTL string from driver
        mov     ah,44h          ; service IOCTL
        int     21h             ; transfer to ms-dos
        jc      error4          ; jump if write failed
        mov     ax,0            ; set no write error
        jmp     flagr
error4: mov     ax,1            ; set read error
flagr:  ret
sadon   endp
        end
```

Interface SAD_OFF.

La interface SAD_OFF, deshabilita el reconocimiento de las interrupciones generadas por la tarjeta SAD en la computadora. La única diferencia entre esta interface y SAD_ON, tiene relación con el valor de un parámetro que ambas le pasan al driver, al solicitarle el servicio 44h función 3. En la rutina sadoff, el valor de la variable a donde apunta el registro dx, queda asociado con 1111h. El driver al ejecutar el comando 12 IOCTL_OUT verifica el valor almacenado en la dirección *dx*, si es igual con 1111h deshabilita la línea de entrada del PIC de la computadora, que se encarga de detectar las interrupciones generadas por la tarjeta SAD.

```
;Interface SAD_OFF     Lenguaje C Modelo Mediano.  flag=sad_off;
.model  medium
.code   sad_off
        public _sad_off
_sad_off proc
;       save machine state on entry
        .
        call    sadoff
        .
;       restore machine state
_sad_off endp
;*****************************************
;SADOFF         Input
;                       AL = 3  AH=44h
;                       BX = handle
;                       CX = number of bytes to write
;                       DS:DX = segment:offset buffer
;               Returns
;               If successful
;                       Carry Flag = clear
;               If unsuccessful
;                       AX = error code
;                       Carry Flag = set
;*****************************************
sadoff  proc    near
        mov     bx,[bp+6]       ; get file handle
        mov     dx,offset  off  ; off it is defined in the data segment as: ON DW 1111h
        mov     cx,2            ; CX =bytes to move
        mov     al,3            ; write IOCTL string from driver
        mov     ah,44h          ; service IOCTL
        int     21h             ; transfer to ms-dos
        jc      error4          ; jump if write failed
        mov     ax,0            ; set no write error
        jmp     flagr
error4:  mov    ax,1            ; set read error
flagr:  ret
sadoff  endp
        end
```

Interface Status y Wstatus.

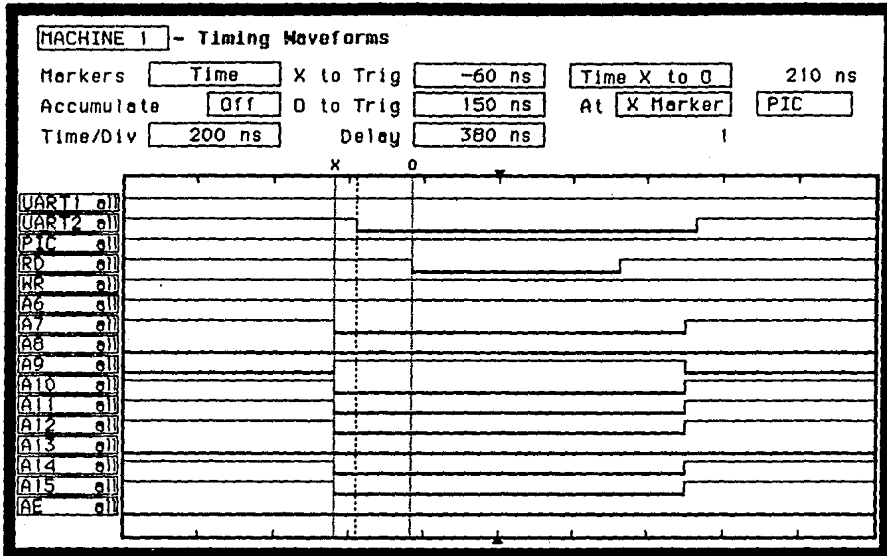
La rutina de interface status, sirve para solicitar al driver los valores de las variables que controlan su operación. La rutina wstatus se usa para cambiar los valores de los parámetros de control del driver.

En la rutina status, se utiliza la función 44h servicio 2 de lectura, para requerir al sistema operativo, genere el comando 3 IOCTL_INPUT. Es necesario recordar que el comando 3, se solicita también cuando se necesita obtener un buffer del Sistema Circular de SAD. (ver interface rext). A fin de que el driver pueda discernir cuál de las dos operaciones le solicitan, en el caso de la rutina status, se asigna un 10 al campo del número de bytes a transferir. En el caso de la rutina rext se asigna un 1 a ese campo. El driver verifica el valor del campo, y determina así la operación a ejecutar.

