MISOSYS, INC.

The Programmer's Guide to TRSDOS Version 6

Misosys, Inc.

The Programmer's Guide to TRSDOS Version 6

By Roy Soltoff, BSEE

First Edition - 1983
Second Edition - 1984

Disclaimer:

While MISOSYS has taken every precaution in the preparation of this book, it assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

MISOSYS Inc.
P. O. Box 239
Sterling, Virginia 22170-0239

This book is dedicated to my first daughter, Stacey Elizabeth, whose birth the eighth of June of 1983 provided me my proudest moment in life. There is no way that I can sufficiently thank my wife, Brenda, for nurturing and bringing forth this new human being - but I'll try.

1. Introduction

Many thousands of users take it upon themselves to explore the workings of an operating system to gain a better understanding of application software interfacing. This has always been such a waste of programmer talent because the system's designers usually know the best interfacing procedures. A complex operating system has many idiosyncrasies. Because of this fact, some procedures work much better than others to accomplish the same goal.

An operating system in this day and age demands that precious talent not be wasted. LDOS Version 6 is a complex operating system. There should not be a void of information that the programmer needs to properly write his or her software. For the programmer, this book should fill that void. It is not intended as an assembly language learning tool nor is it intended as an expose' of "mysteries" concerning the internal workings of the operating system. This book conveys that information which is essential to the job of programming application software, utilities, device drivers and filters.

It is very important for the programmer to keep PORTABILITY paramount in the thinking that goes along with program design. LDOS Version 6 was designed to provide portability for application software by incorporating standard protocols and conventions for all interfacing. Keep that in mind when you explore the contents of this book.

Knowing that the microcomputer community inherently finds distasteful the prospect of reading documentation cover-to-cover prior to jumping in and getting their feet wet, this book includes an index. Then again, what kind of book omits an index? Feel free to access the information randomly, although I recommend that a sequential scanning is more suited to the learning process.

The chapter contents have been designed to be self-contained. Thus, you may find some small repetition of subject matter where it was felt that a term or concept may not have been carried over from an earlier chapter due to an indexed access of the subject matter.

I have tried to be complete within the subjects discussed. As there are some proprietary items within the operating system, confidentiality precludes their appearance in this book. However, any work of this magnitude is bound to omit a detail. If you feel that a subject should have been included, please bring it to the publisher's attention. Remember that the desire to foster the development of portable software may mean that certain points may have been omitted to preclude the writing of non-portable machine specific software. Where you must write machine specific software, it is recommended that you obtain the manufacturer's hardware technical manual.

The programming examples were coded with the PRO-CREATE assembler, which is available from MISOSYS. References to SuperVisor Calls in the form @XXXX should have a corresponding EQU statement, which defines the SVC number.

For those individuals firmly entrenched in operating system exploration, I heartily recommend THE SOURCE, a three-volume set of books that provide the complete set of assembler source listings that constitute LDOS Version 6.2.0. THE SOURCE is available from Logical Systems, Inc.

Lastly, the author is always open to suggestions for improving this book. Certainly if you uncover erroneous data, suggest that it be corrected in the next printing. I wish you successful programming.

2. LDOS VERSION 6 - AN OPERATING SYSTEM OVERVIEW

After spending a few hours at any computer show featuring microcomputers, it becomes obvious that most 8-bit machines look surprisingly similar. Each comes equipped more or less with the following features: CRT monitor, keyboard, one or more 5-1/4" or 8" floppy disk drives (usually 5-1/4" minifloppies), 64K-128K of RAM, and a processor card. With the industry seemingly adopting CP/M as an operating system pseudo-standard, the chip usually chosen is Zilog's Z-80 microprocessor. The design of these machines must be sufficiently straight forward. While each competing manufacturer attempts to make its machine more desirable by implementing greater reliability, flexible interfacing, more peripheral support, additional hardware features, attractive packaging, and lower cost, cognizance of the cost effectiveness of utilizing smarter software may just be the important ingredient sometimes overlooked.

Alternative operating systems are available that bring a great deal of main-frame power to the microcomputer. One such system, LDOS Version 6 [or its licensed dialects such as TRSDOS 6], is a classic example of a truly powerful operating system designed for an eight bit microcomputer using the Z-80 processor chip. LDOS provides a single-user system with total device independence, dynamic file space allocation, extensive file management, job control language structures, a large library of utilities, plus the ability to easily interface to disk storage devices with capacities from 88 kilobyte minifloppies to multi-megabyte Winchester disk drives. Error trapping and an English-like command structure help make LDOS a user-friendly but powerful operating system.

The primary design obligation of LDOS is to ensure MEDIA COMPATIBILITY across all machines running the DOS (within the 5-1/4 or 8" size). This means that a user must be able to take a diskette and use it across all machines running LDOS - so long as the hardware permits that size diskette. To accomplish this, the DOS has a "standard" 5-1/4" structure - both single density and double density. It also has a "standard" 8" diskette structure. The structure goes beyond just the format and allocation schemes - it covers the entire directory makeup.

The hardware architecture chosen for LDOS Version 6 is a Z-80 based microcomputer with a minimum of 64K RAM and 80 by 24 video screen size. The DOS includes a bank-switching SuperVisor Call that implements memory bank switching. The SVC permits switching a memory segment (usually the top 32K) with up to seven auxiliary 32K memory banks. It also supports the controlled transfer of execution to a location within the bank at the option of the user. The system maintains supervision of the resident bank to ensure that the standard bank (bank 0) is always resident during certain operations (disk I/O, character I/O, and interrupt task handling). The DOS is designed to operate starting from address zero (page 0 origin) and is 100% SuperVisor Call (SVC) accessed. System data items needed by application software are also available via SVCs.

Essentially, there are two levels of interaction to the system - command level and primitive level. At the command level, the operator enters a command which requests the execution of some function [perhaps the listing of a file, the displaying of a disk directory, the running of a BASIC program, or the compiling of a C language source file]. The command interpreter parses the user entry, determines whether the request is for a system function or user-supplied function, then arranges for the necessary system resources. Control is transferred to the module necessary to satisfy the request. The system passes parameter pointers to the module and expects a return code upon the module's completion.

System resources and data quantities are requested via a SuperVisor Call (SVC) processor. An SVC is associated with all system primitives (i.e. get a character, put a character, open a file, add a task, rename a file, ...). Application software written in a low-level language (such as assembler) makes direct use of the SVC. Programs using a high-level

language (i.e. BASIC, C, PASCAL, ...) need not bother with the SVC as system interfacing is accomplished within the language interpreter or compiler.

The DOS supports up to eight logical disk packs or volumes logically numbered 0-7. Each floppy, be it one or two sided, is treated as a single volume. Hard disk drives (winchesters) may be treated as a single volume or partitioned into multiple volumes. A Drive Control Table (DCT) contains the parameters associated with each disk (number of cylinders, heads, and sectors per track for example) and also interfaces the disk driver software to the system.

Character Input/Output devices (i.e. keyboard, video display, printer, RS-232 serial ports, ...) and their associated software driver routines are interfaced to the system via Device Control Blocks (DCB). I/O devices are identified by a two-character device name such as KI (keyboard input), DO (video output), PR (printer), and CL (communications line). Whenever a device is specified, it is denoted by an asterisk followed by the device name to form a complete "device specification". The reason for this will soon become evident. Additional devices can be defined to the system once an appropriate software driver is available. The device name selection is left up to the user.

A collection of data stored on disk is termed a file and is denoted by a file specification. A complete file specification consists of five parts: a file name of up to eight characters, a file extension of up to three characters, a file password of up to eight characters, the logical drive specification, and optionally, in certain cases of Partitioned Data Sets (PDS), a member specification of up to eight characters. Whenever users institute a structured naming convention, most files are accessible via the file name reference only. The DOS will search all drives for a file if the drive specification is omitted from the file specification. In addition, many system utilities and user applications can use default file extensions to separate files into classes. For example, PRO-CREATE, a popular assembler running under the DOS, will automatically use the file extension "/ASM" for its source files and "/CMD" for its object code generation thus alleviating the user of the necessity to enter the file extensions (it also helps to prevent inadvertently overwriting one file with another). Similarly, LDOS makes extensive use of default file extensions such as "JCL" for all Job Control Language, "TXT" for ASCII listings, "FLT" for all device filters, etc.

File specifications and device specifications are generally interchangeable. Thus, wherever a file specification is needed, a device specification can usually be entered. This is one of the examples of device independence in the system. The protocol used in character I/O is identical across logical devices (i.e. *KI, *PR, *SO,...) and disk files. Thus, character I/O is handled the same way regardless of the physical device identified in the Device/File Control Block (DCB/FCB) - be it physical keyboard, printer, or disk file. For example, the COPY utility is used primarily to copy a file from one disk to another, as in:

```
COPY ARTICLE/TXT:0 TO ARTICLE/TXT:1
```

which creates a duplicate on drive 1 of the file specified "ARTICLE/TXT" located on drive 0. In lieu of the file specifications, device specifications could equally be used as in the following:

```
COPY *KI TO *PR
```

which copies keyboard input directly to the printer. With ease, a keyboard can be added to a daisy-wheel printer turning it into a temporary typewriter. Perhaps a more useful illustration would be the convenience of directing program output to video display, printer, or a file depending on the device/file specification provided.

The acquisition of disk file space is completely transparent to the user. This frees the user from worrying about sectors, tracks, cylinders, heads, and even disk drives in most cases. File space is obtained dynamically for any given file when space is required. Since directory accesses are dynamic (i.e. any time directory information requires updating, a disk access is made), users can change floppy disk ettes in a disk drive after any open files on the disk have been closed with out having to "log" the action.

Files do not have to occupy contiguous space on a disk but can exist in blocks of space called extents. Linkage maps exist in a file's directory which connect each extent. Access to a file is achieved by placing the file specification in a File Control Block (FCB), referencing a user disk file I/O buffer, and issuing the "OPEN" SVC. The provision of a separate file buffer for each file greatly adds to the system's flexibility. Directory information needed by the file access routines is then placed in the "open" FCB. Thereafter, SVC requests for file positioning, reading, and writing are available to access any record in the file. Fixed record lengths of from one to 256 bytes are available directly at the SVC level. Languages, such as BASIC, generally provide sequential files with variable record lengths.

Although the functions supported are many, a minimum of the machine's RAM space is required by LDOS. This is achieved by having only frequently used routines resident in memory while others are brought in to an overlay region on an as-required basis. All of the functions identified in Table 1-1, including the device and disk drivers (both floppy and hard), are contained in a 9K memory space which includes a 1.5K (1536 bytes) system overlay region. Another 3K region is used for the execution of system library commands but may be used by applications that do not request system library functions. Functionally, the DOS is divided into seven regions: system low core (LOWCORE), Input/Output driver region (IOR), resident system (SYSRES), System Overlay Region (SOR), Library Overlay Region (LOR), User Program Region (UPR), and high memory region (HIMEM). The UPR extends from X'3000' through HIGH$. Table 1-1 illustrates these regions. The DOS normally does not use HIMEM; however, certain user-specified requests must be satisfied by use of high memory. For example, SPOOL filter and buffer space use high memory. KSM filter and data space use high memory. A pointer to the top of HIMEM is available via an SVC and programs must honor this HIGH$ pointer.

The interrupt task scheduler listed in Table 2-1 under SYSRES schedules the execution of small background tasks at periodic intervals. The time intervals are determined primarily by a hardware generated interrupt to the Z-80 processor. A desirable minimum interrupt rate would be 40-60 Hz. This "clock" is software divided to produce "high", medium, and "low" level task control. The DOS provides for eight low level tasks, three medium level tasks, and one high level task. For example, with a 60Hz interrupt rate, one task can be performed at 16.7ms intervals, three discrete tasks can be processed at 33.3ms intervals while eight other tasks are processed at 267ms intervals. The types of tasks generally operating from such a scheduler would be software time of day routines, printer despooling routines, address trace functions, keyboard type ahead scanning, blinking cursor routines, or other processes that need to be examined at periodic intervals.

As a specific example of how software can reduce hardware costs, briefly examine keyboard type-ahead. This feature is quite significant to a fast typist. Even slow operator entry can gain from type ahead by the ability to enter responses in anticipation of known queries. Even if the hardware does not provide an interrupt generating keyboard, the DOS implements a 64-128 (depending on release) character type ahead buffer via task polling which is adequate for all operators.

NAME	START	END	DESCRIPTION
LOWCORE	X'0000	@$SYS	RST vectors, NMI vector, System flags, Date, Time, System FCB, DEBUG register save area, JCL FCB, Command FCB, SVC Table, DCB Table, System stack, Miscellaneous data, Command input buffer, Drive Control Table, Device I/O handler, Clock task, Memory management routines.
IOR	@SYS	X'12FF	Keyboard, Video, Printer, and Disk drivers.
SYSRES	X'1300	X'1DFF	File access routines, SVC processor, System overlay handler, System program loader, Interrupt Task Schedular, System buffer.
SOR	X'1E00	X'23FF	Execution region for system overlays 2-5, 9-13, overlay disk file buffer.
LOR	X'2400	X'25FF	Execution region for system library comands contained in libraries A, B, & C.
UPR	X'3000	(HIGH$)	Execution region for user transient programs (note: programs not accessing the system libraries can start at X'2600'.)
HIMEM	(HIGH$)+1	X'FFFF	Region for relocation of extended system and user static modules.

Table 2-1 System Map

The task scheduler is also used by the despooling function of the printer spooler. The DOS spooler implements a combination of memory and disk buffers to temporarily hold the printer output. This output is despooled to the printer under the control of the task scheduler. The function, being transparent to the user, can continue the despooling even after the application generating the output is finished and another started. When the system contains 128K (or more) of RAM, the extra RAM can be set aside for the spooler's memory buffer.

The primary function of any operating system is to provide the user with a facility for managing and accessing files stored on disk storage devices. Since the user must not be burdened with the physical details of the storage devices themselves, it is the operating system's responsibility to translate all file record access requests into specific drive, track, sector, and head parameters that pinpoint the storage location of each record. The DOS supports a wide range of disk storage capacities. Let's take a brief look at how a disk drive is organized.

Each track is formatted into a specific quantity of 256-byte sectors with a maximum capacity of 32 sectors per track. Sectors are grouped into blocks called "granules" which vary in size according to total track capacity. Whenever additional disk space is needed for a file, an additional granule is allocated. The granule thus becomes the minimum size storage unit. Where multiple headed drives are in use, the track numbers on a surface are duplicated on each surface with all similarly numbered tracks constituting a cylinder. Cylinder capacities also have an upper limit of 256 sectors per cylinder or eight granules per cylinder while the system supports a maximum of eight heads per drive.

In order to evenly use the entire surface of a drive, files are uniformly distributed across each surface [note: LSI unfortunately has changed to a fixed allocation scheme effective with release 6.1]. That means the head has a tendency to be randomly located whenever a directory access is needed. Because of this, each disk drive's directory is placed on the cylinder closest to its midpoint which provides a tendency to minimize the average seek time for directory accesses. The directory, of course, contains information on each file stored on the drive as well as additional tables and codes pertinent to the drive.

The first sector of the directory contains a granule allocation table (GAT). The GAT is bit mapped to each granule of space on the drive. Other fields in the GAT contain the

PACK NAME, DATE of creation, pack PASSWORD, and data pertaining to the configuration of the drive.

The system can support a capacity of 13 Megabytes of directly addressable storage on each of eight drives. Rigid disk drives of greater capacities can be supported by partitioning them into two or more logical drives. Also, where a physical parameter exceeds the upper limits, translation techniques can be used in software. Again, the flexibility of the system provided through intelligent software allows for easy interfacing.

When a file is to be opened for access, the system needs to search the directory for its directory record. Search time is minimized by using a hashing technique to reduce the 11-character string formed from the file name and extension to a one-byte value. The hash code for each file is stored in a Hash Index Table (HIT) which is the second sector of the directory. Each position in this table corresponds to a specific directory entry record. The hash table, being a sector in length, can index a maximum of 256 directory records or files. The directory itself is sized according to disk capacity by being a maximum of one cylinder (up to 34 sectors). Thus, the larger the disk storage capacity, the larger its directory, and the greater the number of file names that can be stored.

To open a file, therefore, the file name and extension are gathered from the specification and put through the hashing algorithm. The HIT sector is read and searched for a matching value. When a match is found, the directory sector containing the corresponding directory record is read. To guard against a different file name/ext hashing to the same value (which is called a collision), the 11-byte string is then checked for a match. If the correct record has not been retrieved, the HIT is examined further.

The directory record contains information such as the date the file was last modified, its update and access password codes, its access level, other attributes such as whether it is a SYStem or PDS file and if a backup has been made, the relative number of the last sector in the file and the last byte within the last sector. The record also contains the physical storage in use by the file by pointing to the cylinder, relative starting granule, and number of contiguous granules for each extent linking up the file. When a file has more than four extents, additional directory records are used as required with forward and backward pointers linking each record.

A feature considered important by many users is the flexibility of the file management utilities. These utilities include such functions as copying files from one drive to another, appending two files together, listing files with structured formatting, renaming files, removing files, obtaining disk directories, and making archival backups of your "favorite" files. All are popular functions with BACKUP being one of the most important in light of the tremendous capacity available when using large storage devices.

Ever since small winchester drives started to appear interfaced to small microcomputers, the question of how to backup these devices loomed large. Although some installations consider streaming tape for backup (relatively expensive as an added cost) while others are incorporating video cassette recorder interfaces (assumes the availability of VCRs at the micro site or another added cost), by far the most popular method has been the use of floppy diskettes (least expensive and widely available). Floppies do have a serious drawback. When comparing the available capacities of a single floppy to a small winchester, it soon becomes obvious that a good handful of diskettes are required to backup the hard drive.

A sophisticated backup utility can ease the frustration of archiving hard disk files. For one thing, with the availability of 80-track 2-headed minifloppies, over 700 Kilobytes can be stored on a single 5-1/4" diskette when recorded in double density. With 2-headed 8" drives, 1.2 Megabytes of storage exist on a floppy diskette. For another thing, the

backup utility provides exceptional flexibility as can be evidenced by the following command examples:

```
BACKUP :4 TO :2
```

will copy all files from logical drive 4 to logical drive 2. If both drives are floppies having the same physical configuration (i.e. both 40-track 2-headed with the same density), then the backup will automatically be performed track by track called "mirror image".

```
BACKUP /TXT:3 TO :5 (OLD)
```

will copy all files with a file extension of "TXT" from logical drive 3 to logical drive 5 but only if the file already exists on logical drive 5. The use of the "OLD" parameter permits organization of archival copies.

```
BACKUP R$S/BAS:4 TO :2 (MOD,DATE="11/09/82-11/15/82")
```

will make copies of all files from logical drive 4 with a filename starting with the character "R", the third character "S", with any character acceptable in all other file name character positions. Also, files must have been last modified between the dates of November 9, 1982 through November 15, 1982 inclusive in order to be included in the backup. In addition, the file must not have been backed up since it was last modified.

These examples illustrate the extreme flexibility of managing archival copies of working files. When used in a hard drive environment, large capacity floppy diskettes can be used to store selected "classes" of files with working files backed up in a structured manor only if they have been modified. Daily "churning" of working files is minimal, thus a procedure that enables a backup only if a modification has been done to a working file within a class certainly lends itself to optimum file management techniques without the need for expensive backup hardware. For those cases where a single file exceeds the capacity of a single floppy, a separate utility provides diskette spanning capabilities for the backup.

The command to obtain a directory display is used frequently in most machine environments. The DOS directory command listing is sorted by file name/ext. When the length of a listing exceeds the line capacity of the video display, paging is performed with a pause at each page. The listing provides data on the protection level, logical record length, file length (in kilobytes), date of last update, and whether a backup copy exists, for each file in the directory. A partial file specification can be requested to limit the listing to those files in the "class" similar to the BACKUP utility.

Disk files are supported with two types of access - Record I/O and character I/O. Logical Records of from one to 256 bytes in length can be read or written using the @READ or @WRITE SVC requests. Record I/O can be random access (by position SVC requests prior to READ/WRITE) or sequential access using repetitive READs or WRITEs. Character I/O is accomplished by @GET and @PUT SVC requests and is essentially the same as record I/O with a Logical Record Length (LRL) equal to one. However, if GET and PUT are used to implement sequential access, then a file can be considered a character I/O device just like a printer, a serial port, or a video display device. A byte I/O request is therefore independent of the physical device "connected" to the control block which is requesting the I/O. This makes the system "device independent".

Routing, filtering, and linking is 100% - devices may be routed to files and subsequently filtered and linked. A priority level hierarchy is established according to bit assignments in the DCB: file, NIL, route, link, and filter (file being the highest). Filters are assigned control blocks in the DCB table area which supports up to 31 entries. Each device driver and filter has its own entry. The establishment of a LINK also uses a DCB

entry to maintain the pointers used for each device in the LINK. Several system library commands, such as the FILTER, LINK, RESET, ROUTE, and SET commands, are provided that are used to support device independence. An illustration of the use of these commands lends well to understanding the full power of device independence. For example, if a suitable software driver (with a filename of RS232/DVR) is available for a serial port (RS-232 channel), then a simple:

```
SET *CL TO RS232
```

will establish the serial port as a device with "CL" as the device name. Now that such a device is available, the user can:

```
LINK *KI TO *CL
LINK *DO TO *CL
```

and the micro is established as a "host" because the serial communications line has been linked to both the machine's keyboard and its video display - the primary input and output devices of the machine.

Device I/O can also be massaged with transformation functions, called filters. For example, an EBCDIC to ASCII translation filter is available that when applied to the serial port by a simple:

```
SET *XL TO XLATE USING EBCDIC
FILTER *CL WITH XLATE
```

the micro can be tied to an IBM mainframe which supports only EBCDIC ports. Want to implement a DVORAK keyboard? By simply filtering the *KI device with the DVORAK translation filter, the keyboard is reorganized - with NO hardware changes required. Many filters are available to format print output, trap specific character codes, perform upper/lower case conversions - the limits are boundless. That's flexibility!

Now that you have a flavor of the capabilities of the DOS, this guide can be used to understand how to interface your programs. The bulk of LDOS Version 6 is machine independent. What this means to you as a programmer is that once you write an application to run under LDOS 6.x, it is portable to any machine running version 6. All you need do is utilize the standard interfacing procedures discussed in this programmers guide. Let the DOS do what an operating system is supposed to do - interface the application to the hardware.

3. Device Input/Output Interfacing

3.1 Device I/O In General

Devices interface to the operating system through driver modules. Character-oriented devices (keyboards, video display tubes, printers, and serial terminals, to name but a few), have their drivers connected to the DOS by Device Control Block (DCB) tables [this is in contrast to disk-type devices which have drivers connected to the system through Drive Control Tables (DCT)]. The purpose of the DCB is to associate a device name with the device hardware itself. A device specification (abbreviated as "devspec") is formed by prefixing an asterisk to the device name. Programs may then reference the device via the device specification in order to identify a particular device for character I/O.

There are three input/output functions that are associated with all character-oriented devices. The "GET" function obtains a character from the device. The "PUT" function sends a character to the device. The "CTL" function provides a means of communicating with the device driver and generally does not invoke input/output with the physical device itself. It is up to the device driver to ensure that the device is currently able to take the character in the case of PUT as well as detect the availability of a character in the case of GET and return the proper condition.

Disk files may also be interfaced via character I/O as well as record I/O [file access via record I/O is discussed in chapter 6, DISK FILE ACCESS AND CONTROL]. A disk file's actual physical storage location on a disk drive is transparent to the user by referencing the file with its associated name (more properly termed its file specification or "filespec"). The operating system permits filespecs and devspecs to be used equivalently in most cases. Character I/O is thus independent of a device or file. The DOS permits the redirection of character I/O at the command level. Because of this, applications must expect character I/O to be associated with a disk file as well as a standard character-oriented device. The DOS provides a uniform protocol for I/O handshaking regardless of character device.

There are three major operations associated with devices. One of these is "routing" which implements the support of I/O redirection. Another is linking which is used to connect two or more devices together. The third operation associated with devices uses filters to achieve filtering. Filters are program modules that can be logically placed between the Device Control Block associated with a device and the device driver connected to the DCB. This operation will form what is called a "device chain". More than one filter module may be placed in the DCB-to-driver chain. These filters bear a very close resemblance to device drivers. In fact, they also utilize the Device Control Block tables to associate their memory storage location with the name assigned to them when they are installed.

This section will discuss the activities that take place between a Device Control Block and a device so that you will better understand the concepts of character I/O. In this manner, you will have no problem in writing device filters and drivers - at least as far as DOS interfacing goes.

3.2 The Device Control Block

The Device Control Block (DCB) is used to interface with various logical devices such as the keyboard, the video display, a printer, a communications line, or other device defined by your hardware implementation. The DCB is composed of eight bytes divided into four fields: TYPE, VECTOR, SYSDATA, and NAME. Figure 3-1 illustrates the DCB. The TYPE field is a one-byte field that describes the capabilities and current state of the DCB (state indicative of routed, linked, filtered, etc.). The VECTOR field is a two-byte field that initially is a pointer to the entry-point of the driver or filter module associated with the DCB. The SYSDATA field is a three-byte field that is used by the

system to support linking and routing. The NAME field is a two-byte field that contains the name associated with the device.

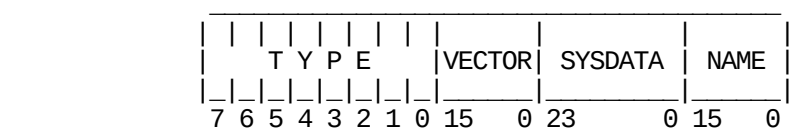

Figure 3-1: DCB Fields

The DCB follows a strict format that defines the utilization of all four fields. The programmer need be concerned only with the TYPE and VECTOR fields. The system requires sole use of the SYSDATA field. It also maintains the NAME field thus usually necessitating no programmer intervention. The DCB format must be followed in all Device Control Blocks established by the user. The following information provides specifications for each field of the DCB.

3.2.1 TYPE Field - Byte 0

Bit 7 This bit specifies that the Control Block is actually a File Control Block (FCB) with the file in an OPEN condition. Since there is a great deal of similarity between DCBs and FCBs, and devices may be routed to files, tracing a path through a device chain may reveal a "device" with this bit set, indicating a routing to a file.

Bit 6 This bit specifies that the DCB is associated with a FILTER module. The VECTOR field then contains the entry point of the filter. A filter initializer must set this bit when the module is assigned to the DCB.

Bit 5 This bit specifies that the DCB (say device AA) is linked to another device associated with a DCB (say device BB). The VECTOR field of AA will point to a dummy LINK DCB (say device LK) which was established by the system when the LINK library command was invoked. The VECTOR field of LK then will point to the original VECTOR contents of AA while the SYSDATA field will contain a pointer to the BB DCB. A picture is said to be worth a thousand words. The device chain linkage will be illustrated later.

Bit 4 This bit specifies that the device defined by the DCB is routed to another character-oriented device or file. The VECTOR field will either point to a DCB if the route destination is a device or it will contain a pointer to the file's FCB field contained in the route module established by the system's ROUTE library command.

Bit 3 This bit specifies that the device defined by the DCB is a NIL device. Any output directed to the device will be discarded. Any input request will be satisfied with a ZERO return condition.

Bit 2 This bit specifies that the device defined by the DCB is capable of handling requests generated by the @CTL SuperVisor Call.

Bit 1 This bit specifies that the device defined by the DCB is capable of handling output requests which come from the @PUT SuperVisor Call.

Bit 0 This bit specifies that the device defined by the DCB is capable of handling requests for input which come from the @GET SuperVisor Call.

3.2.2 VECTOR Field - <Bytes 1 - 2>

This field initially will contain the address of the driver routine that supports the device hardware associated with the DCB. In the case of programmer-installed drivers, the driver initialization code must load the driver's entry point into the VECTOR field of its respective DCB. Likewise, when a filter module is established (via the SET library command), its entry point is placed into the VECTOR field. Once established by either the system or the driver/module initialization code to point to the module's entry point, the VECTOR field is then maintained by the system to effect routing, linking, and filtering.

3.2.3 SYSDATA Field - <Bytes 3-5>

These three bytes are used by the system for routing and linking and are unavailable for any other purpose.

3.2.4 NAME Field - <Bytes 6 - 7>

Byte 6 of this field contains the first character and byte 7 the second character of the device specification name. The system uses the device name field as a reference in searching the Device Control Block tables. When a DCB is assigned by the system during a SET or ROUTE command, this device name field will be loaded by the system with the device specification name passed in the command invocation. Programs requesting a spare DCB via the @GTDCB SuperVisor Call (and a binary ZERO name), are responsible for loading this name field.

If the device has been routed to a file and a search of the device chain shows a TYPE byte with bit-7 set, then the respective control block is an FCB. In this case, byte 6 of the field will contain the DRIVE number of the drive containing the file and byte 7 will contain the Directory Entry Code (DEC) of the file.

3.3 ACCESSING DEVICE CONTROL BLOCKS

The system maintains space in low memory for the storage of the Device Control Block records. There is space sufficient for 31 records. The first DCB will always be associated with the system device named *KI. Therefore, a pointer to the first block may be determined by using the @GTDCB SuperVisor Call as follows:

```
LD      DE,'IK';Load name in reverse order
LD      A,@GTDCB        ;Identify the SVC
RST     40              ;Invoke the SVC
JP      NZ,ERROR        ;Transfer if not found
```

Upon return from the SVC, register HL will contain a pointer to the DCB associated with *KI. An error will result only if the DCB name field was altered. Spare DCB records are filled with binary zeroes. Therefore, a spare DCB record may be located by loading register pair DE with a binary zero value prior to issuing the SVC.

The DOS command "DEVICE (B=Y)" can be used to obtain a linkage map of all device chains. As can be observed from such a listing, all 31 control blocks are not in use. Additional devices are defined by using the SET library command. Any device assigned by the user to a spare control block, may be removed from the system after the device is RESET by using the "REMOVE devspec" command. The DOS defined devices are protected and cannot be removed.

3.4 DEVICE CHAIN ILLUSTRATIONS

Before we can illustrate the device chain, it is necessary to first reiterate the memory module header protocol as required by the system. It is essential that this protocol be used for all modules placed into protected memory so that the system can properly deal with module access and device I/O.

3.4.1 Header Protocol

Each module placed into protected memory will incorporate a facsimile of the following code at the start of the module:

```
ENTRY       JR     BEGIN          ;Branch around linkage
STUFHI      DW     $-$            ;To contain last byte used
DB          MODBGN-ENTRY-5        ;Calculate length of 'NAME'
DB          'MODNAME'             ;Name of this module
MODDCB      DW     $-$            ;To contain DCB pointer for module
SPARE       DW     0              ;Reserved by the DOS
      .
      .                           ;Any data area needed
BEGIN       EQU    $              ;Followed by module code
```

Chapter 8, the appendix, is another source of information concerning the header protocol. It is sufficient for the illustration of device chains to understand that the MODDCB will contain a pointer that points to the Device Control Block established for the module during the execution of the SET library command. This pointer is passed in register pair DE to the module's initialization code by SET. The programmer writing the module code adds a routine which loads this valuue into MODDCB.

3.4.2 Sample DCB Structure

For the purpose of this illustration, let's imagine three active DCB s. The first DCB is associated with the printer driver and has device specification of "*PR" (its devspec). We have also installed a filter via the SET command that performs a backspace followed by the output of a slash when it detects an ASCII zero (0). This filter has a devspec of "*S0". Lastly, we have a filter that toggles a boldface mode for a printer. This filter has a devspec of "*BF". To avoid confusion in the illustration, the devspec will be used to reference the DCB and the module names PRINTER, SLASH0, and BOLDFACE will be used to identify the entry point of the driver or filter module.

We can now show this arrangement of DCB contents and module MODDCB contents as follows:

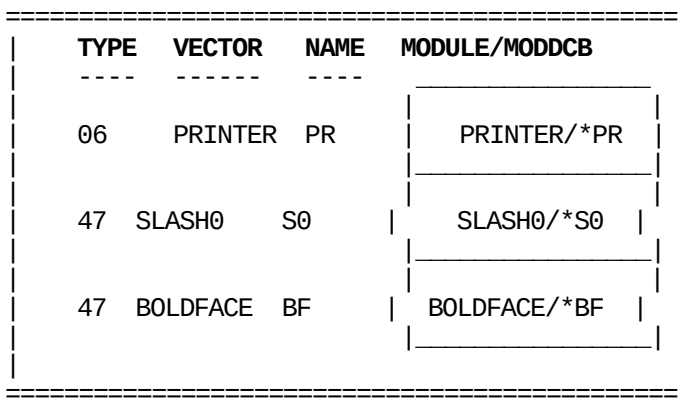

Figure 3-2: Initial DCB Table

Note that the DCBs in figure 3-2 associated with the filters have bit-6 of the TYPE byte set to indicate that they are filters. Also note that the MODDCB pointer points to the DCB which points to the module. Where a filter's MODDCB is pointing to the DCB of the filter, this is indicative of an inactive filter.

3.4.3 Filtering

Filters are written (as you will later learn) to perform all I/O via the @CHNIO SuperVisor Call. This SVC uses the contents of MODDCB within the filter invoking the SVC. Thus, the filter I/O is independent of any address by being handled completely through the SVC. If you perform a system command such as:

```
FILTER *PR USING *S0
```

the operating system will swap the first three bytes of the *PR DCB with the *S0 DCB. This arrangement will establish that shown in figure 3-3.

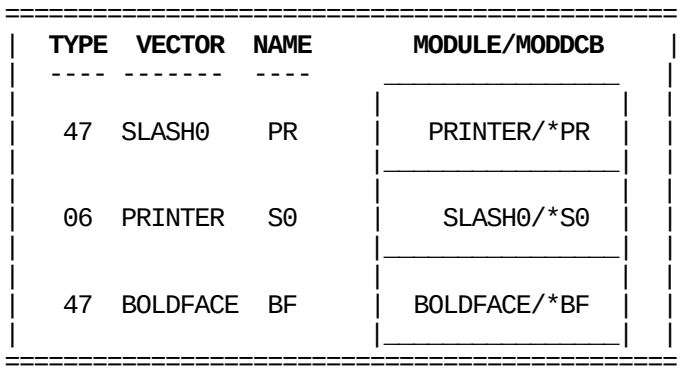

Figure3-3: DCB Table Modified

Let's follow what happens to an @PUT which references the *PR device. The system passes control to SLASH0 (which is pointed to by the *PR vector). This filter performs its character transformation, as required, and sends characters down the chain by picking up the pointer contained in its MODDCB (a pointer to the *S0 DCB) then issuing the @CHNIO SVC. The SVC handles the call by passing control to PRINTER which is the pointer now stored in the VECTOR field of *S0.

If we now try to issue the command:

```
FILTER *PR USING *S0
```

the system will prohibit it since the *S0 Device Control Block does not show up as a filter (bit-6 of the TYPE byte is reset!). However, if we filter *PR using the *BF device, we achieve the arrangement in figure 3-4 after the system swaps the first three bytes of *PR with the first three bytes of *BF.

Examine the arrangement in figure 3-4 closely. Note that the contents of MODDCB for each module are exactly what they were initialized to. Even though the *PR device has been twice filtered, the module itself needs absolutely no change whatsoever. An *PUT to the *PR device (say with an *PRT SVC) may be a little more complicated now, but functions perfectly well. The system first passes control to BOLDFACE (which is pointed to by the *PR vector). This filter performs its necessary device stream massaging and sends characters down the chain by picking up the pointer contained in its MODDCB (a pointer to the *BF DCB) then issuing the @CHNIO SVC. The SVC handles the call by passing control to SLASH0 which is the pointer now stored in the VECTOR field of *BF. The SLASH0 filter performs its character transformation, as required, and sends characters down the chain by picking up the pointer contained in its MODDCB (a pointer to the *S0 DCB) then issuing the @CHNIO SVC. The SVC handles the call by passing control to PRINTER which is the pointer now stored in the VECTOR field of *S0. Upon completion, a series of RET instructions pass the return code back through the modules making up the chain.

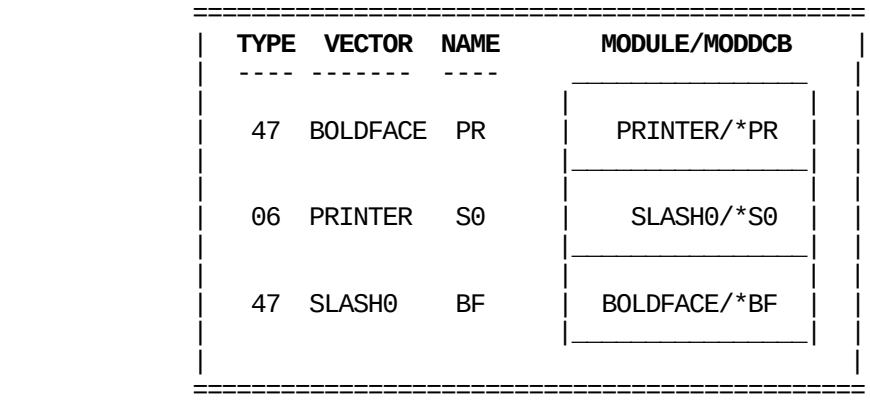

Figure 3-4: DCB Table Further Modified

It is interesting to observe that the process of removing the filters from the device chain is exactly the same as the process to add them into the chain. We can unhook the filters by exchanging the first three bytes of the DCBs in the order of last-in first-out (LIFO). Thus if you exchange the *PR and *BF Device Control Block TYPE and VECTOR fields, you will obtain the arrangement previously shown in figure 3-3. The RESET library command does this for the entire chain.

By now you should be able to notice that we could equally as well remove just the SLASH0 filter if we swap the bytes associated with the *BF and *S0 Device Control Blocks! All that is needed is a facility to do the following:

1. Identify what filter (by module name) is to be removed;
2. Locate the filter in memory via the @GTMOD SuperVisor Call;
3. Obtain the MODDCB pointer to its Device Control Block;
4. Scan through all DCBs to find the DCB pointing to the filter;
5. Then swap the three bytes.

3.4.4 Routing

Routing conveys the facility of I/O redirection. This function allows programs to be independent of the physical device actually handling the I/O. By maintaining a constant reference within a program to a particular DCB, the physical I/O can be channeled to some other device completely transparent to the program. This is achieved through altering the connection between the DCB and its initial driver by reconnecting the DCB to some other driver. The operating system handles all of the functions of implementing the DCB alteration when the ROUTE library command is invoked. The "routed-to" device may be another DCB identified by a devspec or it could be a disk file identified by a filespec. Let's look at an example.

If we, for instance, invoke the command:

```
ROUTE *PR TO FILE/TXT:3
```

the DOS performs a two-stage process. First, it establishes a 32-byte File Control Block and 256-byte buffer for the FILE/TXT:3 disk file. It places this "data" into high memory prefixed with the header protocol. Second, it saves the "route-from" VECTOR and TYPE fields in the SYSDATA field of the DCB while it revises the VECTOR to point to the "routed-to" FCB. The TYPE field is also altered to show a ROUTE is in effect. The DCBs will now look like figure 3-5.

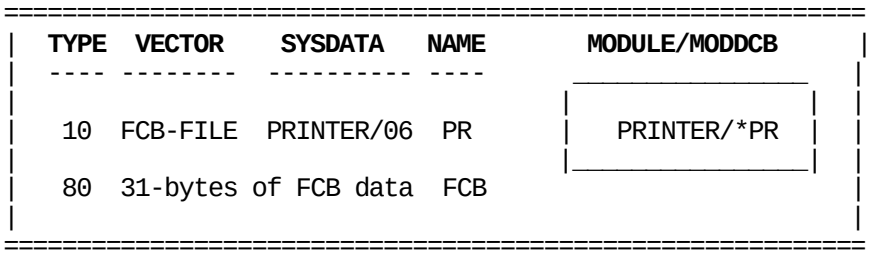

Figure 3-5: DCB Table After ROUTE

Let's now follow an output request to the *PR device. The DOS device I/O handler will recognize the ROUTE bit (bit-4 of the TYPE byte) and update the register linkage so that the FCB will be pointed to instead of the DCB. Noticing that the control block now indicates a disk file (bit-7 of the TYPE byte), the I/O handler will pass control to the character I/O file routines.

The action taken by the operating system to reset a DCB that has been routed is to first close the file, if a filespec was the initial "route-to", then recover the original TYPE and VECTOR from the SYSDATA field.

3.4.5 Filtering a Routed Device

Let's suppose we have a text file that needs line feeds removed (it may be a CP/M file that uses CR-LF as the end-of-line protocol). We could write a program to read the file and write out to another file all characters that are not a line feed. We could also use a trap filter that is handy. We want to be able to filter the file with this trap filter. Using the routing identical to that shown in figure 3-5, establish the trap filter and invoke it with:

```
SET *LF USING TRAP (CHAR=10)
FILTER *PR USING *LF
```

Figure 3-6 will now reflect the DCB structure after this series of commands. It is now easy to LIST the source file with the (P,T=N) option. This will direct a copy of the file to the *PR device (while suppressing tab expansion). As can be observed from the figure, the device handler passes *PR I/O requests to the TRAP filter. After performing whatever filtering is necessary, the @CHNIO request will reference the *LF Device Control Block (which is pointed to by the MODDCB field). The device handler then notes that the ROUTE bit is set and continues to control the @PUT request as was done under figure 3-5. A simple "RESET *PR" upon completion will close the filtered FILE/TXT.

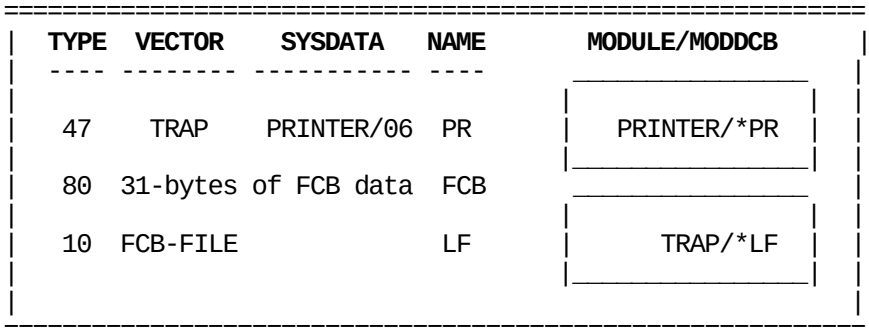

Figure 3-6: Filtering a Route

3.4.6 Linking

Linking is handled by establishing a link Device Control Block storage area for each LINK command invoked. For example, if you "LINK *DO TO *PR", we can illustrate the DCB area as

shown in figure 3-7. The *DO Device Control Block now vectors to the newly established *L0 DCB while the TYPE byte identifies the link. Notice that *L0 has both the VIDEO vectors and a pointer to the *PR DCB (we can conceptualize this as a two legged fork). The system's device handler recognizes that a link is in effect (from *DO's TYPE byte) whereupon it establishes a fork via the link DCB, *L0. It uses the third byte of L0's SYSDATA field to store the direction indicator. After a return from VIDEO without error, the device handler takes the other fork leg (to *PR).

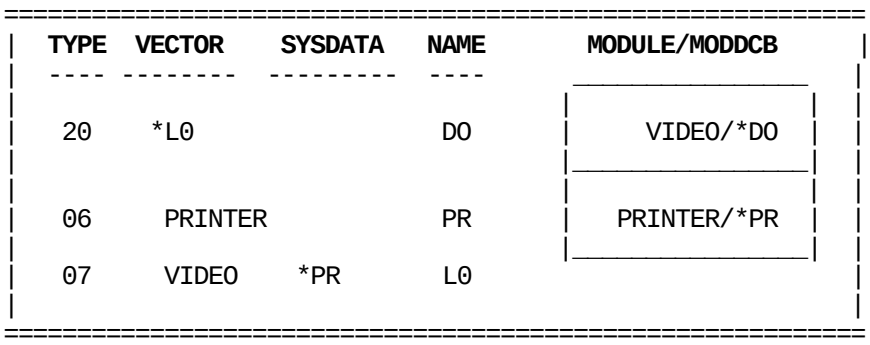

Figure 3-7: Linking Devices

The legs of the fork are entered based on the I/O direction and the return code from a leg. @PUT requests will be sent to the "left" leg of the fork. Providing no error is encountered, the "right" leg of the fork will be entered. The return code passed back to the caller will be either an error from the left leg, an error from the right leg, or a no-error condition. Requests for @GET, will be passed first to the left leg. Only if the left leg has no input available will the right leg be entered. @CTL requests are handled like @PUT.

Linking can be applied to a devspec that has been filtered, routed, or linked. There is no restriction on combinations. Thus, you can link a device that is already linked and filtered and routed. Figure 3-8 depicts the result of linking a device that has already been routed. It is left up to the reader as an exercise to derive the series of commands that composed the associated DCBs/FCB as well as tracing through the device chain for I/O.

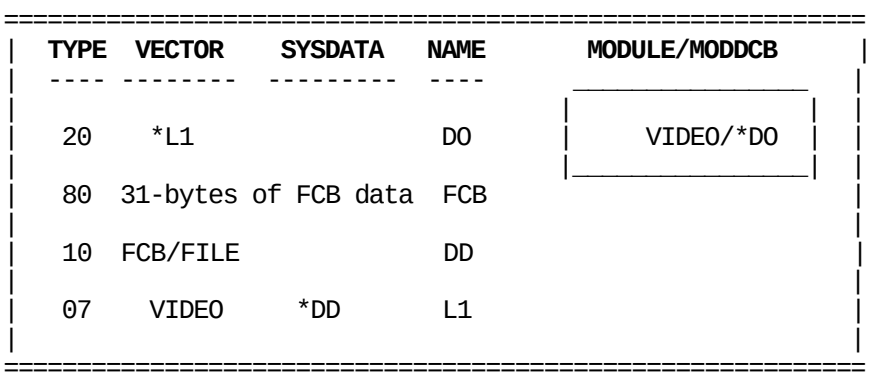

Figure 3-8: Linking a Routed Device

3.4.7 Device Chain Hierarchy

It is possible for the Device Control Block TYPE byte to have more than one bit set in the positions 3-7 (positions 0-2 usually have multiple bits set depending on the I/O supported by the driver). Because of this, the system must utilize a priority level to indicate what function is to be interpreted. The device I/O handler hierarchy is illustrated in figure 3-9.

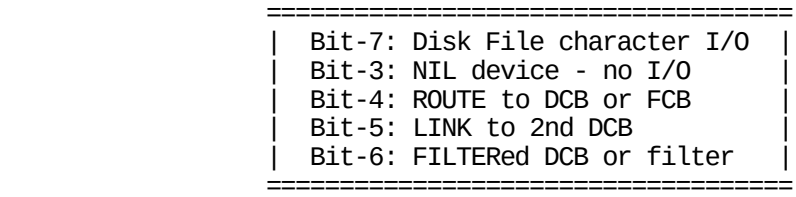

Figure 3-9: DCB Hierarchy

3.4.8 Device Chain Summary

The preceding discussion should shed a great deal of light on the handling of device I/O by the operating system. You should also understand that in order to accomplish this device independence and flexible handling of character I/O, the programmer of device drivers and filters must adhere to a strict protocol of handshaking the modules with the operating system. The next section will explore device I/O looked at from the standpoint of the modules and drivers. Once you grasp these requirements, you will be in total control of filters and device drivers.

3.5 DEVICE DRIVER/FILTER TEMPLATE

The system contains command level procedures that provide easy access to device references so that modifications may be made to the way in which devices are treated by the system. All devices require some type of driving program (a device driver) that is used to handshake the device with the system and cater to the special features and requirements of the device hardware. Some drivers are already implemented within the operating system to handle standard devices. For instance, drivers for handshaking the keyboard, video display, parallel printer port, and RS-232 serial port are included with the system.