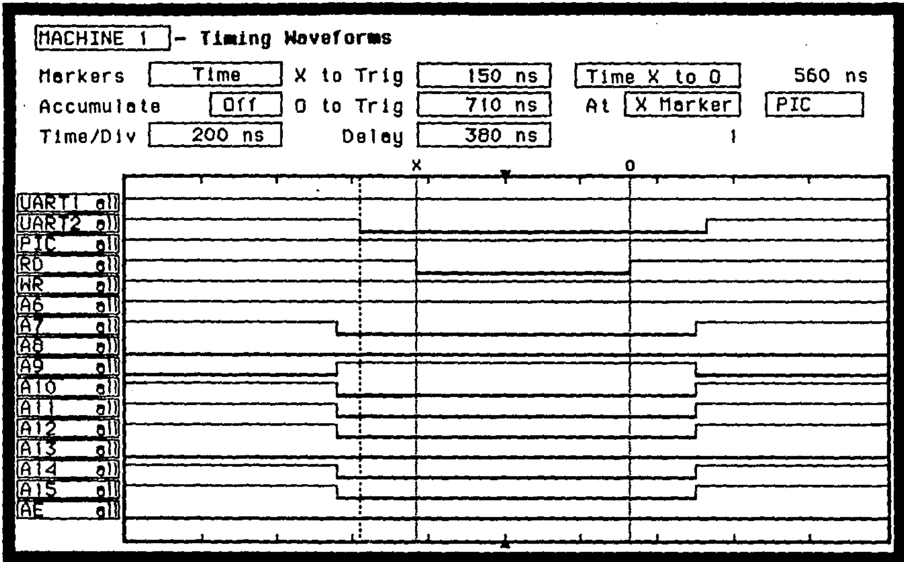
```
; Interface STATUS.ASM
.model  medium
.code   status
        public _status
_status proc
;       preserve machine state
        .
        call    reads
;       restore machine state
        .
        ret
_status endp

reads   proc    near
        mov     bx,[bp+8]       ; get file handle
        mov     dx,[bp+6]       ; get vector address vector2[0]
        mov     cx,10           ; CX =bytes to move
        mov     al,2            ; read IOCTL string from driver.
        mov     ah,44h          ; service IOCTL
        int     21h             ; transfer to ms-dos
        jc      error4          ; jump if write failed
        mov     ax,0            ; set no read error
        jmp     flagr
error4:mov      ax,1            ; set read error
flagr:  ret
reads   endp
        end
```

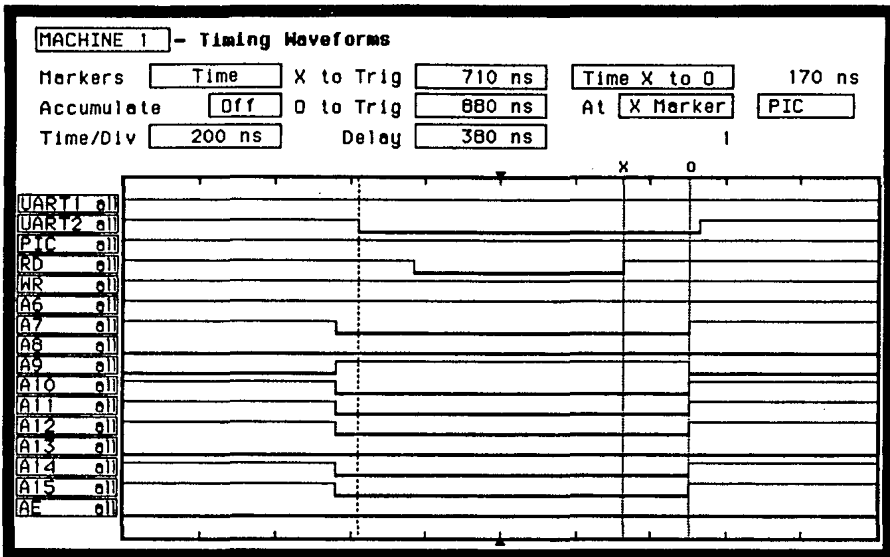
La interface *wstatus* utiliza la función 44h servicio 3, para solicitar al driver, a través del comando ***12 IOCTL_OUT*** una operación de escritura. La estructura de esta rutina es idéntica a la utilizada en la interface status.