Some devices are completely supported with the existing drivers in the total DOS environment. Other devices may need a little more support. The characteristics of a driver may be modified by the introduction of a FILTER. For instance, suppose your printer required a line feed upon receipt of a carriage return to advance the paper. The printer driver does not provide this function. Instead of writing a completely new printer driver, only a filter need be included to add that single function (the FORMS/FLT filter which incorporates this function is usually provided with the system).

The DOS provides two commands to aid in interfacing drivers and filters. The SET command is used to define a new device, re-define an existing device, or install a filter module while assigning it a device name. FILTER is used to place the installed filter into an existing device chain.

The SET command takes the device specification from the command line "SET *XY to filespec" and searches the Device Control Block tables for a matching device name. If the requested device is not defined in your configuration, SET establishes a Device Control Block for the new device. Control then passes to the DRIVER or FILTER with register pair DE containing the address of the Device Control Block record assigned to the "SET" device.

Register pair HL points to the command line character separating the DRIVER/FILTER program filespec and optional parameters. This provides the module initialization routines with the opportunity of parsing a parameter string by using a parameter table and the @PARAM SuperVisor Call. SET provides a default file extension of /FLT since the function of adding filters to the system is the more usual case.

The SET and FILTER commands are designed such that the DRIVER or FILTER program should first load into the User Program Region (starting at X'3000'). After parsing any options or parameters, the module initialization routine automatically relocates the resident

module to high memory (or low memory if sufficient space is available - see the section on Placing Disk Drivers in chapter 4). HIGH$ (or the Driver Input/Output Region pointer) must be properly set after your module relocates.

Samples of filters are provided in Chapter 8, the appendix, which should demonstrate the technique of writing the relocating driver portion of your routine. The remaining sections in this chapter discuss the handshaking and initialization requirements necessary for device drivers and filters.

3.5.1 I/O Primitives

Device independence has its roots in "character I/O". The term shall apply to any I/O passed through a device channel, one character or byte at a time. Three primitive routines are available at the assembly language level for byte I/O. Primitive is not used here to imply rudimentary but rather elementary. Just as the atom is considered a basic building block of molecules, these byte I/O primitives can be used to build larger routines. The three DOS SuperVisor Calls are designated @GET , @PUT, and @CTL. @GET is used to input a byte from a device or file. @PUT is used to output a byte to a device or file. @CTL is used to communicate with the driver routine servicing the device (the character file I/O routines ignore @CTL requests).

Other SuperVisor Calls are available that perform byte I/O, such as @KBD (scan the *KI device and return the key code if a character is available), @DSP (send a character to the *DO device), and @PRT (send a character to the *PR device). These functions operate by first loading register pair DE with a pointer to a specific Device Control Block (DCB) assigned for use by the device, then issuing an @GET or @PUT SuperVisor Call for the respective input or output requests.

When the DOS device handler passes control over to the device driver routine, the Z-80 flag conditions are unique for each different primitive. This provides a method that the drivers can use to establish what primitive was used to access the routine and thus distribute the I/O request to the proper driver or filter subroutine - according to the direction of the request - input, output, or control! Figure 3-10 illustrates the FLAG register conditions prevailing upon entry to a driver or filter.

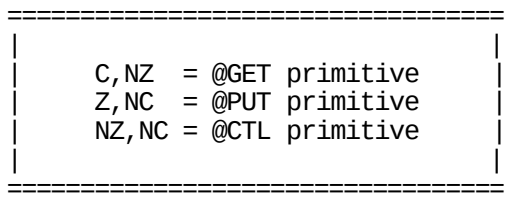

Figure 3-10: Flag Conventions

Register B contains the I/O direction code (1 = GET, 2 = PUT, or 4 = CTL) while register C will contain the character code that was passed in an @PUT or @CTL SuperVisor Call. Register IX will point to the TYPE byte of the Device Control Block being referenced. Registers BC, DE, HL, and IX have been saved on the stack and thus are available for use. Remember that any given module may have been filtered or linked; therefore, do not expect the DCB address in IX to be a constant over time. If the module is a filter, it will be invoking the @CHNIO SuperVisor Call. Thus it will be important to save those registers that must stay unchanged prior to invoking @CHNIO.

3.5.2 I/O Separation

Now let's move on to the device driver linkage used to separate out the @GET, @PUT, and @CTL calls. Remember the FLAG register direction conditions shown in figure 3-10 that were set according to the primitive byte I/O routine that got us to the driver. These conditions provide the key to the separation process. Consider the following protocol for the driver or filter header.

```
ENTRY   JR      BEGIN           ;Branch around linkage
STUFHI  DW      $-$             ;To contain last byte used
        DB      MODDCB-BEGIN-5 ;Calculate length of 'NAME'
        DB      'MODNAME'       ;Name of this module
MODDCB  DW      $-$             ;To contain DCB pointer for module
        DW      0               ;Reserved by the DOS
BEGIN   EQU     $
;*=*=*
;       Actual module code start
;*=*=*
        JR      C,WASGET        ;Go if @GET request
        JR      Z,WASPUT        ;Go if @PUT request
        JR      WASCTL          ;Was @CTL request
```

At the entry of the driver, an absolute relative jump instruction executes which causes a branch around some data. Ignore, for a moment, the header data which is discussed in the Chapter 8, appendix. At the label "BEGIN", a test is made on the CARRY FLAG. If the CARRY was set, then it must have been the result of an input request (@GET). Thus, an input request could be directed to that part of the module which handles character INPUT.

If the request was not from the @GET primitive, the CARRY will not be set. The next test is if the ZERO FLAG is set. The ZERO condition prevailed when an @PUT primitive was the initial request. Thus the jump to WASPUT can transfer to that part of the module that deals specifically with character OUTPUT.

If neither the ZERO nor CARRY flags are set, the routine falls through to the next instruction, a jump to WASCTL - that part of the module that would handle @CTL requests. Obviously, the module code that handles @CTL requests could be placed immediately after the first two tests thereby obviating the need for the "JR WASCTL". Some modules are written to assume that @CTL requests are to be handled exactly like @PUT requests although this is not recommended. The processing of @CTL requests is entirely up to the function of the driver and the author thereof with the exception that the author should not deviate from the functions identified in the @CTL INTERFACING section. When a device has been routed to a disk file, the DOS will ignore @CTL requests. That is, the @CTL codes will not be written to the disk file. The functions of @CTL requests are covered as a separate topic later in this chapter.

3.5.3 Device Driver/Filter Return Codes

One last topic needs to be discussed relating to drivers - the subject of register handshaking conventions. On @GET requests, the character input should be placed in the accumulator. On output requests (either @PUT or @CTL), the character is obtained from register C. It is extremely important for drivers and filters to observe return codes. Specifically, if the request is @GET and no byte is available, the driver returns an NZ condition with the accumulator containing a zero (i.e. OR 1 : LD A,0 : RET). If a byte is available, the byte is placed in the accumulator and the Z-flag is set (i.e. LD A,CHAR : CP A : RET). If there is an input error, the error code is returned in the accumulator and the Z-flag is reset (i.e. LD A,ERRNUM : OR A : RET). On output requests, the Z-flag is set if no output error occurred. The accumulator may be loaded with the character that was output; however, applications invoking an @PUT cannot depend on the accumulator containing the output character on return from the SVC - the character will, however, still be contained in the C register! In the case of an output error, the accumulator must be loaded with the error code and the Z-flag reset as shown above.

3.5.4 Filter Interfacing

A filter module is inserted between the DCB and driver routine (or between the DCB and the current filter when applied to a DCB already filtered). The application of insertion is performed by the DOS FILTER command once the filter module is resident and associated

with a device name. The function of residing a filter module is a responsibility shared by the SET library command and the programmer's filter initialization routine.

The usual linkage for a filter is to access the chained module by calling the @CHNIO SuperVisor Call with specific linkage data in registers IX and BC. Register IX is loaded with the filter's DCB pointer obtained from the memory header MODDCB pointer. Register B must contain the I/O direction code (1 = GET, 2 = PUT, 4 = CTL). This code is already in register B when the filter is entered. You can either keep register B undisturbed or load it with the direction code based on the primitive request. Also, output requests will expect the output character to be in register C.

3.5.5 Filter Initialization

The DCB pointer obtained from MODDCB for the interfacing, is originally obtained from the operating system. It is passed in register DE by the SET command and is loaded into MODDCB by your filter initialization routine. The initialization routine also relocates the filter to high (or low) memory while adjusting any absolute address reference with a suitable relocation routine. The DOS takes care of loading the DCB's NAME field with the associated device name passed in the SET command. The filter initializer must attach itself to the DCB assigned by the SET command by loading the TYPE and VECTOR fields. The TYPE field is loaded with an ORing of the filter bit (bit-6) and any valid direction bits (bits 0-2). If the initialization front end transfers the DCB pointer from DE to IX and loads the filter's entry address into register pair HL, the following code could be used to establish the TYPE byte and vector for a filter which supports GET, PUT, and CTL:

```
        LD      (IX),40H.OR.7  ;Init DCB type to
        LD      (IX+1),L       ;  FILTER, G/P/C I/O,
        LD      (IX+2),H       ;  & stuff vector
```

One final point concerns a test that should be made by the filter initializer. The operating system permits the execution of any load module. A filter program is a load module. To guard against the execution of a filter program by inadvertently entering its full file specification at DOS Ready, the system provides the programmer with an indicator that execution is under control of the SET command. When SET passes control to a filter program, it will set bit-3 of the CFLAG$ (the system request bit). Thus, by testing this bit upon entry to the program, an error exit can be taken if the system request bit is not set. An error message of the form:

```
Must install via SET
```

can be logged and the program aborted. The system automatically resets the system request bit upon regaining control at DOS Ready.

3.5.6 A Partial Filter

A filter module can operate on input, output, control, or any combination based on the author's design. The memory header provides a region for user data storage conveniently indexed by the module. An illustration of a filter follows. The purpose of the filter is to add a line feed on output whenever a carriage return is to be sent. Although the filter requires no data storage, the technique for accessing data storage is shown. Pay close attention to the method of passing characters to the device chain (@CHNIO).

```
ENTRY   JR      BEGIN          ;Branch to start
        DW      FLTEND-1       ;Last byte used by module
        DB      6,'SAMPLE'     ;Name length and name
MODDCB  DW      $-$            ;Ptr to DCB loaded by initialization
        DW      0              ;Reserved
;*=*=*
;       Data storage area for your filter
;*=*=*
DATA$   EQU     $
```

```
DATA1    EQU      $-DATA$
         DB       0              ;Data storage
DATA2    EQU      $-DATA$
         DB       0              ;Data storage
;*=*=*
;        Start of filter
;*=*=*
BEGIN    JR       Z,GOTPUT       ;Go if @PUT
;*=*=*
;        @GET and @CTL requests are chained to the next module
;        attached to the device. This is accomplished by falling
;        through to the @CHAINIO call. Note that the sample filter
;        does not effect the B register, so the filter does not
;        have to load it with the direction code.
;*=*=*
FLTPUT   PUSH     IX             ;Save our data pointer
         LD       IX,(MODDCB)    ;Grab the DCB vector
RX01     EQU      $-2
         LD       A,@CHNIO       ; & chain to it
         RST      40
         POP      IX
         RET
;*=*=*
;        Filter code
;*=*=*
GOTPUT   LD       IX,DATA$       ;Base register is used to
RX02     EQU      $-2            ; index data as (IX+DATA1),...
;
         LD       A,C            ;P/u char to test
         CP       CR             ;If not CR, put it
         JR       NZ,FLTPUT
         CALL     FLTPUT         ; else put it
RX03     EQU      $-2
         RET      NZ             ;Back on error
         LD       C,LF           ;Add line feed
         JR       FLTPUT
FLTEND   EQU      $
;*=*=*
;        Relocation table
;*=*=*
RELTAB   DW       RX01,RX02,RX03
TABLEN   EQU      $-RELTAB/2
```

The relocation table, RELTAB, would be used by the filter initialization relocation routine. Complete filters are listed in Chapter 8, the appendix.

3.5.7 External Access of Module Data

It is sometimes necessary to access the data region of a resident module from outside the module. Perhaps a utility to alter the data is useful (for instance, the SETCOM command alters the data of the COM driver supplied with the system. The @GTMOD SuperVisor Call is used to obtain two pointers. One points to the entry point of the module while the other points to the MODDCB field. If the data is located immediately following the reserved word in the module header, incrementing the MODDCB pointer by four will point it to the data area. The utility uses the module name assigned in the header to locate the module in memory. As an example, let's illustrate an update to DATA1 in the above filter.

```
         LD       DE,FLTSTR$     ;Point to module name
         LD       A,@GTMOD       ;Identify the SVC
         RST      40
         JR       NZ,NOTRES      ;Process "module noot resident"
         LD       HL,4           ;Use pointer in DE to
         ADD      HL,DE          ;  index past MODDCB & reserved
```

```
        LD      A,(VALUE)       ;P/u your new value
        LD      (HL),A          ;  & stuff into resident module
        .
FLTSTR$ DB      'SAMPLE',3      ;Search string
```

3.6 @CTL INTERFACING TO DEVICE DRIVERS

This section discusses the @CTL functions supported by the system supplied device drivers. @CTL functions are invoked by loading register pair DE with a pointer to the Device Control Block (DCB), loading the function code into register C, and issuing the @CTL SuperVisor Call. The DCB address can be located by either using the @GTDCB SVC or OPENing a File Control Block containing the device specification and using the FCB address.

The DOS has assigned function codes for specific operations. Although these operations are not universal across all drivers, the designated function code should be used only for the operation assigned. Rarely will you find a driver that utilizes all of these codes. A driver that accepts a function code to perform an operation should provide a return code as if the request was @PUT. Where a driver does not wish to accept a specific code or codes, it should return a "no-error" result. Function codes in the range <0-31,255> are reserved by the operating system. Function codes in the range <32-254> are available for programmer use. The following operations are assigned function codes:

CODE	OPERATION
0	Return status of device (Z = available, NZ = not available). Where applicable, return an image of the status in the accumulator.
1	Request a <BREAK> or force an attention interrupt.
2	Execute any driver initialization code.
3	Reset any driver buffers and clear any pending I/O.
4	Interface a "wakeup" vector for interrupt driven drivers. Register IY should contain the execution transfer address to be passed control after the driver handles the interrupt. On return from the @CTL call, register IY will contain the previous "wakeup" vector. If a zero is passed in register IY, the "wakeup" vectoring will be disabled.
5	Reserved by the DOS.
6	Reserved by the DOS.
7	Reserved by the DOS.
8	Return the next character in the input buffer but do not empty it from the buffer. A return condition of A = 0 and NZ indicates no character is pending. A <> 0 and NZ indicates an error while Z indicates success while A contains the character.
9-31	These codes are reserved by the DOS.

The system-supplied drivers support some of these functions. The following sections cover what control functions are supported and suggests possible uses. The module name can be used with the @GTMOD SuperVisor Call to obtain the entry point of the driver. This is useful to obtain access to the data areas associated with each driver.

3.6.1 Keyboard driver [system driver assigned to *KI]

A function value of X'03' will clear the type-ahead buffer. This serves the same purpose as repeated calls to @KBD until no character is available. A function value of X'FF' will remain undocumented as its use is proprietary to Tandy Corporation and its function is not supported across all licensed versions of LDOS Version 6. All other function values are treated as @GET requests.

The module name assigned to this driver is "$KI". Its data area includes the following:

Offset	Contents
+0	Contains the last character entered.
+1	Contains the repeat time check which is the system's timer value that when reached will result in a repeat of the last character if the keycode scanned has not changed.
+2	Contains the waiting time in timer units that must transpire before a character can initially be repeated. This value is altered by SETKI (W=dd).
+3	Contains the repeat rate in timer units. This value is altered by SETKI (R=dd).

3.6.2 Video driver [system driver assigned to *DO]

All @CTL requests are treated as if they were @PUT requests.

The module name assigned to this driver is "$DO". Its data area includes the following:

Offset	Contents
+0	Bits 0-2 contain the number of video lines to protect against scrolling. Bit 3 denotes the action to be taken for character values in the range <192-255>. If set, the values are treated as displayable characters. If reset, the values are treated as space compression codes in excess 192 (i.e. 0-63). Bit 4 will denote the action to be taken for character values in the range <1-31>. If set, the value is interpreted as a displayable character. If reset, the value is treated as a video function code as identified in your operating system user manual. Bits 5-7 are reserved by the DOS.
+1	Contains the low order address of the cursor. You must use the @VDCTL SuperVisor to reference the cursor by row,column.
+2	Contains the high order address of the cursor. You must use the @VDCTL SuperVisor to reference the cursor by row,column.
+3	Contains the character that is currently at the cursor position.
+4	Contains the character code defining the cursor.

3.6.3 Printer driver [system driver assigned to *PR]

The printer driver is transparent to all code values when requested by the @PUT SuperVisor Call. That means that all values from X'00' through X'FF' (0-255) can be sent to the printer. The printer driver accepts a function value of X'00' via the @CTL request to return the printer status. If the printer is available, the Z-flag will be set and the usual A register status image is an X'30'. If the Z-flag is reset, the accumulator will contain the four high-order bits of the parallel printer port (bits 4-7).

The module name assigned to this driver is "$PR". There exists no data area within the printer driver.

3.6.4 Forms Filter [non-resident system filter for forms control]

If the FORMS filter is attached to the *PR device, then various codes are trapped and used by the filter according to user options as follows:

Code	Filter Action
X'0D'	Generates a carriage return and optionally a line feed (ADDLF). It will form feed as required.
X'0A'	Is treated the same as X'0D'.
X'0C'	Will form feed (via repeated line feeds if soft form feed).
X'09'	Will advance to the next tab column.
X'06'	Will set top-of-form by resetting the internal line counter to zero.

Other character codes may be altered depending on the user translation option (XLATE).

The FORMS filter's module name is "$FF". Its data area includes the following:

Offset	Contents
+0	Contains the maximum lines per page.
+1	Is used by the filter as a line counter.
+2	Contains the maximum number of lines to print prior to a FORM FEED operation.
+3	Is used by the filter as a character counter.
+4	Contains the character value that is to be translated.
+5	Contains the character value that <+4> is to become.
+6	Contains the number of spaces to indent after an automatic NEWLINE is issued.
+7	Bit 0 specifies that a LINE FEED is to be added after each carriage RETURN. Bit 1 specifies the mode of FORM FEED - a 0 indicates SOFT (multiple line feeds) while a 1 indicates HARD (send X'0C' to the driver).
+8	Contains the maximum number of characters to print on a line prior to issuing an automatic NEWLINE. A value of zero indicates that no automatic NEWLINE is to be issued.
+9	Contains the column of the left hand margin. The filter will provide this count of spaces after a physical carriage RETURN.

3.6.5 COM driver [non-resident system driver for the RS-232C]

This driver handles the interfacing between the RS-232C hardware and character I/O (usually the *CL device).

An @CTL function value of X'00' will return an image of the RS-232 status register in the accumulator. The Z-flag will be set if the RS-232 is available for "sending" (i.e. transmit holding register empty and flag conditions matching as specified by the default protocol or that established by the user via SETCOM). A function value of X'01' will transmit a "modem break" until the next character is @PUT to the driver. A function value of X'02' will re-initialize the serial port hardware to the values last established by SETCOM. A function value of X'04' will enable/disable the WAKEUP feature. All other function values are ignored and the driver will return with register A containing a zero value and the Z-flag set.

The WAKEUP feature deserves additional treatment since it can be quite useful for application software specializing in communications. The RS-232 hardware is usually equipped with the capability of generating a machine interrupt when any of three conditions prevail: transmit holding register empty, received character available, or an error condition has been detected (framing error, parity error, etc.). The COM driver makes use of the "received character available" interrupt to take control when a fully-formed character is in the receive holding register. The COM driver services the interrupt by reading the character and storing it in a one-character buffer. COM would then normally return from the interrupt while it awaits the next @GET request to take the character.

An application can request that instead of returning from the interrupt, control is passed to the application for IMMEDIATE ATTENTION. It is important to note that this action would be occurring during interrupt handling and any processing by the application must be kept at a minimum before control is returned to COM via an RET instruction.

If you use an @CTL function value of X'04', then register IY must contain the address of the handling routine in your application. Upon return from the @CTL request, register IY will contain the address of the previous WAKEUP vector. This should be restored to the COM driver when your application is finished with the WAKEUP feature.

When control is passed to your WAKEUP vector upon detecting a "receive character available" interrupt, certain information is immediately available. Register A will contain an

image of the serial port UART status register. The Z-flag will be set if a valid character is actually available. The character, if any, is in the C-register. Since system overhead takes a small amount of time in the @GET SuperVisor Call, you may only have to @GET the character via standard device interfacing. This will ensure that any filtering or linking in the *CL device chain will be honored. If, on the other hand, your application is attempting to transfer data at a very high rate (9600 baud or higher), you may need to bypass the @GET SuperVisor Call and use the character immediately available in the C-register. Note that this will ignore any device chain linkage.

The module name of the COM driver is "$CL". Its data area includes the following:

Offset	Contents
+0	Contains the handshake mask established according to the default conventions (or those established via SETCOM). This mask is used by COM and needs no concern from the programmer.
+1	Contains the serial port control image (this image may turn out to be dependent on specific RS-232 hardware. Bit 7 Parity [1 = EVEN; 0 = ODD] Bits 6 & 5 Word length [00 = 5; 10 = 6; 01 = 7; 11 = 8] Bit 4 Number of STOP bits [1 = 2 bits; 0 = 1 bit] Bit 3 Parity enable/disable [1 = disable; 0 = enable] Bit 2 Transmit data [1 = enable; 0 = BREAK] Bit 1 Data Terminal Ready lead [0 = ON; 1 = OFF] Bit 0 Request To Send lead [0 = ON; 1 = OFF]
+2	Contains the code for the baud rate.
+3	Flag to indicate KFLAG$ support [1 = ON; 0 = OFF] Effective with LDOS 6.2.0, this byte contains the BREAK character code, LOGBRK. If non-zero, then reception of that byte value from the communications line will cause the BREAK bit of the KFLAG$ to be set. If zero, no input character will be interpreted as a BREAK.
+4	One-character buffer flag [80H = no character; 0 = character]
+5	Storage for the one-character buffer.

4. DISK DRIVE INPUT/OUTPUT INTERFACING

4.1 GENERAL DISK DRIVE CONFIGURATION

This chapter is designed to fully explain the purpose of the Disk Controller Communications SuperVisor Calls. It will also completely describe the fields constituting the Drive Control Table. We will cover the protocol linkage that interfaces the disk driver to the DOS. Finally, we will discuss some of the concepts that are associated with interfacing hard disk drives. There are two reasons for this chapter. On one hand, you may be interested in using the disk primitives to write disk-oriented utility programs. A good foundation in the functions of the controller primitives is essential. On the other hand, you may have the need to write a disk driver that supports a hard disk controller. In this case, it is essential to understand the requirements of the system for communicating with disk devices. Before we can begin these topics, we must gain a knowledge of the configuration of disk storage devices.

The Disk Operating System incorporates the term "disk" because the operating system is associated with and directly supports disk drive storage devices. Although many users of small microcomputers may be used to systems with two or three disk drives, the Version 6 DOS supports up to eight disk storage devices. The most typical type of disk drive used in systems running Version 6 is the floppy disk drive. The hardware that interfaces the floppy disk drive to the computer is called a Floppy Disk Controller (FDC). The controller includes all of the electronics necessary to control and translate operating system commands into control pulses which the drive uses to perform mechanical actions (such as head stepping, drive select, head load, etc) and data transfer.

The floppy disk drives are usually connected to the computer in a multiplexed arrangement. This means that all data and control signals share a common cabling. Where more than one disk drive is connected to the cable, a means of uniquely selecting one drive at a time must be provided. Over the years, a standard of drive selection has been developed that all floppy disk drives adhere to. This standard incorporates four separate drive select lines between the computer and all disk drives. These drive select lines are designated DS0, DS1, DS2, and DS3. Each disk drive is then jumpered to connect to only one of the drive select lines. Sometimes the drives connect to all of the lines while each plug on the cable severs all select lines but one - each cable plug a different select line. Thus, the computer hardware will, in general, support the handling of four floppy disk drives [some companies manufacture a multiplex device that uses the four drive selects as a binary number thus multiplexing up to 15 floppy drives].

Although the typical hardware configuration supports four floppy disk drives, the DOS has provisions for referencing eight distinct logical drives numbered 0-7. We use the term "logical" in case we have a single drive that is partitioned into multiple drives with each partition being referenced by a different drive number. The four extra positions are usually used with installations that connect hard disk drives in addition to the floppies. The DOS stresses device independence. Disk drives are treated no differently. In order to gain a high level of independence, the DOS uses a standardized set of SuperVisor Call functions we will term "Disk Controller Communications". These SVCs are primitive functions that should provide all of the activities needed to communicate I/O requests to the disk controller that's interfacing a disk drive.

The system also maintains a Drive Control Table (DCT) that stores the parameters associated with each of the eight logical drives. Disk drive parameters refer to how the total storage space on a drive is divided up into addressable units. Floppy disk drives use a removable flexible media which has one or two surfaces coated with a magnetic layer of particles. Hard disk drives use either fixed rigid platters or removable cartridges that contain rigid platters also containing magnetic layers of particles. Each platter of a hard drive contains two surfaces. Regardless of the disk drive type, the magnetic layer of particles on each surface is magnetized into concentric circles of storage areas

called TRACKs. Each track is then divided into subareas called SECTORs. Each sector is uniquely identified by a pattern of information preceding each sector called an ID FIELD. The division of a surface into sectors may be envisioned as a pie cut up into equal sized pieces. The process of generating each of the tracks and sectors is termed the formatting process. The physical length of a sector will be greater on the outer tracks of the surface than the inner tracks of the surface (similar to the grooves of a phonograph record). Although the number of sectors per track may vary from one media type to another, the number of sectors in each track of the same media must always be a constant.

The DOS assigns numbers to every sector, every track, and every surface. Surfaces are numbered consecutively by one starting from zero. Tracks are numbered consecutively by one starting from zero at the outermost portion of the disk giving the innermost track the highest number. A CYLINDER consists of the like-numbered tracks on all surfaces. For example, on a two-surface media, track zero of surface zero and track zero of surface one are grouped together into cylinder zero.

Floppy disk drives use a read/write head that is positioned lateral to the disk surface. The head can step in towards the center of the disk and step out to the circumference of the disk while the disk rotates on its hub. The rotational speed is 300 rpm for 5-1/4" floppy disk drives and 360 rpm for 8" floppy disk drives. Hard disk drives rotate at speeds of 3600 rpm and higher. Because the physical lengths of the sector vary from the outer to the inner track, the bit density of each sector varies per track. Therefore, the amount of information stored in all sectors is dependent on the maximum bit density permitted in its shortest sized sector. Some manufacturers of computer systems are using a design which keeps the bit density per sector constant by use of a variable speed drive which maintains a constant linear velocity of the surface across the head regardless of the track position. This technique promotes a greater capacity for storage but requires a more precisely controlled drive. If such a drive control were utilized under this DOS, a suitable translation filter would be needed which would permit the DOS to think that each track still contained the same number of sectors.

If we concern ourselves with a 5-1/4" double density floppy drive rotating at 300 rpm, we can calculate that a disk makes one complete rotation every 200 ms (60/300). Since there are 18 sectors per track, a sector's ID FIELD passes by the drive's head every 11.1 ms. In a system where the transfer of data to and from the disk is under the control of the CPU rather than through auxiliary Direct Memory Access (DMA) hardware, the CPU spends its time handshaking with the controller while transferring each byte of data. If we are trying to access a series of sectors sequentially (as would be the case with a sequentially accessed file), there will rarely be sufficient time for the CPU to establish the handshaking with the controller for the access of the next sector once it has finished transferring the current sector. Thus, if we number the sectors consecutively, most likely the ID FIELD of the sector we next want to read has just passed by the head and we must wait a complete revolution of the disk before getting to the ID FIELD again. In fact, the worst case would require us to wait just under 211.1 ms per sector while the time to read an entire track would be 3.8 seconds!

A practical solution to increasing the data transfer is to stagger the sector numbers so that the next sector to transfer is arriving at the head just after we start looking for it. If we could read many sectors per single rotation, we could speed up the transfer of data. This can be done when the disk is formatted. It can also be done when the disk is accessed by means of a lookup table that translates a logical sector number to a staggered physical sector number. The process of staggering the sector numbers is termed INTERLEAVE. An interleave of two means that sequential sector numbers are in every second physical sector. An interleave of three uses every third position. For a single density 5-1/4 diskette, this pattern would be 0-5-1-6-2-7-3-8-4-9. An 18 sector per track diskette with an interleave of three would have a pattern of 0-6-12-1-7-13-2-8-14-3-9-15-4-10-16-5-11-17. The interleave can be precisely calculated with knowledge of the total time it takes to execute the machine instructions between sector I/O. This is generally a most difficult task; therefore, interleave patterns are generally derived empirically.

Sometimes, the apparent difference in access speed across different systems stems from a poor selection of the sector interleave. The Version 6 DOS uses the method of applying the interleave during the formatting process. The sectors in each track are therefore numbered in a staggered order. [Most CP/M systems format sequential sector numbers and use a sector interleave translation table to translate sequential access requests to the staggered number when the access is made].

One other attempt at increasing the sequential access of sectors is to examine the time between transferring the last sector number of a track and sector zero of the next higher track [for the moment let's not compound the situation of two sided diskettes where the sectors on the second side rotate in an order reverse of the obverse side]. The time lag will include the sector interleave plus the track-to-track step time. Thus it might make sense to not start each track with sector number zero, but to optimize the starting number so that the position of sector zero will have its ID FIELD just coming up to the head by the time that the drive has stepped and is ready to scan for the ID FIELD. This staggering is termed TRACK SKEW. The DOS introduces such a skew during the formatting process; however, such a skew is probably optimum for only one track-to-track stepping rate. With all of this, we still can state that each track contains like numbered sectors - regardless of track number or surface. Therefore, each sector on a disk is designated unique by its respective sector, surface, and track numbers.

When the operating system formats a diskette (or hard disk), all of the parameters associated with the diskette are predetermined. Thus the number of sectors per track, number of sectors per granule and thus the granules per track, number of sides (or surfaces), and number of cylinders are all designated as well as the density of the media in the case of floppy diskettes. Some of these figures (density, sides, granules per track) are written to fields in the Granule Allocation Table which is part of the directory (see chapter 5). Others (sectors per track, sectors per granule, in addition to the former quantities) are part of the DCT fields. When the system attempts to open a file on a disk, it uses the @CKDRV SuperVisor Call function to ascertain the availability of the disk and then logs the disk once it finds it available. The function of "logging" will update the DIRCYL field (providing the driver returns proper system sector error codes), then update the DBLBIT field and the MAXCYL field based on information stored in the GAT. It is up to the driver to sense the density of the floppy media [the "data record not found" controller error is the usual indication that the driver must toggle to the alternate density. If a data record ID FIELD is not readable under both single density and double density, then the assumption is that the corresponding sector is not on the disk and the error is passed back to the system]. The toggling function of the driver includes the updating of the CONFIGURATION FIELD in the DCT appropriate to the density being selected.

The SVC disk primitives are funneled through a common system routine that establishes a linkage protocol between the operating system and the disk device driver(s). When an I/O request is invoked by a higher level SVC, such as a request to READ a file record, the request is translated to that disk primitive needed to satisfy the function. The linkage protocol is uniform across all disk devices that are connected to the system. This makes the access of files transparent to size or nature of the disk device within the scope of the DCT parameters acceptable to the system.

4.2 DRIVE CONTROL TABLE (DCT)

The Drive Control Table (DCT) is the way in which the DOS interfaces the operating system with specific disk driver routines. This table is one of the examples of the versatility of the system as it embodies within it the method of customizing the parameters of a drive so that each disk drive may incorporate a unique set of parameters. For instance, one drive may be a 35-track single headed drive. Another may be an 80-track dual headed. While a third may yet be a 5 megabyte hard drive. Ingenuity and oddball hardware will mix well to provide an easy interface.

The DCT contains the information relating to the granule size. In the case of floppies, granule sizes are standardized by the system according to the disk size and density. Chapter 5 contains more information on granule allocation sizes. Data on the number of sectors per track, number of heads, number of partitions, and maximum number of cylinders is also contained in the DCT for each drive. This data is an essential ingredient in the allocation and accessibility of file records and therefore must be accurately introduced. The table contains a maximum of eight DCT records - one record for each logical drive designated 0-7. Each DCT record is fielded as follows:

4.2.1 DCT VECTOR - <Bytes 0-2>
This three-byte field specifies whether the logical drive position is enabled or disabled. The system will not attempt to communicate with a logical drive number whose DCT position is considered disabled. If the position is enabled, then the field will also contain the address vector of the disk driver module that communicates with the controller interfacing the disk drive. The first byte of the DCT VECTOR would contain an X'C3' value if the drive position is enabled (an X'C3' represents an absolute jump [JP nnnn] instruction in Z-80 machine code). If the drive is disabled, this byte will be an X'C9' value (an X'C9' represents an absolute return [RET] from subroutine instruction in Z-80 machine code).

The second and third bytes of the field will contain the vector transfer address of the disk driver module that communicates with the controller. The operating system typically places the disk drivers in the low memory driver region. A "stock" system has available in this region, memory sufficient to store additional drivers that are not supplied by the system. The DOS will dynamically use this low memory region based on requests to invoke system drivers and filters (such as the COM/DVR or FORMS/FLT). A retrievable pointer to the first available memory address in this region can be used to locate the origin of a user-supplied driver or filter (if sufficient space is available). This will be discussed in a later section.

4.2.2 DCT FLAG-1 - <Byte 3>
This field contains a series of sub-field parameters associated with the disk drive specifications. The field is encoded as follows:

Bit 7 Set to 1 will indicate the disk device is "software" write protected. It is the responsibility of the disk driver to check this bit on any disk primitive that references a WRITE operation (i.e. write sector, write system sector, format track, or format device) and return a "Write protected disk" error code (error 15) if set.

Bit 6 If set to a "1", it indicates that the floppy diskette currently being accessed is formatted in double density. If set to a "0" it indicates that the diskette is single density. The disk driver is responsible for maintaining this bit by recognizing the density of the disk it is accessing. The bit is used both by the driver in the drive selection process and by the system in informative messages by such things as DEVICE displays, DIRectory displays, and FREE displays. This bit is not referenced by the system if the DCT is associated with a hard drive (see bit 3 of this field).

Bit 5 If this bit is set to a "1", the drive associated with the DCT position is an 8" drive. This bit will be a "0" if the drive associated with the DCT position is a 5-1/4" drive. This bit is initially set by whatever installs the disk driver (see the FLOPPY/DCT utility). In the installation of a hard disk driver, this bit should be set according to the size of the hard drive - 5" or 8". In the case of floppy drives, the system formatter will use this bit to adjust its formatting data to 5" or 8". It is also used to adjust informative messages as mentioned under bit-6.

Bit 4 This bit is used to store the side selection number for a current access of a diskette. It is a storage area usable by the disk driver to place the side number calculated from the relative sector passed in the disk primitive request. The system passes a relative sector number based upon the number of sectors per cylinder. On a two-headed floppy disk drive, by dividing the relative sector number by the number of sectors per track, the result will be indicative of the side selection number, 0 or 1. The routine performing the calculation can then place the result in this bit of the DCT for the use of the drive selection routine. The bit value will match the side indicator bit in the sector header as written by the FDC. Hard disk drivers will use storage space internal to the driver to hold such a result.

Bit 3 If this bit is set to a "1", it indicates that the DCT position is associated with a hard drive (Winchester). A "0" in this bit position indicates a floppy disk drive is associated with the DCT position. The bit is used by the system in informative messages by such things as DEVICE displays, DIRectory displays, and FREE displays. In addition, the system's @CKDRV routine uses this bit to inhibit its automatic logging of a hard drive while it restricts its checking to write protect status only.

Bit 2 This bit is set by the system to indicate the minumum time delay required after selecting a floppy disk drive whose motors are not currently running. It must be used by floppy disk drivers to adjust their time delay between selection of the floppy drive and the first poll of the status register. A "1" value indicates the minimum delay to be 0.5 seconds while a "0" value indicates the delay to be 1.0 seconds. The time delay can be introduced via a request of the @PAUSE SuperVisor Call with an appropriate count.

Bits 1-0 This subfield is used for different purposes depending on whether the drive associated with the DCT is a floppy drive or a hard drive. For floppies, the field contains the step rate specification code (0-3) for the floppy disk controller. With a Western Digital 179X FDC or equivalent, the codes correspond to a step rate of 6, 12, 20, and 30ms at an FDC clock speed of 1 MHz and 3, 6, 10, and 15ms at an FDC clock speed of 2 MHz. For hard disk drives, this field is usually associated with the drive select code of the hard disk drive (binary value 0-3).

4.2.3 DCT FLAG-2 <Byte 4>

This byte contains additional drive specifications and parameters. The field is encoded as follows:

Bit 7 Effective with 6.2, this bit is used to inhibit @CKDRV . If set to a "1", no @CKDRV will be performed by @OPEN when accessing that drive.

Bit 6 This bit is used as a flag to the formatter. If set to a "1", it indicates that the controller is capable of double density operation. In this case, the formatter defaults to double density formatting unless the user overrides the default. If set to a "0", the formatter will default to single density formatting. For controllers capable of double density operation, this bit is usually set.

Bit 5 This bit is used for different purposes depending on whether the drive associated with the DCT is a floppy drive or a hard drive. For floppies, a "1" indicates that the diskette currently mounted in the drive is a two sided diskette while a "0" indicates that the diskette is a single-sided diskette. This bit is updated whenever the disk is logged by the system or whenever a program invokes the @CKDRV SuperVisor Call. Note that if a dual sided diskette is placed into a two-headed disk drive that previously

accessed a single-sided diskette, the system will not recognize the second side of the new diskette until the logging process. When the DCT is associated with a hard disk drive, this bit may be used to indicate that a logical cylinder represents two physical cylinders thereby providing support for twice as many cylinders as limited by the Granule Allocation Table (the GAT limits the number of logical cylinders to 203 - thus by using this bit, hard drives to 406 cylinders can be supported as a single logical drive). In the case of hard drives, this bit is termed the "DBLBIT" bit.

Bit 4 This bit is used to indicate the controller associated with the DCT position is an "alien" controller. The term, "alien", refers to a controller that does not return index pulses in its status register. The system uses index pulse transitions in a finite time period (usually 0.5 seconds) to detect the presence of a rotating diskette. If a disk drive does not contain a diskette, or does but the drive door is open, the status obtained on continuous selection of the drive will not indicate the presence of any index pulse transitions. By examining the state of the index pulse over a period of time corresponding to 2.5 possible rotations of a disk, the lack of an OFF-ON-OFF transition state will indicate that the drive is not available. If a controller does not return the state of an index pulse in the controller status byte, then the system will never be able to detect the availability of the drive if it maintains the state transition examination in the logging process. This bit should be set when such controllers are used to inhibit the @CKDRV routine from performing such an examination and proceed to the configuration logging.

Bits 3-0 This subfield is used for different purposes depending on whether the drive associated with the DCT is a floppy drive or a hard drive. For floppies, the field contains the physical drive address (1, 2, 4, or 8) corresponding to the drive select line (DS0, DS1, DS2, or DS3). Thus, only one of the four bits will ever be set. Hard drive installations that partition a drive by head, may use this field to indicate the relative starting head number of the logical drive partition. This provides support for a drive of up to 16 heads although 4 heads is typical.

4.2.4 CURCYL - <Byte 5>

This field is used for different purposes depending on whether the drive associated with the DCT is a floppy drive or a hard drive. For floppies, the field is used by the disk driver to store the current cylinder position of the disk drive assigned to the DCT position. Since a Floppy Disk controller is used to access up to four different drives, when it accesses a drive, its track register must be loaded with correct information as to the current track position of the head. The current cylinder position is maintained by the disk driver in this storage field. The driver can then be use this field to reload the FDC track register prior to a seek operation and update the field to the cylinder requested in the seek. Hard disk controllers generally contain their own internal track register that is not accessible to a software driver. This means that hard disk drivers do not need to maintain the current cylinder position in this field. The field is thus available for the storage of other data items as required by the hard disk driver. Other data items may include the total quantity of heads on the physical drive (as needed by XEBEC controllers), the complex drive select code (as used by Lobo Drives UniVersal Controller), or data associated with drive partitioning by cylinder rather than by head.

4.2.5 MAXCYL - <Byte 6>

This field contains the highest numbered logical cylinder on the drive referenced from a starting cylinder numbered "0". Thus, a 35-cylinder drive would be entered as X'22', a 40-cylinder drive as X'27', and an 80-cylinder drive as X'4F'. A typical 153-cylinder ST-506 compatible winchester drive would have an entry of X'98'. If a hard drive has more than 203 cylinders but less than 407 cylinders and is to be maintained as a single drive (or one partitioned by heads), then the system must access it as if each two physical

cylinders were a single cylinder with twice as much capacity (although the system will still limit the logical cylinder to not exceed 256 sectors). In that case, the MAXCYL entry will be half of the actual quantity and bit-5 of the FLAG-2 field will be set. For example, an SA-1000 drive (8" winchester) has 256 cylinders, four surfaces, and 32 sectors per track. If this drive is treated as a single volume (no partitioning), the MAXCYL entry is X'7F' indicating the highest numbered cylinder is 127 (128 cylinders). The DBLBIT bit is set indicating a logical cylinder is composed of two physical cylinders.

4.2.6 CONFIGURATION FIELD - <Bytes 7-8>

This two-byte field contains information concerning the physical space parameters of the disk drive and how space is allocated per cylinder. Its entries are encoded as follows:

4.2.6.1 Byte 7

Bits 7-5 This subfield contains the number of heads (surfaces) assigned to the logical partition of a hard disk drive. In the case of floppy disk drives, this entry should be a B'000'. For example, a four-head hard drive with a two-head partition would have a B'001' in this subfield. The entry is zero relative, thus a one-head partition is B'000', a two-head partition would be B'001', and an eight-head partition would be B'111'.

Bits 4-0 This subfield contains the highest numbered sector on a track numbered relative from zero. A ten-sector-per-track drive would show an X'09' entry. A 32-sector-per-track hard drive would show an X'1F'.

4.2.6.2 Byte 8

Bits 7-5 This subfield contains the quantity of granules per track allocated to the disk drive according to the number of sectors per granule. Since the field is 3-bits in length, the entry is offset from zero. Thus, one granule per track is entered as B'000', two as B'001', etc. In the case of floppy disk drives, this figure is standardized for 5-1/4" and 8" media as identified in chapter 5. If the DCT is associated with a hard drive, then the figure entered here refers to the number of granules in a physical cylinder according to the number of surfaces. If the DBLBIT bit is set, this entry then represents half of the granules on a logical cylinder. The total granules per logical cylinder is computed by the doubling the value contained in this field if bit-5 of DCT FLAG-2 is set. Let's illustrate this again using the SA-1000 drive. If we configure the drive as a single volume with 16 sectors per granule, a physical track has two granules per track. Since the drive has four surfaces, a physical cylinder has eight granules. However, since the DBLBIT bit must be set to indicate double the 128 cylinders shown in the MAXCYL field, the system would have to double the granules per cylinder computing 16 GPC. This is clearly in violation of the system's upper limit of eight granules per cylinder maximum. Therefore, our example SA-1000 drive would be configured with 32 sectors per granule, one granule per track, four granules per physical cylinder. The DBLBIT bit would provide eight logical granules per logical cylinder. Therefore, this subfield would have an entry to indicate four granules.

Bits 4-0 This field contains the quantity of sectors per granule that is used in the configuration of the disk. In the case of floppy disk drives, this figure is standardized for 5-1/4" and 8" media as identified in chapter 5. Hard disk drive granule sizes are assigned by the implementor of the hard disk drive system.

4.2.7 DIRCYL - <Byte 9>

This field contains the cylinder where the directory is located. For any directory access, the system will use the contents of this field as a pointer to the cylinder

containing the disk's directory. The system attempts to maintain the integrity of this field by using the status returned when the driver reads a system sector in contrast to a non-system sector (chapter 5 discusses the use of data address mark conventions in disk sectors). If the system expects to be reading a directory sector but does not get the error code 6 ("Attempted to read system data sector"), it will read the BOOT sector and obtain the directory cylinder storage byte located therein for a second attempt to read the directory sector. After an unsuccessful second attempt (including whatever retries are performed per attempt by the driver), the system posts a read or write error depending on the original request. This error will eventually be classified as a GAT , HIT or DIRECTORY error if the attempt was an I/O request for the GAT, HIT or a directory entry sector respectively. Realizing that most hard disk controllers do NOT support a data address mark convention, the hard disk driver must simulate the READ SYSTEM SECTOR error code when an @RDSEC or @VRSEC request is made to the directory cylinder. Since the only indication of where the directory is located is contained in this field, it is paramount to the functioning of the hard disk environment that this field be correctly maintained. The system's LOG command will always reload this field with the BOOT sector's directory cylinder pointer. Thus, it may be necessary to highlight the function of LOG in any written information pertinent to the hard disk system user.

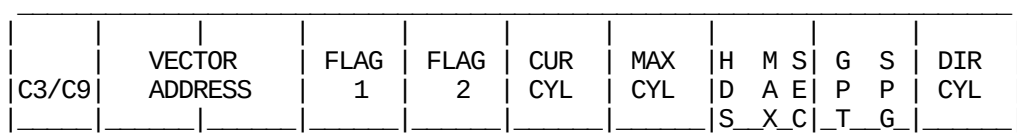

Figure 4-1: Drive Control Table Record

4.3 DISK CONTROLLER COMMUNICATIONS

The function of DISK CONTROLLER COMMUNICATIONS is to communicate operating system commands to a disk driver so that the driver can translate these commands into commands acceptable to the disk controller. Before we look at the command functions provided by the system, let's take a look at the commands available in a typical floppy disk controller - the Western Digital 179X series. Figure 4-2 summarizes these commands. If you are interested in the detailed specifications of such a controller, you should obtain the "FD 179X-02 Floppy Disk Formatter/Controller Family" manual published by the Western Digital Corporation.

Command	Purpose
RESTORE	Recalibrate drive to cylinder 0 position
SEEK	Reposition head to a specified cylinder
STEP	Move the head one cylinder position
STEP IN	Move the head one cylinder to the higher track
STEP OUT	Move the head one cylinder to the lower track
READ SECTOR	Transfer the specified sector from disk to CPU
WRITE SECTOR	Transfer the specified sector from CPU to disk
READ ADDRESS	Transfer data from the next ID FIELD encountered
READ TRACK	Transfer an entire track of data from disk to CPU
WRITE TRACK	Transfer an entire track of data from CPU to disk
FORCE INTERRUPT	Abort the pending controller operation

Figure 4-2: Floppy Disk Controller Commands

Since the DOS also supports hard disk drives, let's look at the commands available in some typical hard disk controllers. The following three figures will summarize the commands supported by the Lobo Drives UniVersal (UVC), the Western Digital WD-1000, and the XEBEC S-1410 controllers.