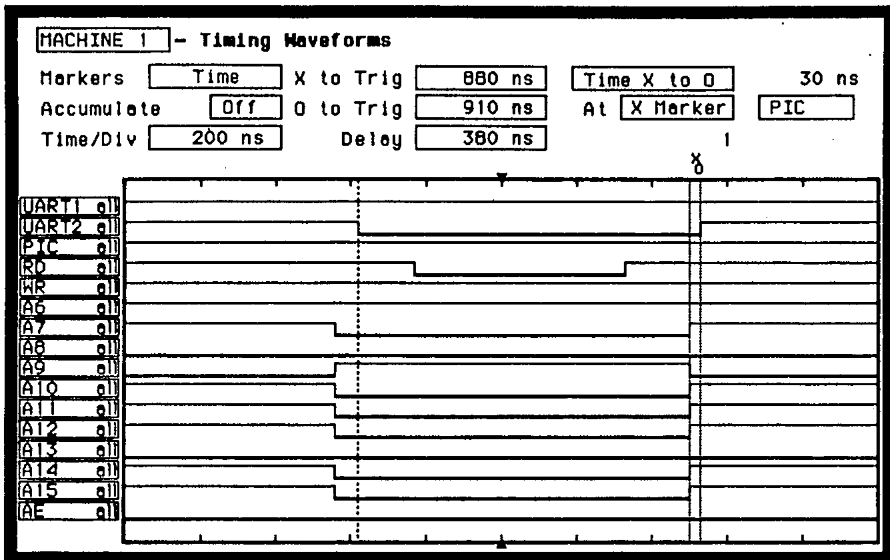
En la gráfica anterior se observan los tiempos de activación, entre las señales de: dirección **A_{15}-A_6** y las de habilitación del UART2 **CS1** (**X** to **Trig = 60 nS**), de dirección **A_{15}-A_6** y lectura **RD** (**X** to **0** = **210 nS**) y de habilitación del Uart2 **CS1** y de lectura **RD** (**0** to **Trig** = **150 nS**). Con lo cual se demuestra que estan operando conforme a los diagramas de tiempo de operación del UART **SCC2698B** del **Apéndice A**, esta gráfica y las subsecuentes, fueron obtenidas de la operación de la tarjeta de Adquisición de Datos del Sistema **SAD**.



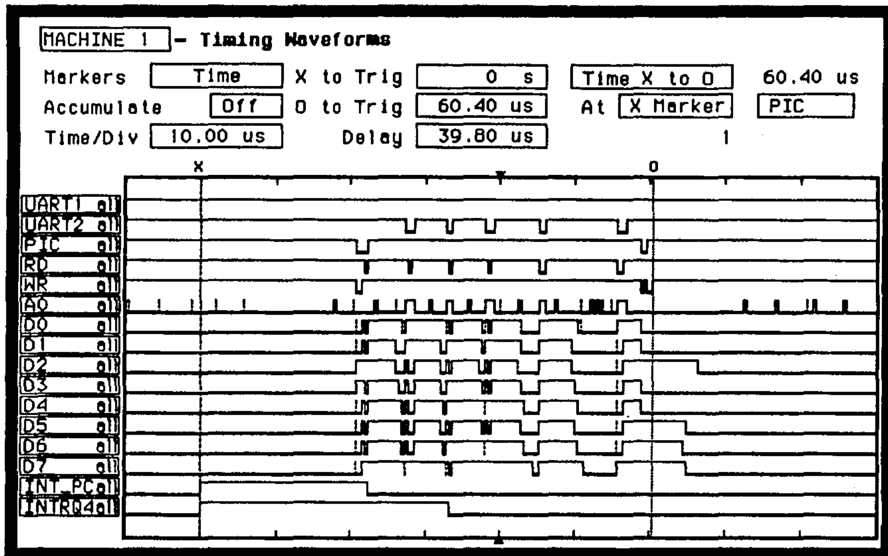
La gráfica anterior, muestra el tiempo de activación de lectura **RD** (**X** to **0** = **560 nS**), en un solo acceso de un comando de lectura a el UART2.



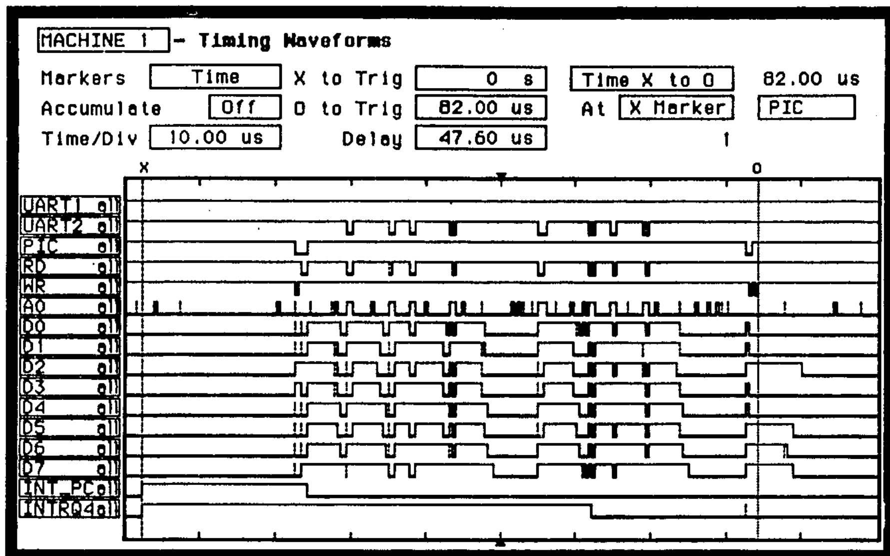
En la gráfica anterior, se muestra el tiempo en el cual las señales de direcciones **A_{15}-A_6** son retenidas posteriormente a la de lectura **RD** (**X** to **0** = **170 nS**).



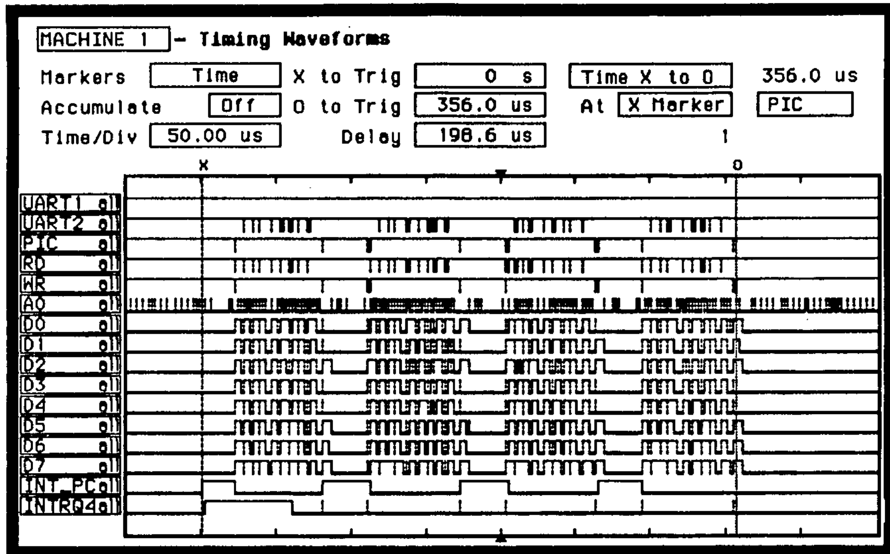
En la gráfica anterior, se muestran los tiempos de activación de la señal de habilitación del UART2 **CS1** (**0** to **Trig** = **910 nS**) y el tiempo en que dicha señal de habilitación **CS1** es retenida posterior a las señales de dirección **A_{15}-A_6** (**X** to **0** = **30 nS**).



En la gráfica anterior, se muestra el tiempo de procesamiento que le toma a el CPU de la computadora, desde que se genera la interrupción y hasta que manda el comando de fin de interrupción a el controlador de interrupciones del computador (**X** to **0** = **60.4** μS). En esta gráfica, se pueden apreciar 5 accesos al UART2 (el primero y el último son de lectura de Status y los 3 intermedios de lectura de datos), 3 accesos a el PIC de la tarjeta de interface (2 del comando de sondeo y el último del comando de fin de interrupción) y finalmente el comando de fin de interrupción a el PIC de la computadora. Aquí están incluidas todas las funciones de servicio para un solo canal de recepción activo.



En la gráfica anterior, se muestra el tiempo de procesamiento que le toma a el CPU del computador, en atender los requerimientos de servicio que tiene el UART2, con 2 canales de recepción activos (**X** to **0 = 82** μ**S**).