Command	Purpose
NO OPERATION	Test if controller available
READ SECTOR	Transfer the specified sector from disk to CPU
READ DISK	Read entire disk without data transfer
WRITE SECTOR	Transfer the specified sector from CPU to disk
FORMAT DISK	Format entire disk
READ UNTIL FLAW	Read disk until encountering an error

Figure 4-3: Lobo-UVC Controller Commands

If we compare the typical Hard Disk Controller [let's abbreviate this term to "HDC"] commands to the commands available in the typical Floppy Disk Controller [we will also abbreviate this term to "FDC"], we find that the HDC generally has very few commands for communication between the CPU [most hard disk systems refer to the CPU as the "HOST"] and the controller. The S-1410 HDC has a preponderance of commands; however, close examination reveals many commands for testing and diagnostics. Each HDC mentioned performs its own automatic SEEK operation; therefore, it is generally not even necessary for the HDC driver to utilize that command. The HDC driver will most typically involve READ, WRITE, and FORMAT operations.

Command	Purpose
RESTORE	Recalibrate drive to track 0
SEEK	Position the read/write head to a cylinder
READ SECTOR	Transfer the specified sector from disk to CPU
WRITE SECTOR	Transfer the specified sector from CPU to disk
FORMAT TRACK	Initialize the ID and DATA fields of the track

Figure 4-4: WD-1000 Controller Commands

Command	Purpose
TEST DRIVE READY	Test if drive is ready
RECALIBRATE	Recalibrate drive to track 0
REQUEST SENSE STATUS	Return the 4-byte drive/controller status
FORMAT DRIVE	Format entire disk
CHECK TRACK FORMAT	Check track for correct ID and interleave
FORMAT TRACK	Initialize the ID and DATA fields of the track
READ	Read the specified sector(s) from disk to CPU
WRITE	Write the specified sector(s) from CPU to disk
SEEK	Position the read/write head to a cylinder
INITIALIZE DRIVE CHARACTERISTICS	Configure controller for drive
READ ECC BURST ERROR LENGTH	Read the byte containing ECC data
RAM DIAGNOSTIC	Test the controller's RAM buffer
DRIVE DIAGNOSTIC	Test the drive-to-controller interface
CONTROLLER INTERNAL DIAGNOSTICS	Perform controller self-test
READ LONG	Read a sector and four ECC bytes
WRITE LONG	Write a sector and four ECC bytes

Figure 4-5: S-1410 Controller Commands

The process of drive selection is unique from HDC to HDC as well as the adapter that electronically interfaces the HDC to the host. FDC drivers are typically more involved with the additional commands for stepping and seeking while performing a little more bookkeeping operations. There is also a great more involvement in the format operation for the FDC driver over the HDC driver.

The DOS provides 16 SuperVisor Calls that are used to pass operating system function requests to a disk controller - be it an FDC or an HDC. Figure 4-6 reviews these functions that are detailed in chapter 7. If we try to correlate the SVC functions with the FDC commands, we observe that the DOS provides no facility for requesting a STEP, STEP OUT, nor a FORCE INTERRUPT. This is not an oversight. The force interrupt is a function that is not needed from a higher level such as the DOS, but would most likely be usable directly within the FDC driver. Also, since the FDC does its own track stepping via the SEEK request, the STEP command from the DOS is only needed during the format operation. The DOS limits this to STEP IN since the disk only needs to be stepped in one direction during the format operation. The remaining SVCs supply the higher level functions to communicate all of the DOS requests to the controller.

NAME	NUMBER	FUNCTION	DESCRIPTION
@DCSTAT	40	0*	Test disk controller status
@SLCT	41	1*	Select a disk drive
@DCINIT	42	2	Initialize a disk controller
@DCRES	43	3	Reset a disk controller
@RSTOR	44	4*	Restore a drive to cylinder 0
@STEPI	45	5*	Issue track step-in to controller
@SEEK	46	6*	Seek to a disk cylinder
@RSLCT	47	7*	Reselect a busy drive until available
@RDHDR	48	8	Read ID field
@RDSEC	49	9*	Read a disk sector
@VRSEC	50	10*	Verify the readability of a disk sector
@RDTRK	51	11	Read a disk track
@HDFMT	52	12*	Format an entire drive
@WRSEC	53	13*	Write a disk sector
@WRSSC	54	14*	Write a disk directory sector
@WRTRK	55	15*	Write a disk track (format data)

Figure 4-6: Disk Controller Communications

Note: Functions asterisked are supported by the DOS floppy driver

Before taking a look at the HDC commands versus the disk controller communications functions, let's address exactly what functions are used in the DOS. The DOS spends a great percentage of the controller's time in reading and writing. These DOS functions use @RDSEC to read disk sectors, @WRSEC and @WRSSC to write non-system and system sectors respectively. Where the application is requesting verification (or where the DOS is writing a system sector), then the @VRSEC function is used which should read the designated sector without disturbing the disk file I/O buffer. Next, the logging function uses @SEEK and @RSLCT to obtain status from the disk. FORMAT uses @WRTRK for the FDC and @HDFMT for the HDC as well as @SLCT, @RSTOR, and @STEPIN in addition to the previous SVCs. BACKUP and FORMAT also use @DCSTAT to make sure that the drive is enabled. These functions are indicated by an asterisk in figure 4-6. The four remaining functions, @DCINIT, @DCRES, @RDHDR, and @RDTRK are provided in case utility software needs these requests for communications with custom drivers [NOTE THAT THE FDC DRIVER SUPPLIED WITH THE DOS DOES NOT SUPPORT THESE FUNCTIONS].

If we look at the HDC commands, we observe that although the DOS commands provided can not uniquely request all of the commands of every controller, the DOS commands do provide the means to satisfy all of the necessary functions. In fact, some DOS functions are not even needed in the case of the HDC and hard disk system.

When the operating system passes the SVC request to the disk driver The manner in which the driver controller linkage is established is by passing a function value contained in register "B" to the software driver that interfaces to the controller. Sixteen functions have been defined within the DOS. The table in figure 4-6 briefly describes these functions.

At this point, it would be beneficial to discuss exactly what operations are performed by the operating system when it receives one of the Disk Controller Communications SVC requests. All of the requests use register C to reference the logical drive number. The DOS uses this value to index the Drive Control Table and obtain a pointer to the DCT record associated with the logical drive. After saving the index register, the DOS places the pointer into IY.

The DOS saves register pair BC and places the function code corresponding to the function as shown in figure 4-6 into register B. The DOS will also issue an @BANK request to bring in bank zero. This operation will ensure that bank zero is resident for a disk I/O operation. It also limits the location of disk drivers or disk filters [like MONITOR

available from Logical Systems, Inc.] to reside in either the low memory driver region or in upper memory of bank zero. Upon return from the disk driver, the DOS will restore the previously resident RAM with another @BANK request.

The DOS then places an "Illegal drive number" error code (32) into the accumulator, resets the Z-flag, then executes a "CALL" to a "JP (IY)" instruction. The purpose of this strange linkage becomes evident when we examine the result. The first byte of the DCT is interpreted as an RET instruction if the drive is disabled. Since register IY is pointing to that byte, the linkage will return back to the caller with the "Illegal drive number" error. If the drive is enabled, the first DCT byte is interpreted as a JUMP instruction which will transfer control to the entry point of the driver. We can now show the uniform register protocol upon entry to a disk driver. This protocol is illustrated in figure 4-7.

Register	Direction	Condition/Value
AF	=>	Irrelevant upon entry to the driver
B	=>	Contains the function code of the request <0-15>
C	=>	Contains the logical drive number <0-7>
D	=>	Contains the cylinder being requested <0-202>
E	=>	Contains the relative sector being requested <0-255>
HL	=>	Contains a pointer to the I/O buffer, where applicable
IY	=>	Contains a pointer to the proper Drive Control Table entry
A	<=	Must be loaded with one of the error dictionary codes
BC	<=	Can be altered by the disk driver
DE	<=	Must be preserved by the disk driver
HL	<=	Must be preserved by the disk driver
IY	<=	Should be preserved by the disk driver
F	<=	The Z-flag should be set if A=0, otherwise reset the Z-flag

Figure 4-7: Disk Driver Register Protocol

The remainder of this section introduces a skeletal disk driver. It will contain only the functions that are associated with protocol required by the DOS. There is no expectation that you will learn how to write a disk driver from this publication; you will learn how to put the functions into your driver that are required by the DOS!

4.4 Skeletal Disk Driver

```
ENTRY   JR      BEGIN           ;The driver starts with the
        DW      DVREND          ;  DOS standard header
        DB      MODPTR-ENTRY-5  ;Length of 'MODNAME'
        DB      'MODNAME'       ;Name for @GTMOD requests
MODPTR  DW      0               ;These pointers are unused
        DW      0
BEGIN   LD      A,B             ;The first test will return
        OR      A               ;  to the caller on @DCSTAT
        RET     Z               ;  and set the Z-flag with A=0
        CP      7               ;
        JP      Z,RSLCT         ;Transfer on @RSLCT
        JP      NC,DISKIO       ;Transfer on physical I/O request
;*=*=*
;       FUNCTIONS 1-6 NEED TO BE PARSED
;*=*=*
SLCT    .                       ;As required
;*=*=*
RSTOR   .                       ;As required
        LD      (IY+5),0        ;Needed if a floppy
;*=*=*
STEPI   .                       ;As required if a floppy
        INC     (IY+5)          ;Bump CURCYL
```

```
;*=*=*
SEEK    .                       ;As required
        LD      (IY+5),D        ;Update CURCYL
;*=*=*
;       The RSLCT function should return with the hardware
;       write protection status. Set bit 6 of the accumulator
;       to indicate the drive is write-protected
;*=*=*
RSLCT   .                       ;As required
;*=*=*
DISKIO  BIT     2,B             ;Test if read or write commands
        JR      NZ,WRCMD        ;Transfer if functions <12-15>
;*=*=*
;       Functions 8-11 need to be parsed
;*=*=*
RDHDR   .                       ;If you want to support it
;*=*=*
RDSEC   .                       ;Read a sector of data
VRSEC   .                       ;Don't alter the buffer
;*=*=*
;       On RDSEC and VRSEC, if the read referenced the
;       directory cylinder and was successful,
;       then you need to return an error code 6. A floppy
;       disk controller will provide the indicated status.
;       Hard disk users may have to compare the requested
;       cylinder to DIRCYL in the DCT.
;*=*=*
RDHDR   .                       ;If you want to support it
;*=*=*
WRCMD   BIT     7,(IY+3)        ;Check for software write protect
        JR      Z,WRCMD1        ;Transfer if no soft WP
        LD      A,15            ;Set "Write protected disk" error
        RET
WRCMD1  .                       ;Now parse functions 12-15
;*=*=*
HDFMT   .                       ;May be used for hard drives
;*=*=*
WRSEC   .                       ;Write with X'FB' data address mark
;*=*=*
WRSSC   .                       ;Write with X'F8' data address mark
;*=*=*
WRTRK   .                       ;May be for floppy or hard drives
;*=*=*
;       NOTE: Hard disk drivers may want to exclude the FORMAT
;       function from the driver if a separate formatter is
;       supplied. This guards against program crashes inadvertantly
;       entering the driver with a register setup depicting FORMAT
;*=*=*
;       Error codes returned to the system under abnormal
;       conditions must be in the error dictionary. Hard disk
;       drivers should attempt to translate the controller error
;       code to the most reasonable DOS equivalent.
;*=*=*
DVREND  EQU     $-1
```

4.5 HARD DISK ALLOCATION SCHEMES

The integrator of a hard disk usually has to consider some form of hard disk partitioning. Why is this to be considered? A hard disk has a minimum of 5 megabytes of storage space. The demand for storage never abates; thus, 10 megabyte, 20 megabyte, and higher capacities are being integrated into the microcomputer environment. The version 6 DOS has limitations on the total size of a storage device that is addressable as a single volume. These are limitations stemming from the size of the directory. A device is limited to a maximum of 256 sectors per logical cylinder, and 203 logical cylinders. Given a standard sector size of 256 bytes, the DOS can address 13.3 megabytes total. If the target drive

exceeds this capacity, then it must be divided into more than one drive in order to address its total capacity.

The DOS also limits the number of files per logical drive to 256 (of which two are taken up by the BOOT/SYS and DIR/SYS files). Although data base applications may find the most practical arrangement is a single volume, the typical use of even a 5 megabyte drive will find the file slots filled before all of the space is allocated - thus space is wasted [It is possible and highly practical for the hard disk integrator to consider combining individual static files into members of a partitioned data set to free up multiple file slots. PRO-PaDS is a utility program capable of creating and maintaining such files]. Therefore, even with the smaller 5 megabyte drive, there exists a rationale for partitioning.

Once the decision is made to divide a drive, the question arises as to how to go about such a division. There are three methods of partitioning. One is to divide the drive by cylinder. For example, Take a 306 cylinder, four head, 10 megabyte drive. This can be divided into two drives with the first logical drive using cylinders 0-152 while the second uses cylinders 153-306. The DOS actually uses logical cylinder numbers 0-152 for both partitions and the hard disk driver must recognize that it needs to translate the 0-152 for the second partition into the range 153-306. Obviously, one can divide up the drive into partitions smaller than 5 megabytes. A second method is to divide the drive so that all of the cylinders are included in a single logical volume, but volumes use different heads. Thus, the previously mentioned drive could be divided into two, three, or four logical drives. A third method would be to translate the drive's physical parameters into quantities acceptable to the system while staying within the maximum number of 256 sectors per logical cylinder.

There are advantages and disadvantages to each method. First, our discussion of floppy configurations pointed out a use for addressing as much capacity in a single cylinder prior to having to step the drive. This means that we would lean towards divisions by cylinder. However, if we are alternately selecting different partitions, the drive must be stepped a great distance to get to each partition. Another problem is that a head crash would essentially wipe out all drives since a single head is used on all partitions. Of course, if the drive physically has more than 406 cylinders, it must be partitioned by cylinders (or translation) to address the higher cylinders.

Partitioning by head provides less sectors per physical cylinder; however, since hard drives today usually use very fast buffered seek, the stepping time to advance a track is minimal. A head crash will also only wipe out a single logical drive.

Translation methods can be useful with drives whose parameters do not lend themselves to the DOS limits (a 39 sector per track drive, for instance). A drawback to translation methods is the difficulty in keeping logical cylinders referencing a physical cylinder.

The important point in any method, is that the driver must be written to do the conversions as the operating system's reference is to logical cylinder and sector within that cylinder when it issues an I/O request. The driver may make use of the CURCYL byte and FLAG-2, bits 3-0 for storage of partition specific data. The driver can also establish its own table when these DCT fields do not provide sufficient space to store the quantities needed by the driver.

Let's take a look at a few examples. The number of file slots identified assumes that all logical drives are considered to be data drives. Subtract 14 from the number for each SYSTEM drive. In the first, case we will use an ST-506 type drive which has four heads and 153 cylinders. This will be the division of a 5 megabyte drive partitioned by head. Figure 4-8 illustrates the DCT parameters to divide the drive into two logical drives of 2.5 megabytes each. Notice that we are using 8-sector granules (2K). Since we can have at

most, eight granules per cylinder, the minimum granule size is 2K. We could have allocated sixteen sectors per granule providing four granules per cylinder.

START HEAD	MAX CYL	# OF HEADS	MAX SEC	GPT	SPG	DIR CYL	FILE SLOTS
0	152	2	32	8	8	76	254
2	152	2	32	8	8	76	254

Figure 4-8: 5 Meg divided; 2-2.5

We could just as well divide this drive into a 1.25 megabyte volume and a 3.75 megabyte volume. This arrangement is illustrated in figure 4-9. This arrangement forces us to allocate granules in 16-sector blocks.

START HEAD	MAX CYL	# OF HEADS	MAX SEC	GPT	SPG	DIR CYL	FILE SLOTS
0	152	1	32	4	4	76	238
1	152	3	32	6	16	76	254

Figure 4-9: 5 Meg divided; 1.25-3.75

If we divide up the drive into three logical volumes, we will develop two volumes of 1.25 megabytes each and one volume of 2.5 megabytes. This arrangement will also provide more file slots.

START HEAD	MAX CYL	# OF HEADS	MAX SEC	GPT	SPG	DIR CYL	FILE SLOTS
0	152	1	32	4	4	76	238
1	152	1	32	4	4	76	238
2	152	2	32	4	8	76	254

Figure 4-10: 5 Meg divided; 2-1.25, 1-2.5

The last division of a 5 megabyte 4-head drive to illustrate is as four separate drives of 1.25 megabytes each. This partitioning provides the greatest number of file slots. Where the environment will have a great deal of small files, it is probably best to use this arrangement.

START HEAD	MAX CYL	# OF HEADS	MAX SEC	GPT	SPG	DIR CYL	FILE SLOTS
0	152	1	32	4	8	76	238
1	152	1	32	4	8	76	238
2	152	1	32	4	8	76	238
3	152	1	32	4	8	76	238

Figure 4-11: 5 Meg divided; 4-1.25

DBLBIT	START HEAD	MAX CYL	# OF HEADS	MAX SEC	GPT	SPG	DIR CYL	FILE SLOTS
1	0	152	2	32	4	16	76	254
1	0	152	1	32	4	8	76	254
1	1	152	1	32	4	8	76	254

Figure 4-12: 5 Meg divided; 2-2.5

Five megabyte drives exist that use 2 heads (a single platter) and incorporate 306 cylinders. If we want to divide up this type of drive by head, we can have at most, two partitions. Since this drive requires the DBLBIT, it will be illustrated in figure 4-12 as both a single and a dual volume. An important observation is that a logical cylinder

is two physical cylinders. Although the drive has 306 cylinders, the cylinder figures in the DCT reflect the logical quantities of half as many. Also, the granules per track figures are representative of a PHYSICAL cylinder. These figures will be doubled by the system in the calculation of granules per cylinder since the DBLBIT is set.

From these figures illustrating the configurations of 5 megabyte drives, it should be relatively easy to develop the necessary Drive Control Table data for drives of 10, 15, 20, and higher megabyte capacity.

4.6 Placement of Disk Drivers

Disk drivers are usually placed into memory by an initialization program which executes from the **SYSTEM (DRIVE=n,DRIVER="filespec")** library command. This DOS facility will load and execute your driver initializer identified by the "filespec". A file extension of "/DCT" is the default. Upon passing control to this DCT driver, register pair DE will be pointing to the DCT record associated with the **DRIVE=n** entry. If the DRIVE parameter was omitted from the SYSTEM command, register pair DE will contain a zero. The function of the initializer is to prepare the driver and DCT tables according to any parameters required for setup of the driver. The initializer then identifies where in memory the driver is to be placed, relocates any absolute address references, then places it into memory. The last function is to insert the entry address into the Drive Control Table.

One other point concerns a test that should be made by the driver initializer that is to be invoked by the SYSTEM command. The operating system permits the execution of any load module. A driver program is a load module. To guard against its execution from DOS Ready by inadvertently entering its full file specification, the system provides the programmer with an indicator that execution is under control of the SYSTEM command. When SYSTEM passes control to a driver program, it will set bit-3 of the CFLAG$ (the system request bit). Thus, by testing this bit upon entry to the program, an error exit can be taken if the system request bit is not set. An error message such as the following can be logged and the program aborted.

```
Must install via SYSTEM (DRIVE=n,DRIVER="filespec")
```

The DOS provides a limited device driver region in low memory. This is where the keyboard, video, printer, and floppy disk drivers are located. User specified device drivers (such as the COM driver) are placed in this region if sufficient space is available. Otherwise, they are relocated to the high memory region and protected. The MemDISK driver must reside in the low memory device driver region. A hard disk driver supplied by LSI is usually placed in low memory. The low memory driver region is filled from the bottom up in contrast to the high memory region which is filled from the top down. The maximum address usable is X'12FF'. The system has a pointer which maintains the first available memory address in this region. This driver I/O region pointer is always positioned as the two bytes just prior to the *KI Device Control Block. Let's take a look at some partial routines to obtain and use this driver pointer.

```
;*=*=*
;       Obtain low memory driver pointer
;*=*=*
        LD      DE,'IK'         ;Locate pointer to *KI DCB
        LD      A,@GTDCB        ;  via @GTDCB SVC
        RST     40
        JP      NZ,IOERR        ;No error unless KI clobbered!
        DEC     HL              ;Decrement to driver pointer
        LD      D,(HL)          ;P/u hi-order of pointer,
        DEC     HL              ;  decrement to and p/u
        LD      E,(HL)          ;  lo-order of pointer
        LD      (LCPTR+1),HL    ;Save ptr for later
        ;*=*=*
;       Make sure driver will fit into (POINTER)-X'12FF'
```

```
;*=*=*
        LD      HL,DVREND-DVRBGN ;Calculate driver length
        ADD     HL,DE            ;Start address + driver length
        LD      (SVEND+1),HL     ;Temp save of new pointer
        LD      BC,1300H         ;Maximum address + 1
        XOR     A                ;Reset carry flag
        SBC     HL,BC            ;No room if START+LENGTH >= 1300H
        JP      NC,NOROOM        ;  fit in low core
        .
        .
        .
;*=*=*
;       Move driver into low memory after relocating
;       any absolute adddress references
;*=*=*
LCPTR   LD      HL,$-$           ;P/u saved driver pointer
        LD      E,(HL)           ;Get the lo-order,
        INC     HL               ;  bump to hi-order,
        LD      D,(HL)           ;  & get it for start of move
        PUSH    DE               ;Save start address for ENTRY
        PUSH    HL               ;Save driver memory pointer
        LD      HL,DVRBGN        ;Point to start of driver
        LD      BC,DVREND-DVRBGN;Calc driver length
        LDIR                     ;  & move into driver region
        POP     HL               ;Now pick up the saved
        LD      (HL),D           ;  pointer again and reset
        DEC     HL               ;  it to point to the
        LD      (HL),E           ;  NEW first available address
        POP     DE               ;Recover for ENTRY stuff into DCT
```

If insufficient room exists in the low memory driver region (perhaps it is already filled with COM/DVR, MemDISK/DCT, FORMS/FLT, or some additional driver/filter), then your initialization program should obtain the high memory pointer (HIGH$) via the @HIGH$ SuperVisor Call and relocate the driver to high memory. Remember the HIGH$ pointer points to the first available high memory address but the memory is filled towards lower addresses. The sample filter listed in Chapter 8, the Appendix, illustrates a high memory relocation.

5. The DOS Directory Structure

5.1 GENERAL DIRECTORY CONVENTIONS

The disk operating system uses a one-level directory structure to logically associate a file specification (including the access of any record in that file) to the physical storage space on a disk occupied by the file. This DOS directory occupies an entire cylinder on the disk drive (or logical disk drive if a hard disk is partitioned into multiple logical drives). The directory itself is considered a file with the specification "DIR/SYS".

The directory is composed of three primary parts: A Granule Allocation Table (GAT) contains information pertinent to the allocation of physical disk space. The GAT also contains data that may be considered the disk pack identification. The second part of the directory is a Hash Index Table (HIT) which is used by the DOS to speed access to individual directory records associated with each file stored on the disk. The last part of the directory contains the access information pertinent to each disk file. This information is termed the FILE DIRECTORY ENTRY records.

Before delving into the detailed descriptions of each part, one important item must be discussed concerning the directory. The soft-sectored floppy disk format was first designed by IBM for the 3740. This format defined an identification field for each physical sector on the disk. Preceding the sector is a byte termed the "Data Address Mark". IBM defined two distinct data address marks: An X'FB' was assigned for a sector that contained actual data. An X'F8' was assigned to a "deleted" sector (i.e. one whose data is deleted and the sector is available for use). The convention of use for these data address marks in this operating system is to assign the X'FB' to indicate any "ordinary" sector on the disk - an "ordinary" sector is any sector that is not part of the directory. The X'F8' data address mark is used for all sectors constituting the directory cylinder.

Disk controllers used to access the disk will generally return an indication in a status register of the data address mark detected when reading any given sector. The DOS capitalizes on this scheme by using the returned status as an indicator of what type of sector was read - a directory sector or non-directory sector. When a read-sector (@RDSEC) service request is satisfied by a disk driver, it is the responsibility of the driver to return this status to the caller. If a "normal" sector is successfully read, the driver returns a no-error indication. If a directory sector is successfully read, the driver returns an error code 6 - "Attempted to read system data record".

The first sector (cylinder 0, sector 0) of each disk contains a pointer to the cylinder containing the directory. This pointer is the third byte of the sector. There is also a field in the Drive Control Table which contains a copy of that pointer. When the system requests a read of a directory sector and is returned status which indicates that a regular sector was read instead of a directory sector, it assumes that the disk has been changed since it last accessed the directory and the new disk has its directory on a different cylinder. The system then updates the Drive Control Table (DCT) field by reading the first sector and retrieving its directory cylinder pointer. This condition is used by the system to constantly keep current information on the disk each time the directory cylinder is accessed [the @OPEN and @INIT SuperVisor Calls also act to keep the system current on the disk structure by logging the disk identification via the @CKDRV SuperVisor Call and updating its DCT fields accordingly].

Because of the Data Address Mark conventions employed in the DOS, two SuperVisor Calls have been provided to read/write directory sectors. The @RDSSC (SVC-85) will read a directory sector and update, where necessary, the Drive Control Table directory cylinder field. The @WRSSC (SVC-54) can be used to write a sector to the directory and properly

identify the correct Data Address Mark. Directory sector writes should be verified with the @VRSEC SuperVisor Call. Expect to obtain an error code 6 as previously noted.

5.2 THE GRANULE ALLOCATION TABLE (GAT)

The Granule Allocation Table (GAT) contains a section of information pertinent to the allocation of physical storage space on the disk. For floppy disk drives, this section is composed of two tables: The ALLOCATION table specifies what areas of the disk are allocated or unavailable for use while the LOCKOUT table specifies what areas of the disk are physically unusable. For winchester drives (hard drives), the LOCKOUT table is not used and the ALLOCATION table is extended to include the GAT space normally used by the floppy lockout table. The GAT is wholly contained in the first sector of the directory cylinder. Additional fields are stored within the GAT sector that describe the disk (its pack identification). The GAT also contains certain data specific to the formatting configuration of the disk.

An entire disk is divided into cylinders (tracks) and sectors. The standard sector size is 256 bytes in length. Each cylinder has a specified constant quantity of sectors. Because the DOS uses a single 8-bit register to communicate sector numbers, it will support a maximum of 256 sectors per cylinder. A group of sectors is allocated whenever additional space is needed. This group is termed a GRANULE and is always a constant size for any given disk. This does not mean that the granule is the same size for all disks. The size of a granule generally increases with the increasing size of the disk storage device. The choice of a granule size is a compromise over minimum file lengths and overhead during the dynamic allocation process. It is somewhat dependent on the number of sectors per cylinder because the number of sectors per granule must divide evenly into the number of sectors per cylinder.

The ALLOCATION and LOCKOUT tables are actually bit maps that associate one granule of space per bit. One byte is used to store the information on a single cylinder; therefore, the GAT is configured to provide for a maximum of eight granules per cylinder. In these tables, each bit that is set indicates a corresponding granule in use (or locked out). A reset bit indicates a granule free to be used. In the GAT allocation and lockout bytes, bit 0 corresponds to the first relative granule on a cylinder (denoted as granule 0). Bit 1 corresponds to the second relative granule (denoted as granule 1), bit 2 the third (denoted as granule 2), and so on through bit 7 for the eighth granule (denoted as granule 7). This is illustrated in figure 5-1.

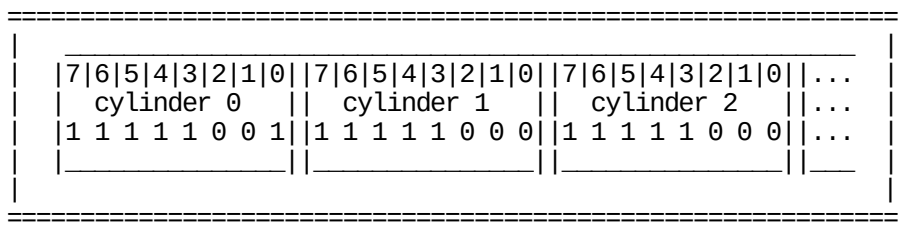

Figure 5-1: Allocation Table Representation

A 5-1/4" single density diskette is formatted at ten sectors per track, five sectors per granule, two granules per track. A two-sided diskette has twice the number of granules per track available on each cylinder. Thus, the single density single, sided 5-1/4" configuration will use only bits 0 and 1 of each GAT byte. The remaining GAT byte will contain all 1's - thereby denoting unavailable granules. A 5-1/4" double density diskette is formatted at 18 sectors per track, six sectors per granule, three granules per track. Thus, this configuration will use bits 0, 1, and 2 of each GAT byte. The standard granule allocation conventions used by the DOS for floppy diskettes are as shown in figure 5-2.

	SECTORS PER TRACK	SECTORS PER GRANULE	GRANULES PER TRACK	MAXIMUM CYLINDERS
5" SDEN	10	5	2	96
5" DDEN	18	6	3	96
8" SDEN	16	8	2	77
8" DDEN	30	10	3	77

Figure 5-2: Allocation for Single-Sided Floppy Media

Figure 5-2 assumes single sided media. The DOS supports two-sided operation within the confines of the hardware interfacing the physical drives to the CPU. A two-headed floppy drive functions as a single volume with the second side treated as an extension of the first in a true cylinder structure. A bit in the Drive Control Table (DCT) indicates one-sided or two-sided drive configuration.

A winchester-type hard disk also has a similar configuration. However, since many different sizes of winchesters are available, the recommended configurations for representative hard drives are covered in chapter 6 - DISK FILE ACCESS AND CONTROL. For the purposes of this chapter, it is sufficient to mention that hard drives may use the first 203 GAT bytes to reference ALLOCATION information (positions X'00' through X'CA'). Hard drives that exceed 203 physical cylinders require remapping or partitioning. Methods of achieving remapping and partitioning are also discussed in chapter 6.

The following describes the structure of the Granule Allocation Table and the information contained in it. The numbers in angle brackets indicate the relative positions of the field within the GAT. Figure 5-3 illustrates the entire GAT.

5.2.1 ALLOCATION TABLE - <Bytes X'00' - X'5F'>

This table contains a bit image of what space is available for use (and conversely what space is not available). GAT+0 corresponds to cylinder 0, GAT+1 corresponds to cylinder 1, GAT+2 corresponds to cylinder 2, and so forth. As previously noted, bit 0 of each byte corresponds to the first granule on the cylinder, bit 1 corresponds to the second granule, etc. A "1" indicates the granule is not available for use. The amount of GAT space assigned to this table permits a maximum of 96 cylinders; however, the formatter restricts the format of 8" media to 77 cylinders.

5.2.2 LOCKOUT TABLE - <Bytes X'60' - X'BF'>

This table contains a bit image of what space has been locked out from use. Granules may be locked out because they either do not physically exist (i.e. granules 3-7 on 5-1/4" double density floppy media) or the verify process of the floppy formatter had detected a bad sector in a granule. The table corresponds on a cylinder for cylinder basis as does the allocation table. It is used specifically during mirror-image backup functions to determine if the disk has the available capacity to effect a backup of the source diskette.

5.2.3 EXTENDED ALLOCATION TABLE - <Bytes X'C0' - X'CA'>

This table is used in hard drive configurations by extending the ALLOCATION table from X'00' through X'CA' and omitting a distinct lockout table. The table then provides a capacity of up to 203 cylinders. The hard drive DBLBIT bit is available in the Drive Control Table to permit combining two physical cylinders into a single logical cylinder provided the limit of 256 sectors per cylinder is not exceeded. This arrangement therefore provides support for up to 406 cylinders. Lockout information, where available, is generally denoted by setting the appropriate bit assigned in the ALLOCATION table.

Hard drives generally cannot be backed up in a mirror-image manner and the BACKUP utility will prohibit it by automatically entering the RECONSTRUCT mode.

5.2.4 DOS VERSION - <Byte X'CB'>
This field contains the operating system version used in formatting the disk. Disks formatted under DOS 6.0 will have a value of X'60' contained in this byte. It is used to determine whether or not the disk contains all of the parameters needed for DOS 6.0 operation.

5.2.5 CYLINDER EXCESS - <Byte X'CC'>
This byte contains the number of logical cylinders in excess of 35. It is used to minimize the time required to compute the maximum cylinder formatted on the diskette and to update the Drive Control Table. It is designed to be excess 35 so as to provide complete compatibility with previous systems that restricted the floppies to 35 tracks and did not maintain the byte. This field is read to update the Drive Control Table during the process of logging the disk by the @CKDRV SuperVisor Call process.

5.2.6 DISK CONFIGURATION - <Byte X'CD'>
This byte contains data specific to the formatting of the diskette. It is fielded as follows:

Bit 7 Set to "1" indicates the disk is a DATA disk; thus all but two directory slots are available for data files. Set to a 0 indicates that the disk is a SYSTEM disk which reserves 14 additional directory slots for system files providing a maximum of 240 directory entries for data files.

Bit 6 Set to "1" implies double density formatting. Set to 0 implies single density formatting.

Bit 5 Set to "1" indicates two-sided floppy media. Set to 0 indicates single-sided floppy media.

Bit 4 This is reserved for internal system use.

Bit 3 This is reserved for internal system use.

Bits 2-0 Contain one less than the number of granules per track that were used in the formatting process.

5.2.7 DISK PACK PASSWORD - <Bytes X'CE' - X'CF'>
This field contains the 16-bit hash code of the disk master password. Its storage is in standard low-order high-order format. The password itself must be composed of the characters <A-Z, 0-9> with the first character alphabetic. The 16-bit hash code can be obtained from the DOS for any given password. This is done by placing the password string into an 8-character buffer left-justified and padded with spaces, and then invoking a system overlay.

The following code illustrates this operation.

```
HASHMPW LD    DE,PSWDPTR    ;Point to the 8-char buffer
        LD    A,0E4H        ;Specify password hash function
        RST   40            ;Issue the RST instruction
```

The 16-bit password hash code will be returned in register pair HL. Registers AF, B, DE, and HL are altered. The operating system will not return to the address following the RST 40 instruction when the SVC function code is an internal system request code (i.e. has bit-7 set) but will return to the previous caller. Thus, it is necessary to CALL this routine.

5.2.8 PACK NAME - <Bytes X'D0' - X'D7'>
This field contains the diskette pack name. This is the same name displayed at boot up if the diskette is a system diskette used for the boot operation [specifically, the boot name is obtained from the System Information Sector but is managed coincidentally by FORMAT and ATTRIB. It is also the name displayed during a FREE or DIR or obtained by the @DODIR SuperVisor Call. The name is assigned during the formatting operation or re-assigned during an ATTRIB renaming operation.

5.2.9 PACK DATE - <Bytes X'D8' - X'DF'>
This field contains the date that the disk was formatted or the date that it was used as the destination in a mirror-image backup operation. If the diskette is used during a BOOT, this date will be displayed adjacent to the pack name [actually, the boot date is obtained from the System Information Sector but is managed coincidentally by BACKUP].

5.2.10 RESERVED FIELD - <Bytes X'E0' - X'F4'>
This field is reserved for future use under DOS version 6. It formerly contained the AUTO command buffer under earlier versions of the DOS; however, since Version 6 supports 79-character command lines, the System Information Sector now holds the AUTO command buffer for use during a BOOT operation.

5.2.11 MEDIA DATA BLOCK - <Bytes X'F4' - X'FF'>
Effective with LDOS 6.2.0, this field contains a header sub-field and a sub-field replicating the last seven bytes of the drive control table in use and associated with the media when the media was formatted.

Bytes 0-3 contains an X'03' followed by the string, "LSI".

Bytes 4-10 replicates the last seven bytes of the DCT during format.

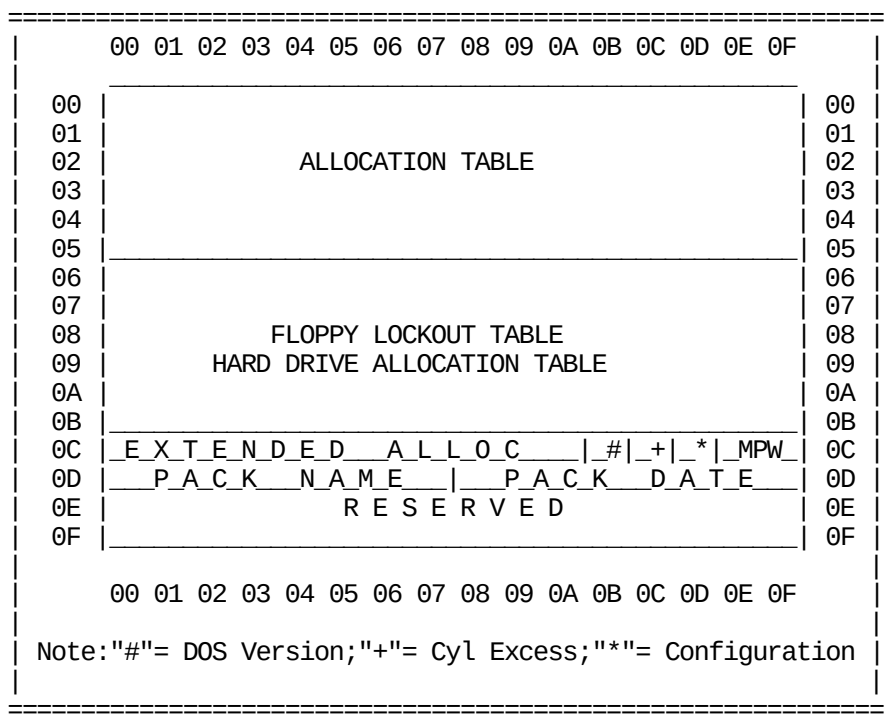

Figure 5-3: Granule Allocation Table Illustrated

5.3 THE HASH INDEX TABLE (HIT)

The Hash Index Table is the key to addressing any file in the directory . It is designed so as to pinpoint the location of a file's primary directory entry with a minimum of disk accesses. A minimum quantity of disk accesses is useful to keep system overhead low while at the same time providing for rapid file access.

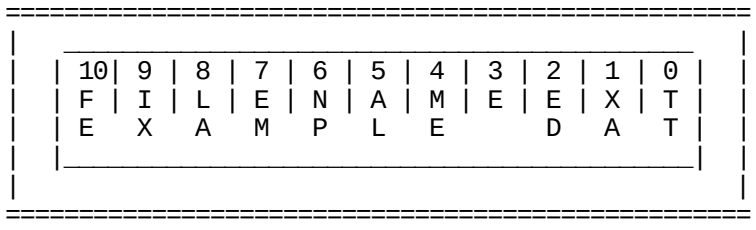

Figure 5-4: File NAME/EXT buffer

When an application requests the system to open a file, the system must locate that File's Primary Directory Entry (FPDE) record which contains the disk storage data needed to address the file. The procedure that the system uses to locate a file's FPDE is to first take the file name and extension and construct an 11-byte field with the file name left justified and padded with blanks so as to fill out eight positions. The file extension is then inserted, padded with blanks, and will occupy the three least significant bytes of the 11-byte field. The resulting string is illustrated in figure 5-4. This field is then processed through a hashing algorithm which produces a single byte value in the range X'01' through X'FF' (a hash value of X'00' is reserved to indicate a spare HIT position). The following code may be used to obtain the one-byte hash code for an 11-character NAME/EXT buffer.

```
HASHSPEC LD    HL,SPECPTR    ;Point to the 8-char buffer
         LD    A,0D4H        ;Specify filename hash function
         RST   40            ;Issue the RST instruction
```

The one-byte hash code is returned in the accumulator. Registers AF, B and HL are altered. The operating system will not return to the address following the RST 40 instruction when the SVC function code is an internal system request code (i.e. has bit-7 set) but will return to the previous caller. Thus, it is necessary to CALL this routine.

Each file's hash code is stored in the Hash Index Table (HIT) at a position which is associated with the FPDE record containing the file's access information. After the OPEN routine obtains the hash code for the file identified in the file specification, it searches the HIT for a matching hash code. Since more than one 11-byte string can hash to identical codes, the opportunity for a "collision" exists (a collision is where two or more file names result in the same hash code). For this reason, the search algorithm will sequentially scan the HIT for a matching code entry and when found, will then read the FPDE record corresponding to the matching HIT position. OPEN will then compare the file name/ext stored in the FPDE record with that provided in the file specification. If both match, the file's FPDE directory record has been found. If the two fields do not match, the HIT entry was a collision and the algorithm continues its search from where it left off. If a match to the hash code is not found in the HIT, the file does not exist on that disk drive. If the user passed a drive specification (drivespec) as part of the file specification, a "File not found" error will be returned. If no drivespec was passed, the system will search all drives in logical number order starting with drive 0. If the @INIT SuperVisor Call was used to open the file, the system will first use @OPEN to determine the possible existence of the file. If @OPEN advises that the file has not been found, then @INIT will create the file by obtaining a spare HIT position then constructing the corresponding FPDE.

The position of a file's hash code entry in the Hash Index Table is called the Directory Entry Code (DEC) for the file. All files will have at least one DEC. A contiguous block of granules allocated to a file is termed an EXTENT. The FPDE record contains fields to hold the data on four extents. Files that use more than four extents because they are either large (an extent can address a maximum of 32 contiguous granules) or fractured into non-contiguous space require extra directory records to hold the additional extents. These additional records are termed the File's Extended Directory Entries (FXDE) which also have four extent fields each. A Directory Entry Code is also used to associate an FXDE with a HIT entry. Thus, a file will have DECs for each FXDE record and use up more than one filename slot in the HIT. Therefore, to maximize the quantity of file slots available, you should keep your files below five extents wherever possible.

The FPDE and FXDE records are contained in the remaining sectors of the directory cylinder. The Directory Entry Codes are mapped to the FPDE/FXDE records by each DEC's position in the Hash Index Table. Conceptualize the HIT as eight rows of 32-byte fields as shown in figure 5-5. Each row will be mapped to one of the directory entry records in a directory sector. The first HIT row to the first directory entry record, the second HIT row to the second directory entry record, and so forth. Each column of the HIT field (the 0-31) is mapped to a directory entry sector. The first column is mapped to the first directory entry sector in the directory cylinder (not including the GAT and HIT). Therefore, the first column corresponds to sector number 2, the second column to sector number 3, and so forth. The maximum quantity of HIT columns actually used will be governed by the disk formatting according to the formula: N = (number of sectors per track times the number of sides) minus two.

In the 5-1/4" double density single-sided configuration, there exist eighteen sectors per cylinder - of which two are reserved for the GAT and HIT. Since only sixteen directory entry sectors are possible, only the first sixteen positions of each HIT field are used. Other formats will use more or less columns of the HIT, depending on the quantity of sectors per cylinder in the formatting scheme.

This arrangement works nicely when dealt with in assembly language for interfacing. Consider the DEC value of X'84'. If this value is loaded into the accumulator, a simple:

```
AND     1FH    ;Strip off row and
ADD     A,2    ;  calculate sector
```

will extract the sector number of the directory cylinder containing the file's directory entry. If that same value of X'84' was operated on by:

```
AND     0E0H   ;Strip off sector and keep row
```

the resultant value will be the low-order starting byte of the directory entry record assuming that the directory sector was read into a buffer starting at a page boundary. This procedure makes for easy access to the directory record. The system provides two routines, @DIRRD and @DIRWR, that will read/write the correct directory entry sector corresponding to a DEC. The directory I/O uses the system buffer and a pointer in the HL register pair is automatically positioned to the proper FPDE (the buffer is on a page boundary for physical I/O). @DIRWR performs verification after write!

The following figure may help to visualize the correlation of the Hash Index Table to the directory entry records. Each byte value shown represents the position in the HIT and is, in fact, the Directory Entry Code value. The actual contents of each byte will be either an X'00' indicating a spare DEC, or the one-byte hash code of the file occupying the corresponding directory entry record.

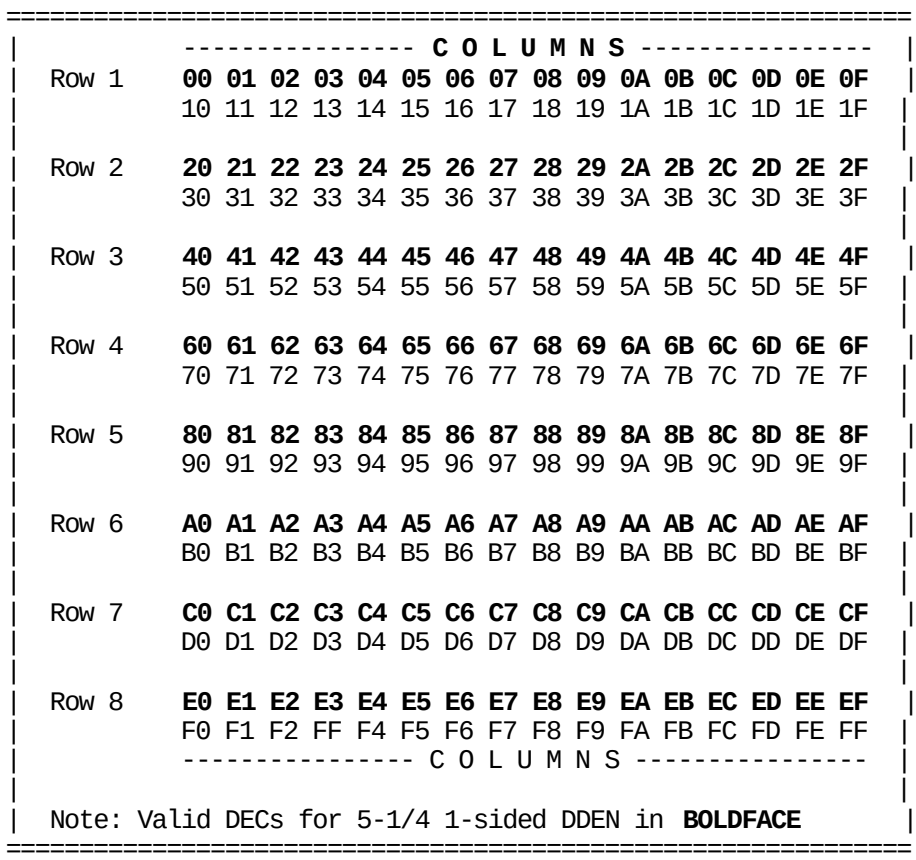

Figure 5-5: Directory Entry Codes

The eight directory entry records for the directory entry sector numbered 2 would correspond to DEC assignments in HIT positions 00, 20, 40, 60, 80, A0, C0, and E0. The positions shown in figure 5-6 are reserved for system overlays on a system disk (as determined from the configuration field defined in the section on the Granule Allocation Table). These entry positions, of course, correspond to the first two rows of each directory entry sector for the first eight directory entry sectors. Since the operating system accesses these overlays by the DEC position in the HIT rather than by file name, these positions are always reserved for system disks. Data disks reserve only positions 00 (BOOT/SYS) and 01 (DIR/SYS).

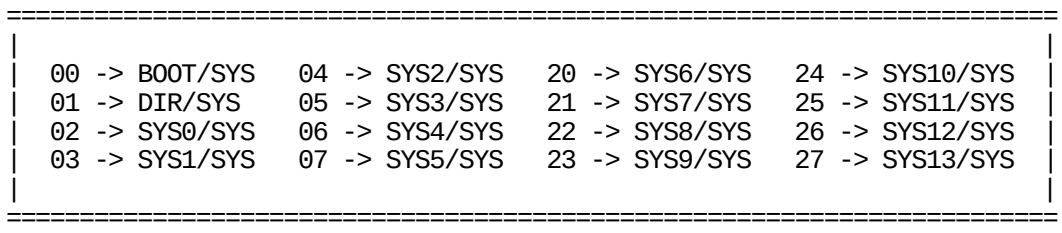

Figure 5-6: Directory Entry Codes reserved for SYSTEM files

The Hash Index Table limits the design of the system to a maximum support of 256 files on any one logical drive. With the current state of the art in hard disk drive technology, that limit may prove too small a number. Obviously, additional file slots are available by partitioning a hard drive into two or more logical drives with each partition containing its own directory. The customized hard disk driver then translates the logical cylinder/sector information to physical parameters. This concept is discussed in detail in chapter 4.

5.4 THE DIRECTORY RECORD STRUCTURE

The disk directory contains the information sufficient to access all files on the disk. We have already shown that disk space allocation is defined in the Granule Allocation Table. We have also revealed in the previous section how the operating system uses file hash codes stored in the Hash Index Table to locate the Directory Entry Code for each file. Each DEC refers to a specific directory entry record. A directory record is 32-bytes in length. Thus, each directory entry sector contains eight directory entry records.

The HIT was shown to contain a maximum of 256 Directory Entry Codes. Since there are eight entries per sector, the maximum number of directory entry sectors is 32 (256 divided by 8). If we add one sector for the GAT and one for the HIT, we discover that the maximum length of the entire directory can be 34 sectors. The directory must be contained completely on a single cylinder. Therefore, the exact length of the directory and hence the number of directory entries is highly dependent on the size of a cylinder. For example, an 18-sector per cylinder formatted disk will have 16 directory entries and hence 16 times 8 or 128 directory entries. Consult the section on the HIT for the formula calculating the number of directory sectors.

	SECTORS PER CYLINDER	DIRECTORY RECORDS	FILES AVAILABLE PER DIRECTORY TOTAL	SYSTEM DISK	DATA DISK
5" SDEN-1	10	8	64	48	62
5" SDEN-2	20	18	144	128	142
5" DDEN-1	18	16	128	112	126
5" DDEN-2	36	32	256	240	254
8" SDEN-1	16	14	112	96	110
8" SDEN-2	32	30	240	224	238
8" DDEN-1	30	28	224	208	222
8" DDEN-2	60	32	256	240	254
5" HARD-<1>	128	32	256	240	254
5" HARD-<2>	64*2	32*2=64	256*2=512	240*1+ 254*1=494	254*2=508
5" HARD-<4>	32*4	30*4=120	240*4=960	224*1+ 238*3=938	238*4=952

Note: Hard drive values show total entries for all partitions.
"<x>" denotes the number of logical drives.

Figure 5-7: Directory entries for various media

The first two directory entries of the first eight directory entry sectors are reserved for system overlays on a SYSTEM disk. A DATA disk reserves only the first directory entry of the first two directory entry sectors. The total capacity of files is equal to the number of directory sectors times eight (since 256/32 = 8). The quantity available for use will always be reduced by 16 on a SYSTEM disk or by two on a DATA disk to account for those entries reserved for the operating system. Figure 5-7 shows the record capacity (file capacity) of each floppy format type. The dash suffix on the density indicator represents the number of sides formatted. The figure also lists representative values for 5 megabyte winchester drives (typical ST-506 compatible: 4 heads, 32 sectors per track, 153 tracks per head).

Because of the Data Address Mark conventions employed in the operating system, two SuperVisor Calls have been provided to read/write directory entry sectors. The @DIRRD (SVC-87) will read a directory entry sector into the system buffer when passed a drive and DEC. Register pair HL is automatically positioned to the proper directory entry in the buffer corresponding to the DEC (the buffer is on a page boundary for physical I/O). This buffer can be written back to the directory using the @DIRWR (SVC-88), again by specifying only the drive and DEC.

Any sector of the directory may be requested for I/O by using either @RDSSC (SVC-85) for reading (which will update the Drive Control Table directory cylinder field where required) or @WRSSC (SVC-54) can be used to write a sector to the directory and properly identify the correct Data Address Mark. Directory sector writes should be verified with the @VRSEC SuperVisor Call. Expect to obtain an error code 6 as previously noted. This procedure makes for easy access to the GAT and HIT directory records. Abbreviated contents of the directory may also be retrieved via the @DODIR and @RAMDIR Supervisor Calls.

Finally, since the directory is conceptualized as a data file and contains its own directory entry, DIR/SYS, the directory can be treated as a file and OPENed - just like any other file. READ access is granted for this method. Under no circumstances should you attempt to write to the directory by defeating the password protection when the directory is opened as a file and accessed as such. Failure to heed this warning may make the directory unreadable.

The expert programmer may find useful information in the directory - especially for those that write catalog programs. Since the directory information is so vital to the friendliness of programs, the system displays a great deal of information on each file via the directory command. The following provides detailed information on the contents of each directory entry field. The numbers contained in angle brackets refer to the relative byte(s) of the field in the record.

5.4.1 ATTRIBUTES - <Byte 0>

This byte contains the entire attributes of the designated file. It is encoded as follows:

Bit 7 This bit flag is used to indicate whether the directory entry is the file's primary directory entry (FPDE) or one of its extended directory entries (FXDE). Since a directory entry can contain information on up to four extents, a file that is fractured into more than four extents requires additional directory records. If this bit is a "0", the entry is an FPDE. If set to a "1", the entry is an FXDE.

Bit 6 A SYStem file is noted by setting this bit to a "1". If set to a "0", the file is declared a non-system file. It is used as a reference in DOS utilities and as a double check when the DOS overlay loader accesses a file in the reserved HIT entries.

Bit 5 This bit is used to designate the corresponding file as a Partitioned Data Set. The PDS is a library file managed by a utility program called PRO-PaDS. The utility is available from MISOSYS.

Bit 4 This activity bit is used to indicate whether the directory record is in use or not. If set to "1", the record is in use. If set to a "0", the directory record is not active although it may appear to contain directory information. A previously active file is removed only by resetting this bit, removing its HIT entry, and deallocating its space. Thus, the FPDE is left intact except for this bit.

Bit 3 Specifies the visibility; if "1", the file is INVisible to a DIRectory display or other library function where visibility is a parameter. If a "0", then the file is declared VISible.

Bits 0-2 Contain the access protection level of the file. The 3-bit binary value is encoded as follows:

0 - FULL 1 - REMOVE 2 - RENAME 3 - WRITE
4 - UPDATE 5 - READ 6 - EXEC 7 - NO ACCESS

5.4.2 FLAG FIELD - <Byte 1>

This field contains four file flags in bits 7-4. The low-order nibble is associated with the DATE field. The flags are encoded as follows:

Bit 7 When this bit is set, the system will be kept from deallocating any unused space at the end of the file when the file is closed. This bit will be set to a "1" if the file was "CREATEd" by the DOS library command, CREATE. Such a file will never shrink in size. The file will remain as large as its largest allocation.

Bit 6 This flag is termed the "MOD flag". If this flag is set to a "1", it indicates that the file has not been backed up since its last modification. The BACKUP utility is the only DOS facility that will reset this flag. It is set during the file close operation if the File Control Block (FCB+0, Bit 2) indicated a modification of file data.

Bit 5 This bit is set by the system when a file is opened with UPDATE or greater access. It is used to detect the presence of an open file for subsequent OPENs of the same file. The bit is reset by the CLOSE operation.

Bit 4 This bit is used internally by the system.

If the ATTRIBUTE field identifies the record as an FXDE, then this entire byte (flags and month) will contain the Directory Entry Code of the directory entry forward linked to this one. This entry is the backward link.

5.4.3 MODIFICATION DATE - <Bytes 1 - 2>

This field is composed of 12 bits, the low-order nibble of DIR+1 and the entire byte of DIR+2. It contains the month, day, and year for the day that the file was last modified. The field is encoded as follows.

Bits 11-8 Contain the binary month of the last modification date. If this field is a zero, the system date was not set when the file was established nor since if it was updated.

Bits 7-3 Contain the binary day of last modification.

Bits 2-0 Contain the binary YEAR - 1980. That is to say that 1980 would be coded as 000, 1981 as 001, 1982 as 010, etc.

5.4.4 EOF OFFSET - <Byte 3>

This field contains the end-of-file offset byte. It points to the position in the ending sector of where the next byte can be placed. If EOF OFFSET is a zero, it means that a full sector of 256 bytes had been written to the last sector of the file and the next byte must be written to a new sector. This byte, and the ending record number (ERN), form a triad pointer to the byte position immediately following the last byte written.

5.4.5 LOGICAL RECORD LENGTH - <Byte 4>

This field contains the Logical Record Length (LRL) specified when the file was initially generated (via @INIT) or subsequently changed by being overwritten with some file that has another LRL via "COPY (CLONE)" or "BACKUP". A value of "0" indicates that the LRL is equal to 256.

5.4.6 FILE NAME - <Bytes 5 - 12>

This field contains the name portion of the file specification. The file name will be left justified and padded with trailing blanks. The name will always be in upper case characters <A-Z, 0-9>. If a file has FXDE records in addition to the FPDE, only the FPDE will contain the filename in this field.

5.4.7 FILE EXTENSION - <Bytes 13 - 15>

This field contains the extension portion of the file specification. As in the name field, it is left justified and padded with trailing blanks. If a file has FXDE records in addition to the FPDE, only the FPDE will contain the file extension in this field.

5.4.8 OWNER PASSWORD - <Bytes 16 - 17>

This field contains the hash code of the OWNER password. The OWNER password is used to gain full access to a password protected file. Passwords are assigned at file creation and/or changed with the ATTRIB library command. The 16-bit hash code for a file password can be obtained using the method shown for obtaining the disk master password hash code.

5.4.9 USER PASSWORD - <Bytes 18 - 19>

This field contains the hash code of the USER password. THE USER password is required to access the file at the level of protection identified in the attribute field. Passwords are assigned at file creation and/or changed with the ATTRIB library command. The 16-bit hash code for a file password can be obtained using the method shown for obtaining the disk master password hash code.

5.4.10 ENDING RECORD NUMBER - <Bytes 20 - 21>

This field contains the ending record number (ERN) which is based on full sectors. If the ERN is zero, it indicates a file where no writing has taken place (or a lot of writing whereby you forgot to close the file). If the LRL is not 256, the ERN value represents the sector where the EOF occurs. Each time a sector is written to the disk, the ERN is advanced by one - even if the sector is not a full sector. Thus, if ERN shows 3, and EOF OFFSET shows 0, then three full sectors have been written (relative 0, 1, and 2). If ERN shows 3 and EOF OFFSET shows 62, then two full sectors and one partial sector of 62 bytes have been written.

5.4.11 EXTENT DATA FIELDS - <Bytes 22 - 29>

The extent data fields contain data on the allocation of disk space for the file. Each field is composed of 16-bits and can contain the allocation information for a maximum of 32 contiguous granules. Their contents tell you what cylinder stores the first granule of the extent, what is the relative number of that granule, and how many contiguous granules are in use in the extent. Each extent is encoded according to the pattern illustrated for extent field 1.

5.4.11.1 Extent Field 1 - <Bytes 22-23>

Bits 15-8 Contain the cylinder number for the starting granule of that extent. The extent uses space on the disk starting from this cylinder and the sector based on the starting granule, for as many granules as are noted in bits 4-0.

Bits 7-5 Contain the relative granule number (0-7) in the cylinder which is the first granule of the file for that extent. This value is numbered starting from zero. (i.e. a "0" indicates that the first granule in use is the first granule on the cylinder. This would be sector 0. A "1" would indicate that

the first granule in use is the second granule on the cylinder. If there are 6 sectors per granule, sector 6 would start the extent. A "2" would indicate that the first granule in use is the third on the cylinder. If there are 6 sectors per granule, then the first sector in use would be sector 12.)

Bits 4-0 Contain the quantity of contiguous granules in the extent. The value is relative to 0. Therefore a "0" value implies one granule, "1" implies two, and so forth. Since the field is 5 bits, it contains a maximum of X'1F' or 31, which would represent 32 contiguous granules.

5.4.11.2 Extent Field 2 - <Bytes 24-25>

Structured the same as 1.

5.4.11.3 Extent Field 3 - <Bytes 26-27>

Structured the same as 1.

5.4.11.4 Extent Field 4 - <Bytes 28-29>

Structured the same as 1.

5.4.12 FXDE LINK FLAG - <Byte 30>

This field is a flag noting whether or not a link exists to an extended directory record. If no further directory records are linked, the byte will contain X'FF'. If the value is X'FE', a link is recorded to an extended directory entry.

5.4.13 FXDE LINK POINTER - <Byte 31>

This is the forward link to the extended directory noted by the FXDE LINK FLAG. The link pointer is the Directory Entry Code (DEC) of the extended directory record. The FXDE will then contain the Directory Entry Code of this directory entry in the FLAG field and the month sub-field of the DATE field. This other DEC becomes the backward link.

Figure 5-8 represents one directory entry record illustrating a file with two extents.

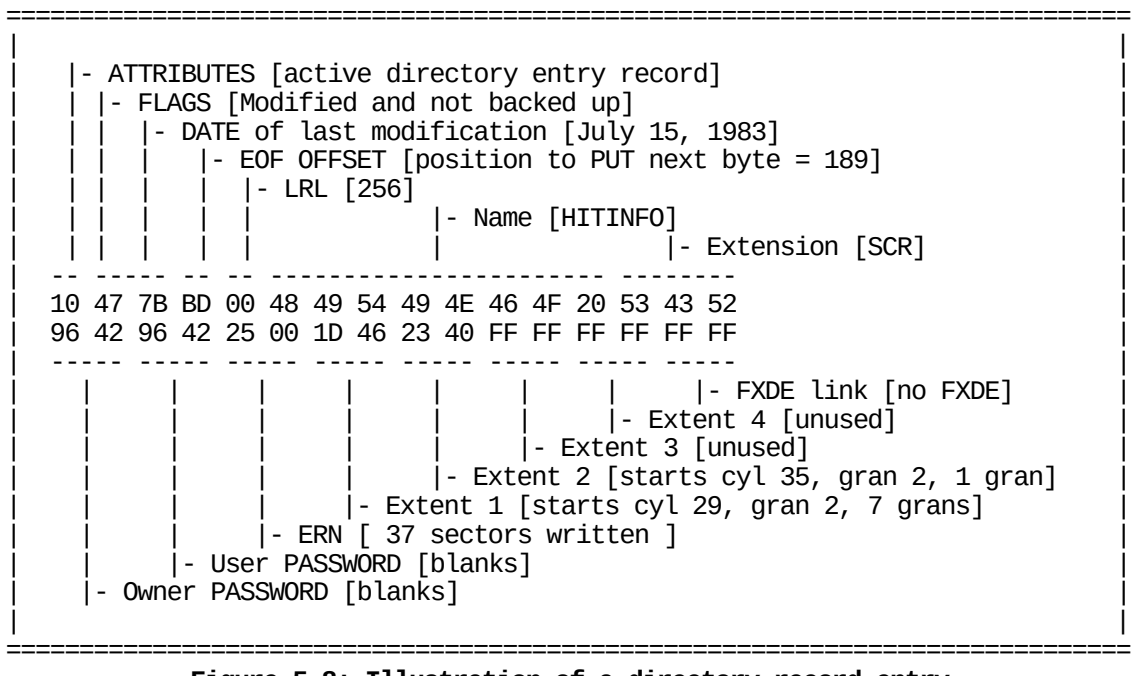

Figure 5-8: Illustration of a directory record entry

6. Disk File Access and Control

6.1 GENERAL FILE STRUCTURES

The primary reason we make use of computer systems is to aid us in managing large volumes of data. Our computers utilize the Disk Operating System (DOS), the fundamental purpose of which is to make an easier job of handling the storage of that data. We usually want rapid access to data; therefore, the random access disk storage device is the selected storage medium due to its inherent speed in accessing data. These devices take two forms, floppy disks with either one or two heads which use a single diskette with corresponding one or two surfaces, and winchester hard disk drives which consist of one or more platters with each platter consisting of two surfaces. The hard disk drive may use either a fixed or removable media.

Regardless of the disk drive type, each surface is divided into concentric circles of storage area called tracks. Each track is then subdivided by a fixed number of subareas called sectors. Although the number of sectors per track may vary from one media type to another, the number of sectors in each track of the same media is constant. The DOS assigns numbers to every sector, every track, and every surface. Surfaces are numbered consecutively by one starting from zero. Tracks are numbered consecutively by one starting from zero at the outermost portion of the disk giving the innermost track the highest number. A CYLINDER consists of the like numbered tracks on all surfaces. For example, on a two-surface media, track zero of surface zero and track zero of surface one are grouped together into cylinder zero. The sectors in each track are numbered starting from zero. Thus, each track contains like numbered sectors - regardless of track number or surface. Therefore, each sector on a disk is designated unique by its respective sector, surface, and track numbers.

Data is stored in these sectors. Obviously, if your program had to keep track of all the sectors your data was occupying, you would have to make the program necessarily complex [if this is not obvious, you will become a believer after reading the section on file access]. The DOS alleviates you of this task by totally managing the storage space. It does this by associating an 8-character name with the storage areas assigned to a logically connected set of data called a file. Thus, the name becomes a FILENAME. The DOS also permits a 3-character extension to be affixed to that name to better classify the type of file: data, text, command program, etc. This extension is termed the FILE EXTENSION. You can attach a unique PASSWORD and access level such as EXECute only or READ only to each file in order to provide a greater degree of protection to the information contained in the file. Furthermore, the file can be placed on any of up to eight disk storage devices. Each disk drive is assigned a DRIVE number from zero to seven. Therefore, to uniquely reference a file, we put together the NAME, EXTENSION, PASSWORD, and DRIVE and refer to the result as a FILE SPECIFICATION. The term, file specification, is rather long so we shorten it to "filespec".

In order to assign space on a disk for storage of file data, the DOS groups together a quantity of sectors into a GRANULE. The size of the granule varies according to the capacity of the media. This variation in size was discussed in the GRANULE ALLOCATION TABLE section. The DOS assigns space dynamically to a file. This means that space is reserved for the file only when the file needs it. The process whereby the system looks for additional space is termed the ALLOCATION process. The DOS would prefer to allocate granules that are connected sequentially to each other. The sequential connections are only logical in nature, not physical connections. The DOS prefers to access a disk drive device in a particular order to optimize the transfer of data. Since the time to step the head from one cylinder to another is greater than the time to access a sector in the cylinder where the head is positioned, it is far preferable to access all sectors of a cylinder before stepping to another cylinder. If we look at sequential access of a file, we then would want to conceptualize a sequential connection to start from track zero, surface zero, sector zero incrementing the numbers like the odometer in a car as it

travels the turnpike. In this manner, all sectors of a cylinder are accessed before the disk drive has to step to the next cylinder.

It is not always possible to allocate space consecutively. For instance, say we want to add a granule to an existing file but the next granule consecutive to the last granule of the file has already been allocated to another file. Our file must then be fractured into more than one piece. We term each piece of the file an EXTENT. The system's file access routines logically connect each EXTENT so to a program accessing the file, it appears as if the file exists as one continuous allocation of space.

The disk directory stores all the allocation data on each file contained on the disk. Allocation data on a particular file is stored in a directory entry record. Each record can hold the allocation information on up to four extents. The first record is termed the File's Primary Directory Entry or FPDE while all succeeding directory records are considered to be the File's Extended Directory Entries or FXDE records. In order to access the file data, the system's file access routines must utilize the information contained in the file's FPDE.

It is impractical to have to read the FPDE each time another sector of data is transferred. Therefore, the scheme employed is to access the directory once in a process to obtain all of the file's access information and place the information into a memory area termed a File Control Block (FCB). The actual process is termed "opening the file". The reverse process, that of updating the directory entry once the access of a file is complete is termed "closing the file". The DOS provides SuperVisor Call requests to perform the OPEN and CLOSE functions. These type of requests are called "file control" functions since they give you the means of controlling the disk file. Other types of requests are associated with accessing the data in a file and are thus called "file access" requests. INTERFACING VIA SUPERVISOR CALLS, chapter 7, describes each access and control SuperVisor Call.

Data is generally collected into units called RECORDS. These may be fixed-length records with each record being exactly the same length or they may be variable length records where the length of the record varies from record to record. Fixed-length records can be accessed sequentially (i.e. starting from record zero and continuing to the last record of the file). This type of access is termed RECORD I/O. The DOS supports fixed length records from one to 255 characters in length by automatically handling the blocking and deblocking of records into and out of the disk file I/O buffer. Since the DOS standardizes disk file I/O buffer sizes at 256 characters each, record lengths of 256 are handled directly without recourse to the blocking and deblocking used on shorter records and these records can also be transferred to and from the disk more quickly. Record sizes larger than 256 can be used in an application program; however, the blocking and deblocking of records must be performed entirely within the application while, in general, the application will use 256-character records to and from the system. Henceforth, any reference to the term RECORD will consider to be associated with a record which ranges from 1 to 256 characters in length.

Fixed length records can also be accessed directly by record number (which is customarily called RANDOM ACCESS). The DOS provides SuperVisor Call requests to position the record pointer maintained in the File Control Block to the record of choice. The application can then address the record via READ or WRITE SuperVisor Call access requests. Additional SVCs provide other functions associated with the access of a file.

The structure of variable length records is highly dependent on the programming language used to code the program. Most high-level languages (BASIC, FORTRAN, etc) provide variable length file structures which may not be equivalent across each language. One common structure which is supported by more than one language is to use a character or character combination to represent the end of the record. The BASIC language operating under Version 6 uses the ASCII code X'0D' which is a CARRIAGE RETURN to indicate the end

of a variable length record. Some systems use CARRIAGE RETURN followed by LINE FEED (X'0A'). Some languages use a one-byte or two-byte length indicator within the record to indicate the actual length of the record. Program files that are directly executable are, in fact, variable length record files which use a one-byte length field within each record. These "load module" files even include a record TYPE character which permits the specification of different records for different purposes within the same file.

Some files may not even be able to be conceptualized as containing fixed or variable length records. You might consider a word processing text file as not falling into the above classification although each paragraph may, in fact, be a "record". Other files may be variable length but include an index which points to the beginning of each record or group of records. The records are accessed sequentially after the record pointer is extracted from the index. This type of access is usually called Indexed Sequential Access Method (ISAM). Both the operating system's library files and the Partitioned Data Set files supported under the PRO-PaDS utility are ISAM files. The bottom line is for you to determine the type of access you want to employ after exploring the nature of your data and understanding how the system accesses disk files.

There are three methods which are used in application programs to access disk files. The first method is to consider the file as a stream of characters. This access method uses the GET and PUT character I/O SuperVisor Call functions and was discussed in chapter 3, DEVICE INPUT/OUTPUT INTERFACING. The second method is where your file contains physically consistent fixed length records. In this case, it is probably practical to consider RECORD I/O. The third method is to use 256-byte records and perform your own blocking or deblocking as required.

The following sections describe the methods used to control and access files. The last section completely describes the fields in the File Control Block which is used in all interfacing of disk files.

6.2 CONTROLLING DISK FILES

When a file is to be opened for access, the application program initially provides the file specification to the DOS by placing it in the File Control Block (FCB) which will be used for the file. The program then invokes the OPEN function. The DOS, in turn, searches the disk drive(s) for the file's directory entry. Once found, it replaces the filespec in the FCB with information needed by the file access routines. The system then manages the FCB contents according to the demands of the file access requests. The following sections will illustrate some of these control functions.

6.2.1 Getting Filespecs

From where does a program obtain the filespec? You are already familiar with the DOS commands that appear to get the filespec from the command line. Let's take a look at this method. You will learn from the chapter on SuperVisor Calls that when the system transfers control to a program, register pair HL contains a pointer to the first non-blank character on the command line which terminated the name of the executing program. Let us assume that our program will use a command line syntax as follows:

```
PROGRAM-NAME FILE-SPECIFICATION (PARAMETERS)
```

The command-line pointer will be pointing to the first character of the file specification. For the moment, let's make the filespec entry mandatory. We can then code the routine to fetch the filespec as follows:

```
ENTRY   LD    DE,FCB1      ;Point to FCB
        LD    A,@FSPEC     ;Identify the SVC
        RST   40           ;Invoke the SVC
        JP    NZ,SPCERR    ;Transfer on error
```

The @FSPEC SVC will transfer the filespec contained on the command line into the FCB . Any conversion to upper case will be performed as required which permits the entry of the filespec in upper or lower case. Typically, you would want to provide a default file extension to save the user the time it takes to enter up to four additional characters when the application is designed for a class of file (such as TXT, ASM, JCL extensions). A default file extension will not override any extension entered with the filespec. A default will add an extension provided by the program only if the user omitted one. This default can be added as follows:

```
            PUSH  HL          ;Don't disturb command line pointer
            LD    HL,TXTEXT   ;Point to storage of default
            LD    DE,FCB1     ;Point to FCB as required
            LD    A,@FEXT     ;Identify the SVC
            RST   40          ;Invoke the SVC
            POP   HL          ;Restore the pointer
            .
            .
            .
    TXTEXT  DB    'TXT'       ;Data  field for default extent
```

Other times we may want to prompt the user to enter a filespec. This is achieved through a combination of @DSPLY and @KEYIN as follows:

```
            LD    HL,SPCMSG$  ;Point to message
            LD    A,@DSPLY    ;Identify the SVC
            RST   40          ;Invoke the SVC
            LD    HL,FCB1     ;Use the FCB for input buffer
            LD    BC,31<8.OR.0 ;Specify 31 chars & C=0
            LD    A,@KEYIN    ;SVC for line input
            RST   40          ;Invoke the SVC
            JP    C,GOTBRK    ;Transfer on <BREAK>
            LD    D,H         ;Copy the FCB pointer to DE
            LD    E,L
            LD    A,@FSPEC    ;Now parse the entry to
            RST   40          ;  handle l/c to U/C
            JP    NZ,SPCERR
            .
            .
    SPCMSG$ DB    'Enter the input filespec',13
```

This routine will display the "Enter the input filespec" message and place the user input into the FCB. The @FSPEC request will then process the user entry to convert any lower case to upper case while it tests the validity of the entry.

6.2.2 Password Protection of Files

Any discussion concerning the opening of disk files must begin with a discussion of file password protection. This is a subject that has not been too well understood and deserves sufficient explanation. File protection is a process whereby access to a file can be limited to either a level of access (read, write, remove, etc,), to the entry of a password, or to both a level of access and a password requirement. The DOS achieves this file protection capability through a combination of two password fields and a protection level field for each file. The file password fields are termed the OWNER password and the USER password. Users familiar with earlier versions of the DOS may be familiar with the earlier corresponding terms of UPDATE and ACCESS which were changed in release 6 to OWNER and USER respectively to avoid any confusion with the protection level.

The protection level field (we will use the term PROT) is associated with the USER password and indicates what level of access to the file is granted when the USER password is part of the file specification at the time that the file is opened. The different levels of access granted are shown in figure 6-1. Suppose that the access level is READ. If the filespec includes the USER password, then the file will be opened but the system

will only permit the opener to read the file, not to write to it. Any SuperVisor Call request for updating, writing, renaming, or removing will return the "Attempt to access protected file" error. If the OWNER password is part of the filespec when the file is opened, the system will permit all levels of access regardless of any USER password or protection level.

Prot	Effect
NONE	You cannot access the file. This PROT is used for system files.
EXEC	You can only run the program file.
READ	You can read the file.
UPDATE	You can write to an existing file without extending it.
WRITE	You can write to and extend the file.
RENAME	You can change the name/extension of the file.
REMOVE	You can delete the file from the disk.
FULL	You can change the protection level and passwords of the file.
Note:	Each level grants the access listed above it.

Figure 6-1 Access protection levels

Passwords are assigned to files in one of two ways. If a password is part of the filespec when the file is first created with the @INIT SuperVisor Call function, then that password will become both the OWNER and USER passwords. The protection level will be FULL but since both password fields are in use, the password must be entered for any access to the file. The second method of applying password protection is to use the ATTRIB library command. This command allows you to change both passwords and protection level - assuming you have the access authority based on the file's existing protection.

A password can be composed of nothing but blanks. This is in effect, no password at all since the entry of NOTHING is interpreted as a blank field and thus will grant access according to the level associated with the password field. For instance, if the OWNER password field is blank, the file has no protection whatsoever even if the USER password field is non-blank because a filespec without a password entry will match the blank OWNER password thus granting full access. It is important for the OWNER password to be non-blank if the file is to be protected in any manner.

A common situation is to find the OWNER password kept private to those individual(s) either maintaining the application or responsible for the integrity of the file contents while providing a blank USER password with a protection level set to the minimum level of access needed by the user. For instance, if the user only needs to read a file, set the protection level to READ. This user can then read the file without having to bother with a password but that user cannot write to the file, cannot remove it from the disk, cannot rename the file, nor can the user change the protection level of the file. However, the maintainer can step in to deal with file maintenance at a higher level of access given the OWNER password.

Where use of a file needs to be restricted to an individual out of a group of individuals, then the USER password field should have a non-blank password that is distinct from the OWNER password. The access protection level is still kept to the minimum necessary for the user. This scenario will then permit that individual the minimum access to the file while excluding all others (unless, of course, the user shares his knowledge of the password with others).

It may be practical for any given installation to consider protecting all files to the minimum access level expected of them. Thus any file whose primary access is READ only would be protected accordingly. There will be less chance to inadvertently remove the file by mistake or mistakenly write to it - a common error when dealing with applications that frequently prompt the user for the entry of file specifications.

A high level language permits you the opportunity of indicating your access level in the language syntax. For example, BASIC requires you to specify whether a sequentially accessed file is to be INPUT or OUTPUT corresponding to READ or WRITE. The operating system has no facility for identifying the maximum level of access desired for any particular opening of the file except through the passwords and access protection level.

6.2.3 Opening Files

Files opened with UPDATE or greater access are indicated as open in their directory entry record by the setting of a "file open bit". Any subsequent open attempt will result in a force to READ access protection and return the appropriate "File already open" error code. This is designed primarily for the use of shared access multiplexed disk drives where files are shared among a number of users. This arrangement will restrict the altering (but not reading) of file data to only one user at a time. It is therefore important for applications to CLOSE files as soon as the application is finished with the file access. It is also important for applications to trap the "File already open" error and take appropriate action. Realize that files protected to READ only, may be opened by multiple users and still be opened for updating by the maintainer providing the proper OWNER password is provided. The importance of maintaining proper levels of file protection through the use of passwords and protected access levels should not be taken lightly.

For the convenience of applications that access files only for reading, a facility for forcing the file access to READ only when a file is opened has been provided in the DOS. This facility will inhibit the "file open bit" and set the File Control Block access permission to READ (providing that the access permission level granted according to the password entered was READ or greater). Under this linkage, it is not necessary to close the file when you are finished accessing it as no directory updating will be done. Of course if you want the system to recover the filespec and place it into the FCB , you will have to close the file. Check the discussion covering the FORCE-to-READ flag (bit-0 of the SFLAG$) in the @FLAGS SuperVisor Call. Note that once the FORCE-to-READ flag has been set, the next @OPEN or @INIT SuperVisor Call request will automatically reset the bit after satisfying the request.

When a file is opened, the system needs to be told where the disk file I/O buffer is located. This buffer is used to transfer a full sector of data to and from the disk. The system also needs to be told what Logical Record Length (LRL) is to be used while the file is open. If the LRL at open time differs from the LRL of the file as noted in the directory, the OPEN routine will return an "LRL open fault" error code BUT THE FILE WILL STILL BE PROPERLY OPENED ACCORDING TO THE LRL PASSED IN THE OPEN REQUEST. The error code is your indication that a different LRL is being used. If the LRL is 256, then the system does not block and deblock the data records and will expect that all data to I/O will be using the disk file I/O buffer. If the LRL is in the range <1-255>, then the disk file I/O buffer is used only for transferring full sectors to and from the disk. Say, for example, a file has 200-byte records, the second record of the file is partially contained in the first sector and partially contained in the second sector. The file is said to SPAN two sectors. This requires a separate buffer to hold the record data while the system uses the disk file I/O buffer for the transfer of the sector. The program then will specify a USER RECORD buffer (UREC) that will be used by the system to transfer the data records to and from the disk file I/O buffer on each I/O request. Thus, whenever a file record spans two sectors, the system will have the necessary buffering regions to fully block and deblock the record. Note that the arrangement of separate disk file I/O buffers for each file provides greater flexibility for accessing multiple files coincidentally.

To illustrate the linkage necessary to open an existing file, we will be referencing an 80-byte record length file with the specification, BULKLOAD/DAT:2. The file has an OWNER password, blank USER password with protection level of WRITE. The filespec has been placed into the File Control Block as shown in figure 6-2. Note that the filespec is left justified and is terminated with an ETX (X'03') character. The ETX is automatically

placed as the terminator when a file specification is parsed into the FCB by the @FSPEC SuperVisor Call function. A carriage RETURN (X'0D') could equally be used if your program is completely controlling the placement of the filespec into the FCB. The remainder of the FCB contents is inconsequential as anything past the ETX or RETURN is completely ignored by the OPEN process.

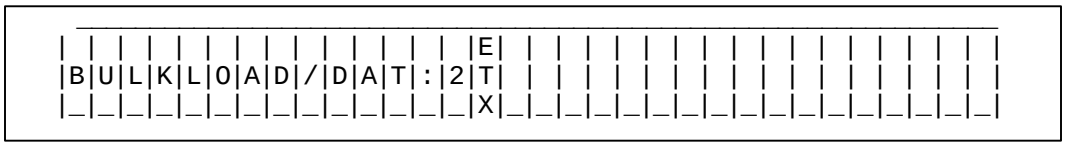

Figure 6-2 FCB prior to OPEN

Once the FCB is filled with the filespec, we can open the file using linkage such as this:

```
            LD    HL,FILEBUF   ;Point to the disk file I/O buffer
            LD    DE,FCB1      ;Point to the File Control Block
            LD    B,80         ;Specify the Logical Record Length
            LD    A,@OPEN      ;Identify the SVC
            RST   40           ;Invoke the SVC
            JP    NZ,IOERR     ;Transfer on a returned error
            .
            .
            .
            ORG   $<-8+1<8     ;Set PC to page origin
FILEBUF     DS    256          ;Reserve space for file buffer
```

Many programs are coded so that the data areas are placed at the end of the program. As you become adept at file handling, you will discover that accessing file buffers that are placed at a page boundary is not only easier, but sometimes more efficient depending on your specific use of the buffer. The "ORG" pseudo-OP in the above routine serves the purpose of establishing the program counter at a page origin. This provides for the access of each byte in the buffer by indexing the low-order byte of a 16-bit register pair.

If you are going to create a new file, all that needs to be changed in the routine illustrated is to replace the "LD A,@OPEN" with "LD A,@INIT". Specifics on the protocol of @INIT are located in chapter 7. The @INIT SuperVisor Call can also be used to open an existing file. Your use of either @OPEN or @INIT is dependent on the purpose of the file. If your application is going to write a file that can be either existing or new, then @INIT is the choice. @INIT will inform you as to whether it located an existing file or created a new one (the carry flag is set if a new file is created). This information may be useful to your application. If it is a requirement that the file be existing, then @OPEN should be used.

If it is mandatory that the file NOT be existing, then the system provides a few capabilities to support this requirement. You can first @OPEN the file. If the file is successfully opened, then you know that the file is existing and can take the appropriate action. If the file did not open successfully, you should check the error code returned by the system to verify that it returned a "File not found" error as other errors may not imply the non-existence of the file [for instance, the LRL provided with @OPEN may be different than that stored in the directory entry giving an "LRL open fault" error]. Another interesting technique for detecting the existence of a file is to attempt to RENAME it using the same name. This can be done with the @RENAME SuperVisor Call by copying the filespec into a second FCB for use as the "new" but identical name. The @RENAME routine will always first check the existence of the file before determining RENAME permission and verifying that the new name differs from the old. If @RENAME returns a "File not found" error, you will know that the file does not exist. If the file

does exist, @RENAME should return either an "Illegal access to protected file" error (if you do not have RENAME permission) or an "Illegal file name" error due to the duplicate name. The @RENAME method uses slightly less system overhead and thus will execute faster. It also will not attempt to set the directory's "file open bit" thereby performing one less directory write.

6.2.4 Closing Files

The reverse operation of opening a file, be it @INIT or @OPEN, is the CLOSE operation. Remember that files opened with UPDATE or greater access must be closed in order to update the directory entry record. The updating process will change the modification date and set the MODification flag bit if any writing has occurred. The updating process also alters the end-of-file information if a sequentially accessed file has been either extended or shortened. Finally, the updating process resets the "file open bit". The CLOSE operation uses the information that the system has been maintaining in the FCB . Thus, you close a file simply by passing the FCB pointer to the SuperVisor Call as follows:

```
        LD    DE,FCB1      ;Point to the open File Control Block
        LD    A,@CLOSE     ;Identify the SVC
        RST   40           ;Invoke the SVC
        JP    NZ,IOERR     ;Transfer on a returned error
```

6.2.5 Miscellaneous File Control

Before we leave the topic of file control let's address some lesser used control requests. First we have the removal of a file. The system's REMOVE library command can delete a file from the disk when at DOS Ready or command level. You could also remove a file by passing a "REMOVE filespec" command line to the system via the @CMNDR SuperVisor Call request. If we consider the DOS command level to be the highest level, then the lowest level is via assembly language SVCs. The SVC method of file removal requires that the file first be opened. The reason for this requirement is based on the overlay structure of the system. The file control routines are resident in system overlays rather than in the memory resident portion of the system like the file access routines. It so happens that the routines to open a file are in an overlay (SYS2) different from the overlay containing the routines to remove a file (SYS10). Since the system has no provision for system overlays to invoke functions in other overlays, your application program "supervises" the two functions of opening and removal. This linkage is as follows:

```
        LD    DE,FCB1      ;Point to the FCB holding the filespec
        LD    A,@OPEN      ;Identify the SVC
        RST   40           ;Invoke the SVC
        JR    Z,OPENOK     ;Continue if no open error
        CP    42           ;Check on "LRL open fault"
        JR    NZ,RMVERR    ;Error if anything else
OPENOK  LD    A,@REMOV     ;Identify the SVC
        RST   40           ;Invoke the SVC
RMVERR  JP    NZ,IOERR     ;Transfer on a returned error
```

Notice that we did not need to reference a disk file I/O buffer since no I/O was going to be performed (why waste the three bytes for the instruction?). Also, since we are going to ignore "LRL open fault" errors, there is no need to put an LRL value into register B.

When the system removes a file, it first deallocates the space taken up by the file by resetting the appropriate bits in the Granule Allocation Table. In the deallocation process, all of the file's extended directory entry (FXDE) records are zeroed and their corresponding Directory Entry Code (DEC) positions freed for future use. Then the hash code is removed from the file's primary directory entry (FPDE) record DEC position of the Hash Index Table used by the file. Finally, the ACTIVE bit of the FPDE record is reset. The rest of the information in the FPDE is left unaltered. It is thus possible to "unremove" a file that had a maximum of four extents by activating its FPDE, restoring

the hash code in the proper DEC, and reallocating the space in the GAT provided the space has not been reused by some other file.

Two other lesser used SuperVisor Call requests are @LOAD and @RUN. It's more important to explain their use rather than illustrate their use. Most programs are stand-alone programs. They are totally self contained in terms of the program code. When programs get large or when programs must access large amounts of data in memory, it may be necessary to segment the program into two or more sub-programs. Depending on the functions performed by the program, this segmentation can take two forms. Where the functions can be divided into separately chained processes (such as a language compiler that can separate parsing from code generation), one sub-process can RUN the other sub-process. Where the functions of the program must be divided up and controlled by a supervising sub-program, the available memory can be divided into a resident sub-program region and an overlay sub-program region - similar to the overlay structure of the operating system. Thus the supervisor will LOAD each overlay as required and transfer control into the loaded sub-program.

When an executing program needs to either @RUN or @LOAD another program, there is one point that is most important to understand. Although the @RUN and @LOAD functions utilize the system file buffer, they require a user File Control Block. Also, either request will return to the calling program if an error is detected in the loading of the program file. Therefore, it is essential that the program being loaded must not overwrite either the FCB used to access it nor the error handling routines following the @LOAD or @RUN linkage requests! To ignore this situation is to invite disaster to come knocking at your door.

6.3 ACCESSING DISK FILES

The concept of accessing disk files conveys the idea of transferring data to and from the disk file. Before the file can be accessed, it must be opened as discussed in the preceding section. Once a file has been opened, any of a number of file access SuperVisor Call requests can be made depending on the specific nature of the desired function.

It may be useful to understand exactly how the operating system's file access routines react in order to satisfy our request. Let us say, for example, that we want to read the 100th record of the BULKLOAD/DAT file. The 100th record has a record number of 99 since records are numbered starting from record 0. We establish the linkage to accomplish this as follows:

```
LD    DE,FCB1      ;Point to the opened FCB
LD    BC,99        ;Specify the record number
LD    A,@POSN      ;Identify the positioning SVC
RST   40           ;Invoke the SVC function
JP    NZ,IOERR     ;Transfer on error
LD    HL,UREC1     ;Point to our record buffer
LD    A,@READ      ;Identify the SVC request
RST   40           ;Invoke the SVC
JP    NZ,IOERR     ;Transfer on an error return
```

The first part of the linkage positions the FCB so that the next I/O operation will deal with record number 99 - the 100th record. After a successful positioning, the record will be read into the record buffer. This is a very brief explanation. Let's examine in detail, the sequence of steps actually executed by the file positioning routine, @POSN.

First, since the file's LRL is less than 256, the 100th record must be deblocked from the sector containing the record (or sectors if by chance the 100th record spans two sectors). By multiplying the record number (99) by the logical record length (80), the value 7920 is obtained. This represents the first byte of the record in OFFSET position 240 of relative sector number 30.

Next, it would be very useful if the disk file I/O buffer already contained relative sector number 30. The Next Record Number (NRN) is the relative sector number. However, before we can make use of the NRN, we have to make sure that the buffer currently contains the sector identified by the NRN. To determine this, @POSN first checks the "buffer current" flag. If the buffer contains the sector identified by the NRN, @POSN then checks if the NRN and the sector number needed to satisfy the position of record 99 are in agreement. If the file buffer currently holds the needed sector, it immediately transfers to a routine which checks on end-of-file conditions and returns to the caller.

If the buffer does not contain the needed sector, then the NRN must be changed to the relative sector needed. But first the system must check to see if it has to write the buffer contents back to the disk file. This determination is based on whether the buffer is current and contains changed data which has not yet been written to disk (perhaps the result of a previous record written that did not span two disk sectors and thus did not require any physical writing).

When the @READ request is passed to the system, again the system must first check if the disk file I/O buffer contains any data which is updated but not yet written to disk. The @READ routine does not know that an @POSN request immediately preceded it. Then, since the LRL is less than 256, the @READ routine passes a series of character read requests for as many characters as that identified by the LRL. Each character is placed into a consecutive location of the user record buffer, which in this case is UREC1. The character read requests are virtually identical to those requested by an @GET SuperVisor Call request as both are performed by the same routines. Finally, the system adjusts the Next Record Number and OFFSET pointers so that the next @READ references the next consecutive record.

We now have to look at what happens when a character read is requested. First, the system checks to see if the end of the file has been reached so it can return the "End of file encountered" error code. Next, it checks to see if the byte is contained in the current disk file buffer (i.e. if the buffer is current). If the buffer is not current, the sector identified by the NRN must be read from disk. Before the system even wants to calculate what sector that represents, it has to ensure that the requesting user has READ permission to the file. This it can do by examining the access level stored in the FCB.

When it concludes that proper access is available, it proceeds to calculate the logical cylinder and sector that the file's NRN relative sector represents. If you thought the process was complex up to this point, hang on to your hat! The relative sector (remember number 30?) is converted to a relative granule number and relative sector offset in that granule. In this case, we will assume that the file is stored on a 5-1/4" floppy disk ette formatted in double density with six sectors per granule. The system obtains the sectors per granule data from the Drive Control Table (DCT) for the drive containing the file. This means that the relative granule needed is granule number 5 (30 divided by 6). Since the remainder of the calculation is zero, the relative sector offset in that granule is number 0 which is the first sector of the granule.

The system then examines the EXTENT fields of the FCB to determine what extent contains data covering relative granule number 5. To do this, the system uses the cumulative granule figures contained in the EXTENT fields. After determining that the granule is in one of the existing extents, the system can calculate the needed cylinder and relative sector in that cylinder by the following process. A few numbers may help this explanation. Say the file has two extents. The first extent contains three granules (numbered 0-2), while the second extent contains twelve granules (numbered 3-14) and starts on the third granule of cylinder 25. Figure 6-3 illustrates part of the second extent by cylinder and granule. First subtract off the number of granules contained in all extents previous to the desired extent and add the result to the starting granule number of the extent (5-3+2=4). Next, divide that result by the number of granules per cylinder derived from DCT information and keep the remainder (4/3=1 remainder 1). The

result is the relative cylinder from the starting cylinder while the remainder is the relative granule offset in that cylinder. If we now add the relative cylinder (1) to the starting cylinder (25), we compute the desired granule is in cylinder 26. Furthermore, the relative granule offset is granule number 1 (the second granule). Thus, by using the starting cylinder and granule of the extent, the relative cylinder and sector numbers for the starting sector of the needed granule are obtained. Finally, the granule offset is used to get the sector number of the desired sector. Since the granule offset is zero, our needed sector is the first sector of granule 1 which is sector 6. Thus, cylinder 26, sector 6 is passed by the system to the disk driver which reads that sector into the file buffer. Are you still with us?

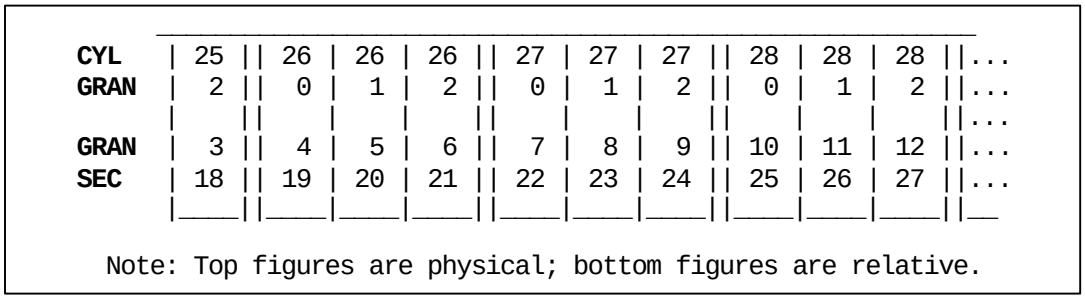

Figure 6-3 Illustration of 2nd extent for BULKLOAD/DAT

If, by chance, the system cannot find the desired granule in any of the extent fields of the FCB, it must go back to the directory using the DEC and DRIVE fields of the FCB and see if the granule is actually part of the file. This would only happen if the file had more than four extents or the access was extending the file (at which point additional space would be allocated).

Upon recognition of the complexity of the preceding discussion, it will severely limit your desire to control your own file allocations. The DOS does the job well; however, the system must entertain sufficient overhead in order to access the proper disk sector and dynamically allocate additional file space as required. Also, the system must inhibit the requesting program from violating protection levels.

Most of the file access SuperVisor Call requests are self-explanatory and their use is evident from the descriptions contained in chapter 7, INTERFACING VIA SUPERVISORY CALLS. An important point worth remembering is that the system will automatically advance the record pointers (NRN and OFFSET) on each @READ and each @WRITE request AFTER PERFORMING THE OPERATION so that the next record accessed is consecutively sequential to the one just accessed. This provides sequential I/O without need of @POSN calls. What we would like to discuss here is some suggested uses for these file access SVCs.

6.3.1 Specific Access Requests

The @GET and @PUT requests are fundamentally useful when the program is to be device independent. By using character I/O, the specification can be either a devspec or filespec. Of course if a device was opened, all of the other file access routines would return a "File not open" error code so you may want to restrict the access to @PUT and/or @GET or use bit-7 of the FCB as an indicator of file versus device and take the appropriate action.