En la gráfica anterior, se observa el tiempo de procesamiento que le toma a el CPU del computador, en atender los 8 canales activos de recepción del UART2 de la tarjeta de interface del Sistema **SAD (X to 0 = 356 µS).**

MACHINE 1 - Timing Waveforms

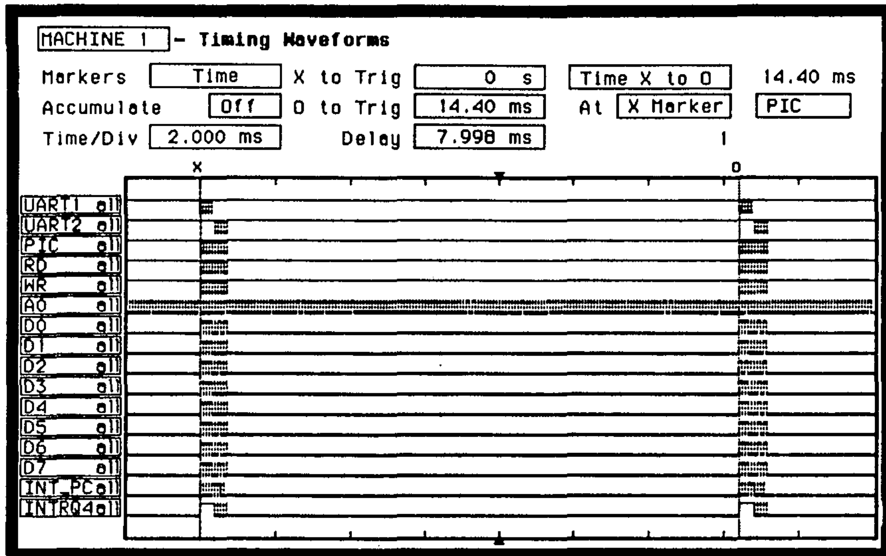
Markers Time X to Trig 0 s Time X to 0 716.0 us

Accumulate Off 0 to Trig 716.0 us At X Marker PIC

Time/Div 100.0 us Delay 398.6 us 1

UART1 all
UART2 all
PIC all
RD all
WR all
A0 all
D0 all
D1 all
D2 all
D3 all
D4 all
D5 all
D6 all
D7 all
INT_PC all
INTRQ4 all

En la gráfica anterior, se observa el tiempo de procesamiento que le toma a el CPU del computador, en atender los 16 canales de recepción activos de la tarjeta de interface del Sistema **SAD (X to 0 = 716 μS)**.



En la gráfica anterior, se observa el tiempo que transcurre entre la atención de los 16 canales de recepción activos, hasta que son nuevamente atendidos, hay que resaltar que se tiene suficiente tiempo posterior a la atención de los 16 canales de recepción como para ejecutar cualquier otro programa de aplicación, que incluya la graficación en pantalla de todos los canales, detección de evento de todos los canales, así como la grabación de dichos eventos en el disco duro; en todas las gráficas anteriores se utilizó una estación **GEOS-3** de tres componentes y operando a una velocidad de 2100 bauds.

Bibliografía.

Cirrus Logic	(1988)	Preliminary Data Sheet, CL-CD180 June, 1988.
Cirrus Logic	(1988)	CL-CD180 Update August, 1988.
Duncan, R.	(1988)	Advanced MSDOS Programming. Second Edition.
Eggebrecht, L.	(1990)	Interfacing to the IBM Personal Computer. Second Edition.
IBM.	(1984)	Personal Computer AT, Hardware Reference Library. Technical Reference.
IBM.	(1987)	Disk Opertating System, Version 3.30 Technical Reference.
Intel.	(1987)	Microprocessor and Peripheral Handbook. Volume I - Microprocessor.
Intel.	(1989)	Microprocessor and Peripheral Handbook. Volume II - Peripheral.
Lai, R.	(1987)	Writing MSDOS Device Drivers.
Mastrianni, S.	(1991)	OS/2 Device Drivers. Byte Magazine, July 1991. pp 241.
Microsoft.	(1987)	Mixed-Language Programing Guide. Microsoft Macro Assembler 5.1.
Microsoft.	(1987)	Reference. Microsoft Macro Assembler 5.1.
Norton, P.	(1985)	Programmer's Guide to the IBM PC.
Planet Earth.	(1982)	Earthquake. Time Life Books.
Signetics	(1986)	High-Speed CMOS Data Manual.

Signetics (1987) Microprocesor Data Manual.

Signetics (1988) SCC2698B, No. AN410B (Octal UART) Aplication Note.

Signetics (1990) SCC2698B. Enhanced Octal Universal Asynchronous Receiver/Transmitter (Octal UART). Document No. 853-1127. Date of Issue: September 7, 1990.

Signetics (1990) SCC2698B. Octal UART, Application Note. Octuber 1990.

Townsend, C. (1989) Advanced MS-DOS. Expert Techniques for Programmers.

Tan, Yi (1989) Control de una Estación Sismológica Digital. Tesis de Maestría. Facultad de Ingeniería, U.N.A.M.