The function of @BKSP is to backspace one record based on the LRL. When a disk file is accessed via @PUT and @GET, it is usually opened with an LRL of 256. However, if you try to perform a character backspace, the system will backspace a full sector. The easy way around this is to temporarily change the LRL in the FCB to 1 prior to issuing the @BKSP then restoring the LRL after the @BKSP call. The following code illustrates this method:

```
        LD      DE,FCBX         ;Point to the open FCB
```

```
LD    HL,FCBX+9    ;Point to the LRL field
LD    B,(HL)       ;P/u the current LRL
LD    (HL),1       ;  & reset to LRL=1
LD    A,@BKSP      ;Identify the SVC request
RST   40           ;Invoke the SVC
LD    (HL),B       ;Reset to original LRL
JP    NZ,IOERR     ;Transfer on error
```

If you want to add sequential data to the end of an existing file, you will need to position to the end of the file after it is opened. Use the @PEOF SuperVisor Call request for this purpose. The SVC will return an "End of file encountered" error if the request is successful. Any other error code indicates a malfunction. This is one of the few system requests that return an error code upon success so you should be careful when you use it.

The @RREAD request is useful when reading nested files. Nested files are those where you are accessing each consecutively but not coincidentally. In this case, the same disk file I/O buffer can be used for each file. When you switch from one file to another, issue a @RREAD so that the system reloads the buffer with the sector that was being accessed for the last record read or for the last character obtained from @GET . The @RREAD request will force a rereading of the sector identified by the NRN provided that the LRL is either 1-255 or the file was accessed via @PUT or @GET. What do you do if you were using LRL=256 and @READ requests while maintaining your own offset pointer. All you need do in this case is to decrement the NRN and issue another @READ. For example, the PRO-CREATE editor assembler available from MISOSYS uses sector I/O for reading source files. PRO-CREATE maintains its own offset pointer as it extracts lines of code from the disk buffer. When it detects the "*GET filespec" request for including a nested file, it saves the current FCB in a save area and then opens the requested file using the same file buffer. When the end of the second file is reached, PRO-CREATE restores the saved FCB of the original file and executes the following code:

```
LD    DE,FCB       ;Point to the opened FCB
LD    HL,(FCB+10)  ;Obtain the current NRN,
DEC   HL           ;  decrement by one
LD    (FCB+10),HL  ;  and update the FCB
LD    A,@READ      ;Identify the SVC function
RST   40           ;Invoke the SVC
JP    NZ,IOERR     ;Transfer on error
```

The @RWRIT SuperVisor Call request would be used where you want to read a full sector (LRL=256) into the disk file I/O buffer, alter it directly in the buffer, then immediately write that buffer back to disk. The @RWRIT will force the NRN that was automatically advanced by the @READ request to be decremented by one so that it repoints to the sector corresponding to the buffer contents. It then performs the requests necessary to write the buffer to disk. Note that @RWRIT is not to be used when the LRL is not equal to 256 as this SuperVisor Call does not reference the user record buffer.

The @WEOF SuperVisor Call request allows you to update the end-of-file (EOF) information in the directory while still keeping the file in an open state. Obviously, a similar function can be performed with an @CLOSE followed by an @OPEN; however, complications can prevail with a CLOSE-OPEN combination. Remember that the close operation restores the filespec to the FCB but cannot reclaim the password. Therefore, if the FCB was referencing a password protected file, the subsequent OPEN will fail unless you had saved the original filespec somewhere in the program and restuffed the FCB prior to the second OPEN request. Also, the CLOSE-OPEN combination updates the MOD flag and date as required, and checks to see if it can deallocate any unused file space. This takes time. If all you want to do is to update the EOF, use the @WEOF function.

One last function that can be performed by the file access routines is the allocation of disk space to a file. A file can be pre-allocated by the CREATE library command but that

also inhibits any deallocation of unused space. The following routine will allocate file space without any restriction on deallocation to a file opened with LRL equal to 256. Register pair DE is expected to be pointing to the file's FCB. The file's size is passed in register pair BC as the number of 256-byte records. A successful allocation will be indicated by the setting of the Z flag.

```
WRERN   LD      A,B             ;If space = 0, don't
        OR      C               ;  do any allocation
        RET     Z
        DEC     BC              ;Adjust for 0 offset
        LD      A,@POSN         ;Position to the "size"

        RST     40
        LD      A,@WRITE        ;Write a dummy sector
        RST     40
         JR      NZ,WRERN1      ;Branch on error
         LD      A,@REW         ;Now rewind the file
         RST     40
         LD      HL,0           ;Set ERN record to 0
         LD      (FCB1+12),HL
         RET
WRERN1  CP      27              ;Disk Full?
         RET     NZ             ;Back on some other error
         LD      A,@REMOV       ;Remove what can't fit
         RST     40
         LD      A,27           ;Back with error code
         OR      A              ;  and NZ flag
         RET
```

Examine the functions of the file access routines listed in chapter 7. They will relate the scope of access permitted by the operating system. More complex levels of access such as ISAM, or random access of variable length records can be supported by building appropriate routines from the provided record I/O and character I/O routines. The following section will provide details on each field of the File Control Block. Most applications will not have to bother with the contents of the FCB. If you feel the need, go to it.

6.4 The FILE CONTROL BLOCK (FCB)

The File Control Block (FCB) is a 32-byte region that is used by the system to interface with a file that has been "opened" for access . Its contents are extremely dynamic. As records are written to or read from the disk file, specific fields in the FCB are modified. It is extremely important that during the time period that a file is open, you avoid changing the contents of the FCB unless you are sure that its alteration will in no way effect the integrity of the file.

The FCB initially contains the specification of the file that is to be opened for access. Upon a successful "open", the system will replace the specification with data derived from the file's directory entry. The file specification (without any password field) will be returned to the FCB when the file is closed. The information contained in each field of the FCB is as follows:

6.4.1 TYPE code of the control block - <Byte 0>

This byte contains certain attributes of the control block. It correlates to the TYPE byte of the Device Control Block, especially in light of the fact that both the DCB and the FCB can be associated with a device specification (the FCB by the nature of a ROUTE to a file). The TYPE byte uses each bit as a flag per the following specifications:

Bit 7 If set to a "1", it will indicate that the file is in an open condition; if set to a "0", the file is assumed closed. This bit can be tested to determine the "open" or "closed" status of an FCB and is used by the

operating system for such a purpose. The system's device I/O handler also makes use of this bit to determine the necessity for disk file character I/O.

Bit 6 This bit will be set to a "1" if the file was opened with UPDATE or greater access. It indicates to the CLOSE routine that the application has the authority to reset the "file open bit" in the directory entry record for the respective file. The CLOSE routine will not update the directory entry of a file without this bit being set in the FCB.

Bit 5 This bit indicates that the opened file is a Partitioned Data Set. The system will set this bit when the file is opened if it detects the presence of the PDS attribute in the directory entry of the file (DIR+0, bit 5).

Bit 4 This bit is reserved for future use by the DOS.

Bit 3 This bit is reserved for future use by the DOS.

Bit 2 This bit will be set to a "1" if any WRITE operation is performed by the system on this file while it is open. The bit is used specifically to update the MOD flag in the file's directory entry record when the file is closed.

Bit 1 This bit is reserved for future use by the DOS.

Bit 0 This bit is reserved for future use by the DOS.

6.4.2 Input/Output Status - <Byte 1>

This byte contains I/O buffer status flag bits used in read/write operations by the system. The STATUS byte uses each bit as a flag per the following specifications:

Bit 7 If this bit is set to a "1", it indicates that I/O operations will be either record operations of logical record length (LRL) less than 256 (1-255) or character I/O. If set to a "0", only full sector operations or character I/O will be performed. If you are going to utilize only full sector I/O, system overhead is reduced by specifying the LRL at open time to be 0 (indicating 256). An LRL of other than 256 will set bit 7 to a "1" when the file is first opened.

Bit 6 When a file's records have been accessed randomly rather than (or in addition to) sequentially, the system must be prohibited from the altering the Ending-Record-Number (ERN) unless the file is extended beyond its current ERN. This bit is used for that status. If set to a "1", it indicates that the ERN is to be set to the Next-Record-Number (NRN) only if the NRN exceeds the current value of ERN. Whenever the position SVC (@POSN) is invoked, it will automatically set bit 6. If bit 6 is set to a "0", then ERN in the FCB will be updated on every WRITE operation.

Bit 5 It is always necessary for the system to know whether or not the file buffer contains the current disk sector as specified by the NRN. This bit is maintained for that use. If it is set to a "0", then the disk file buffer contains the current sector denoted by NRN. If it is set to a "1", then the file buffer does not contain the current sector. When a sector is read into the disk buffer, the system will reset this bit to show that the buffer currently holds the disk sector specified by the NRN. During character I/O, the first character GET request will force the system to transfer a full disk sector into the file buffer and reset the "buffer current" bit. Bit 5 is automatically set when the character in the last byte of the buffer has been transferred to the application in the GET requests. This will then indicate that the buffer is not current so that the next GET will force a read of the next sector.

Bit 4 During file I/O, an application may request a repositioning of the file's NRN-OFFSET pointer. This may be requested via an @BKSP, @POSN, @REWIND,

@SKIP, @PEOF, or @SEEKSC SuperVisor Call. It is important for the system to know whether or not the disk file buffer has been changed since it was read from the file. If the buffer has been altered, it is necessary to write the buffer back to the file prior to any movement of the file pointer. This flag conveys such status. If it is set to a "1", it indicates that the buffer contents have been changed since the buffer was read from the file. If it is set to a "0", the indication is that the buffer has not been modified. The system will set this bit whenever a WRITE operation is performed on the buffer by either a PUT or the write of a record (of LRL < 256). The bit is reset by the system when the buffer is physically written to the disk via the @WRSEC SuperVisor Call request.

Bit 3 The normal method to reflect changes in a file's directory entry record data is to update the directory entry only when the file is closed. Thus, the FCB contains all of the information pertinent to the modifications. This keeps the directory accesses to a minimum and results in faster file throughput. However, it is important to note that if the system crashes after extensive file updating (specifically where the file has been extended), the added information will be unrecoverable without manual corrections to the file's directory entry record. It is possible to force the system to always update the directory whenever the system extends the file by writing another sector. Unattended operation may utilize this extra measure of file protection. It is specified by appending an exclamation mark "!" to the end of a file specification when the filespec is requested at open time. This bit will then be set by the system. It is used to specify that the directory record is to be updated every time that the NRN exceeds the ERN.

Bits 2-0 These bits will contain the access protection level as retrieved from the directory entry record of the file when the file is first opened. The specific bit pattern will be adjusted to the protection level granted according to the password (OWNER vs USER) entered at file open time.

6.4.3 PDS Member Origin Offset - <Byte 2>

When a Partitioned Data Set (PDS) has been opened for individual member access (a sector origin member), the PDS linkage routines will adjust the EOF contained in the FCB to be the logical EOF of the member. The member origin offset is the number of relative sectors between the logical ERN of the member and the first relative sector of the member. This byte will contain that forward offset so that the linkage routines may be able to calculate the logical beginning of the member. The calculation is required for linkage to all SuperVisor Calls that reference file positioning forward of the NRN (@BKSP, @REWIND, @POSN, @SEEKSC).

6.4.4 Disk File Buffer Pointer - <Bytes 3-4>

This is a pointer to the disk file buffer that is used for all disk I/O associated with the file. The pointer is a 16-bit address stored in normal low-order - high-order format. This pointer is the buffer address specified in register pair HL at open time.

6.4.5 Next Record Number Byte Offset - <Byte 5>

When a file is accessed with either character I/O or record I/O of Logical Record Length less than 256, requests for I/O may not necessarily require the transfer of a physical sector from/to the disk. Therefore, the system needs a pointer to the byte position within the buffer that is to be used for the next I/O operation. This field contains that position - it is termed an OFFSET within the sector pointed to by the NRN. If this offset is a zero value, then the next byte to be transferred during an I/O operation is dependent on whether or not the buffer contains the current sector as noted by FCB+1, bit 5. The system automatically maintains this OFFSET byte during record and character I/O. If your application is performing full sector I/O for writing data while it is maintaining its own character buffering, then it is important for it to maintain this byte when the file is closed if the true end-of-file offset is not at a sector boundary. Remember, this offset is a pointer to the next available buffer position and not to the

position where the last character is placed. For instance, after writing three bytes into positions 0, 1, and 2 of the buffer, the offset must be incremented to "3" since the next available buffer position is byte 3.

6.4.6 Logical Drive Number - <Byte 6>
This contains the logical drive number in binary of the drive containing the file. It is absolutely essential that this byte be left undisturbed. It is used by the system's file access routines to obtain the logical disk drive number that physical I/O is to reference. It, and the Directory Entry Code contained in FCB+7 are the only links to the directory information for the file. Since the operating system supports a maximmum of eight logical drives, the logical drive number is contained in a 3-bit field. The remaining bits are reserved for future use in large disk segmentation.

Bits 7-3 This field is reserved by the DOS for future use.

Bits 2-0 This field contains the logical drive number where the file is stored.

6.4.7 Directory Entry Code - <Byte 7>
This field contains the Directory Entry Code (DEC) which points to the file's primary directory entry. This code is the relative position in the Hash Index Table where the hash code for the file's directory entry appears. Whenever the system needs to access the directory for the open file, it must use both this DEC and the logical DRIVE to uniquely specify the proper directory record. Do not tamper with this byte. It may be interesting to note that the device name, which uniquely identifies a device, and the DEC-DRIVE, which uniquely identifies a file, are contained in the same fields of their respective control blocks.

6.4.8 Ending Record Number Byte Offset - <Byte 8>
This field contains the byte offset in the Ending Record Number which points to one byte past the end-of-file. This byte is similar to FCB+5 except it pertains to the ERN rather than the NRN. If a file has been extended during the time it was open, then the NRN byte offset and NRN become the new ERN byte offset and ERN when the file is closed.

6.4.9 Logical Record Length - <Byte 9>
This field contains the logical record length in effect when the file was opened. This may not be the same LRL that exists in the directory. The directory LRL is generated at the file creation and will never change unless another file is cloned to it.

6.4.10 Next Record Number <Bytes 10-11>
This field contains the Next-Record-Number (NRN), which is a pointer to the relative sector for the next I/O operation. When a file is opened, NRN is set to zero indicating a pointer to the beginning of the file. Each physical sector I/O advances NRN by one. An @REWIND SuperVisor Call request will reset the NRN to zero.

6.4.11 Ending Record Number <Bytes 12-13>
This field is a pointer to the last sector of the file regardless of whether the sector is a full sector (i.e. all bytes occupied and EOF-OFFSET has a zero value) or a partial sector (i.e. EOF-OFFSET is not equal to zero). In a null file (one with no records), ERN will be equal to zero. If one sector had been written, ERN would be equal to one.

6.4.12 Starting Extent - <Bytes 14-15>
This field contains the same information as the first extent of the directory. This represents the starting cylinder of the file (FCB+14) and the starting relative granule within the starting cylinder (FCB+15). FCB+15 also contains the number of contiguous granules allocated in the extent. This can always be used as a pointer to the beginning of the file referenced by the FCB. During any file access, this field will be searched first to see if it contains the granule which stores the physical sector that is being referenced.

6.4.13 Extent Quad 1 - <Bytes 16-19>
The QUAD is a 4-byte field that contains the granule allocation information for one extent of the file as well as the total quantity of granules contained in the file logically prior to this extent. Relative bytes zero and one contain the cumulative number of granules allocated to the file up to but not including the extent referenced by this field. This quantity is calculated by the system by adding up all the number of contiguous granules allocated in previous extents. Relative byte two contains the starting cylinder of this extent. Relative byte three contains the starting relative granule for the extent and the number of contiguous granules. Relative bytes two and three are obtained directly from an extent field of the directory entry record. Figure 6-4 illustrates the Extent Quad.

6.4.14 Extent Quad 2 - <Bytes 20-23>
This field contains information similar to the first Extent Quad but for a second extent of the file.

6.4.15 Extent Quad 3 - <Bytes 24-27>
This field contains information similar to the first Extent Quad but for a third extent of the file.

6.4.16 Extent Quad 4 - <Bytes 28-31>
This field contains information similar to the first Extent Quad but for a forth extent of the file.

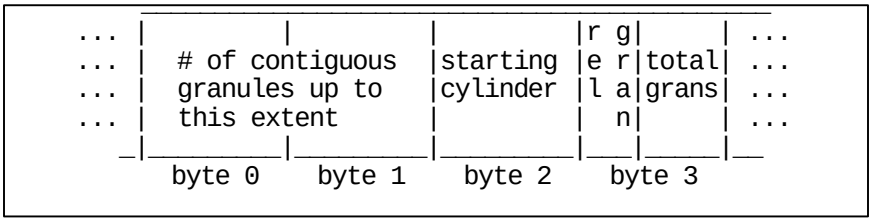

Figure 6-4 An FCB Extent Quad

The File Control Block contains information on only five extents at any one time - one of which is always the first extent of the file (that which is placed into STARTING EXTENT. When a file is first opened, data for the STARTING EXTENT is extracted from the first extent of the file's primary directory entry record (the FPDE). If the file has more than one extent, data for the EXTENT QUADS is calculated for each additional extent that contains allocation information in the FPDE. This leaves, at a minimum, one EXTENT QUAD vacant.

Each time a record is accessed, the system determines if the record is located in the starting EXTENT. If not, then the system searches the extent QUADs. If the record is located in one of the QUADS, then the data contained in all QUADS to the left of the "desired" QUAD is shifted right by one QUAD and the data from the "desired" QUAD is placed in the first extent QUAD field. This action is undertaken so that extent QUADs that contain records recently accessed will be searched first. If a record in a file is accessed which is not contained in any FCB extent QUAD field, then the DOS must access the directory entries for the file and locate that extent which contains the needed granule (and hence the needed record). Once the extent is located, the data in extent QUAD fields 1-3 will be shifted to occupy fields 2-4 and the new data will be placed into extent QUAD field 1. If the desired record cannot be located in any extent of the file, the system will attempt to allocate additional space necessary to position the record.

Although the operating system can handle a file of any number of extents, it is wise to keep the total number of extents small. If the file has more than five extents, additional directory accessing must be done to locate the extent containing the desired

record. If a file has more than four extents, then it will occupy more than one directory entry record and thereby reduce the number of file slots available. The most efficient file is one with a single extent although the file can be at most 32 granules in size. The number of extents can be reduced by copying the file to a diskette containing a great deal of free space.

7. Interfacing via SuperVisor Calls

7.1 SUPERVISOR CALL LINKAGE

This chapter discusses specific linkage necessary to communicate with the operating system for service requests at the assembly language level. Requests for system resources are accomplished via SuperVisor Calls (SVCs). The following sections describe each SVC and the register contents passed to and from the system.

The DOS does not affect the contents of the Z-80's alternate registers (AF', BC', DE', and HL'). Where the DOS makes use of index registers IX and IY, it will save them prior to their use and restore them when that use is completed. The exception, of course, is where IX and/or IY are used to pass information to or from the DOS.

Each SVC specifies what registers are altered by the system. The AF register will always be altered. Most SVCs incorporate return codes. Where applicable, the return code is passed in the accumulator and the Z-flag status is indicative of an error or success [Z = success, NZ = error]. Some SVCs use only the state of the Z-flag to indicate a pass/fail situation. The return code convention is specified under the linkage shown for each SVC.

SuperVisor Calls utilize a number from 0 to 127. Numbers from 128 to 255 are not interpreted as SVCs but are used internally by the DOS for other system overlay invocations. The SVC number is placed in the accumulator once the registers particular to the SVC are set up and control is passed to the operating system by issuing a RST 40 (RST 28H) Z-80 instruction.

7.1.1 Adding or Changing SVC Entries

Some programmers may find it useful to alter the performance of existing SuperVisor Calls to suit unique situations. A program may even be written that could utilize additional SVCs. Four SVC slots [numbers 124-127] have been provided for application programs. An examination of the following SVC tables will reveal a good handful of SVC numbers that have not yet been assigned by Logical Systems. Caution is to be observed in utilizing any of these reserved slots since you may find your program unusable with a future release of the operating system. [Remember that four RST instructions: RST 8, RST 16, RST 24, and RST 32 are available for use by application software.]

In any event, be it modification of the vector for an existing SVC or the addition of your own into a "user" SVC, the interface is simple. The SVC table is always (and will always) be origined at the start of a RAM page. The page address (i.e. the high-order byte of the SVC table) can be obtained from the system via the FLAGS pointer returned by the @FLAGS SVC. Since the low order byte starts out with 0 for SVC-00, you can locate the exact address for the SVC vector by multiplying the SVC number by two, loading the result into the low order byte of a register pair (say L), then loading the high order byte of that register pair (say H) with the SVC table base address (FLAGS$+26). This will then index the low order byte of the SVC vector. The SVC vectors are stored in standard low-high order.

7.2 PROGRAM ENTRY AND EXIT CONDITIONS

When the operating system executes a program either from DOS Ready or via an SVC (@CMNDI , @CMNDR, or @RUN), certain conditions prevail. These conditions relate to the register contents and the stack location. The useful register contents are as follows:

BC — Contains a pointer to the start of the command line. This is useful for those applications desiring to know what program name caused their invocation (as in the command-line arguments applicable to C programs).

DE — Contains a pointer to the File Control Block used to open the program file being run. This may prove useful to access the program file as data since the file is already in an open condition (the PRO-PaDS utility from MISOSYS makes use of this condition).

HL — Contains a pointer to the first non-blank character on the command line which terminated the parsing of the name entered in order to execute the program. This pointer should be used if you are going to parse command-line file specifications using @FSPEC or parameters via @PARAM.

If the program was executed from DOS Ready or via @CMNDI, the stack pointer (SP) will point to the system stack which has approximately 150 bytes of storage space. If the program was executed via @RUN or @CMNDR, then the stack pointer contains whatever was established by the invoking program. In any event, the top of the stack will contain the return address to the module which is invoking the program, be it another program or the system.

If you are going to switch stacks, you should be aware that the system's task processor requires possibly 40 bytes of stack space. The exact amount will depend on what tasks are active. Release 6.0.0 of the DOS also has a restriction that limits the stack to reside below X'F400'. If you are going to use the @BANK request to toggle memory banks, then the stack must reside below X'7FFE'.

When your program terminates, it should load register pair HL with a return code. If the program terminates without error, use a return code of 0. If the termination is due to a DOS I/O error or other error being returned by an SVC as noted in the error dictionary, load that error number into HL. For all other error conditions, the suggested procedure is to load a -1 (X'FFFF') into register pair HL. After loading HL, you can either issue a RET instruction or issue an @EXIT SuperVisor Call. Note that the RET exit method mandates that you maintain the integrity of the stack pointer so that it is pointing to a valid return address. You may want to establish exit code that reloads the stack pointer with the SP contents that you saved when first executing the program. Thus, the SP will always be correct for an RET. An @EXIT termination will always restore control to the operating system even if the program was invoked via an @CMNDR. Therefore, if you suspect that your program will be invokable from another program, you should use the RET method for program termination.

7.3 SUPERVISOR CALLS LISTED ALPHABETICALLY

Name	Svc #	Purpose
@ABORT	SVC-21	Abnormal program exit
@ADTSK	SVC-29	Add a task process
@BANK	SVC-102	RAM bank switching
@BKSP	SVC-61	File record backspace
@BREAK	SVC-103	Establish <BREAK> vector
@CHNIO	SVC-20	Device chain character I/O
@CKBRKC	SVC-106	Check for a keyboard BREAK
@CKDRV	SVC-33	Check disk drive availability (& log)
@CKEOF	SVC-62	Check for file's end-of-file (EOF)
@CKTSK	SVC-28	Check task slot availability
@CLOSE	SVC-60	Close an open disk file
@CLS	SVC-105	Clear the Video screen
@CMNDI	SVC-24	Interpret and execute a command
@CMNDR	SVC-25	Execute a command and return
@CTL	SVC-05	Control a device chain
@DATE	SVC-18	Obtain system date
@DCINIT	SVC-42	Initialize a disk controller

@DCRES	SVC-43	Reset a disk controller
@DCSTAT	SVC-40	Test disk controller status
@DEBUG	SVC-27	Enter system DEBUG package
@DECHEX	SVC-96	Convert decimal string to binary
@DIRRD	SVC-87	Read a DEC's directory record
@DIRWR	SVC-88	Write a DEC's directory record
@DIV16	SVC-94	16-bit by 8-bit unsigned division
@DIV8	SVC-93	8-bit by 8-bit unsigned division
@DODIR	SVC-34	Obtain or display directory data
@DSP	SVC-02	Character output to *DO (video display)
@DSPLY	SVC-10	Line output to *DO (video display)
@ERROR	SVC-26	Post an error message
@EXIT	SVC-22	Exit program with return code
@FEXT	SVC-79	Fetch a default file extension
@FLAGS$	SVC-101	Obtain system flags pointer
@FNAME	SVC-80	Obtain filespec given DEC and drive
@FSPEC	SVC-78	Fetch and parse a file specification
@GET	SVC-03	Character input from a device/file
@GTDCB	SVC-82	Obtain DCB pointer given devspec
@GTDCT	SVC-81	Obtain DCT pointer given drive
@GTMOD	SVC-83	Obtain entry point given module name
@HDFMT	SVC-52	Pass "format device" to controller
@HEX16	SVC-99	Convert 16-bit binary to ASCII hex
@HEX8	SVC-98	Convert 8-bit binary to ASCII hex
@HEXDEC	SVC-97	Convert 16-bit binary to ASCII decimal
@HIGH$	SVC-100	Obtain or alter HIGH$/LOW$
@INIT	SVC-58	Open a new or existing file
@IPL	SVC-00	Reboot the system
@KBD	SVC-08	Scan the *KI device
@KEY	SVC-01	Obtain a character from the *KI device
@KEYIN	SVC-09	Obtain a line of characters from *KI (or JCL)
@KLTSK	SVC-32	Remove task assignment during execution
@LOAD	SVC-76	Load a program file
@LOC	SVC-63	Return file's current record number
@LOF	SVC-64	Return file's ending record number
@LOGER	SVC-11	Send a message to the Job Log (*JL)
@LOGOT	SVC-12	Display and log a message (*DO and *JL)
@MSG	SVC-13	Send a message line to a device
@MUL16	SVC-91	16-bit by 8-bit into 24-bit multiplication
@MUL8	SVC-90	8-bit by 8-bit into 8-bit multiplication
@OPEN	SVC-59	Open an existing file
@PARAM	SVC-17	Parse a command line of parameters
@PAUSE	SVC-16	Delay execution for a time period
@PEOF	SVC-65	Position to the end of a file
@POSN	SVC-66	Position to a designated record of a file
@PRINT	SVC-14	Send a message line to *PR device
@PRT	SVC-06	Send a character to *PR device
@PUT	SVC-04	Send a character to a device/file
@RAMDIR	SVC-35	Obtain directory information
@RDHDR	SVC-48	Read ID field (where supported)
@RDSEC	SVC-49	Read a disk sector
@RDSSC	SVC-85	Read a disk's directory sector
@RDTRK	SVC-51	Read a disk track (where supported)
@READ	SVC-67	Read a file record
@REMOV	SVC-57	Remove a file from disk

@RENAM	SVC-56	Rename a file on disk
@REW	SVC-68	Rewind a file to its beginning
@RMTSK	SVC-30	Remove a task assignment
@RPTSK	SVC-31	Replace a task assignment during execution
@RREAD	SVC-69	Reread the last sector read
@RSLCT	SVC-47	Reselect a busy drive until available
@RSTOR	SVC-44	Restore a drive to cylinder 0
@RUN	SVC-77	Run a program given its filespec
@RWRIT	SVC-70	Rewrite the last sector written
@SEEK	SVC-46	Seek to a disk cylinder
@SEEKSC	SVC-71	Seek a record of a file
@SKIP	SVC-72	Skip the next record of a file
@SLCT	SVC-41	Select a disk drive
@SOUND	SVC-104	Activate hardware sound generation
@STEPI	SVC-45	Issue track step-in to controller
@TIME	SVC-19	Obtain the system time
@VDCTL	SVC-15	Various video control functions
@VER	SVC-73	Write then verify a file record
@VRSEC	SVC-50	Verify the readability of a disk sector
@WEOF	SVC-74	Directory update a file's end-of-file
@WHERE	SVC-07	Resolve run-time address
@WRITE	SVC-75	Write a file record
@WRSEC	SVC-53	Write a disk sector
@WRSSC	SVC-54	Write a disk directory sector
@WRTRK	SVC-55	Write a disk track (format data)

7.4 SUPERVISOR CALLS LISTED NUMERICALLY

Name	Svc #	Purpose
@IPL	SVC-00	Reboot the system
@KEY	SVC-01	Obtain a character from the *KI device
@DSP	SVC-02	Character output to *DO (video display)
@GET	SVC-03	Character input from a device/file
@PUT	SVC-04	Send a character to a device/file
@CTL	SVC-05	Control a device chain
@PRT	SVC-06	Send a character to *PR device
@WHERE	SVC-07	Resolve run-time address
@KBD	SVC-08	Scan the *KI device
@KEYIN	SVC-09	Obtain a line of characters from *KI (or JCL)
@DSPLY	SVC-10	Line output to *DO (video display)
@LOGER	SVC-11	Send a message to the Job Log (*JL)
@LOGOT	SVC-12	Display and log a message (*DO and *JL)
@MSG	SVC-13	Send a message line to a device
@PRINT	SVC-14	Send a message line to *PR device
@VDCTL	SVC-15	Various video control functions
@PAUSE	SVC-16	Delay execution for a time period
@PARAM	SVC-17	Parse a command line of parameters
@DATE	SVC-18	Obtain system date
@TIME	SVC-19	Obtain the system time
@CHNIO	SVC-20	Device chain character I/O
@ABORT	SVC-21	Abnormal program exit
@EXIT	SVC-22	Exit program with return code
rsvd	SVC-23	reserved
@CMNDI	SVC-24	Interpret and execute a command

@CMNDR	SVC-25	Execute a command and return
@ERROR	SVC-26	Post an error message
@DEBUG	SVC-27	Enter system DEBUG package
@CKTSK	SVC-28	Check task slot availability
@ADTSK	SVC-29	Add a task process
@RMTSK	SVC-30	Remove a task assignment
@RPTSK	SVC-31	Replace a task assignment during execution
@KLTSK	SVC-32	Remove task assignment during execution
@CKDRV	SVC-33	Check disk drive availability (& log)
@DODIR	SVC-34	Obtain or display directory data
@RAMDIR	SVC-35	Obtain directory information
rsvd	SVC-36	reserved
rsvd	SVC-37	reserved
rsvd	SVC-38	reserved
rsvd	SVC-39	reserved
@DCSTAT	SVC-40	Test disk controller status
@SLCT	SVC-41	Select a disk drive
@DCINIT	SVC-42	Initialize a disk controller
@DCRES	SVC-43	Reset a disk controller
@RSTOR	SVC-44	Restore a drive to cylinder 0
@STEPI	SVC-45	Issue track step-in to controller
@SEEK	SVC-46	Seek to a disk cylinder
@RSLCT	SVC-47	Reselect a busy drive until available
@RDHDR	SVC-48	Read ID field (where supported)
@RDSEC	SVC-49	Read a disk sector
@VRSEC	SVC-50	Verify the readability of a disk sector
@RDTRK	SVC-51	Read a disk track (where supported)
@HDFMT	SVC-52	Pass "format device" to controller
@WRSEC	SVC-53	Write a disk sector
@WRSSC	SVC-54	Write a disk directory sector
@WRTRK	SVC-55	Write a disk track (format data)
@RENAM	SVC-56	Rename a file on disk
@REMOV	SVC-57	Remove a file from disk
@INIT	SVC-58	Open a new or existing file
@OPEN	SVC-59	Open an existing file
@CLOSE	SVC-60	Close an open disk file
@BKSP	SVC-61	File record backspace
@CKEOF	SVC-62	Check for file's end-of-file (EOF)
@LOC	SVC-63	Return file's current record number
@LOF	SVC-64	Return file's ending record number
@PEOF	SVC-65	Position to the end of a file
@POSN	SVC-66	Position to a designated record of a file
@READ	SVC-67	Read a file record
@REW	SVC-68	Rewind a file to its beginning
@RREAD	SVC-69	Reread the last sector read
@RWRIT	SVC-70	Rewrite the last sector written
@SEEKSC	SVC-71	Seek a record of a file
@SKIP	SVC-72	Skip the next record of a file
@VER	SVC-73	Write then verify a file record
@WEOF	SVC-74	Directory update a file's end-of-file
@WRITE	SVC-75	Write a file record
@LOAD	SVC-76	Load a program file
@RUN	SVC-77	Run a program given its filespec
@FSPEC	SVC-78	Fetch and parse a file specification
@FEXT	SVC-79	Fetch a default file extension

@FNAME	SVC-80	Obtain filespec given DEC and drive
@GTDCT	SVC-81	Obtain DCT pointer given drive
@GTDCB	SVC-82	Obtain DCB pointer given devspec
@GTMOD	SVC-83	Obtain entry point given module name
rsvd	SVC-84	reserved
@RDSSC	SVC-85	Read a disk's directory sector
rsvd	SVC-86	reserved
@DIRRD	SVC-87	Read a DEC's directory record
@DIRWR	SVC-88	Write a DEC's directory record
rsvd	SVC-89	reserved
@MUL8	SVC-90	8-bit by 8-bit into 8-bit multiplication
@MUL16	SVC-91	16-bit by 8-bit into 24-bit multiplication
rsvd	SVC-92	reserved
@DIV8	SVC-93	8-bit by 8-bit unsigned division
@DIV16	SVC-94	16-bit by 8-bit unsigned division
rsvd	SVC-95	reserved
@DECHEX	SVC-96	Convert decimal string to binary
@HEXDEC	SVC-97	Convert 16-bit binary to ASCII decimal
@HEX8	SVC-98	Convert 8-bit binary to ASCII hex
@HEX16	SVC-99	Convert 16-bit binary to ASCII hex
@HIGH$	SVC-100	Obtain or alter HIGH$/LOW$
@FLAGS$	SVC-101	Obtain system flags pointer
@BANK	SVC-102	RAM bank switching
@BREAK	SVC-103	Establish <BREAK> vector
@SOUND	SVC-104	Activate hardware sound generation
@CLS	SVC-105	Check for keyboard BREAK
@CKBRKC	SVC-106	Clear the Video screen
rsvd	SVC-107	reserved
rsvd	SVC-108	reserved
rsvd	SVC-109	reserved
rsvd	SVC-110	reserved
rsvd	SVC-111	reserved
rsvd	SVC-112	reserved
rsvd	SVC-113	reserved
rsvd	SVC-114	reserved
rsvd	SVC-115	reserved
rsvd	SVC-116	reserved
rsvd	SVC-117	reserved
rsvd	SVC-118	reserved
rsvd	SVC-119	reserved
rsvd	SVC-120	reserved for ARCNET use
rsvd	SVC-121	reserved for ARCNET use
rsvd	SVC-122	reserved for ARCNET use
rsvd	SVC-123	reserved for ARCNET use
rsvd	SVC-124	Available for user programs
rsvd	SVC-125	Available for user programs
rsvd	SVC-126	Available for user programs
rsvd	SVC-127	Available for user programs

7.5 SUPERVISOR CALLS LISTED BY FUNCTION GROUP

7.5.1 Character I/O

Name	Svc #	Purpose
@KEY	SVC-01	Obtain a character from the *KI device
@DSP	SVC-02	Character output to *DO (video display)
@GET	SVC-03	Character input from a device/file
@PUT	SVC-04	Send a character to a device/file
@CTL	SVC-05	Control a device chain
@PRT	SVC-06	Send a character to *PR device
@KBD	SVC-08	Scan the *KI device
@VDCTL	SVC-15	Peek/Poke video by row,column
@CHNIO	SVC-20	Device chain character I/O

7.5.2 Line I/O

Name	Svc #	Purpose
@KEYIN	SVC-09	Obtain a line of characters from *KI (or JCL)
@DSPLY	SVC-10	Line output to *DO (video display)
@LOGER	SVC-11	Send a message to the Job Log (*JL)
@LOGOT	SVC-12	Display and log a message (*DO and *JL)
@MSG	SVC-13	Send a message line to a device
@PRINT	SVC-14	Send a message line to *PR device
@VDCTL	SVC-15	Video RAM <-> User RAM

7.5.3 Data Conversion

Name	Svc #	Purpose
@PARAM	SVC-17	Parse a command line of parameters
@MUL8	SVC-90	8-bit by 8-bit into 8-bit multiplication
@MUL16	SVC-91	16-bit by 8-bit into 24-bit multiplication
@DIV8	SVC-93	8-bit by 8-bit unsigned division
@DIV16	SVC-94	16-bit by 8-bit unsigned division
@DECHEX	SVC-96	Convert decimal string to binary
@HEXDEC	SVC-97	Convert 16-bit binary to ASCII decimal
@HEX8	SVC-98	Convert 8-bit binary to ASCII hex
@HEX16	SVC-99	Convert 16-bit binary to ASCII hex

7.5.4 Disk Controller Communications

Name	Svc #	Purpose
@DCSTAT	SVC-40	Test disk controller status
@SLCT	SVC-41	Select a disk drive
@DCINIT	SVC-42	Initialize a disk controller
@DCRES	SVC-43	Reset a disk controller
@RSTOR	SVC-44	Restore a drive to cylinder 0
@STEPI	SVC-45	Issue track step-in to controller
@SEEK	SVC-46	Seek to a disk cylinder
@RSLCT	SVC-47	Reselect a busy drive until available
@RDHDR	SVC-48	Read ID field (where supported)
@RDSEC	SVC-49	Read a disk sector
@VRSEC	SVC-50	Verify the readability of a disk sector
@RDTRK	SVC-51	Read a disk track (where supported)
@HDFMT	SVC-52	Pass "format device" to controller

@WRSEC	SVC-53	Write a disk sector
@WRSSC	SVC-54	Write a disk directory sector
@WRTRK	SVC-55	Write a disk track (format data)

7.5.5 File Access

Name	Svc #	Purpose
@GET	SVC-03	Character input from a device/file
@PUT	SVC-04	Send a character to a device/file
@BKSP	SVC-61	File record backspace
@CKEOF	SVC-62	Check for file's end-of-file (EOF)
@LOC	SVC-63	Return file's current record number
@LOF	SVC-64	Return file's ending record number
@PEOF	SVC-65	Position to the end of a file
@POSN	SVC-66	Position to a designated record of a file
@READ	SVC-67	Read a file record
@REW	SVC-68	Rewind a file to its beginning
@RREAD	SVC-69	Reread the last sector read
@RWRIT	SVC-70	Rewrite the last sector written
@SEEKSC	SVC-71	Seek a record of a file
@SKIP	SVC-72	Skip the next record of a file
@VER	SVC-73	Write then verify a file record
@WEOF	SVC-74	Directory update a file's end-of-file
@WRITE	SVC-75	Write a file record

7.5.6 File Control

Name	Svc #	Purpose
@RENAM	SVC-56	Rename a file on disk
@REMOV	SVC-57	Remove a file from disk
@INIT	SVC-58	Open a new or existing file
@OPEN	SVC-59	Open an existing file
@CLOSE	SVC-60	Close an open disk file
@LOAD	SVC-76	Load a program file
@RUN	SVC-77	Run a program given its filespec
@FSPEC	SVC-78	Fetch and parse a file specification
@FEXT	SVC-79	Fetch a default file extension
@FNAME	SVC-80	Obtain filespec given DEC and drive

7.5.7 System Control

Name	Svc #	Purpose
@IPL	SVC-00	Reboot the system
@VDCTL	SVC-15	Various video control functions
@PAUSE	SVC-16	Delay execution for a time period
@ABORT	SVC-21	Abnormal program exit
@EXIT	SVC-22	Exit program with return code
@CMNDI	SVC-24	Interpret and execute a command
@CMNDR	SVC-25	Execute a command and return
@ERROR	SVC-26	Post an error message
@DEBUG	SVC-27	Enter system DEBUG package
@HIGH$	SVC-100	Obtain or alter HIGH$/LOW$
@FLAGS$	SVC-101	Obtain system flags pointer
@BANK	SVC-102	RAM bank switching
@BREAK	SVC-103	Establish <BREAK> vector

@CKBRKC	SVC-106	Check for keyboard BREAK
@CLS	SVC-105	Clear the Video screen

7.5.8 System Data

Name	Svc #	Purpose
@VDCTL	SVC-15	Obtain the video cursor position
@DATE	SVC-18	Obtain system date
@TIME	SVC-19	Obtain the system time
@CKDRV	SVC-33	Check disk drive availability (& log)
@DODIR	SVC-34	Obtain or display directory data
@RAMDIR	SVC-35	Obtain directory information
@GTDCT	SVC-81	Obtain DCT pointer given drive
@GTDCB	SVC-82	Obtain DCB pointer given devspec
@GTMOD	SVC-83	Obtain entry point given module name
@RDSSC	SVC-85	Read a disk's directory sector
@DIRRD	SVC-87	Read a DEC's directory record
@DIRWR	SVC-88	Write a DEC's directory record
@HIGH$	SVC-100	Obtain or alter HIGH$/LOW$
@FLAGS$	SVC-101	Obtain system flags pointer

7.5.9 Task Process Control

Name	Svc #	Purpose
@CKTSK	SVC-28	Check task slot availability
@ADTSK	SVC-29	Add a task process
@RMTSK	SVC-30	Remove a task assignment
@RPTSK	SVC-31	Replace a task assignment during execution
@KLTSK	SVC-32	Remove task assignment during execution

7.5.10 Miscellaneous

Name	Svc #	Purpose
@WHERE	SVC-07	Resolve run-time address
@PARAM	SVC-17	Parse a command line of parameters

7.6 SUPERVISOR CALL DETAILS

7.6.1 @ABORT SVC-21

This SVC will cause an abnormal program exit and return to DOS. Any JCL execution in progress will cease. @ABORT functions by loading the HL register pair with a value of X'FFFF' and passing control to @EXIT.

Registers Affected: Not applicable.

7.6.2 @ADTSK SVC-29

This SVC will add an interrupt level task pointed to by your Task Control Block (TCB) to the real time clock task processor Task Control Block Vector Table. The task slot can be 0-11; however, some slots are already assigned to certain functions in the DOS. The SVC, @CKTSK, can be used to test for slot availability. Slot assignments 0-7 are low priority tasks, slots 8-10 are medium priority tasks, and slot 11 is a high priority task. Note: The TCB is a pointer to a word of RAM containing the address of the task driver entry point and not to the location of your task driver. Detailed interfacing on background tasks is in Chapter 8, the Appendix, on TASK PROCESSOR.

Registers Affected: AF, HL.

Entry

DE	Pointer to your Task Control Block (TCB).
C	Contains the task slot assignment number.

7.6.3 @BANK SVC-102

This SVC deals with memory bank use. The top half of the first 64K block is bank 0, and the second 64K is banks 1 and 2. DOS supports a total of 8 memory banks of 32K each (numbered 0-7). See Chapter 8, the Appendix, on BANK SWITCHING for programming details and illustrations. Internally, the DOS makes use of three storage bytes: the BAR contains the bit-image of Bank Available RAM; the BUR contains the bit-image of Bank Used RAM; and LBANK$ contains the number (0-7) of the currently resident bank. These storage areas are not directly accessible to the programmer but are referenced through the SVC functions. In the interfacing register protocol identified below, register-B passes a function code.

Registers Affected: AF, BC, [HL if a transfer is requested].

Bank Request [optional transfer]

Entry:

B	0
C	Bank number (0-7). Optionally set bit-7 to transfer to the address specified in register pair HL.
HL	Optional address to transfer to in the new bank. This option is selected by setting bit-7 of register-C.

Exit:

B	Returns a 0.
C	Returns the previously resident bank number (0-7). If a transfer has been specified (via bit-7 set), bit-7 will remain set.
A	Returns any error code if NZ condition.
NZ	Bank not there.

Bank Release

Entry:

B	1; Reset bank in C.
C	Bank number.

Bank Availability Test

Entry:
B 2; Test if bank C in use.
C Bank number.

Exit:
NZ In use.

Bank Reservation Request

Entry:
B 3; Set bank in C.
C Bank number.

Exit:
NZ Already in use.

What Bank is Resident

Entry:
B 4; Return current installed bank.

Exit:
A Returns the bank number (0-7) of the currrently resident bank.

Note: The coding of the @BANK routine will not return an error if you try to reset a Bank Used RAM (BUR) that is "in-use" because it is not installed. The way in which bank-reset should be performed is to know which one you were using and made in-use. Note that even though @BANK permits you to reset a non-existant bank, if you try to enable it, you will get an error since the enabling routine will not permit the selection of a bank not installed.

7.6.4 @BKSP SVC-61

This SVC will perform a backspace of one logical record in the referenced file.

Registers Affected: AF.

Entry:
DE A pointer to the FCB of the file to backspace.

Exit:
A Error return code.
Z Set if the operation was successful.

7.6.5 @BREAK SVC-103

This SVC is used to establish or reset a <BREAK> key vector. The <BREAK> condition is observed as a background interrupt task. Once activated, a <BREAK> will pass control to your vectored routine providing the current program counter is above the resident DOS and below HIGH$.

Registers Affected: AF.

Entry:
HL Address of your break vector.
HL X'0000' to restore to system break handler.

Note: @EXIT in SYS1 automatically restores BREAK to the system handler. This is not done for @CMNDR. Also, don't forget that if DEBUG is enabled, then entry to DEBUG takes precedence over the BREAK (of course, even though DEBUG has been enabled, if you only have EXEC access, DEBUG is effectively disabled).

Your break handling routine will need to debounce the BREAK key and obviously deal with the stack pointer (since the stack could be anywhere depending on when and where the break was detected). Something of the following is suitable:

```
ENTRY  LD      (STKSAV),SP ;Save the stack pointer
       PUSH    HL
       LD      HL,MYBRK    ;Point to your BREAK handler
       @@BREAK             ;Set up "MYBRK" as break entry
       .
       .
MYBRK  DI                  ;Don't permit further BREAKs
       LD      B,80H       ;Wait for fingers to get off
       @@PAUSE             ;  of the BREAK key
       LD      SP,$-$      ;P/u the orig stack pointer
STKSAV EQU     $-2
       EI                  ;Interrupts back on
       what ever you want
       RET                 ;To what invoked the program
```

7.6.6 @CHNIO SVC-20

This SVC is used to pass control to the next module in a device chain. Its use is restricted to device filters. Detailed information on the use of @CHNIO will be found in chapter 3, DEVICE INPUT/OUTPUT INTERFACING.

Registers Affected: Depends on the filter modules chained.

Entry:

IX Contains a pointer to the Device Control Block assigned to the filter module. This is recovered from the MODDCB field located in the module header. Note: IX should be saved before loading and restored upon return from @CHNIO.
B Contains the I/O direction code (GET=1, PUT=2, CTL=4).
C Contains the output character for PUT or GET.

7.6.7 @CKBRKC SVC-106

This SVC was installed effective release 6.2.0. It checks to see if the BREAK key has been pressed. It also clears the BREAK bit of the KFLAG$ if a break condition is detected.

Registers Affected: AF.

Exit:

Z BREAK was not detected.
NZ BREAK was detected. SVC returns only when BREAK is released.

7.6.8 @CKDRV SVC-33

This routine will check a drive reference to ensure that the drive is in the system and a formatted diskette is in place. It will also "log" the disk as far as density, number of sides, and directory cylinder so that the Drive Control Table information is correct.

Registers Affected: AF.

Entry:

C	Logical drive number

Exit:

Z	If drive is ready.
NZ	If drive is not ready
A	Indeterminate and irrelevant.
CF	Set if disk is write protected.

@CKEOF SVC-62

This SVC will check for the end-of-file at the current logical record number.

Registers Affected: AF.

Entry:

DE	A pointer to the FCB for the file to check.

Exit:

A	Error return code. If not X'1C', then some other error has been encountered. It is necessary to get NZ and A=X'1C' for the proper EOF indication.
Z	Set if not at the end of file and no error is encountered.

7.6.9 @CKTSK SVC-28

This SVC will check if the referenced task slot (0-11) is available for use. See Chapter 8, the Appendix, on TASK PROCESSOR for further details.

Registers Affected: AF, HL.

Entry:

C => The task slot number (0-11).

Exit:

Z	Indicates that the task slot is available.
NZ	Indicates that the task slot is in use.

7.6.10 @CLOSE SVC-60

This SVC will close a file or device. If a file is closed, the directory is updated which is essential. All files that have been opened with UPDATE access or greater must be closed.

Registers Affected: AF.

Entry:

DE A pointer to your File or Device Control Block.

Exit:

A Will contain any error return code.
Z Set if no error was encountered.

7.6.11 @CLS SVC-105

This SVC was installed in release 6.2.0. It will clear the video screen via an @DSP of HOME and CLEAR-TO-END-OF-FRAME.

Registers Affected: AF

Exit:

Z Set if no error was encountered, otherwise reset (i.e. NZ).
A Contains the error code under an NZ condition.

7.6.12 @CMNDI SVC-24

This SVC passes control to the command interpreter. Your command string will be invoked just as if it was entered in response to a "DOS Ready".

Registers Affected: Not applicable..

Entry:

HL A pointer to the start of a line buffer containing your command string terminated with an <ENTER> (X'0D'). Only the first 79 characters of your command string will be used.

7.6.13 @CMNDR SVC-25

This SVC will execute a command similarly to @CMNDI; however, upon completion of the command, control will be returned to the address following the @CMNDR invocation. It is necessary for all executing commands to maintain the stack pointer and exit via an RET instruction after loading HL with the return code. It is possible to limit the execution to DOS LIBrary commands by setting bit-4 of the CFLAG$ (see @FLAGS SVC).

Registers Affected: Dependent on command executed.

Entry:

HL A pointer to the start of a line buffer containing your command string terminated with an <ENTER> (X'0D'). Only the first 79 characters of your command string will be used.

Exit:

HL Will contain the return code of the executing command.

7.6.14 @CTL SVC-05

This SVC will output a control byte to a logical device. If a device control block is referenced, the TYPE byte must permit CTL operation. The file access routines will ignore @CTL requests and provide a "no error" return code. Control protocol is very unique to each device. See chapter 3, DEVICE INPUT/OUTPUT INTERFACING, for additional information.

Registers Affected: AF.

Entry:
DE A pointer to the DCB or FCB to control output.
C Byte to output.

7.6.15 @DATE SVC-18

Get today's date in display format (XX/XX/XX). The SVC can also be used to obtain the address of the binary storage for the system date. This may be useful for hardware clock add-ons.

Registers Affected: AF, BC, DE.

Entry:
HL Buffer area to receive date string.

Exit:
DE Returns a pointer to the 5-byte binary date storage:
DATE+0 = year in excess 1900;
DATE+1 = day (1-31);
DATE+2 = month (1-12);
DATE+3 = bits 0-7 of the year's day;
DATE+4 = holds bit-8 of the year's day in bit-0, the day of the week (1-7) in bits 1-3, and bit-7 is set for a leap year.

7.6.16 @DCINIT SVC-42

This SVC passes a function 2 to a disk driver. It is commonly used for disk controller initializing. See chapter 4 for additional information.

Registers Affected: AF [Note: DOS saves BC, IY; drivers should save any other registers they use].

Entry:
C Logical drive number (0-7).

Exit:
A Error return code, if any.
Z Set if the operation was successful.

7.6.17 @DCRES SVC-43

This SVC passes a function 3 to a disk driver. It is commonly used for disk controller resetting. See chapter 4 for additional information.

Registers Affected: AF [Note: DOS saves BC, IY; drivers should save any other registers they use].

Entry:
C Logical drive number (0-7).

Exit:
A Error return code, if any.
Z Set if the operation was successful.

7.6.18 @DCSTAT SVC-40

This SVC passes a function 0 to a disk driver. It is commonly used for testing the status of a logical drive. A disk driver should return with no error on function 0. Thus, if a particular drive is disabled, the system will return an error-32 to the calling program. Chapter 4 has more information.

Registers Affected: AF [Note: DOS saves BC, IY; drivers should save any other registers they use].

Entry:
C Logical drive number (0-7).

Exit:
A Error return code, if any.
Z Set if the operation was successful.

7.6.19 @DEBUG SVC-27

This SVC will force the system to enter the DEBUGging package.

Registers Affected: None except those changed by the user.

7.6.20 @DECHEX SVC-96

This SVC performs the conversion of a decimal string of digits <0-9> to their binary value in a 16-bit field. Overflow is not trapped. The conversion stops on the first digit found not to be in the range <0-9>. The linkage is:

Registers Affected: AF, BC, HL.

Entry:
HL => A pointer to your decimal string.

Exit:
BC Returns the resultant 16-bit binary value of "string".
HL Points to 1st non-decimal digit.
Z-flag is indeterminate

7.6.21 @DIRRD SVC-87

This SVC will read a directory sector containing the directory entry for a specified Directory Entry Code (DEC). The sector will be written to the system buffer, SBUFF$, and the register pair HL will point to the first byte of the directory entry specified by the DEC. Note that this is a method to recover the page address of the system's buffer by keeping register-H after an @DIRRD invocation. See the sections on HASH INDEX TABLE and DIRECTORY RECORD FORMAT for additional information.

Registers Affected: AF, HL.

Entry:
B Directory Entry Code of the file.
C Logical drive number (0-7).

Exit:
HL Points to the DEC's directory entry.
A Error return code, if any.
Z Set if no error is encountered.

7.6.22 @DIRWR SVC-88

This SVC will write the system buffer, SBUFF$, back to the disk directory sector that contains the directory entry of the DEC specified in the calling linkage. See the sections on HASH INDEX TABLE and DIRECTORY RECORD FORMAT for additional information.

Registers Affected: AF, HL.

Entry:
B Directory Entry Code of the file.
C Logical drive number (0-7).

Exit:
A Error return code, if any.
Z set if no error.

7.6.23 @DIV16 SVC-94

This SVC will perform a division of a 16-bit unsigned integer by an 8-bit unsigned integer.

Registers Affected: AF, HL.

Entry:
HL Should contain the dividend value.
C Should contain the divisor value.

Exit:
HL Returns the resultant value.
A Returns the remainder value.

7.6.24 @DIV8 SVC-93

This SVC performs an 8-bit unsigned integer divide.

Registers Affected: AF, E.

Entry:
E Should contain the dividend value.
C Should contain the divisor value.

Exit:
A Returns the resultant value.
E Returns the remainder value.

7.6.25 @DODIR SVC-34

This SVC will capture selected directory information for the logical drive referenced in the SVC's invocation and either pass the information to your designated buffer or display formatted information on the *DO device. A function number is passed in register B to control the desired output.

Registers Affected: AF.

Display Filespecs

Entry:
B 0; Function to display the directory of visible files to *DO.
C The logical drive number (0-7) of the selection.

Directory to Buffer

Entry:
B 1; Function to stuff your buffer with directory information.
C The logical drive number (0-7) of the selection.
HL A pointer to your buffer. The data returned by @DODIR is the first 16-bytes of each directory record followed by the ERN. The buffer will be terminated by an X'FF'.

Display Filespecs Matching EXT

Entry:
B 2; Function to display the directory of visible files to *DO. The display is limited to files matching the given extension.
C The logical drive number (0-7) of the selection.
HL A pointer to a 3-character file extension. The use of a dollar sign in any position represents a global match.

Directory Matching EXT to Buffer

Entry:
B 3; Function to stuff your buffer with directory information. The data is limited to files matching the given extension.
C The logical drive number (0-7) of the selection.
HL A pointer to your buffer. This pointer is also interpreted to be a pointer to a 3-character file extension. The use of a dollar sign in any position represents a global match. Note that this function implies that the start of your buffer is stuffed with the file extension to be matched.

Obtain Free Space

Entry:
B 4; Function to stuff your buffer with free space information. The information passed will be DISK NAME and DISK DATE in positions 1-16; total space on the disk (in K) in positions 17-18; and FREE SPACE available (in K) in positions 19-20.
C The logical drive number (0-7) of the selection.
HL A pointer to your buffer.

7.6.26 @DSP SVC-02

This SVC will output a byte to the video display devspec *DO.

Registers Affected: AF, DE.

Entry:
C Byte to display

Exit:
Z Set if no error was encountered, otherwise reset (i.e. NZ).
A Contains the error code under an NZ condition.

7.6.27 @DSPLY SVC-10

This SVC will display a message line to the *DO device. The line must be terminated with either an <ENTER> (X'0D') or an ETX (X'03'). If an ETX terminates the line, the cursor will be positioned immediately after the last character displayed.

Registers Affected: AF, DE.

Entry:
HL points to the 1st byte of your message.

7.6.28 @ERROR SVC-26

This SVC will provide an entry to post an error message. @ERROR will normally terminate to the @ABORT SVC. If bit 7 of the error register is SET, the error message will be displayed and return will be made to the calling program. If bit 6 of the error register is reset, the complete error information shown below is displayed. If bit 6 is set, then only the "Error message string" [see Chapter 8, the Appendix, Error Message Dictionary] is displayed.

Registers Affected: AF [Note: not applicable if @ABORT option].

Entry:
C Error number with bits 6 and 7 optionally set.
DE Optional string buffer pointer used with CFLAG option.

It is possible to have @ERROR return the message string associated with the error by setting bit-7 of the CFLAG$ (see SVC-101). This can be useful if you want to control the positioning of the message. Also, in the case of compilers and interpreters, it can be useful to use this option as a means of providing greater flexibility to the application program.

```
*** Error code = xx, Returns to X'dddd'
<filespec, devicespec, or open FCB/DCB status>
Last SVC = nnn, Returned to X'rrrr'
```

7.6.29 @EXIT SVC-22

This is the normal SVC to perform a program exit and return to DOS. Alternatively, if your program maintains the integrity of the stack pointer, then a simple RET instruction will return to the system.

Registers Affected: Not applicable.

Entry:
HL Must be loaded with the return code (0 = no error).

7.6.30 @FEXT SVC-79

This SVC will set up a default file extension in the FCB if the file specification entered contains no extension.

Registers Affected: AF.

Entry:
DE A pointer to the File Control Block.
HL Pointer to the 3-character default extension which must be stored in upper case.

7.6.31 @FLAGS$ SVC-101

This SVC will return a pointer to the base of the flags table. The pointer is returned in register IY. The flag table is a table of 26 flags lettered A-Z. Certain additional system variables are indexed relative to this pointer. Once the pointer is obtained, each flag may be referenced relative to IY. For instance, if the SFLAG$ is needed, use "IY+'S'-'A'" to reference the storage address of the flag. The following presents the flag assignments available to the programmer:

Registers Affected: AF, IY.

Exit:
IY Returns the pointer to the base of the flag table.

AFLAG$

This "allocation" flag contains the starting cylinder number that is used by the system's file space allocation routine when searching for free space on disk media. The system defaults this value to cylinder 1.

CFLAG$

Bit 0 If set, then the system will not permit the change of HIGH$ via SVC -100. This flag is reset by @EXIT and @CMNDI. This function is useful for applications invoking system resources via @CMNDR while still wanting control of the entire memory region through HIGH$.

Bit 1 If set, @CMNDR is executing. This flag is reset by @EXIT and @CMNDI. Note that once an @CMNDR invocation is performed, the flag cannot be reset by the system until "exit" of the application has been made via @EXIT or @CMNDI.

Bit 2 If set, it indicates that the command interpreter in SYS1 is requesting the line input from the keyboard. This condition is important for keyboard filters that may change the resident system overlay. If SYS1 is resident and overwritten when bit-2 is set, you will crash the DOS upon passing control back to the keyboard driver unless SYS1 is restored.

Bit 3 If set, then the system is requesting execution from either the "SET" or "SYSTEM (DRIVER=" commands. This bit should be tested by drivers or filters upon installation to ensure that they are being installed by the proper system command rather than just by RUN or execution.

Bit 4 If set, then the @CMNDR SVC will only execute system LIB commands. Bear in mind that "RUN" will be invokable which could then be used to override the limitation.

Bit 5 If set, the SYSGEN library command will be inhibited. This may be useful to inhibit application environments from altering the boot initialization configuration.

Bit 6 If set, then @ERROR will not display any error message. This can be used to inhibit the posting of error messages by programs invoked from @CMNDR.

Bit 7 If set, then @ERROR will pass the error message to the buffer pointed to by register pair DE. See @ERROR for more data.

DFLAG$

Bit 0 Set to "1" if SPOOL is active

Bit 1 Set to "1" if TYPE AHEAD is to be active. Type-ahead can betoggled on/off via this bit.

Bit 2 If set, it indicates VERIFY (ON) has been set.

Bit 3 If set, it indicates that SYSTEM (SMOOTH) is active.

Bit 4 If set, then MemDisk is active.

Bit 5 If set, it indicates that FORMS is active.

Bit 6 If set, it indicates that KSM is active.

Bit 7 Set if printer supports block graphics for screen print.

EFLAG$

This flag byte is used to indicate the presence of an Extended Command Interpreter (ECI) program in the SYS13/SYS slot. A non-zero value indicates that the user's ECI be used to interpret the command line instead of the system's command interpreter. On entry to your ECI, bits 4-6 of this flag are imaged in the accumulator and are available for immediate test.

IFLAG$

This flag is used in international systems. Bit assignments are:

Bit 0 Set to indicate French.

Bit 1 Set to indicate German.

Bit 2 Set to indicate Swiss.

Bit 3 reserved

Bit 4 reserved

Bit 5 reserved

Bit 6 Special DMP mode on/off.

Bit 7 Set 7-bit ASCII mode on/off.

KFLAG$

Bit 0 Set to "1" if BREAK pressed (see KFLAG interfacing and the @CKBRKC SVC-105).

Bit 1 Set to "1" if PAUSE pressed (see KFLAG interfacing).

Bit 2 Set to "1" if ENTER pressed (see KFLAG interfacing).

Bit 3 Reserved by DOS.

Bit 4 Reserved by DOS.

Bit 5 Set to "1" if in CAPS lock mode of the keyboard.

Bit 6 Reserved by DOS.

Bit 7 Set to "1" if a character is in the type-ahead buffer.

LFLAG$

Bit 0 If set, FORMAT will not prompt for step rate.

Bit 1 reserved

Bit 2 reserved

Bit 3 reserved

Bit 4 If set, FLOPPY/DCT will inhibit the 8" query.

Bit 5 If set, FORMAT will not prompt for number of sides.

Bit 6 Reserved for Interrupt Mode 2 hardware.

Bit 7 Reserved for Interrupt Mode 2 hardware.

MFLAG$

This flag is machine specific. It is used to contain an image of a particular CPU port. For instance, on the TRS-80 Model 4, this is an image of the MODOUT port (X'EC').

NFLAG$

This "network" flag is used for control in network situations. The bits are assigned as follows:

Bit 0 If set, the "file-open" bit will be written to the directory when a file is opened with update or higher access.

Bit 1 reserved

Bit 2 reserved

Bit 3 reserved

Bit 4 reserved

Bit 5 reserved

Bit 6 Set if the system's task processor is in control. NOTE: do not execute an EI instruction within any driver or filter routine if this bit is set.

Bit 7 - reserved

OFLAG$

This flag is machine specific. It is used to contain an image of a particular CPU port - generally dealing with memory management. For instance, on the TRS-80 Model 4, this is an image of the OPREG port (84).

PFLAG$

This flag is assigned to printer operations. Bits are as follows:

Bit 0 - reserved

Bit 1 - reserved

Bit 2 - reserved

Bit 3 - reserved

Bit 4 - reserved

Bit 5 - reserved

Bit 6 - reserved

Bit 7 - Set to 1 if the SPOOLer is in a paused state.

SFLAG$

Bit 0 This is the FORCE-TO-READ flag. If set prior to issuing an @OPEN, then the system will not check for matching LRL nor will the system set the "file open bit" in the directory for the opened file. However, the file will be restricted to READ access (unless a lower access is detected during the open. This bit will be automatically reset by @OPEN.

Bit 1 This bit will be set by @OPEN if an EXEC-only file is opened and bit-2 of SFLAG$ is set. Under these conditions, @OPEN will change the access granted to READ so that @LOAD can load the file. Thus, the application (for instance BASIC) can load an EXEC-only file to be RUN while still detecting the EXEC protection status.

Bit 2 Set this bit to enable the loading of an EXEC-only file. This bit works in conjunction with bit-1.

Bit 3 Set to "1" if SYSTEM (FAST) has been established.

Bit 4 Set to "1" to disable the BREAK key.

Bit 5 Set to "1" if DO is in effect executing Job Control Language.

Bit 6 Set to "1" to force extended error messages. This is only practical in a debugging environment.

Bit 7 Set to "1" if DEBUG is to be turned on after the execution of the program just loaded for execution. The use is internal to the system. If DEBUG is active, the DOS will not enter DEBUG when running an EXEC-only program but will maintain the DEBUG status via this bit.

TFLAG$

This is the machine type flag. It's value indicates the computer model running the DOS. Some of the typical TRS-80 values are: 2 = model 2; 4 = model 4; 5 = model 4P; 12 = model 12; 16 = model 16.

UFLAG$

This is a user flag. It is available for whatever purpose you wish to make of it. It will remain unused by the system; however, the flag contents will be part of any SYSGEN configuration file.

VFLAG$

Bits 0-3 Are used in controlling the cursor blink rate.

Bit 4 If set, the clock will be displayed on the video screen.

Bit 5 This bit is used by the system to toggle the cursor state.

Bit 6 If set, the cursor is non-blinking; otherwise blinking.

Bit 7 Used by the system to suppress blinking while in the *DO driver to inhibit the blink task from changing state.

WFLAG$

This is a machine dependent flag commonly used to store an image of mode-1 interrupt masking. For instance, on the TRS-80 Model 4, it stores an image of the WRINTMASK register (E0).

OTHER DATA

The other system information accessible relative to the flags pointer is as follows:

FLAGS-47 contains the release number of the DOS (OSRLS$). For instance, OSRLS$ is X'10' for version/release 6.0.1 (see FLAGS+27 for the version).

FLAGS-1 contains the overlay entry number of the system overlay currently resident in the overlay region. The low-order four bits reference the overlay number (1-13).

FLAGS+26 contains a one-byte pointer to the memory page which contains the SVC vector table (SVCTAB). This is useful to hook into system routines by indexing into the proper SVCTAB position according to the SVC number. The SVCTAB is always located on a page boundary.

FLAGS+27 contains the version number of the DOS (OSVER$). For instance, OSVER$ is X'62' for version 6.2.x

FLAGS+28 through FLAGS+30 contain a jump vector for @ICNFG. See the Chapter 8, the Appendix, on @ICNFG interfacing for details on this vector.

FLAGS+31 through FLAGS+33 contain a jump vector for @KITSK. See Chapter 8, the Appendix, on @KITSK interfacing for details on this vector.

7.6.32 @FNAME SVC-80

This SVC will recover the file name and extension from the directory for the referenced directory code and drive. It is used by the system to recover the filespec when closing a file. Although @FNAME can be used for a "directory" function, @DODIR or @RAMDIR are better candidates for performing that function.

Registers Affected: AF.

Entry:

DE	Buffer to receive file name/ext
B	DEC of file desired
C	drive number of drive containing the file

7.6.33 @FSPEC SVC-78

This SVC will fetch a file or device specification from an input buffer. Conversion of lower case to upper case will be made.

Registers Affected: AF, HL.

Entry:

HL	A pointer to the buffer containing file specification.
DE	A pointer to the 32-byte File Control Block.

Exit:

HL	Points to the terminating character found.
A	Will contain the terminating character.
Z	Set if valid file specification found.

7.6.34 @GET SVC-03

This SVC will fetch a byte from a logical device or a file. Note that if the DCB references the *KI device, an NZ condition with error code of 0 (A=0) will indicate that no character was available.

Registers Affected: AF.

Entry:

DE	A pointer to the DCB or FCB for the device/file.

Exit:

A	Byte fetched or error return code.
Z	Set if byte was fetched without error.

7.6.35 @GTDCB SVC-82

This SVC will locate the address of the Device Control Block (DCB) associated with the device name passed in the invocation.

Registers Affected: AF, HL.

Entry:

DE	2-character device name (E has 1st char, D has 2nd char). Note: If DE=0, then a pointer to the first available DCB will be returned.

Exit:

HL	Address of the Device Control Block.
Z	set on no error, else error 8 (device not avail).

7.6.36 @GTDCT SVC-81

This SVC will obtain a pointer to the Drive Control Table (DCT) associated with the requested logical drive. See the section on DRIVE CONTROL TABLE in chapter 4 for detailed information on the DCT.

Registers Affected: AF, IY.

Entry:
C logical drive number (0-7).

Exit:
IY the Drive Code Table address.

7.6.37 @GTMOD SVC-83

This SVC will locate the entry address of a module resident in memory provided all resident modules use the established header protocol.

Registers Affected: AF, DE, HL.

Entry:
DE Pointer to the module name terminated with an ETX (or any character in the range (X'00'-X'1F').

Exit:
HL Returned entry address of the module.
DE Pointer to address of first byte past the module name storage within the module header.
Z Set if the module is found in memory.

7.6.38 @HDFMT SVC-52

This SVC is used to pass a function 12 (X'0C') to a disk driver. It is commonly used to pass a "format drive" command to a hard disk controller. See chapter 4 for more information.

Registers Affected: AF [Note: DOS saves BC, IY; drivers should save registers any other registers they use].

Entry:
C The logical drive number (0-7).

Exit:
A The return code if an error.
Z Set if no error.

7.6.39 @HEX16 SVC-99

This SVC will convert a 16-bit binary number to hex ASCII.

Registers Affected: AF, HL.

Entry:
DE Contains the value to be converted.
HL A pointer to your 4 character buffer.

Exit:
HL Points to end of buffer + 1.

7.6.40 @HEX8 SVC-98

This SVC will convert a 1-byte number to hex ASCII.

Registers Affected: AF, HL.

Entry:
C Contains the value to convert.
HL A pointer to your 2-character buffer.

Exit:
HL Will point to end-of-buffer + 1.

7.6.41 @HEXDEC SVC-97

This SVC converts a 16-bit binary number into decimal ASCII.

Registers Affected: AF, BC, HL.

Entry:
HL Contains the value to convert.

DE A pointer to your 5-character buffer.

Exit:
DE Will point to end-of-buffer + 1.

7.6.42 @HIGH$ SVC-100

This SVC will alter or return the current value of HIGH$/LOW$. Note that neither can be altered if bit-0 of the CFLAG$ is set. HIGH$ is a word containing the highest RAM address usable by the system. User modules that need be protected from being overwritten are placed in high memory. The module's last address should occupy the current HIGH$ and HIGH$ is then lowered to correspond to the memory location just prior to the module. LOW$ needs to be set by those programs using @CMNDR that want to protect memory starting from their lowest address (LOW$ defaults to X'2FFF').

Registers Affected: AF [HL if originally set to 0].

Entry:
B 0, SVC deals with HIGH$
B 1, SVC deals with LOW$
HL If a non-zero value is contained in HL, then HIGH$/LOW$ is changed the that value. If HL contains a zero value, then the current value of HIGH$/LOW$ is returned.

7.6.43 @INIT SVC-58

INIT will open an existing file. If the file is not found, it will be created according to the file specification.

Registers Affected: AF.

Entry:
HL The 256-byte disk I/O buffer to be used during I/O.
DE File Control Block containing the file specification.
B Logical Record Length to be used while the file is open.

Exit:

A	Error return code
CF	Set if a new file was created
Z	Set if no error is encountered during the INIT.

7.6.44 @IPL SVC-00

This SVC will reboot the system. It functions the same as pressing the hardware RESET button. A usable booting system disk must be available in physical drive 0.

Registers Affected: Not applicable

7.6.45 @KBD SVC-08

This SVC will scan the *KI device and return the fetched character, if any character is available. Note that it is possible to generate an end-of-file (EOF) error from the physical keyboard (NZ with A=X'1C'). Consult the DOS manual for your particular installation to ascertain what key entry establishes the EOF indication. On the TRS-80 Model 4, for instance, the entry <CONTROL><SHIFT><@> generates the EOF.

Registers Affected: AF, DE.

Exit:

A	Contains the value of the key depressed or error return code.
Z	Set to indicate register-A contains the entered key code. If reset, then either no key was depressed or an error occurred. Register-A will contain a zero (X'00') under no-key, no-error. Register-A will contain a non-zero error code if an error was detected during the character "get" (perhaps a route?).

7.6.46 @KEY SVC-01

This SVC will continuously scan the *KI device until a character is available. It will not return until a character is available.

Registers Affected: AF, DE.

Exit:

A	Contains the character entered or error code.
Z	Set if no error is encountered.

7.6.47 @KEYIN SVC-09

This SVC will accept a line of input until terminated by either an <ENTER> or <BREAK>. During the input, the routine will display the entries. Backspace, tab, and line delete are supported. KEYIN exits with the cursor in whatever state it was in at the time KEYIN was entered.

Registers Affected: AF, BC, DE.

Entry:

HL	Pointer to user line buffer of length = B+1.
B	Maximum number of characters to input.
C	Should contain a zero (possible enhancement of KEYIN will use register C to contain a fill character).

Exit:

B	Contains the actual number of characters input.
CF	Set if <BREAK> terminated the input.
Z	Set if no error was encountered.

7.6.48 @KLTSK SVC-32

This SVC will remove the task assignment from the task table and return to the foreground application that was interrupted when called by an executing task driver. See Chapter 8, the Appendix, section on TASK PROCESSING for detailed information.

Registers Affected: Not applicable..

7.6.49 @LOAD SVC-76

This SVC will load a program file (a file in load module format).

Registers Affected: AF, B, HL.

Entry:
DE FCB containing the filespec of the file to load.

Exit:
HL Will contain the program's transfer address if no error is detected during the load; otherwise it will contain the error return code.
Z Set if the load was successful.

7.6.50 @LOC SVC-63

This SVC will calculate the current logical record number for the file referenced.

Registers Affected: AF, BC.

Entry:
DE A pointer to the FCB for the file to check.

Exit:
BC Returns the current logical record number.
A Error return code if an error is encountered.
Z Set if the operation was successful.

7.6.51 @LOF SVC-64

This SVC will calculate the logical record number where an end-of-file (EOF) error would be encountered for the referenced file.

Registers Affected: AF, BC.

Entry:
DE A pointer to the FCB for the file to check.

Exit:
BC Returns the EOF logical record number.
A Error return code if an error is encountered.
Z Set if the operation was successful.

7.6.52 @LOGER SVC-11

This SVC will issue a log message to the Job Log device (*JL). The "message" is any character string terminating with an <ENTER> (X'0D'). The current time string will be automatically prefixed to the message.

Registers Affected: AF, DE.

Entry:
HL A pointer to the message line to log.

Exit:
A Error return code if an error is encountered.
Z Set if the operation was successful.

7.6.53 @LOGOT SVC-12

This SVC will display and log a message. It will perform the same function as @DSPLY followed by @LOGER.

Registers Affected: AF, DE.

Entry:
HL A pointer to the message line to log.

Exit:
A Error return code if an error is encountered.
Z Set if the operation was successful.

7.6.54 @MSG SVC-13

This SVC is a message line handler used to output a message string to any device.

Registers Affected: AF.

Entry:
DE A pointer to a Device or File Control Block to receive output.
HL A pointer to the message line.

7.6.55 @MUL16 SVC-91

This SVC will perform an unsigned integer multiplication of a 16-bit multiplicand by an 8-bit multiplier. The resultant value is stored in a 3-byte register field.

Registers Affected: AF, DE.

Entry:
HL Contains the multiplicand value.
C Contains the multiplier value.

Exit:
HL Returns the two high order bytes of resultant value.
A Returns the low-order byte of the resultant value.

7.6.56 @MUL8 SVC-90

This SVC will perform an 8-bit by 8-bit unsigned integer multiplication. Since overflow out of the 8-bit register is not returned as an error, the routine should only be used on small integer values.

Registers Affected: AF, DE.

Entry:
C Contains the multiplicand value.
E Contains the multiplier value.

Exit:
A Returns the resultant value.

7.6.57 @OPEN SVC-59

This SVC will open an existing file or device. The Logical Record Length (LRL) passed in register B should match the LRL stored in the directory. If it does not, an "LRL open fault" error will be returned; however, the file will still be opened. If the file is already in an open state, the file's directory record will indicate the condition. In this case, the file will still be opened; however, only READ access (or less depending on the access permitted by the password) will be granted. A "File already open" error will also be returned.

Registers Affected: AF.

Entry:

HL A pointer to your buffer for disk I/O.
DE A pointer to the File or Device Control Block containing the filespec or devicespec.
B Should contain the Logical Record Length for the open file.

Exit:

A Error return code
Z Set if open was successful

7.6.58 @PARAM SVC-17

This SVC can be used to parse an optional command line parameter string. Its primary function is to parse command parameters contained in a command line totally enclosed within parentheses. The parameter formats acceptable for the command line entries are as follows:

```
PARM=X'hhhh'        hexadecimal entry
PARM=ddddd          decimal entry
PARM="string"       alphanumeric entry
PARM=ON             switch entry indicating TRUE
PARM=YES            switch entry indicating TRUE
PARM=Y switch       entry indicating TRUE
PARM=OFF            switch entry indicating FALSE
PARM=NO             switch entry indicating FALSE
PARM=N              switch entry indicating FALSE
```

The user-entered parameters that are to be accepted by your application are contained in a parameter table (PRMTBL$). This table stores the parameter names and a pointer to indicate where the user response is to be placed. Two forms of the PRMTBL$ are supported.

The first form uses a fixed width table with a maximum name length of six characters. The PRMTBL$ is coded as follows. A 6-character NAME left justified and filled with blanks followed by a 2-byte address VECTOR which points to the location which will receive the parsed values. The 2-byte memory address denoted by the address VECTOR field of your table receives the value of PARM if PARM is non-string. If a string is entered, the 2-byte memory address receives the address of the first byte of "string". NAME and VECTOR may be repeated for as many parameters as are desired. A byte of X'00' must be placed at the end of the table to indicate its ending point.

The second PRMTBL$ format permits a greater degree of flexibility in parameter handling. It also provides feedback as to each parameter entered by the user. Its format begins with a byte of X'80' to indicate the enhanced table. Each parameter is then identified with four fields. These fields are as follows:

CONTROL

Bit 7	Set if numeric values are to be accepted.
Bit 6	Set if switch values are to be accepted.
Bit 5	Set if string values are to be accepted.
Bit 4	Set if the first character of NAME is accepted as an abbreviation for the parameter.
Bits 0-3	Contain the length of the NAME field (1-15).

NAME

Contains the parameter name used to reference the parameter on the command line. This field must be in upper case.

RESPONSE

Bits 7-5 Are set by @PARAM as appropriate to the type of entry made by the user.

Bits 0-4 Contain the length of the string entry if a string was entered. A length of 0 is indicative of either a NULL string or a string longer than 31 characters. This can be differentiated by testing the first character of the string. If a double quote ("), then a NULL string was entered. Any other character indicates a string longer than 31 characters which will be terminated by a (").

VECTOR

This word is a pointer to the memory location that will receive the parsed value. It is filled in the same manner as that identified in the first format.

Note: Caution is to be observed in the proper use of the enhanced mode when you have something like the following: ON and ONLY in the table; if ON islisted first, then ON, ONx, ONxx, etc will match. This is because the parsing stops as soon as the length of the table entry has been reached. Alternatives are to add an appending space to the table entry, or order the table ONLY followed by ON.

See Chapter 8, the Appendix, USING THE SYSTEM PARAMETER SCANNER, for detailed information. The @PARAM protocol is as follows:

Registers Affected: AF, BC, HL.

Entry:

DE	A pointer to the beginning of your parameter table.
HL	A pointer to the command line to parse.

Exit:

HL	Returns pointing to the terminating character.
Z	Set if either no parameters found or valid parameters.
NZ	If a bad parameter was found.
A	Effective with 6.2.0, contains error code 44 on NZ return.

7.6.59 @PAUSE SVC-16

This SVC will suspend program execution and go into a "wait" state for a period of time determined by your count. The delay is approximately 15 microseconds per count regardless of the system FAST/SLOW option.

Registers Affected: AF, BC.

Entry:
BC delay count

7.6.60 @PEOF SVC-65

This SVC will position an open file to the end-of-file position. If the SVC is successful, an error 28 - "End of file encountered" will be returned.

Registers Affected: AF.

Entry:
DE A pointer to the FCB of the file to position.

Exit:
A Will return the error return code.

7.6.61 @POSN SVC-66

This SVC will position a file to a logical record. This will be useful for positioning to records of a random access file. When the @POSN routine is used, Bit 6 of FCB+1 is automatically set to ensure that the EOF will be updated when the file is closed only if the NRN exceeds the current ERN. This action will guard against any inadvertant deallocation of space in the random access file. A file can be extended by positioning to its EOF (see @PEOF) then writing to it.

Registers Affected: AF.

Entry:
DE A pointer to the FCB for the file to position.
BC Contains the logical record number for the positioning.

Exit:
A Will contain an error return code if an error was encountered.
Z Set if the operation was successful

7.6.62 @PRINT SVC-14

This SVC will output a message string to the printer device, *PR. The message string must conform to the syntax specified under @DSPLY.

Registers Affected: AF, DE.

Entry:
HL A pointer to the message to be output.

Exit:
A Will contain an error code if the SVC was unsuccessful.
Z Set if the SVC was successful.

7.6.63 @PRT SVC-06

This SVC will output a byte to the printer device, *PR. All character codes are passed unaltered to the device unless the forms filter is filtering the device. If the *PR device is not available, the SVC will time out after approximately 10 seconds and return a "Device not available" error.

Registers Affected: AF, DE.

Entry:
C Contains the character to print.

Exit:
A Will contain the error code if the SVC was unsuccessful.
Z Set if the SVC was successful.

7.6.64 @PUT SVC-04

This SVC will output a byte to a logical device or a file.

Registers Affected: AF.

Entry:
DE A pointer to the Device or File Control Block of the output device.
C Contains the byte to output.

Exit:
A Will contain an error return code if the SVC was unsuccessful.
Z Set if the SVC was successful.

7.6.65 @RAMDIR SVC-35

This SVC provides abbreviated information from the directories of visible files as well as free space information for a disk. It will provide information similar to the RAMDIR vector on earlier Model III TRSDOS 1.3. Register C is used to pass a function code to the SVC. Linkage is as follows:

Total Directory

Registers Affected: AF.

Entry:
C 0; Obtain directory records of all visible files.
B Should contain the logical drive (0-7) for the disk.
HL A pointer to your buffer which will be passed the data.

Exit:
A Returns an error code if the operation encountered an error.
Z Set if the SVC was successful.

File Directory

Registers Affected: AF.

Entry:
C 1-254; Obtain the directory record for the file whose Directory Entry Code (DEC) is equal to register C+1.
B Should contain the logical drive (0-7) for the disk.
HL A pointer to your buffer which will be passed the data.

Exit:
A Returns an error code if the operation encountered an error.
Z Set if the SVC was successful.

The information passed to your buffer will consist of 22-byte records. The buffer is terminated by a plus sign ("+"). Each record is fielded as follows:

0-14 FILENAME/EXT:D - left justified and buffered with spaces
15 Protection level (0-6)
16 End of File (EOF) offset byte
17 Logical Record Length (0 implies 256)
18-19 Ending Record Number (ERN) of the file
20-21 Space allocated for the file (in K)

Free Space

The SVC linkage to accomplish a retrieval of free space is as follows:

Registers Affected: AF.

Entry:
C 255; Obtain free space information.
B Should contain the logical drive (0-7) for the disk.
HL A pointer to your buffer which will be passed the data.

Exit:
A Returns an error code if the operation encountered an error.
Z Set if the SVC was successful.

The total space allocated to files (in K) is returned in the first two bytes of the buffer while the total space left available (in K) is stored in the third and fourth bytes of the buffer.

7.6.66 @RDHDR SVC-48

This SVC passes a function 8 to a disk driver. It is commonly used for reading sector header information from the next encountered sector ID field of a floppy disk . See chapter 4 for additional information.

Registers Affected: AF [Note: DOS saves BC, IY; drivers should save any other registers they use].

Entry:
C Logical drive number (0-7).
HL A pointer to the buffer which will receive the data transfer.

Exit:
A Contains an error return code, if any.
Z Set if the operation was successful.

7.6.67 @RDSEC SVC-49

This SVC passes a function 9 to a disk driver. This is used to transfer a sector of data from the disk drive to your buffer. See chapter 4 for additional information.

Registers Affected: AF [Note: DOS saves BC, IY; drivers should save any other registers they use].

Entry:
HL A pointer to the buffer to receive the sector of data.
D Contains the logical cylinder number to read (0-255).
E Contains the logical sector number to read (0-255).
C Contains the the logical drive number.

Exit:
A Passes the error return code if an error is encountered.
Z Set if no error is encountered.

7.6.68 @RDSSC SVC-85

This SVC will read the directory system sector identified by the calling linkage. The cylinder number containing the directory that is loaded into register D is recovered from the Drive Control Table (DCT). The DCT for the each drive is obtained via the @GTDCT SVC.

Registers Affected: AF.

Entry:
HL A pointer to the buffer to receive the sector of system data.
D Contains the logical cylinder number to read (0-255).
E Contains the logical sector number to read (0-255).
C Contains the the logical drive number.

Exit:
A Passes the error return code if an error is encountered.
Z Set if no error is encountered.

7.6.69 @RDTRK SVC-51

This SVC passes a function 11 to a disk driver. It is commonly used for reading an entire track of a floppy disk where permitted by the controller. See chapter 4 for additional information.

Registers Affected: AF [Note: DOS saves BC, IY; drivers should save any other registers they use].

Entry:
C Logical drive number (0-7).
HL A pointer to the buffer which will receive the data transfer.

Exit:
A Contains an error return code, if any.
Z Set if the operation was successful.

7.6.70 @READ SVC-67

This SVC will read a logical record from an open file. If the LRL defined at open time was 256 (0), then the next sequential sector identified by the Next Record Number (NRN) contained in the File Control Block (FCB) will be transferred to the buffer established at open time. For Logical Record Lengths (LRLs) between 1 and 255, the next logical record will be placed into the user record buffer, UREC, identified in the @READ SVC. The 3-byte NRN is updated after the read operation so as to prepare for the next sequential read operation.

Registers Affected: AF.

DE A pointer to the FCB for the file to read.
HL A pointer to the UREC (needed if LRL <> 0).

Exit:
A Will contain an error return code if an error was encountered.
Z Set if the operation was successful.

7.6.71 @REMOV SVC-57

This SVC will remove a file. The FCB must be in an open condition prepared by @OPEN or @INIT. The file's directory will be updated by resetting the activity bit (bit-4 of DIR+0), the corresponding Directory Entry Code (DEC) in the Hash Index Table (HIT) will be set to zero, and the space occupied by the file will be deallocated from the Granule Allocation Table (GAT). The 32-byte FCB will be set to zeroes upon successful commpletion of the file's removal. If the control block contained data appropriate to an opened device, the @REMOVE SVC will treat the request as if it were an @CLOSE request. Devices can only be removed via the RESET library command.

Registers Affected: AF.

Entry:
DE A pointer to the open File Control Block (FCB) of the file.

Exit:
A Will contain an error code if an error is encountered.
Z Set if no error is detected.

7.6.72 @RENAM SVC-56

This SVC can be used to change the filename or extension fields of a file stored on disk. The access protection level must permit renaming for the operation to be successful.

Registers Affected: AF.

Entry:
DE A pointer to the File Control Block (FCB) containing the filespec of the file to be renamed.
HL A pointer to the FCB containing the new filename/extension.

Exit:
A Will contain an error code if an error is encountered.
Z Set if no error is detected.

7.6.73 @REW SVC-68

This SVC will rewind a file to its beginning and reset the 3-byte NRN pointer to 0. The next record that will be transferred for I/O with a @READ /@WRITE request will be the first record of the file.

Registers Affected: AF.

Entry:
DE A pointer to the FCB for the file that you want to rewind.

Exit:
A Will contain an error return code if an error was encountered.
Z Set if the operation was successful.

7.6.74 @RMTSK SVC-30

This SVC will remove an interrupt level task from the Task Control Block Vector Table (TCBVT). See Chapter 8, Appendix, on TASK PROCESSOR for detailed information on the use of this SVC.

Registers Affected: AF, DE, HL.

Entry:
C Contains the task assignment slot (0-11) to remove.

7.6.75 @RPTSK SVC-31

This SVC must be invoked only from an executing task. It will exit the task process currently executing and replace the task's vector address in the Task Control Block Vector Table (TCBVT) with the address following the SVC instruction. Return is made to the foreground application that was interrupted. See the TASK PROCESSOR section in Chapter 8, the Appendix, for detailed information on the use of this SVC.

Registers Affected: Not applicable..

7.6.76 @RREAD SVC-69

This SVC will cause a reread of the current sector providing the file was opened with an LRL between 1 and 255 or the file was accessed via character I/O (@GET /@PUT). Its most probable use would be in applications that reuse the disk I/O buffer for multiple files and want to reload the buffer with the proper file sector.

Registers Affected: AF.

Entry:
DE A pointer to the FCB for the file to reread.

Exit:
A Will contain an error return code if an error was encountered.
Z Set if the operation was successful.

7.6.77 @RSLCT SVC-47

The SVC is used to pass a function code 7 to a disk driver. This function will perform a test of the selected drive to see if it is in a busy state (i.e. if the disk controller is still executing a command). If busy, the drive will be re-selected until it is no longer busy. See chapter 4 for additional information.

Registers Affected: AF [Note: DOS saves BC, IY; drivers should save any other registers they use].

Entry:
C Should contain the logical drive number.

7.6.78 @RSTOR SVC-44

This SVC will restore a disk drive to cylinder 0 by passing a function 4 to a disk driver. See chapter 4 for additional information.

Registers Affected: AF [Note: DOS saves BC, IY; drivers should save any other registers they use].

Entry:
C Logical drive number (0-7).

Exit:
A Contains an error return code, if any.
Z Set if the operation was successful.

7.6.79 @RUN SVC-77

This SVC will load and execute a program file. Your FCB should not be located in the memory region that will be loaded with the file you want to execute.

Registers Affected: AF, BC [Note: HL alterted on an error].

Entry:
DE A pointer to the FCB containing the program's filespec.

Exit:
BC Returns a pointer to the start of the system command buffer.
HL Contains the error return code if an error was encountered.

7.6.80 @RWRIT SVC-70

This SVC will rewrite the current sector following a write operation. The @WRITE function advances the Next Record Number (NRN) after the sector is written. @RWRIT will decrement the NRN and write the disk buffer again.

Registers Affected: AF.

Entry:
DE A pointer to the FCB for the file sector to rewrite.

Exit:
A Contains an error return code if an error was encountered.
Z Set if the operation was successful.

7.6.81 @SEEK SVC-46

This SVC will pass a function code 6 to a disk driver. It is used to issue a controller SEEK command. Disk controllers optionally verify only the track address, therefore it is not necessary to pass a sector number to @SEEK. See chapter 4 for additional information.

Registers Affected: AF [Note: DOS saves BC, IY; drivers should save any other registers they use].

Entry:
C Contains the logical drive number.
D Contains the logical cylinder requested.

7.6.82 @SEEKSC SVC-71

This SVC is used to seek a specified file record prior to attempting to read or write the record. The record identified for the seek operation will be that determined by the Next Record Number (NRN) identified in the File Control Block (FCB). The SEEK operation may require that the current file buffer be written back to disk if it contains updated information and the desired record is located in a different disk sector. If an error occurs in this operation, the error code will be returned. The return code condition will never reflect an error for the actual SEEK itself. @SEEKSC serves a useful purpose only when asynchronous I/O is implemented permitting disk seeking external to CPU control. On the TRS-80 Model 4, it is unnecessary.

Registers Affected: AF.

Entry:
DE A pointer to the File Control Block of the file.

Exit:
A Contains an error code if an error is encountered in writing.
Z Set will indicate that the SEEK operation "completed".

7.6.83 @SKIP SVC-72

This SVC will cause a skip past the next logical record. The SKIP operation may require that the current file buffer be written back to disk if it contains updated information and the desired record is located in a different disk sector. If any error is encountered

in this operation, an error will be returned. The Next Record Number (NRN) contained in the FCB will be changed accordingly.

Registers Affected: AF.

Entry:
DE A pointer to the FCB for the file to skip.

Exit:
A Will contain an error return code if an error was encountered.
Z Set if the operation was successful.

7.6.84 @SLCT SVC-41

This SVC will pass a function code 1 to a disk driver. See chapter 4 for additional information. The function will select a drive. The appropriate time delay specified in your configuration (SYSTEM (DELAY=Y/N)) should be undertaken if the drive selection requires it.

Registers Affected: AF [Note: DOS saves BC, IY; drivers should save any other registers they use].

Entry:
C Contains the logical drive number (0-7).

Exit:
A Will contain an error return code if an error was encountered.
Z Set if the operation was successful.

7.6.85 @SOUND SVC-104

This SVC will interface to the sound generator if one is provided with the computer. Note that the maskable interrupts are disabled during the duration of the tone generation. The routine should function the same regardless of FAST/SLOW. All regs except the accumulator are left unchanged. The Z-flag is always set on exit. For those generators capable of multiple sounds, the linkage is as follows:

Registers Affected: AF.

Entry:
B Contains a function code packed as follows:

Bits 0-2 tone selection (0-7) with 0=highest & 7=lowest.

Bits 3-7 Contain the tone duration (0-31) with 0=short, 31=long. Short approx 3/32 sec, long approx 3 sec.

7.6.86 @STEPI SVC-45

This SVC passes a function 5 to a disk driver. It is commonly used for specifying a step-in controller command. See chapter 4 for more information.

Registers Affected: AF [Note: DOS saves BC, IY; drivers should save any other registers they use].

Entry:
C Logical drive number (0-7).

Exit:
A Error return code, if any.
Z Set if the operation was successful.

7.6.87 @TIME SVC-19

This SVC will return the time of day in display format (HH:MM:SS). It also will recover a pointer to the binary time storage which may be useful for those implementing hardware clocks.

Registers Affected: AF, BC, DE.

Entry:
HL A pointer to the 8-character buffer to receive the time string.

Exit:
DE Returns a pointer to the binary time storage, TIME$. The 3-byte region contains seconds, minutes, and hours. TIME$-1 stores the 30 Hertz rate system timer.

7.6.88 @VDCTL SVC-15

This SVC performs various video control functions depending on the function code passed in register B. It is very useful for handling direct video access. The functions are as follows:

VIDEO "PEEK"

Registers Affected: AF, BC, DE.

Entry:
B 1; Gets the character at the position identified by HL.
HL Contains the row (0-23) in register H, and column (0-79) in L.

Exit:
A Will be returned with the character at "HL".
Z Set if the operation was successful.

VIDEO "POKE"

Registers Affected: AF, BC, DE.

Entry:
B 2; Puts the character at the position identified by HL.
HL Contains the row (0-23) in register H, and column (0-79) in L.
C Contains the character to put at "HL".

Exit:
Z Set if the operation was successful.

SET CURSOR POSITION

Registers Affected: AF, B, DE.

Entry:
B 3; Moves the cursor to the position identified by HL.

HL Contains the row (0-23) in register H, and column (0-79) in L.

Exit:
A Will contain the error code if an error was encountered.
Z Set if the operation was successful.

OBTAIN CURSOR POSITION

Registers Affected: AF, B, HL.

Entry:
B 4; Obtains the current cursor position by row and column.

Exit:
HL Contains the row (0-23) in register H, and column (0-79) in L.
A Will contain the error code if an error was encountered.

BUFFER TO VIDEO

Registers Affected: AF, BC, DE, HL.

Entry:
B 5; Moves a BLOCK of RAM to the video RAM.
HL A pointer to the user's RAM BLOCK.

Exit:
A Will contain the error code if an error was encountered.
Z Set if the operation was successful.

BLOCK is 1920 bytes for 6.2, 2048 bytes for 6.0 and 6.1

VIDEO TO BUFFER

Registers Affected: AF, BC, DE, HL.

Entry:
B 6; Moves the video RAM image to a RAM BLOCK.
HL A pointer to the user's RAM BLOCK.

Exit:
A Will contain the error code if an error was encountered.
Z Set if the operation was successful.

BLOCK is 1920 bytes for 6.2, 2048 bytes for 6.0 and 6.1

SCROLL PROTECT

Registers Affected: AF, B.

Entry:
B 7, Inhibit scrolling of lines at the top of the video screen.
C Contains the number of lines to protect (0-7).

CURSOR CHARACTER

Registers Affected: AF, B.

Entry:
B 8; Change the cursor character.
C Contains the new cursor character (or code value).

Exit:
A Will be returned with the current cursor value (for 6.0.1+).
Z Set if the operation was successful.

VIDEO LINE TRANSFER

Registers Affected: AF, BC, DE, HL.

Entry:
B 9; Invoke line transfer
C transfer direction; 0 = buffer to video, 1 = video to buffer.
H video row to transfer (0-23).
DE A pointer to the user's 80-character buffer.

Exit:
A Will contain the error code if an error was encountered.
Z Set if the operation was successful.

7.6.89 @VER SVC-73

This SVC will perform a @WRITE operation followed by a test read of the sector (assuming that the WRITE required physical I/O) to verify that it will be readable. The test read will not cause data to be transferred to the file buffer.

Registers Affected: AF.

Entry:
DE A pointer to the FCB for the file to verify.
HL A pointer to the user record buffer (UREC) containing the logical record (where the LRL is <> 256).

Exit:
A Will contain an error return code if an error was encountered.
Z Set if the operation was successful.

7.6.90 @VRSEC SVC-50

This SVC will pass a function 10 to a disk driver. The function should verify the readability of a sector without transferring any data from the disk to the buffer. See chapter 4 for additional information.

Registers Affected: AF [Note: DOS saves BC, IY; drivers should save any other registers they use].

Entry:
C Contains the logical drive number.
D Contain the cylinder number to verify.
E Contains the sector number to verify.

Exit:
A Will contain an error return code if an error was encountered.
Z set if the operation was successful.

7.6.91 @WEOF SVC-74

This SVC will force the system to update the directory entry with the current end-of-file information. The file's FCB will remain in an open state.

Registers Affected: AF.

Entry:
DE A pointer to the FCB for the file to WEOF.

Exit:
A Will contain an error return code if an error was encountered.
Z Set if the operation was successful.

7.6.92 @WHERE SVC-07

This SVC can be invoked to determine the address of the calling routine. It can be useful for small routines that are to be made run-time relocatable.

Registers Affected: AF, HL.

Exit:
HL Returns the memory address following the SVC instruction.

7.6.93 @WRITE SVC-75

This SVC will cause a write to the next record identified in the FCB. If the file's Logical Record Length (LRL) identified in the FCB is less than 256, then the logical record in the user buffer will be transferred to the file. If LRL is equal to 256, a full sector I/O will be made using the disk I/O buffer identified at file open time.

Registers Affected: AF.

Entry:
DE A pointer to the FCB for the file to write.
HL A pointer to the user record buffer (UREC) containing the logical record (where the LRL is <> 256).

Exit:
A Will contain an error return code if an error was encountered.
Z Set if the operation was successful.

7.6.94 @WRSEC SVC-53

This SVC will pass a function code 13 to a disk driver. It is used to write a physical sector of data to the disk. See chapter 4 for additional information.

Registers Affected: AF [Note: DOS saves BC, IY; drivers should save any other registers they use].

Entry:
C Contains the logical drive number.
D Contains the number of the cylinder to write.
E Contains the number of the sector to write.
HL A pointer to the buffer containing the sector of data.

Exit:

A Will contain the error code if an error was encountered.
Z Set if the operation was successful.

7.6.95 @WRSSC SVC-54

This SVC will pass a function code 14 to a disk driver. It is used to write a system sector (used in the directory cylinder). Where the disk controller supports the IBM Data Address Mark convention, the controller command should denote the "deleted data mark", or X'F8' in lieu of the standard data mark (X'FB'). This distinct mark is used in the @RDSEC command to detect the presence of a system (directory) sector. Other than this Data Address Mark variation, @WRSSC is the same as @WRSEC ; however, the DOS will use @WRSSC for all writes to the directory cylinder. See chapter 4 for additional information.

Registers Affected: AF [Note: DOS saves BC, IY; drivers should save any other registers they use].

Entry:

C Contains the logical drive number.
D Contains the number of the cylinder to write.
E Contains the number of the sector to write.
HL A pointer to the buffer containing the sector of data.

Exit:

A Will contain the error code if an error was encountered.
Z Set if the operation was successful.

7.6.96 @WRTRK SVC-55

This SVC will pass a function code 15 to a disk driver. It is used to format a physical track on a disk drive. Where the data pattern is under software control (as is the case for floppy disk drives), the data format must conform to that identified in your controller's reference manual. Hard drives that are formatted by track may use this SVC to control the track to track formatting. If the target drive is a floppy disk, then it is necessary to precede the @WRTRK SVC with a drive select via SVC @SLCT . See chapter 4 for additional information.

Registers Affected: AF [Note: DOS saves BC, IY; drivers should save any other registers they use].

Entry:

C Contains the logical drive number.
HL Contains a pointer to the buffer containing the format data
D Contains the number of the cylinder to write.

Exit:

A Will contain the error code if an error was encoun tered.
Z Set if the operation was successful.

8. APPENDIX

8.1 BOOT INITIALIZATION ICNFG INTERFACING

In order to bring up the "DOS Ready" message when first powering up your computer, all that you need do is place a SYSTEM diskette into the disk drive physically assigned to the zero slot and depress a RESET button. In a few short moments, the ready prompt appears on the display screen. Although, to the casual observer, not much appears to have taken place, the machine has executed many "behind-the-scenes" procedures in order to make the operating system available for your commands. The appendix section on SYSTEM DISK BOOTING covers the individual steps undertaken. Here we discuss one of the final steps - the execution of an initialization configuration routine.

Certain items of hardware require an initialization process before they can be used. For instance, the RS-232 hardware needs to have parameters such as baud rate, word length, and number of stop bits initialized before it can be used. This initialization process could be a software routine which transfers the required parameters to the UART and Baud Rate Generator. Certain hard disk controllers (the XEBEC controller, for instance) may also need to be initialized before the attached disk drive can be used. This initialization process may be implemented as a program executing under the AUTO command or it may be a small routine that is part of the disk driver. If the latter, it would be useful to have it execute prior to the "DOS Ready" message. You may also develop a complex system function that takes over one or more SuperVisor Call functions. Since such a function could reside in memory as part of a configuration, it would be useful to have it automatically hook into the SVC table. Again, if the interfacing routine were part of the function code in memory and the system provided a method to execute such a routine, it would alleviate the problem of executing the hook.

After the system booting process loads a configuration file, it CALLs a vector, called the @ICNFG vector. The contents of the vector are accessible from the FLAGS pointer returned by the @FLAGS$ SuperVisor Call. Thus, any initialization routine that is part of a memory configuration can be executed if its entry address is made available to @ICNFG. This is accomplished by placing your entry address into @ICNFG while you save the former address - eventually transferring control to the former address when your routine completes its execution. This process is called "chaining into @ICNFG". If you need to configure your own routine that requires initialization when the machine is booted, you chain into @ICNFG.

Let's first look at a sample initialization configuration routine linkage. Your initialization routine would obviously be unique to the function it was to perform so we will not illustrate that part. A template for such a routine would appear as:

```
INIT    CALL    ROUTINE         ;Start of init
LINK    DB      'Roy'           ;Pass to the chain
ROUTINE .
        Your initialization routine
        .
        .
        RET                     ;End with a RET instruction!
```

The relocated address identified by the label "INIT" is the entry point that will be placed into the @ICNFG vector field. The 3-byte field identified as "LINK" will be used to store the original contents of the @ICNFG vector field. Thus, when INIT receives control, it "calls" your initialization routine then passes back to the next routine chained into @ICNFG.

We will now illustrate a procedure to accomplish the chaining linkage. The chaining procedure is performed by that part of your program which is going to place the memory-

resident routine into its execution location in memory. The first thing that must be done is to move the contents of the @ICNFG vector into your initialization routine. The code:

```
LD      A,@FLAGS$       ;Get flags pointer
RST     40              ;  into register IY
LD      A,(IY+28)       ;Get @ICNFG byte 1
LD      (LINK),A        ;  & save in LINK+0
LD      L,(IY+29)       ;Get address LOW and HIGH
LD      H,(IY+30)       ;  then save in the
LD      (LINK+1),HL     ;  LINK address vector
```

does this by transferring the three byte vector to your routine. You then need to relocate your routine to its execution memory address. Once this is done, transfer the relocated initialization entry point to the @ICNFG vector as a jump instruction with this code:

```
LD    HL,INIT           ;Get (relocated)
LD    (IY+29),L         ;  init address
LD    (IY+30),H
LD    A,0C3H            ;Set JP instruction
LD    (IY+28),A
```

It is sometimes necessary to have your initialization program execute the initialization routine so that the function of the module is immediately available. You probably do not want to execute any other routines that may be chained into @ICNFG so you should not CALL the chain! Your initialization routine can be executed by calling its relocated address as in:

```
CALL  ROUTINE           ;Initialize only mine
```

Don't forget to SYSGEN after linking in your routine. The SYSGEN process includes saving the revisions to @ICNFG so that any changes will be part of the system configuration the next time the disk is booted. By following these procedures, you can effect the invocation of your routine every time you boot the operating system disk which contains this configuration.

8.2 THE KFLAG$ SCANNER

Many applications have the need to detect a PAUSE or BREAK condition while they are in execution. BASIC does this after every logical statement is executed (i.e. after each end of line or ":" statement separator). That's how, in BASIC, you can stop a program with the <BREAK> key or pause a listing. The classical method that programmers have used to detect the condition was to scan the keyboard via the @KBD SuperVisor Call. If a character was input, and it was a <BREAK> or a <PAUSE>, the appropriate action would be taken. Any other entry that was available would be ignored which would discard all other keyboard entries. Unfortunately, if the user was trying to make use of keyboard type-ahead, each @KBD request looking for <BREAK> or <PAUSE> would extract one character from the type-ahead buffer; thus the user's typed-ahead entries would be lost.

Another method could be used on a matrix keyboard that is accessible to the application. This method does not request entries via the @KBD call but scans the keyboard physically examining the keyboard matrix. A problem with this method is that accessible matrix keyboards are not always available. A second problem is that if such a keyboard was available, the application would not be portable across Version 6 installations.

If an application uses the KFLAG$ keyboard function latch to observe the BREAK or PAUSE condition, it overcomes these deficiencies [a third condition - that of the ASCII CR is also supported]. KFLAG$ contains three bits associated with the "keyboard" functions of BREAK, PAUSE (sometimes interpreted as <SHIFT-@>), and CR (sometimes interpreted as

<ENTER>). An interrupt task processor routine (herinafter called the KFLAG$ scanner or just scanner) examines the physical keyboard and sets the appropriate KFLAG$ bit if any of the conditions are observed. Similarly, the system's COM serial driver routine also sets the appropriate KFLAG$ bits if it detects the matching conditions being received. In the KFLAG$, bit-0 is assigned for BREAK, bit-1 is assigned for PAUSE, and bit-3 is assigned for CR.

It is important to note that the interrupt KFLAG$ scanner does NOT reset the condition bits - it only sets them. Thus, it is up to the application using these flag conditions to reset the bits as required. Now, you may ask, why wasn't the scanner coded so that it resets the bits? Well, if that was the case, you would never sense the "events" as they would occur too fast. Think of the KFLAG$ condition bits as a latch. Once a condition is detected (latched), it remains latched until some routine resets the latch, usually after examining a condition and taking action - a function to be performed by a KFLAG$ examination routine that is part of the application using it.

With this introduction, let's look at an illustrative routine designed to use the <BREAK> and <PAUSE> conditions of the KFLAG$ latch. This routine assumes that index register IY can be altered with impunity.

```
CKPAWS  LD      A,@FLAGS$       ;Get Flags pointer
        RST     40              ;  into reg IY
        LD      A,(IY+'K'-'A')  ;P/u the KFLAG$
        RRCA                    ;Bit 0 to carry
        JP      C,GOTBRK        ;Go on BREAK
        RRCA                    ;Bit 1 to carry
        RET     NC              ;Return if no pause
        CALL    RESKFL          ;Reset the flag
        PUSH    DE              ;Don't alter reg DE
FLUSH   LD      A,@KBD          ;Flush type-ahead
        RST     40              ;  buffer while
        JR      Z,FLUSH         ;  ignoring errors
        POP     DE
PROMPT  PUSH    DE
        LD      A,@KEY          ;Wait on key entry
        RST     40
        POP     DE
        CP      80H             ;Go on <BREAK>
        JP      Z,GOTBRK
        CP      60H             ;Ignore <PAUSE>
        JR      Z,PROMPT        ;  else ...
RESKFL  PUSH    HL              ;Reset KFLAG$ without
        PUSH    AF              ;  altering AF or HL
        LD      A,@FLAGS$       ;P/u flags pointer
        RST     28H             ;  into reg IY
RESKFL1 LD      A,(IY+'K'-'A')  ;P/u the flag
        AND     0F8H            ;Strip ENTER,
        LD      (IY+'K'-'A'),A  ;  PAUSE, BREAK
        PUSH    BC              ;Don't alter register BC
        LD      B,16
        LD      A,@PAUSE        ;Pause a bit to "debounce"
        RST     40              ;  the key entry
        POP     BC
        LD      A,(IY+'K'-'A')  ;Check if finger is
        AND     7               ;  still on key
        JR      NZ,RESKFL1      ;Reset it again
        POP     AF              ;Restore registers
        POP     HL              ;  and exit
        RET
```

In order to understand this KFLAG$ examination routine, the best thing to do would be to take apart the entire routine and explain each sub-routine. The first piece:

```
CKPAWS  LD      A,@FLAGS$       ;Get Flags pointer
        RST     40              ;  into reg IY
        LD      A,(IY+'K'-'A')  ;P/u the KFLAG$
        RRCA                    ;Bit 0 to carry
        JP      C,GOTBRK        ;Go on BREAK
        RRCA                    ;Bit 1 to carry
        RET     NC              ;Return if no pause
```

reads the KFLAG$ contents. The @FLAGS$ SuperVisor Call is used to obtain the flags pointer from the DOS. Be aware that if your application is using the IY index register, then you better save and restore it within the CKPAWS routine (alternatively, you could use memory loads in lieu of IY indexing, use @FLAGS at the beginning of your application to calculate the location of KFLAG$, and stuff the address into the CKPAWS memory LD instructions.) The first rotate instruction places the BREAK bit into the carry flag. Thus, if a <BREAK> condition was in effect, the sub-routine would branch to "GOTBRK" - which is your break handling routine. If there is no pending BREAK, the second rotate places what was originally in the PAUSE bit into the carry flag. If a <PAUSE> condition is not in effect, the routine returns to the caller. This sequence of code gives a higher priority to <BREAK> (i.e. if both BREAK and PAUSE conditions are pending, the <BREAK> condition has precedence). It is important to note that the GOTBRK routine needs to clear the KFLAG$ bits after it services the <BREAK> condition. This is simply done via a call to RESKFL.

The next part of the routine is executed on a <PAUSE> condition.

```
        CALL    RESKFL          ;Reset the flag
        PUSH    DE              ;Don't alter reg DE
FLUSH   LD      A,@KBD          ;Flush type-ahead
        RST     40              ;  buffer while
        JR      Z,FLUSH         ;  ignoring errors
        POP     DE
```

First the KFLAG$ bits are reset via the call to RESKFL. Next, we take care of removing any characters that are stored in the type-ahead buffer (the system will automatically clear the type-ahead buffer when a BREAK condition is latched). This can be done by repeatedly invoking the @KBD request until it returns a "no character available" condition code.

Now that the routine is in a PAUSEd state and the type-ahead buffer is cleared, it must wait for a key input. The following routine does this:

```
PROMPT  PUSH    DE
        LD      A,@KEY          ;Wait on key entry
        RST     40
        POP     DE
        CP      80H             ;Go on <BREAK>
        JP      Z,GOTBRK
        CP      60H             ;Ignore <PAUSE>
        JR      Z,PROMPT        ;  else ...
```

The PROMPT routine is coded to accept a <BREAK> and branch to your BREAK handling routine so that the user can "abort" from a PAUSE. It will ignore repeated <PAUSE> entries (the 60H is the standard byte value that is interpreted as a PAUSE entry). Any other character will cause it to fall through to the following routine which clears the KFLAG$ latch.

```
RESKFL  PUSH    HL              ;Reset KFLAG$ without
        PUSH    AF              ;  altering AF or HL
        LD      A,@FLAGS$       ;P/u flags pointer
        RST     40              ;  into reg IY
RESKFL1 LD      A,(IY+'K'-'A')  ;P/u the flag
        AND     0F8H            ;Strip ENTER,
```

```
LD      (IY+'K'-'A'),A  ;  PAUSE, BREAK
PUSH    BC              ;Don't alter register BC
LD      B,16
LD      A,@PAUSE        ;Pause a bit to "debounce"
RST     40              ;  the key entry
POP     BC
LD      A,(IY+'K'-'A')  ;Check if finger is
AND     7               ;  still on key
JR      NZ,RESKFL1      ;If so, reset it again
POP     AF              ;Restore registers
POP     HL              ;  and exit
RET
```

The RESKFL subroutine needs to be called when you first enter your application. This is necessary to clear the flag bits that were probably in a "set" condition. This "primes" the detection. The routine also needs to be called once a BREAK, PAUSE, or ENTER condition is detected and handled.

Another method that can be used to detect the BREAK condition is to use the @CKBRKC SuperVisor Call - SVC-105. This SVC essentially performs all of the code needed to test the BREAK bit of the KFLAG$ and reset it as required. Thus, instead of using your own code to test the KFLAG$'s BREAK bit, you can invoke @CKBRKC. An NZ return indicates that the BREAK key was depressed. Since the SVC also clears the BREAK bit, it should be invoked once at the beginning of your program to ensure that the bit is first reset.

8.3 DISK LOAD MODULE FORMATS

A load module is simply a disk file that can be loaded into memory by the system loader. The file is made up of variable length records and is usually a program. Many different types of records are included in a load module - the DOS makes extensive use of distinct record types in load modules. One record type is a load record which contains information on where it is to load into memory. If the file can be directly executed as a program, it then becomes known as an executable load module (ELM). The usual term that has been applied to such a file is "CMD". That's because a directly executable load module can be invoked as if it were a system CoMmanD. We further use the default file extension of /CMD for these command files.

A load module can be conceptualized as a sequence of RECORDS. Note that we did not say an ordered sequence. Thus, the implication is that the records do not have to be in an ascending order (contiguous load addresses). Each record contains three fields: a TYPE field, a LENGTH field, and a DATA field. It has a one-byte indicator as to what TYPE of record it is. This TYPE code is used to denote a record as a HEADER record, a TRANSFER record, an ISAM directory entry record, a LOAD record, or other meaningful structure. Each record also has a one-byte LENGTH field which is the length of the data area field. The data field length thus ranges from <1-256> in value. The remaining part of the record is its DATA AREA and is used to store program code, directory information, messages, or other pertinent information. If you are familiar with BASIC random access files, you will see the similarity in the fielding of records - except in this case, we have variable length sequentially accessed records [with partitioned data sets provided in the PRO-PaDS utility, you also have variable length indexed sequential accessed records]. Figure 8-1 lists the various TYPE codes currently used in the operating system.

TYPE	DATA AREA
01	Object code load block
02	Transfer address
04	End of partitioned data set member
05	Load module header
06	Partitioned data set header
07	Patch name header
08	ISAM directory entry
0A	End of ISAM directory
0C	PDS directory entry
0E	End of PDS directory
10	Yanked load block
1F	Copyright block

Figure 8-1: Load Module TYPE Codes

Any code above X'1F' is invalid as a record type. In addition, any code not listed in figure 8-1 is reserved for future use.

Let's look at a sample file. Start by listing the first sector of the FLOPPY/DCT utility via the command: **LIST FLOPPY/DCT (H)**. Notice that it starts out with:

```
05 06 46 4C 4F 50 50 59 1F 2A 43 6F ...
 .  .  F  L  O  P  P  Y  .  .  C  o ...
```

stretched across the screen. What you have here is a load module header (TYPE=05). The length byte (LENGTH=06) follows the TYPE code. The 6-byte DATA AREA field is the header name. All records follow this "fielding" order. A record is organized with a TYPE, LENGTH, DATA sequence. The X'1F' begins the second record. It happens to be a copyright record with a LENGTH of X'2A' or 42 decimal bytes. Incidentally, the TYPE=1F record is generated automatically by the "COM" pseudo-op in PRO-CREATE, the assembler used to develop and maintain the operating system.

Note that each record begins with the TYPE code and the first byte following the end of a record is always the TYPE code of the next record. The only exception is when a TYPE code indicates the end of a file. If you look further in the record displayed at relative position X'34', or if you count 42 bytes down from the "C" of "Copyright", you will see:

```
01 02 00 2C D5 ...
```

The record TYPE is a load block (TYPE=01), and the length of the data area is X'02', or 258 data bytes. Yes, we previously stated that the length ranged up to 256 and here we have 258! This TYPE-01 record is a special case. The two-byte field following the LENGTH is the starting load address for the rest of the field. Since the LENGTH value includes the 2-byte load address, a length of X'03' would indicate only one load byte. A length of X'04' would indicate two load bytes. A length of X'FF' would indicate 253 load bytes. A length of X'00' would indicate 254 load bytes. To be able to have a data area with up to 256 bytes of loadable data, the LENGTH values of X'01' and X'02' are indicative of 255 and 256 load bytes respectfully. This is accomplished by having the system loader decrement the length value by two when reading a load address. The resultant value becomes the true length of the loadable data.

If you let the program listing proceed to the end of the file, the last four bytes should appear as:

```
02 02 00 2C
```

This will represent the TRANSFER record (TYPE=02). Again, we have a LENGTH byte which shows a 2-byte data field. The data field contains the transfer address or entry point to the program in standard low-order, high-order sequence. The system uses this address as an entry to the program after successfully loading it into memory. This address is also what is returned in register pair HL by the @LOAD SuperVisor Call.

So far we have discussed the HEADER, the COPYRIGHT, the LOAD, and the TRANSFER records. These are the four common record types you will find in most load module files. We also observe that our discussion of program load modules was limited to a single program per file. Another kind of file is one that contains many program modules (or data modules) as sub-files. Since the file is divided into sub-files, it is considered a "partitioned data set" abbreviated as "PDS". The PDS contains a directory of its sub-files with each sub-file being termed a MEMBER of the PDS and having an entry in the directory. The system loader supports a particular kind of PDS used to contain the library overlays: SYS6/SYS, SYS7/SYS, and SYS8/SYS (LIB A, B, and C respectively).

Let's take a look at one of these libraries. List the first record of SYS6/SYS via the command: **LIST SYS6/SYS.LSIDOS (H)**. Look at the area just past the copyright message. You will see something like this:

```
08 06 21 00 24 00 00 CB 08 06 61 ...
```

The TYPE code of X'08' indicates an ISAM DIRECTORY ENTRY record. The LENGTH byte denotes a DATA area of six bytes. After the sixth byte, you will see another TYPE=08 starting another ISAM directory entry record. SYS6 is a partioned data set. The TYPE=08 records are the directory entries for its members.

The ISAM directory data area is used by the SYSTEM loader to locate where a particular member can be found in the file. The data area includes positioning information indicating the exact byte position in the PDS which is the first record of the member. The six-byte data field is further divided into sub fields. The first byte (in this case, 21) is the ISAM entry number. This entry number is provided to the system loader when a library command is parsed by the command interpreter. The entry number is the PDS member that will execute your request. The system loader searches the PDS directory for a matching directory record. The next two-byte sub-field is the transfer address of the member. The transfer address is contained in the directory so that more than one transfer address can be applied to a member. Therefore, a member can have multiple entry points. The last three-byte field is the triad pointer which points to the first byte of the member. The triad pointer is composed of the Next Record Number (NRN) and Relative Byte Offset for the member's first record byte. The system then positions to the pointer and loads the member. Thus you have six bytes of data as specified by the LENGTH byte. Since the process uses an index (the directory) to locate the member's starting byte then proceeds to sequentially read the member, the access method is termed "Indexed Sequential Access Method" (ISAM).

A TYPE-08 record can also have a 9-byte data area. In the PRO-PaDS utility available from MISOSYS, the ISAM directory entry record includes a three-byte subfield which contains the TRUE length of the member. The position of a member's logical end-of-file (EOF) can thus be calculated by adding its length to its position and adjusting for sector boundary alignment.

While you are looking at the first sector of SYS6, proceed to the first byte following the last ISAM directory record. You will observe the sequence:

```
0A 01 00 04 01 00 01 02 00 26 ...
```

The TYPE=0A indicates that it is the end of a PDS directory. The SYSTEM loader will return a "file not found" error if it reaches this record without finding a match of the ISAM number. The LENGTH=01 is needed because ALL load module records MUST have a length byte. The DATA area contains only a single arbitrary byte, X'00'. We cannot indicate a null record because a length byte of X'00' indicates 256 data area bytes. Thus, the X'0A' record type must have a minimum of one byte in its data area.

The following record is a TYPE=04 to indicate the end of a PDS member. This record serves but one purpose when used immediately following the directory - it will result in the return of a "Load file format error" if a library file is executed as if was a CMD file. When not expecting a partitioned data set file, the SYSTEM loader will ignore record types other than X'01' and X'02' except for the X'04'. The file reading will terminate at the X'04' with the above-mentioned error message.

The record type X'04' is usually used at the end of each partitioned data set member. If you list through SYS6, you will discover that each member ends with "04 01 00" rather than a TYPE=02 record. The system loader uses the X'04' type code in lieu of the transfer address code because the SYSTEM loader recovers the transfer address from the ISAM directory. Thus it needs to take action different from that when a standard load file has been completely loaded.

The next record types to discuss are those used in a generalized PDS file as exemplified in the PRO-PaDS utility. Such a file starts with a record type X'06' in lieu of an X'05' which is the normal header type for a load module. The first release of PRO-PaDS uses the X'06' in certain utility commands to note whether the referenced file is a partitioned data set compatible with PRO-PaDS utilities. The DOS does, in fact, make this information available known by setting a bit in the FCB when a PDS file is opened.

The PRO-PaDS partitioned data sets include a MEMBER DIRECTORY which correlates the member NAME with its associated ISAM entry number. A representative PDS MEMBER DIRECTORY entry looks like this:

```
0C 0B 64 69 72 20 20 20 20 20 01 01 7A 0C ...
 .  .  D  I  R                .  .  z  . ...
```

The TYPE=0C record indicates a PDS member directory entry record. The LENGTH byte specifies that the data area is an 11-byte field. The DATA AREA is subfielded as an 8-byte member name (stored in lower case), a one-byte ISAM entry number that is used to match up with a corresponding ISAM directory entry record, and a 2-byte field of member data. The first byte uses bit-7 to indicate a data member in contrast to an executable CMD program. Bit-6 indicates that the member has been established as "sector-origin" and can be directly accessed by linkage to the standard file access routines supported in PRO-PaDS Version 2. Bit positions 5-4 are reserved for future use. Bits 3-0 and the next byte contain the 12-bit DATE field formatted as in the standard directory entry record. This entry is the date that the member was added to the PDS. The end of the MEMBER DIRECTORY is indicated by a TYPE=0E record with its expected length and data field (as in "0E 01 00"). The purpose of this record is similar to the TYPE=0A record for the ISAM directory. It indicates the end of the MEMBER directory. The ISAM directory is positioned in the PDS to follow the MEMBER directory.

One last set of record types to discuss is the records associated with the PATCH utility. When you apply an X-patch to a file, the name of the patch file is used as a header name with a record type of X'07'. Thus, if you want to YANK the patch, the PATCH program can read through the file and search for a like-named header. If a matching header is found, PATCH will change the header record type to a X'09' to indicate a yanked patch. Also, since it may be impossible to remove the patch without bubbling up any code blocks following the patch (another patch maybe?), PATCH will change the TYPE=01 records to TYPE=10 records. The TYPE=10 records will not be loaded by the SYSTEM loader but will be

considered as non loadable comment records. It is thus possible to "un-yank" a yanked patch; however, this feature is not implemented in the PATCH utility.

8.4 ERROR MESSAGE DICTIONARY

Any time a SuperVisor Call experiences a malfunction, it returns an error code to the caller. The error codes possible are in the range <0-63>. The operating system associates a message string with each error code. Each string can be displayed or obtained via the @ERROR SuperVisor Call request. The words contained in the messages are stored in an error dictionary which is in a system overlay. This section of the appendix is a compilation of those error code messages and associated meanings.

Error 00: **No error**

A return code of zero indicates that there is no error.

Error 01: **Parity error during header read**

During a read request, the sector ID FIELD could not be satisfactorily read. Repeated failures would most likely indicate media or hardware failure.

Error 02: **Seek error during read**

During a read sector disk I/O request, a sector ID FIELD noting the requested cylinder was not located within the time period allotted by the controller. Either the cylinder is not formatted on the diskette, or the step rate designated is too low a value for the hardware to properly respond.

Error 03: **Lost data during read**

During a read sector request, the CPU was late in accepting a byte from the FDC data register and subsequently lost one of the bytes from the sector. For more information, consult the reference manual for the floppy disk controller used in your disk controller.

Error 04: **Parity error during read**

During a read request, the FDC sensed a CRC error. Possible media failure would be indicated. The Drive hardware could also be at fault.

Error 05: **Data record not found during read**

A disk sector read request was generated with a sector number not found on the cylinder referenced.

Error 06: **Attempted to read system data record**

A read request for a sector located within the directory cylinder was made without using the directory read routines. Directory cylinder sectors are written with a data address mark that differs from the data sectors data address mark. See chapter 4 and chapter 5 for additional information concerning address marks.

Error 07: **Attempted to read locked/deleted data record**

This error indicates that a request was entered which required a system overlay that had been purged from the system disk.

Error 08: **Device not available**

A reference was made for a logical device that either could not be located in the Device Control Blocks or the hardware associated with the device was not available (for example, a printer that was off-line).

Error 09: **Parity error during header write**

This is the same type of error as error-01 except that the operation requested was sector WRITE.

Error 10: **Seek error during write**

This is the same type of error as error-02 except that the operation requested was sector WRITE.

Error 11: **Lost data during write**

The CPU was not fast enough in transferring a byte to the FDC during a sector write request so it could be written to the disk. Therefore, one or more of the sector bytes were lost.

Error 12: **Parity error during write**

A CRC error was generated by the FDC during a sector write operation.

Error 13: **Data record not found during write**

This is similar to error-05. The sector number requested for the write operation, could not be located on the cylinder being referenced. Either the request is erroneous, or the cylinder is improperly formatted.

Error 14: **Write fault on disk drive**

This error message results when the disk controller returns a "write fault" error. Consult your FDC or HDC reference manual.

Error 15: **Write protected disk**

A write request was generated to a disk which either had a write protected diskette or the drive was write protected via software (see the SYSTEM (WP) DOS command). On 5-1/4" diskettes, covering the notch will protect the diskette from being written. On 8" media, exposing the notch will perform the same thing. If you want to write on a diskette, you must observe the proper notch condition.

Error 16: **Illegal logical file number**

A Directory Entry Code was referenced that was invalid for the referenced drive.

Error 17: **Directory read error**

Any disk error sensed during the reading of directory entry record sectors will result in this error. It could be media failure, hardware failure, or program crashes. The system's directory read accesses replace any lower level error (such as parity error) with this code.

Error 18: **Directory write error**

This error is similar to error-17 but the error condition is sensed while attempting to write a directory sector back to the disk. The integrity of the directory is now suspect.

Error 19: **Illegal file name**

The file specification provided to the system contains a character not conforming to the file specification syntax.

Error 20: **GAT read error**

Disk errors sensed while reading the Granule Allocation Table will cause this error. It could be media failure, hardware failure, or program crashes.

Error 21: **GAT write error**

This error is similar to the error-20 except that the error was sensed during a WRITE request. The integrity of the GAT is suspect.

Error 22: **HIT read error**

This error is similar to error-20 but occurred during a READ of the Hash Index Table.

Error 23: **HIT write error**

This error is similar to error-21 but occurred during a WRITE of the Hash Index Table.

Error 24: **File not in directory**

This error indicates that a file specification was referenced for OPEN that could not be located in the directory. Note that if the request was to LOAD a program load module file, the error code returned would be "Program not found". Most likely the cause was a misspelled filespec.

Error 25: **File access denied**

This indicates that an access request was made for a file that was password protected and the access protection level was NONE.

Error 26: **Directory space full**

An open of a new file was requested and the target disk either was not available or its directory was entirely in use. Use another diskette or remove uneeded files.

Error 27: **Disk space full**

While a file was being written, all available space on the disk was allocated before the file was completely written. Whatever space was already allocated to the file will still be allocated although the file's end of file pointer will not be updated. It may be useful to remove the file to recover the space after writing the file to another diskette.

Error 28: **End of file encountered**

The end of a file was reached during a read or position access. The file was probably smaller than the application expected. This error can also be used within an application to determine the end of a sequentially read file.

Error 29: **Record number out of range**

A request was made to read a sector of a file where the Next Record Number of the sector was beyond the Ending Record Number.

Error 30: **Directory full - can't extend file**

This error will result when the system must allocate an extended directory entry (FXDE) to a file because it has used all extent fields of its last directory entry record and no spare directory slot is available. All available directory entry records are in use. The solution would be to repack the disk by individually copying its files to a freshly formatted diskete.

Error 31: **Program not found**

The execution of a CMD program file could not be completed because the file was not located in the directory. Either the filespec was misspelled or the disk that contained the file was not mounted.

Error 32: **Illegal drive number**

This error will occur whenever a reference is made to a disk drive that is not included in your system. It may be disabled, or the drive requested was not ready for access (no diskette, drive door open, etc.).

Error 33: **No device space available**

This error will generally be returned by the SET command when you enter a request to establish a new device in the system and all of the resident system area reserved for Device Control Block tables is already in use. It is suggested that you use the "DEVICE (B=Y)" command to see if any currently defined non-system devices can be eliminated by using RESET.

Error 34: **Load file format error**

This error will be returned by the system loader when an attempt is made to LOAD a file that does not conform to the load module format structure. Most likely, the file referenced is a data file or a BASIC program file.

Error 35: **Memory fault**

This error indicates that a memory cell malfunctioned during the process of loading a program file.

Error 36: **Attempted to load read only memory**

This error would be returned if the program file being loaded referenced a memory cell that could not be altered. Either the cell was part of the read only memory (ROM), or the address was referencing an area of the machine not containing any read/write memory (RAM). Do not expect to see this error.

Error 37: **Illegal access attempted to protected file**

This indicates that an access request was made for a file that was password protected and the access protection level was not met for the request. Check if the disk is write protected.

Error 38: **File not open**

A file access operation was requested using a File Control Block that indicated a closed file. Most likely, there was a program error.

Error 39: **Device in use**

A request was made to REMOVE an active device from the Device Control Block table. It is necessary to first RESET a device before removing it.

Error 40: **Protected system device**

A request was made to REMOVE a standard system device. You cannot remove system devices such as *KI, *DO, *PR, *JL, *SI, and *SO.

Error 41: **File already open**

A request was made to open a file that was already open with an access level of UPDATE or greater. If you are in a single user environment and you know that the file is not open, you can reset the "open" indication by issuing a "RESET filespec" command.

Error 42: **LRL open fault**

This error indicates that a file was opened with a logical record length passed in the open linkage that differed from the file's LRL as stored in its directory. The file will be properly opened with the LRL passed in the open. This error is for information only.

Error 43: **SVC parameter error**

This error will be returned by a SuperVisor Call when one or more parameters associated with its register linkage contain invalid values.

Error 44: **Parameter error**

This error is returned by the parameter scanner when it detects in invalid command line parameter string. The error is usually caused by a misspelled parameter name or value, use of an unsupported abbreviation, or by entering a parameter that does not exist for the command invoked.

Errors 45-62: **Unknown error code**

Error codes in this range may not be defined by the operating system. Any time the @ERROR routine is called with an error number in this range, the "Unknown error code" message will be displayed. It most likely indicates a software problem.

Error 63: **Extended error**

This error code is used to indicate that an extended error code is in register pair HL. The @ERROR routine will display "** Extended error, HL = X'nnnn'" if called with error-63.

8.5 HEADER PROTOCOL OF MEMORY MODULES

A module of code can be relocated into high memory so that it's last byte is positioned at the value returned from the @HIGH$ SuperVisor Call. The module is then protected from being overwritten by other modules by adjusting HIGH$ to point to the address preceding the start of the module. Modules relocated and protected in this manner, must include a standard header that identifies the module. Modules placed into the low memory I/O driver region also must adhere to this standard. The header is used by the system to accomplish a number of important functions. First, it provides a locatable storage region for pointers used in the device independent library operations. Second, it provides a name string used by the @GTMOD SuperVisor Call to locate a specific module. Other data contained in the header provides the information needed to identify the entry address of each module so protected.

The following code describes this standard header:

```
ENTRY   JR      BEGIN           ;Branch around linkage
STUFHI  DW      $-$             ;To contain last byte used
        DB      MODDCB-ENTRY-5 ;Calculate length of 'NAME'
        DB      'MODNAME'       ;Name of this module
MODDCB  DW      $-$             ;To contain DCB pointer for module
        DW      0               ;Reserved by the DOS
;*=*=*
;       Area that can be used to store data
;*=*=*
        .
BEGIN   EQU     $
;*=*=*
;       Actual module code start
;*=*=*
```

Let's examine this module header line by line so that you gain an understanding of its purpose. At the label "ENTRY", the header always will have a relative jump instruction. The operand of the jump will almost always reference the starting address of your module. An exception to this would occur if the data area was extensive so that it placed the label "BEGIN" beyond the range of the jump relative instruction. If such was the case, you must provide an absolute jump (JP) instruction just prior to the data area. The address of this instruction will then be used as a reference in the operand field of the ENTRY jump relative.

It is also possible that the "module" is not a program but rather a data area that you have reserved. This data area must still have a memory header; however, since there exists no BEGIN address, it is recommended that you reference the operand of the ENTRY jump relative instruction so that it jumps to ENTRY (i.e. jumps to itself). This is the second exception.

The 2-byte storage region identified by the label STUFHI must be loaded with a value equal to the last memory address used by the module. The program routine that relocates the module into its memory position is responsible for loading this value. The system's @GTMOD routine uses the value to be able to branch sequentially from module to module. If the module is placed into high memory, this address value is the value returned by @HIGH$.

The next two fields of the header are the name LENGTH and NAME fields. The NAME field will contain the module's name as assigned by the programmer. This is the name string that is used in the @GTMOD SuperVisor Call to locate the module. The name must range from <1-15> characters in length and cannot have any character value below X'20'. The length of the name is then placed into bit positions 0-3 of the LENGTH field. The system uses the length value to determine how many characters must be matched in the NAME field. Bits 4-7 of the LENGTH byte are reserved by the operating system.

If the module is a device driver or filter, then it was assigned a Device Control Block when the driver or filter was invoked with the SET command. The SET command passes a pointer to this DCB in register pair DE when the initializing program first executes. It is the responsibility of the initializing program to load the DCB pointer into the 2-byte MODDCB storage field. The system requires this pointer for proper operation of its character I/O device chains.

The last 2-byte field is loaded with a binary zero. It's use is reserved by the operating system. You may conveniently use the memory region after this address for the storage of any data. Thus, the pointer returned from a successful @GTMOD search for the module will be easily used to index the data area.

8.6 INTERRUPT TASK PROCESSOR INTERFACING

The operating system is designed to function on hardware that can provide a maskable interrupt (mode 1). This interrupt can be generated either by a standard Clock Timer Chip (CTC) or it can be derived by other clocking methods (synchronized to the AC line frequency or decoded from some other frequency generator). An operating system Task Processor (TP) manages this interrupt to perform background tasks neccessary to perform specific functions of the DOS (such as the time clock where a hardware clock is not provided, blinking cursor where a CRTC blinking cursor is not provided, etc.).

The TP provides twelve individual TASK SLOTS that are executed on a "time-sharing" basis. The interrupt rate is software divided into three different timing groups spread across the task slots. One of these task slots is considered "high priority" and functions approximately 60 times a second (the exact time period depends on the interrupt rate provided). Three are considered "medium priority" and execute 30 times a second. The remaining eight are considered "low priority" and execute at a rate of 30/8 times a second (or 15 times every four seconds). The task task slots are numbered 0-11 with 0-7 being "low priority" tasks, 8-10 being "medium priority" tasks, and 11 being a "high priority" task.

The DOS maintains a Task Control Block Vector Table (TCBVT) which contains 12 vectors - one for each of the 12 possible task slots numbered from zero through eleven. Five system SuperVisor Calls that manage the task vectors are provided. These and their functions are:

@CKTSK = Check if a task slot is unused or active
@ADTSK = Add a task to the TCBVT
@RMTSK = Remove a task from the TCBVT
@KLTSK = Remove the currently executing task
@RPTSK = Replace the TCB address for the current task

The next point must be completely understood since it has caused confusion to many attempting to learn how to interface to the TP. The Task Control Block Vector Table (TCBVT) contains vector pointers. The TCBVT vectors POINT TO A 16-BIT LOCATION IN MEMORY WHICH CONTAINS THE ADDRESS OF THE SERVICING ROUTINE. Thus, the tasks themselves are twice indirectly addressed (those programmers familiar with C will observe that the TCBVT is an array of pointers to pointers). Make sure you keep this in mind! When you program an interrupt service routine, the entry point of the routine needs to be stored in memory. If we call this storage location the beginning of a Task Control Block (TCB), the reason for the indirect method of addressing interrupt tasks will become more clear. Let's illustrate an example TCB.

```
MYTCB     DW     MYTASK
COUNTER   DB     15
TEMPY     DS     1
MYTASK    RET
```

This is obviously an extremely useless task since all it does is return from the interrupt. However, note that a TCB location has been defined as "MYTCB" and this location contains the address of the task. A few more data bytes immediately following the task address storage have also been defined. Upon entry to an interrupt task service routine, index register "IX" will contain the address of the TCB. You, therefore, can address any TCB data using index instructions as in "DEC (IX+2)" which will decrement the value contained in "COUNTER". Let's expand the routine slightly.

```
MYTCB   DW      MYTASK
COUNTER DB      15
TEMPY   DB      0
MYTASK  DEC     (IX+2)
        RET     NZ
        LD      (IX+2),15
        RET
```

Here we have made use of the counter. Each time the task executes, the counter is decremented. When the count reaches zero, the counter is restored to its original value. This task still is pretty worthless for its function except for its illustration of data referencing. The big question is how does this task get added to the Task Control Block Vector Table (TCBVT)? We use the @ADTSK SuperVisor Call for that. Assuming we have decided that the task will be low priority, we must locate an unused low-priority task slot. We can see if slot 2 is available for use by invoking the @CKTSK SVC as follows:

```
        LD      C,2             ;Reference slot 2
        LD      A,@CKTSK        ;Identify the SVC
        RST     40              ;An "NZ" indication
        JP      NZ,INUSE        ;  says that the slot
                                ;  is being used.
```

Once you ascertain that the slot is available (i.e. not being used by some other task), you can add your task routine. The following code will add such a task to the TCBVT:

```
        LD      DE,MYTCB     ;Point to the TCB
        LD      C,2          ;Reference slot 2
        LD      A,@ADTSK     ;Identify the SVC
        RST     40           ;Issue the service call
```

We just point register "DE" to the TCB, load the task slot number into register C, then issue the @ADTSK SuperVisor Call. The task, most likely, would have been placed into high memory and protected by adjusting HIGH$ via the @HIGH$ SuperVisor call. The DOS has been designed to make specific use of bank-switched memory. The system's Task Processor will always enable bank zero when the TP takes control to perform background tasks. It restores the previously resident bank when it completes. This ensures that a single memory bank will consistently be available in high memory during interrupt task processing. In order to properly control and manage this additional memory, certain restrictions have been placed on tasks. Any and all tasks must be placed in either low memory (address X'0000' through X'7FFF') or in bank zero of high memory (address X'8000' through X'FFFF'). It is up to the assembly language programmer to ensure that tasks are placed in the correct memory area.

Once a task has been activated, it is sometimes necessary to deactivate it. This can be done in two ways. The most often way is to use the @RMTSK SuperVisor Call in the following manner:

```
        LD      C,2             ;Designate the task slot
        LD      A,@RMTSK        ;Identify the SVC
        RST     40              ;Invoke the service call
```

What could be more simple? We identify what task slot to remove by the value placed into register C, then issue the supervisor call. Another method can be used if we want to remove the task WHILE WE ARE EXECUTING IT. Consider the routine modified as follows:

```
MYTCB   DW      MYTASK
COUNTER DB      10
TEMPY   DB      0
MYTASK  DEC     (IX+2)
        RET     NZ
        LD      A,@KLTSK        ;Identify the SVC
        RST     40              ;Invoke the service call
```

The @KLTSK service routine will remove the currently executing task. Since this task is currently executing, it is the one that gets removed from the TCBVT table. The system will not return to your routine but will continue as if you had executed an "RET" instruction. Therefore, the "@KLTSK" SuperVisor Call should be the last instruction you want executed. In this example, MYTASK will decrement the counter by one on each entry to the task. When the counter reaches zero, the task will be removed from slot 2 (remember it was placed in slot 2).

One additional TP SuperVisor Call is @RPTSK. The function is easy to say in words; however, its function is best illustrated. The @RPTSK function will update the TCB storage vector (the vector address in your task control block) to be the address immediately following the @RPTSK SVC instruction. This is also another case where the system will NOT return to your task routine after the SVC is made but rather continues on with the TP. To illustrate how this TP function is used in a program, the final example should be examined:

First, let's point out that this task routine contains no method of relocating it to protected RAM. The statements starting at label, BEGIN, add the task to TCBVT slot zero (without checking for its availability) and return to DOS Ready. The task contains a four second down counter and a routine to put a character in video RAM (80th character of row 0). At four second intervals, the character toggles between '|' and '-'. The toggling is achieved by toggling the execution of two separate routines which perform the character display. Use is made of the @RPTSK TP call to implement the routine toggling. Examine this task closely to ascertain the functioning of @RPTSK.

```
BEGIN   LD      DE,TCB          ;Point to TCB & add the
        LD      C,0             ;  task to slot 0
        LD      A,@ADTSK
        RST     40
        LD      A,@EXIT         ;Exit to DOS
        RST     40
TCB     DW      TASK
COUNTER DB      15
TASKA   LD      A,@RPTSK        ;Replace current
        RST     40              ;  task with TASKA
TASK    LD      BC,2<8.OR.'|'   ;Put a '|' character
        LD      HL,0<8.OR.79    ;  at Row 0, Col 79
        LD      A,@VDCTL
        RST     40
        DEC     (IX+2)          ;Decrement the counter
        RET     NZ              ;  & return if not
        LD      (IX+2),15       ;  expired else reset
        LD      A,@RPTSK        ;Replace the previous
        RST     40              ;  task with TASKB
TASKB   LD      BC,2<8.OR.'-'   ;Put a '-' character
        LD      HL,0<8.OR.79    ;  at Row 0, Col 79
        LD      A,@VDCTL
        RST     40
        DEC     (IX+2)
        RET     NZ
```

```
        LD      (IX+2),15
        JR      TASKA
```

By firmly understanding the functions of each of the TP SuperVisor Calls discussed, you will be proficient at integrating interrupt tasks into your applications. A final note is to be aware of the task slots already used by the DOS or other applications. Use @CKTSK to find an unused task slot.

8.7 LOW MEMORY DETAILS

The author thought long and hard concerning the inclusion of this section of the Appendix. Why is this section a problem? The Version 6 operating system was designed to promote the development of portable software. The term, portable, means not only should the software function from machine to machine, it should also function under each release of the DOS. The DOS needs access to the storage of data for internal system use. Trying to keep the memory locations of this data constant across all implementations of the system is quite restrictive and usually becomes limiting to the healthy growth of the system. Keeping portability in mind, the designers of the system have provided SuperVisor Calls which return pointers to data that may be useful to a program. Thus, there should usually be no need to access data areas by memory address. We say "usually" since it is possible that user's of the system are writing machine-dependent SYSTEM code. This is the only reason that the Appendix contains this section. It is recognized that once a data address is known, application programmers tend to use it. **RESIST THE IMPULSE**. If the system does not provide via an SVC, data that you think you need, perhaps you don't really need the data. It is entirely possible that the information you need is actually available via an SVC, although not entirely obvious. Remember, when you bypass the SVC structure of the DOS, you most certainly risk portability!

With the preceding discussion in mind, let's first take a look at the general uses of each low core memory page.

Sector	Page	General Contents
n/a	0	RST vectors, Flag tables, misc...
n/a	1	SuperVisor Call Table
0	2	Bank data, 31 Device Control Blocks
1	3	System stack area, Miscellaneous machine dependent routines.
2	4	System Information data, Drive Control Table, Input buffer.
3	5	Start of I/O handling and drivers. Extends to end of page 12H.

The low core area starting at memory page two is actually loaded by and from the BOOT/SYS. The system uses the first two sectors to contain BOOT code needed to bring up the system. A booting ROM reads either the first or second sector of track 0 - the BOOT track. This sector contains code which, in turn, reads the entire BOOT/SYS file. Thereafter, BOOT loads the resident system file, SYS0/SYS, and transfers control to it. Because of this process, part of low memory is loaded directly from the BOOT/SYS file contained on track 0 while other parts of low memory are loaded by SYS0/SYS. A description of the booting process and the boot track is contained in another section of the Appendix. Let's now look at some of the details of low memory. REMEMBER THAT THIS INFORMATION IS PROVIDED FOR USE ONLY IN EXTREME NON-PORTABLE SITUATIONS!

An asterisk following the page byte(s) indicates a quantity that can be obtained from the system via some SuperVisor Call. A pound sign indicates that the address is fixed due to processor assignment.

8.7.1 Details of Low Memory Page 0

Bytes	Use
00-02#	RST 00 - Reserved for system use
03-04	reserved
05-07	reserved
08-0A#	RST 08 - Available to applications
0B-0C	SVCRET$- Return address from SVC invocation
0D	LSVC$ - Last SVC invoked
0E-0F	FDDINT$- Used by FDC driver for SYSTEM (SMOOTH)
10-12#	RST 16 - Available to applications
13-17	USTOR$ - User application storage area
18-1A#	RST 24 - Available to applications
1B	PDRV$ - Physical address of current drive
1C-1D	PHIGH$ - Physical high memory
1E-1F*	LOW$ - Lowest usable address of high memory
20-22#	RST 32 - Available to applications
23	LDRV$ - Logical address of current drive
24-25	JDCB$ - Saved FCB pointer
26-27	JRET$ - Saved I/O return address
28-2A#	RST 40 - System SVC call
2B	TIMSL$ - Time slice counter
2C*	TIMER$ - RTC counter [always precedes TIME$]
2D-2F*	TIME$ - Time string storage area
30-32#	RST 48 - DEBUG call address
33-37*	DATE$ - Date string storage
38-3A#	RST 56 - Maskable interrupt vector
3B*	OSRLS$ - DOS release number
3C	INTIM$ - Interrupt latch image
3D	INTMSK$ - Mask for INTIM$
3E-4D	INTVC$ - Table of 8 interrupt latch vectors
4E-65*	TCBVT$ - Table of 12 interrupt task vectors
66-68#	NMIVCT - Non-maskable interrupt vector
69*	OVRLY$ - Current system overlay resident
6A-83*	FLAGS$ - 26 system flags [A-Z] in order
84*	SVCTP$ - SVC table hi-order byte pointer
85*	OSVER$ - Operating system version
86-88*	@ICNFG - Initialization configuration vector
89-8B*	@KITSK - Keyboard task vector
8C-9F	SFCB$ - System file control block
A0-BF	DBGSV$ - DEBUG register save area
C0-DF	JFCB$ - JCL File Control Block
E0-FF	CFCB$ - Comand interpreter File Control Block

8.7.2 Details of Low Memory Page 1

Bytes	Use
00-FF*	SVCTAB$ - 128 vectors for SVC's 0-127

8.7.3 Details of Low Memory Page 2

Bytes	Use
00*	BUR$ - Bank Used RAM image
01*	BAR$ - Bank available RAM image
02*	LBANK$ - Currently resident RAM bank
03-05	JCLCB$ - Mini DCB for JCL line input
06-07*	DVRHI$ - First available byte in I/O driver region
08-0F*	KIDCB$ - Keyboard Input Device Control Block
10-17*	DODCB$ - Video Device Control Block
18-1F*	PRDCB$ - Printer Device Control Block
20-27*	SIDCB$ - Standard Input Device Control Block
38-2F*	SODCB$ - Standard Output Device Control Block
30-37*	JLDCB$ - Job Log Device Control Block
38-FF*	spare DCBs [25 of them]

8.7.4 Details of Low Memory Page 4

Bytes	Use
00	reserved
01	ZERO$ - set to X'00'
02-0D	MAXDAY$- [31,28,31,30,31,30,31,31,30,31,30,31]
0E-0F*	HIGH$ - Highest free address in user RAM
10-1F	reserved
20-6F*	INBUF$ - Command line input buffer
70-BF*	DCT$ - Drive Control Table records
C0-C6	reserved for use by system
C7-DB	DAYTBL - Days of the week [SunMon...]
DC-FF	MONTBL$- Months of the year [JanFeb...]

8.8 MEMORY BANK SWITCHING

This section discusses the techniques of using the @BANK SuperVisor Call. The control of an assembly-coded application operating in a memory banked environment requires a high degree of skill in assembly language coding and should not be undertaken by the novice. The professional is advised to carefully read the information contained in this section which discusses how bank switching is supported within the operating system.

The DOS can support eight multiple RAM banks of 32K each in addition to a resident 32K bank. This brings the total RAM configuration to 288K. The non-resident RAM banks are designated as banks zero through seven. The 32K of bank zero (generally considered as "high memory") and the resident 32K are considered the standard 64K of the DOS. Banks one through seven may be used for buffers or data storage. Through sophisticated techniques, they can even be used to store executable code. An entire bank is reserved for a particular function. The system maintains a pointer (HIGH$) for bank 0 only. At any one time, only one of the banks are resident. All are imaged at address X'8000' through X'FFFF'. When a bank transfer is performed, the specified bank becomes addressable and the previous bank is no longer available. Since memory refresh is performed on all banks, nothing in the previously resident bank is altered during whatever time it is not addressable (i.e. not resident).

The DOS provides support in accessing this additional RAM by means of the @BANK SuperVisor Call (SVC-102). Let's take a look at how this RAM is handled. When the operating system is booted, it examines what banks of RAM are installed in the machine. The DOS maintains a byte bit-map with each bit representing one of the banks of RAM. This byte is called "Bank Available RAM" (BAR), and its information is set when the DOS is booted. BAR bit-0 corresponds to bank 0, BAR bit-1 corresponds to bank 1, and so on to BAR bit-7 corresponding to bank 7. A machine may have only one bank, bank 0.

Another byte bit-map is used to indicate whether a bank is reserved or available for use. This byte is designated the "Bank Used RAM" (BUR). Again, a bit assignment corresponds one-for-one with the bank number. The management of any memory space within a particular bank of RAM (excluding bank 0) is the sole responsibility of the application program "reserving" a particular bank.

The DOS I/O device handler will always enable bank 0 upon execution of any character I/O service request (@PUT, @GET, @CTL, as well as those other character I/O SVCs that use @PUT/@GET/@CTL). The DOS also enables bank 0 at the initial entry to the task processor and when a disk I/O communications function is requested. This requires that any device driver or filter that is relocated to high memory (X'8000'-X'FFFF') must reside in bank 0. The same holds true for interrupt task routines and disk drivers/filters. The system provides this restriction to make sure that any filter, driver, or task routine that control passes to will be occupying enabled RAM memory. If a RAM bank other than 0 was resident during these operations, it would be restored upon return from the device/drive/task handler. The limitation will ensure that device I/O, task processing, and disk I/O will never be impacted due to bank switching of RAM by an application.

Another restriction requires that the stack pointer (SP) is not pointing to an adddress above X'7FFE' when a bank transfer is requested. This is because that stack range would have placed the stack in the memory region that is being swapped thereby making the stack contents erroneous. The @BANK SVC will inhibit the request and return an "SVC parameter error" if this condition is violated. It is acceptable for an interrupt task, filter module, or driver that is located in the bank switched address range to perform a bank transfer to another bank provided the necessary linkage and stack area is being utilized. This will be discussed later in more detail.

All bank transfer requests must be performed using the @BANK SVC. This SVC provides five functions - four of which are interogatory in nature. One of the functions performs

actual bank switching. As previously discussed, the contents of banks other than 0 are managed by the application - not the DOS. Therefore, the application first needs a way of ascertaining the availability of any given bank. For instance, if an application wants to reserve use of bank 1, it must first check if bank 1 is free to use. This is achieved by using function 2 as follows:

```
LD      C,1             ;Specify the bank #
LD      B,2             ;Ck BUR if bank-in-use
LD      A,@BANK         ;Identify the SVC
RST     40
JR      NZ,INUSE        ;NZ if in use already
```

Astute programmers will recognize that the first two instructions could be combined to form one instruction as:

```
LD      BC,2<8.OR.1
```

and save one-byte of code; however, for the sake of clarity in denoting the @BANK function codes, all remaining illustrations will use distinct instructions. Note that the return condition (NZ or Z) is entirely satisfactory for ascertaining whether or not you can use the specified bank or if it is not available for use. The accumulator contains no error code.

If you gain the availability of a specified bank, you then need to reserve it. This is done by using function 3 as follows:

```
LD      C,1             ;Specify bank-1
LD      B,3             ;Set BUR to show in-use
LD      A,@BANK         ;Identify the SVC
RST     40
JR      NZ,ERROR
```

You must check for an error by examining the Z-flag. In general, discounting a system error, an NZ condition returned means that the specified bank is already in use. In fact, if you had validly performed a function 2 (testing if the bank was available) and obtained a "not-in-use" indication but obtained an NZ condition on function 3, the @BANK SVC service routine has been altered and is most likely unusable.

Before actual bank switching is explained, let's look at one more function. When an application no longer requires a memory bank, it can return the bank to a "free" state by means of function 1. This is coded as follows:

```
LD      C,1             ;Specify bank-1
LD      B,1             ;Set BUR to show free
LD      A,@BANK         ;Identify the SVC
RST     40
```

No return code condition is checked as none is supplied by the DOS. In the unlikely event that you mistakenly invoke function 1 with a bank that is non-existent, you will still get an error if you try to later enable the non-existent bank.

If you need to ascertain what bank is resident at any point in time, use function 4 as follows:

```
LD      B,4             ;What bank's resident?
LD      A,@BANK         ;Identify the SVC
RST     40
```

The current bank number will be returned in the accumulator. This information may be useful prior to installing a driver/filter/task module into bank 0.

The more complex bank function is function 0. This request is used to actually exchange the current bank with the specified bank. A very important point to remember here is that since a memory transfer will take place in the address range X'8000' to X'FFFF', the transfer cannot proceed correctly if the stack pointer (SP) contains a value that places the stack in that range. In fact, @BANK will inhibit function 0 and return an SVC parameter error if the stack pointer violates the condition.

A bank can be used purely as a data storage buffer. Most likely, the application's routines for invoking and indexing the bank switching will reside in the user range X'3000' through X'7FFF' (or possibly in the I/O driver range). As an example illustration, the following code will invoke a previously tested and reserved bank (via functions 2 and 3), access the buffer, and then restore the previous bank:

```
        LD      C,1             ;Specify bank-1
        LD      B,0             ;Bring up bank
        LD      A,@BANK         ;Identify the SVC
        RST     40
        JR      NZ,ERROR        ;Whatever error trap
        PUSH    BC              ;Save old bank data
        .
        your code to access the buffer region
        .
        POP     BC              ;Recover old bank data
        LD      A,@BANK         ;Identify the SVC
        RST     40
        JR      NZ,ERROR        ;Whatever error trap
```

Note that the @BANK function 0 conveniently returns a zero in register B to effect a function 0 later, as well as provides the old bank number in register C. This means that you only have to save register pair BC, pop it when you want to restore the previous bank, and then issue the @BANK SVC.

Say you have a need to transfer to another bank from a routine that is executing in high memory. Can this be done? Notice that the only limitation discussed was that the stack must not be in high memory. The @BANK SVC function 0 does provide a technique for automatically transferring to an address in the new bank. This technique is termed the transfer function. It relies on the assumption that since you are managing the 32K bank, your application should know exactly where it needs to transfer (i.e. where the application originally placed the code to execute). The code to perform a bank transfer is similar to the above. Register pair HL must be loaded with the transfer address, Register C, which contains the bank number to invoke, must have its high order (bit-7) set to indicate the TRANSFER activity. After the specified bank is enabled, control is passed to the transfer address that was in HL. Upon entry to your routine in the new bank (we will refer to it as "PROGRAM-B"), register HL will contain the old RETurn address so that PROGRAM-B will know where to return to when it transfers. Register C will also contain the old bank number with bit-7 set and register B will contain a zero. This register setup will provide for an easy return to the routine in the old bank that invoked the bank transfer. An illustration of the transfer code is as follows:

```
        LD      C,1             ;Specify bank-1
        LD      B,0             ;Bring up bank 0
        LD      HL,(TRAADR)     ;Set the transfer address
        SET     7,C             ;  & denote a transfer
        LD      A,@BANK         ;Identify the SVC
        RST     40
RETADR  JR      NZ,ERROR
```

Control will be returned to "RETADR" under either of two conditions. If there was an error in executing the bank transfer (for instance an invalid bank number or the stack pointer being in high memory), the returned condition will be NZ. If the transfer took place and PROGRAM-B transferred back, the returned condition will always have the Z-flag

set. Thus, the Z-flag will be indicative of a problem in effecting the transfer. If, by chance, PROGRAM-B needs to provide a return code, it must be done by using register pair DE, IX, or IY, as registers AF, BC, and HL are used to perform the transfer (or some other technique such as altering the return transfer address to a known error trapping routine).

PROGRAM-B should contain code that is very similar to that shown earlier. For example, PROGRAM-B could be:

```
PROGB   PUSH    BC              ;Save old bank data
        PUSH    HL              ;Save the RET address
        .
        your PROGRAM-B routines
        .
        POP     HL              ;Recover transfer address
        POP     BC              ;Get bank transfer data
        LD      A,@BANK         ;Identify the SVC
        RST     40
        JR      NZ,ERROR        ;Whatever error trap
```

PROGRAM-B saves the bank data (register BC). Don't forget that a transfer was effected and register C has bit-7 already set when PROGRAM-B is entered.

PROGRAM-B also saves the address it needs to transfer back (which is in HL). It then performs whatever routines it has been coded for, recovers the transfer data, and issues the bank transfer request. As explained earlier, an NZ return condition from the @BANK SVC indicates that the bank transfer was not performed. A recommendation is to verify that your application has not violated the integrity of the stack where the transfer data was stored.

Never place disk drivers, device drivers, device filters, or interrupt task service routines in banks other than bank-0. It is possible to segment one of the above modules and place segments in banks 1 through 7 provided the segment containing the primary entry is placed in bank 0. All three types of divisions are incorporated into the system's spooler with transfer between segments being accomplished by the bank transfer techniques discussed above.

It sometimes is necessary to transfer a page of memory from one bank to another. This can only be done in one of two ways. Either a character(s) at a time is passed in a register(s) or a page buffer below X'8000' is used. The system uses the last page of the system overlay region (X'2300'-X'23FF') as an overlay buffer (except for SYS5/SYS which loads into the region). This buffer is generally available for use as a page transfer buffer. Do not use this location if your memory transfer routine is a background task or is using the RAM bank as a disk cache buffer.

8.9 INTERFACING TO @KITSK

Consider for a moment that disk I/O can not take place during an interrupt task. How then can we write "background" routines that perform disk I/O? The system printer spooler does its despooling function as a background task. If we cannot perform disk I/O during interrupt tasks, how can we despool? We achieve this by being able to invoke a background task in a way that does not depend on the interrupt task processor. A function frequently requested in almost every application is that of obtaining characters from the keyboard. If we can "hook into" this keyboard request, we can execute a task every time the keyboard is scanned. For those tasks that require disk I/O, we can make use of this keyboard task process.

At the beginning of the system keyboard driver code is a call to @KITSK. This means that any time that @KBD is called, the @KITSK vector is likewise called (actually, the type-

ahead interrupt task bypasses this entry to inhibit calling @KITSK from the interrupt routine). Therefore, if you want to interface a background routine that does disk I/O, you must chain into @KITSK

The interfacing procedure to @KITSK is virtually identical to that shown for Boot Initialization ICNFG Interfacing (except that FLAGS+31 through FLAGS+33 is used to reference the @KITSK vector) and will not be repeated here. For the sake of clarity, you may want to write your background routine to start with:

```
START   CALL    ROUTINE         ;Invoke task
LINK    DB      'Roy'           ;For @KITSK hook
ROUTINE EQU     $               ;Start of the task
        .
        RET
```

Now that the procedure has been demonstrated, be aware of one major pitfall. The @KBD routine is invoked from @CMNDI and @CMNDR which is in SYS1/SYS. This invocation is from the @KEYIN call which fetches the next command line after issuing the "DOS Ready" message. If your background task executes and opens or closes a file (or does anything to cause the execution of a system overlay other than SYS1), then SYS1 will be overwritten by that system module handling your request). When your routine finishes, the @KEYIN handler returns to what called it - which was SYS1. Unfortunately, SYS1 is no longer resident. You have just crashed the system!

ANY TASK CHAINED TO @KITSK WHICH CAUSES A RESIDENT SYS1 TO BE OVERWRITTEN MUST RELOAD SYS1 PRIOR TO RETURNING.

Okay, how do you accomplish this without knowing system code (point of information: if you are writing background tasks, you are writing system support code!)? You will be able to use the following code to reload SYS1 if SYS1 was resident prior to your task's execution.

```
ROUTINE LD      A,@FLAGS        ;Get flags pointer
        RST     40              ;  into register IY
        LD      A,(IY-1)        ;P/u resident over-
        AND     8FH             ;  lay and remove
        LD      (OLDSYS+1),A    ;  the entry code
        .
        Rest of your task
        .
EXIT    EQU     $
OLDSYS  LD      A,0             ;P/u old overlay #
        CP      83H             ;Was it SYS1?
        RET     NZ              ;Return if not else
        RST     40              ;Get SYS1 per reg A
                                ;  (no RET needed)
```

Another method is to determine if the keyboard request originated from the command interpreter. While the command interpreter is fetching its command line via @KEYIN , it sets bit-2 in the CFLAG$ (see @FLAGS SuperVisor Call). Thus, if your KITSK routine discovers that bit set, then the command interpreter originated the line input. If you cause the system to load some other overlay into the system overlay region, it is your responsibility to restore SYS1!

8.10 SYSTEM DISK BOOT TRACK

The operating system goes through a complicated process to bring itself to a "ready" state. This process is known as BOOTING. All implementations of the DOS require that the machine contain a small routine in Read Only Memory called the BOOT ROM. The operating system uses the first two sectors of track zero of the system disk to contain BOOT code needed to bring up the system. The BOOT ROM has the small job of reading either the first or second sector of track zero, the BOOT track. The track contains a core-image file called BOOT/SYS. The sector that is read contains code which, in turn, reads the entire BOOT/SYS file into low memory starting at memory page 2. The BOOT/SYS file occupies 16 sectors of track zero. Thereafter, BOOT/SYS loads the resident system file, SYS0/SYS, and transfers control to it. SYS0/SYS contains additional code which performs further system initialization. This includes loading the first two pages of memory (page 0 and page 1), loading any system configuration file (CONFIG/SYS), and executing any AUTO command. Because of this process, part of low memory is loaded directly from the BOOT/SYS file contained on track 0 while other parts of low memory are loaded by SYS0/SYS.

The BOOT/SYS file contains two things of limited importance to programmers. First, BOOT/SYS contains a pointer to the cylinder which holds the disk's directory . Second, the BOOT/SYS contains system information in one of its sectors called the SYSTEM INFORMATION SECTOR. It is necessary to discuss only these two items.

The DIRECTORY CYLINDER POINTER is a one-byte pointer that exists as the third byte of both sector zero and sector one. Both locations store this information in order to be media compatible across various implementations of the operating system. Hard disk formatters that perform their own initialization of the directory cylinder must store the logical cylinder number of the directory in these two pointers. The pointer is the only byte of the first two sectors that requires attention.

The SYStem INFOrmation sector (SYSINFO) is sector two of track zero. It contains various pieces of system information as follows:

Bytes	Use
00	Operating system version used when formatting the disk. This number is in hexadecimal (i.e. X'60', X'61', etc...)
01	Configuration byte to specify if a booting disk contains a CONFIG/SYS file [X'C9'=NO, X'00'=YES]
02-1D	MAXDAY$ [31,28,31,30,31,30,31,31,30,31,30,31]
0E-0F	reserved
10-17	Disk Pack name - same as in Granule Allocation Table
18-1F	Disk Pack date - same as in Granule Allocation Table
20-6F	80-character storage area for the AUTO command. This means that the AUTO command buffer on the disk loads directly into INBUF$ by the BOOT loader.
70-BF	Drive Control Table (DCT) records.
C0	Disk type (system=X'FF', data=X'00')
C1	reserved
C2	System BOOT date prompting [X'00'=YES, X'FF'=NO]
C3	System BOOT time prompting [X'00'=YES, X'FF'=NO]
C4	System BOOT floppy disk restores [X'00=NO, X'FF'=YES]
C5	reserved
C6	reserved
C7-DB	Days of the week [SunMon...]
DC-FF	Months of the year [JanFeb...]

8.11 SYSTEM OVERLAY CONTENTS

A system as complex and flexible as LDOS would occupy considerable memory space to be able to provide all of its features. The DOS, however, makes extensive use of overlay segments in order to minimize the amount of memory reserved for system use. The compromize in using an overlay driven system, is that while a user's application is in progress, certain disk file activities requested of the system may require the operating system to load different overlays to satisfy the request. This could cause the system to run slightly slower than a less sophisticated system which has more of its file access routines always resident in memory. The system provides a procedure to permanently place specified overlays into memory to enhance the overall speed of operation (see the SYSTEM command).

The following will describe the functions performed by each system overlay. Numbers in angle brackets represent the system SVC entry.

8.11.1 SYS0/SYS

This is not an overlay. It contains the resident part of the operating system (SYSRES).

8.11.2 SYS1/SYS

This overlay contains the command interpreter. This processes @CMNDI <X'B3'> and @CMNDR <X'A3'>. It contains the routines for processing the @FEXT SVC <X'D3'>, the routines for processing the @FSPEC SVC <X'C3'>, and the routines for processing the @PARAM SVC <X'E3'>. It also contains the @EXIT processor <X'93'>.

8.11.3 SYS2/SYS

This overlay is used for opening <X'94'> or initializing <X'A4'> disk files and logical devices. It contains the functions for @RENAM <X'F4'> and @GTDCB <X'B4'>. It also contains the @CKDRV routines <X'C4'>, and routines for hashing file specifications <X'D4'> and passwords <X'E4'>.

8.11.4 SYS3/SYS

This overlay contains all of the system routines needed to close files and devices <X'95'>. It also contains the routines needed to service the @FNAME SVC <X'A5'>.

8.11.5 SYS4/SYS

This system overlay contains the system error dictionary and @ERROR SVC processing routines.

8.11.6 SYS5/SYS

This overlay contains the system debugger.

8.11.7 SYS6/SYS

This overlay contains all of the algorithms and routines necessary to service the LIBrary commands identified as "Library A" by the LIB command. The following list identifies the commands and their ISAM entry number.

21 DIR	63 RESET	53 RENAME
61 DEVICE	65 SET	1E MEMORY
32 COPY	66 FILTER	91 DO
31 APPEND	41 LIST	81 LOAD
64 ROUTE	18 REMOVE	82 RUN
62 LINK	19 LIB	

8.11.8 SYS7/SYS

This overlay contains all of the algorithms and routines necessary to service the LIBrary commands identified as "Library B" by the LIB command. The following list identifies the commands and their ISAM entry number.

```
14 DEBUG            72 PURGE
1B VERIFY           71 DUMP
15 DATE             13 CREATE
16 TIME             11 AUTO
22 FREE             33 BUILD
51 ATTRIB
```

8.11.9 SYS8/SYS

This overlay contains all of the algorithms and routines necessary to service the LIBrary commands identified as "Library C" by the LIB command. The following list identifies the commands and their ISAM entry number.

```
A1 SYSTEM
1C SYSGEN
B1 FORMS
B2 SETCOM
B3 SETKI
A2 SPOOL
```

8.11.10 SYS9/SYS

This overlay contains the routines necessary to service the EXTended debugging commands available after a DEBUG (EXT) is performed.

8.11.11 SYS10/SYS

This system overlay contains the procedures necessary to service the request to REMOVE a file <X'9C'>.

8.11.12 SYS11/SYS

This overlay contains all of the procedures necessary to perform the Job Control Language execution phase. These are the initial entry for setup and initialization <X'9D'>, the revised @EXIT processor <X'AD'>, keyboard request processing <X'CD'>, and //INPUT keyboard processing <X'DD'>.

8.11.13 SYS12/SYS

This overlay contains the routines to service the @RAMDIR <X'9E'> and the @DODIR <X'AE'> SuperVisor Calls. It also includes the routines to service the @GTMOD function <X'BE'>.

8.11.14 SYS13/SYS

Effective with release 6.2.0, SYS13 can be used by an application environment for an Extended Command Interpreter (ECI). This ECI gains control from SYS1 on any of the following SVCs: @ABORT, @CMNDI, @CMNDR, and @EXIT. The programmer develops the ECI and copies it to the application system disk SYS13/SYS module via the command:

```
COPY usereci SYS13/SYS.LSIDOS:d (C=N)
```

The programmer then sets the EFLAG$ and invokes SYSGEN to save the EFLAG$ configuration. Upon entry to the ECI, the registers will be set as for any other program execution (see page 6-100), with the exception of register A. Bits 4-6 of the accumulator will contain an image of the respective EFLAG$ bits. The ECI programmer may use different EFLAG$ assignments in a multiple module application environment to invoke the ECI with different entry points.

8.12 SYSTEM OVERLAY ACCESS

Practically all of the functions contained in the system overlays are accessed via library commands or standard SuperVisor Calls. Only in a few unique cases is access to overlay functions through the SYSTEM SVC required. The two cases, calculating the file specification hash code and the password string hash code, have been discussed. The system functions provided in the overlays will usually have a standard SuperVisor Call assigned. These SVCs have been discussed in chapter 7. The system translates standard SVC numbers <0-127> within SYSRES to the overlay entry number in order to process the user request. Although it is possible to directly access a function via its overlay entry number or ISAM entry number, this should not be done. The standard SVC linkage protocol should be used to address the overlay functions since there is no guarantee that the routines servicing the overlay functions will remain in the overlay presently assigned.

A user SVC request is via a RST 40 instruction which places the return address at the top of the stack. Since the process to translate the user request to a system overlay request also uses a RST instruction (to minimize the length of the translation code), an extra return address is placed on the stack. The SVC processor adjusts for this by popping the extraneous return address when it is processing a system overlay request. The system's request is easily identifiable since all system request codes have bit-7 set. Because of this, if a user requests a system overlay function directly, it is necessary to CALL the RST instruction so that the return address that is kept on the stack is a pointer to the address following the CALL instruction. System overlays one through five and nine through thirteen, can be loaded into the overlay region by means of the following code:

```
        LD      A,8<4.OR.#+2    ;The "#" represents the
        CALL    RST40           ;  number of the overlay
        .
        .
RST40   RST     40              ;Returns to what called this
```

For a specific example, in order to load SYS3/SYS, the accumulator will be loaded with the value, X'85'. When one of these overlays loads, the last two bytes of the system overlay region will be loaded with the length of the overlay. This information is used by the "SYSTEM (SYSRES)" command.

The library overlays, SYS6/SYS, SYS7/SYS, and SYS8/SYS, are partitioned data sets. The system locates the origin of individual members by means of an ISAM directory. The directory contains an entry number, a NRN-BYTE OFFSET pointer, and a transfer address (this is discussed in the appendix section, DISK LOAD MODULE FORMAT). When the command interpreter recognizes a library command request, it obtains the ISAM entry number from its table and issues a system overlay request. The ISAM entry number is placed in register B while the accumulator contains the corresponding overlay load code as discussed in the preceding paragraph. Again, since it is possible for the members to be located in a different overlay in the future, the proper method to invoke a library overlay member is via an @CMNDR or @CMNDI SuperVisor Call.

8.13 USING @PARAM

The @PARAM SuperVisor Call is used in practically all DOS library commands and utilities as well as filters, drivers, and languages. Since you are already familiar with the DOS commands, you should recognize the wide range of input syntax parsed and interpreted by @PARAM. The SVC is used to decode TRUE/FALSE parameters (by either entering or not entering a parameter word), YES/NO parameters (by using PARM=Y or PARM=N), ON/OFF parameters (by using PARM=ON or PARM=OFF), decimal values (by entering PARM=ddddd), hexadecimal values (by entering PARM=X'xxxx'), and character string values (by entering PARM="characterstring"). Parameter entries can be made in either upper case or lower case - even with hexadecimal digits (A-F equally acceptable as a-f).

The system parses a complex parameter string that may be composed of many parameters - each separated from the other by a comma. The interpreted entries are passed back to @PARAM caller according to the parameter table designed by the programmer. Version 6 supports two types of parameter tables. The first type is the fixed width table which was supported under Version 5.

The second type is a variable width table that supports additional information. In the following discussions, we will first illustrate the former table. You should have already read the information in chapter 7 covering the @PARAM SuperVisor Call.

Let's assume we have an application that offers the user varying options to set up the function of the application. In BASIC, this may be the number of files or protected memory size. In BACKUP, this may be the diskette master password or date range of files to select. In SETCOM, this may be whether CTS is to be honored. How do we get this information to the program? We could prompt the user by a prompt message for each and every parameter that needs to be determined. Experienced users soon get tired of prompts. Inexperienced users get extremely frustrated when the system requires an inflexible syntax for the entry of options. How can everyone be satisfied - from novice to expert? Why, by using @PARAM.

We will propose a hypothetical application requiring the determination of five options:

1. A length field used in ascertaining the number of print columns of output. This should default to 80 to denote an 80 column printer if no entry is made. The range should be limited to 32-255.
2. A module specification field to indicate whether line feeds are to be added after carriage return, removed after carriage return, or no checking is to be performed.
3. A title field to be placed on each page of output. In addition, paging is to be suppressed if no titling is desired. Furthermore, the default is to incorporate paging unless otherwise specified by the user.
4. A prompting specification to note whether prompts for changing paper are to be made at the appropriate time if sheet paper is used or omitted if tractor feed paper is used. The default should be no prompting.
5. A translation option for converting a character on output. This should default to no translation.

The first thing required by the system designer is to designate "words" for the command line parameters. They should be chosen to be as easily remembered as possible. They should be greatly correlated in definition to the function they are specifying. Additionally, abbreviations should be considered in addition to the full "word". Thought should be given to using words whose first character is different for each parameter so as to provide single character abbreviations. However, if any parameter is omnipotent, care should be exercised in designating an abbreviation.

In the example above, we will choose LENGTH, FEED, TITLE, PROMPT, and XLATE parameter words for the options 1-5. We will also abbreviate these as L, F, T, P, and X. Your application's documentation must fully explain the purpose of the parameters. A typical command line entry could be:

```
URPROG (length=132,title="Program Guide",xlate=x'0e00')
```

The command line could just as easily have been entered as:

```
URPROG (t="Program Guide",x=x'0e00',l=132)
```

Note that not only are abbreviations used, but the order of appearance in the command line is irrelevant. Also note that parentheses enclose the command line parameters; however, the closing parenthesis is not required. You can take some liberties with the string and hexadecimal syntax. Hexadecimal entries can drop the closing single quote. Strings are considered terminated by any value less than SPACE. Thus, a closing carriage return validly terminates a string. This leeway permits entry of such command lines as:

```
URPROG (t="Program Guide
URPROG (x=x'0e00,t="Program Guide
```

You're saying there must be a catch. How can @PARAM do all that? Easy - you must follow some rules and implement some coding in your program. Not very much coding is required, though. When you execute a command line, the command interpreter is activated (@CMNDI). If a LIBRARY name is specified, the system's library module is activated. If a program name is entered (the system first tries a default extension of /CMD if the user does not supply one) the program will be loaded and transfer will be performed to the program's transfer address which is located at the end of the load module (following the X'0202'). When control is passed to the program, register pair HL contain the address of the first non-blank character following the program name entered. If @PARAM is requested, it will search the command line for a parameter string left parenthesis starting from the address pointed to by HL. It will ignore blanks while it looks for the "("; however, if it finds a non-blank character other than "(", it will immediately return. If there are going to be additional entries, such as file specifications, on the command line preceding possible parameters, these must be parsed first by your program before issuing the @PARAM SVC.

The prologue of URPROG might go something like this:

```
URPROG  PUSH    HL              ;Hang on to INBUF$ pointer
        LD      HL,HELLO$       ;Point to hello message
        LD      A,@DSPLY        ;Display message to screen
        RST     40
        POP     HL              ;Recover INBUF$ pointer
        LD      DE,PRMTBL$      ;Point to parameter table
        LD      A,@PARAM        ;Go parse all of the parms
        RST     40
        JP      NZ,PRMERR       ;Go to error handler if bad entry
        .
        .                       ;The rest of URPROG
        .
HELLO$  DB      10,'Some friendly message',CR
;*=*=*
;       This is the parameter table. Note its entries are
;       all 6-characters in width. The address specified by
;       the parameter vector follows each parameter "word".
;       In addition, the table is ended with a zero byte.
;*=*=*
PRMTBL$ DB      'LENGTH'        ;Length parameter
        DW      LPARM+1
        DB      'L     '
```

```
        DW      LPARM+1
        DB      'FEED  '        ;Line feed parameter
        DW      FPARM+1
        DB      'F     '
        DW      FPARM+1
        DB      'TITLE '        ;Title parameter
        DW      TPARM+1
        DB      'T     '
        DW      TPARM+1
        DB      'PROMPT'        ;Prompt parameter
        DW      PPARM+1
        DB      'P     '
        DW      PPARM+1
        DB      'XLATE '        ;Translate parameter
        DW      XPARM+1
        DB      'X     '
        DW      XPARM+1
        NOP                     ;This is the ending zero byte
```

The PRMTBL$ is going to be structured similarly for all tables. The convention used of specifying the address vector as "LABEL+1" will become immediately obvious once you inspect the method of using the result in URPROG. As an aside, let's look at two conventions of referencing the second byte of a three-byte instruction.

```
METHOD1 LD      (LABEL+1),HL    ;Load HL into the "nn" field
        .
        .
        .
LABEL   LD      BC,0            ;P/u the value loaded


METHOD2 LD      (LABEL),HL      ;Load HL into the "nn" field
        .
        .
        .
        LD      BC,0            ;P/u the value loaded
LABEL   EQU     $-2             ;The "nn" field is 2-bytes back
```

The first method will be used to illustrate parameter table vector addresses in this appendix section. Use the method you are most comfortable with. It is suggested that you choose one technique and use it exclusively throughout a program. Otherwise you will find yourself getting into trouble as you forget which method you were using.

Now that the @PARAM system function has parsed the entered command line, how do we utilize the "values" it interpreted while still supporting our defaults and conditions? Well, bear in mind that if the user has not entered a parameter word, nothing will be entered by @PARAM into the address vector specified by the parameter table. Therefore, an initial condition can be supplied in the coding. Also, the initial value coded will be dependent on just what condition you want the default to be. Let's see how this would work.

```
        .
        .               ;Some front end code
        .
;*=*=*
;       Here is where we pick up the length parameter. Note
;       that it is initialized to 80 if there is no user entry
;*=*=*
LPARM   LD      BC,80           ;Pick up the entry
        INC     B               ;Test hi-order for zero
        DEC     B               ;It must be zero for range check
        JP      NZ,LBAD         ;Bad length if range >255
        LD      A,C             ;P/u the lo-order length
```

```
        CP      32              ;Must be >= 32
        JP      C,LBAD          ;Bad length if range < 32
;*=*=*
;       The length parameter has been tested for proper range.
;       It can be used in URPROG where needed by either stuffing
;       the accumulator where needed or by picking up the value
;       later by a "LD   A,(LPARM+1)" instruction.
;*=*=*
        .
        .
        .
        .
;*=*=*
;       Here is where we pick up the line feed parameter. Based
;       on the conditions specified, we need a three-way test.
;       What has to be ascertained is whether the user specified
;       FEED=ON, FEED=OFF, or didn't even enter FEED. The ON/OFF
;       entries are the same as TRUE/FALSE specifications and
;       result in a -1/0 value respectively (ON = -1, OFF = 0).
;       We therefore must define a default value which is
;       neither 0 nor -1.
;*=*=*
FPARM   LD      BC,1            ;We will use a "default" of 1
        LD      A,B             ;Merge the hi and lo orders
        OR      C
        JR      Z,RMVFEED       ;Remove line feed if FEED=OFF
        INC     A               ;If FEED=ON was specified, A=X'FF'
        JR      Z,ADDFEED       ; thus A would be zero after the INC
;*=*=*
;       The line feed parameter has now been handled. It is left
;       up to the reader to provide routines for RMV and ADD FEED.
;*=*=*
        .
        .
        .
        .
;*=*=*
;       The title parameter needs to default to ON per our
;       conditions. This would mean that if no TITLE was
;       supplied in the command line, the user would be prompted
;       to enter it (user friendly). On string parameters,
;       @PARAM will load the address of the first character of
;       "string" into the vector address specified in PRMTBL$.
;       URPROG will then have to parse the string until it finds
;       one of the string terminating characters.
;*=*=*
TPARM   LD      BC,-1           ;Force the default to be TITLE=Y
        LD      A,B             ;Check on entry of T=N
        OR      C               ;Merge hi and lo orders
        JR      Z,NOTITLE       ;To user provided routine
        INC     A               ;Check if T=Y or no entry
        LD      HL,PMTITL$      ;Init pointer just in case
        CALL    Z,GETITLE       ;Go prompt & get title if only T=Y
;*=*=*
;       The GETITLE routine would have to display the prompt,
;       provide an input means, then place the address of the
;       first character of string into register pair BC.
;       Otherwise, reg BC already has the address of that char.
;*=*=*
        .
        .                       ;Your routine for parsing the title
        .                       ;character string belongs here.
        .
;*=*=*
;       The prompt parameter will be an easy one. Its default is
;       PROMPT=OFF and no other special conditions need be met.
```

```
;*=*=*
PPARM   LD      BC,0            ;Zero because the default is OFF
        LD      HL,FLAG$        ;Let's set a flag for this one
        RES     0,(HL)          ;Init flag to off
        LD      A,C             ;Only the lo-order is needed
        OR      A               ;Test the entry
        JR      Z,$+4           ;Skip the next instruction if P=N
        SET     0,(HL)          ;Set the flag if P=Y
        .
        .
        .
;*=*=*
;       The translation parameter is the last one to retrieve.
;       In order to provide a default of no translate character,
;       the code will use a zero value for this test. It is
;       important to note that since the entry is a 16-bit
;       value, your documentation must clearly note which order
;       is the character to test. If in X'xxyy', we denote "xx"
;       for the test character and "yy" its translated value,
;       then "yy" becomes the lo-order byte when loaded while
;       "xx" becomes the hi-order byte.
;*=*=*
XPARM   LD    BC,0          ;Note the zero default
;*=*=*
;       That's all there is to it. We could, of course, test
;       for an X'0000' value and set a flag to indicate no XLATE
;       option entered. Then later test the flag first before
;       checking on a XLATE match. However, it would probably take
;       just as long to test for the option as it would to
;       test for the character so we will not use a flag.
;*=*=*
        .
        .
        .
;*=*=*
;       Here is some code that could use the translate feature
;       The character is in the accumulator.
;*=*=*
        LD      BC,(XPARM+1)    ;P/u the test characters
        CP      B               ;Translate this character?
        JR      Z,$+3           ;If match, use translate
        LD      C,A             ;  else use this character
        LD      DE,PRDCB$       ;Point to Device Control Block
        LD      A,@PUT          ; and put the character
        RST     40
        .
```

Sometimes, you may want to provide a parameter that can be entered either as a decimal value, a hexadecimal value, or as a string value. For instance, if you want the user to optionally assign a "separator" character which defaults to a semicolon, it would be very friendly to accept any of the following: [sep=X'3A', or sep=58, or sep=":"]. The decoding can get involved. When the program is expecting a 16-bit value, if we would closely inspect the decoding of the parameter entry, we would find that there is difficulty in differentiating a string parameter which returns a 16-bit address from a decimal or hexadecimal value. Another observation is that while the inclusion of abbreviations for the parameter words is both recommended and desirable, it requires duplicate entries in the parameter table. These entries waste memory space. The second parameter table format solves these problems. First, the system provides feedback as to the type of entry contained in the parameter command string: switch (yes/no, true/false, on/off), value (16-bit decoded decimal or hexadecimal entry), or string (start address and length). In addition, each parameter word can be a different length while single character abbreviations are specified within the one table entry. Let's take a look at our parameter table if it were recoded into the second format.

```
VAL     EQU     80H             ;Set value bit
SW      EQU     40H             ;Set switch bit
STR     EQU     20H             ;Set string bit
ABR     EQU     10H             ;Set abbreviation bit
;
PRMTBL$ DB      80H             ;Indicate format 2
;
        DB      VAL.OR.ABR.OR.6
        DB      'LENGTH'        ;Length parameter
LRESP   DB      0
        DW      LPARM+1
;
        DB      SW.OR.ABR.OR.4
        DB      'FEED'          ;Line feed parameter
FRESP   DB      0
        DW      FPARM+1
;
        DB      STR.OR.ABR.OR.5
        DB      'TITLE'         ;Title parameter
TRESP   DB      0
        DW      TPARM+1
;
        DB      SW.OR.ABR.OR.6
        DB      'PROMPT'        ;Prompt parameter
PRESP   DB      0
        DW      PPARM+1
;
        DB      VAL.OR.STR.OR.5
        DB      'XLATE'        ;Translate parameter
XRESP   DB      0
        DW      XPARM+1
        NOP                     ;This is the ending zero byte
```

When the @PARAM service function completes its parsing and interpreting of the parameter command string, the response byte corresponding to parameter entries will be altered according to any entry parsed. Thus, your program can incorporate code to test the response byte to determine the exact type of entry made in the parameter line. By comparing the response byte to the control byte, the program can ascertain the validity of the entry. It is left for the reader to adjust the decoding routines according to table format 2.

8.14 TRAP Filter Illustrated

```
;TRAP/ASM - Filter to trap a single character - 07/31/83
;
        COM     '<Copyright 1983 by Roy Soltoff>'
;*=*=*
;       This FILTER will trap a single character
;       as specified by the command line entry.
;
;       A single byte to trap can be passed in the
;       command line as a parameter. If not entered,
;       it will default to X'0E', the infamous cursor
;       on character which if sent to a printer, will
;       cause expanded character mode on a lot of dot
;       matrix printers if CURSOR ON is sent to *PR.
;
;       To filter the printer output, issue:
;          SET *TP to TRAP (CHAR=dd)
;          FILTER *PR using *TP
;
;*=*=*
LF      EQU     10              ;Line feed
CR      EQU     13              ;Carriage return
@CHNIO  EQU     20
@HIGH$  EQU     100
@DSPLY  EQU     10
@FLAGS$ EQU     101
@PARAM  EQU     17
@LOGOT  EQU     12
;
        ORG     3000H
BEGIN   PUSH    DE
        POP     IX              ;Get DCB pointer into IX
        LD      (MODDCB),DE     ;Stuff DCB pointer
        PUSH    HL              ;Save command line ptr
        LD      HL,HELLO$
        LD      A,@DSPLY        ;Display hello
        RST     40
        POP     HL              ;Rcvr command line ptr
;*=*=*
;       Check if entry from SET command
;*=*=*
        LD      A,@FLAGS$       ;Get flag pointer
        RST     40
        BIT     3,(IY+'C'-'A')  ;System request?
        JP      Z,VIASET
        LD      DE,PRMTBL$      ;Point to parameter table
        LD      A,@PARAM        ;Get parms if any
        RST     40
        JR      NZ,PRMERR
CPARM   LD      BC,14           ;Init to X'0E'
        LD      A,(CRESP)       ;P/u the response
        OR      A               ;  & see if any entry
        JR      Z,CDEFLT        ;Default if none
        BIT     7,A             ;Value entry?
        JR      NZ,CDEFLT       ;Value is in reg C
        BIT     5,A             ;String value?
        JP      NZ,PRMERR       ;Error if anything else
        LD      A,(BC)          ;BC contains a pointer
        LD      C,A             ;Shorter than a jump
CDEFLT  LD      A,C             ;Xfer the value to reg A
        LD      (TRAPBYT+1),A   ;  & stuff in filter
;*=*=*
;       install new HIGH$ and move filter code
;*=*=*
        LD      HL,0            ;Get current HIGH$
        LD      B,L
        LD      A,@HIGH$
        RST     40
        JR      NZ,NOMEM
        LD      (OLDHI),HL      ;Put in filter header
;*=*=*
;       Move module into memory
;*=*=*
```

```
        EX      DE,HL           ;Destination address to DE
        LD      HL,MODDCB-MODEND
        ADD     HL,DE           ;Relocate one address
        LD      (RX01),HL
        LD      HL,MODEND       ;Last byte of module
        LD      BC,LENGTH       ;Length of filter
        LDDR
        EX      DE,HL           ;Move new HIGH$ to HL
        LD      A,@HIGH$        ;Set new HIGH$ into the system
        RST     40
        INC     HL              ;Bump to filter entry
        LD      (IX+0),40H.OR.7 ;Stuff TYPE byte
        LD      (IX+1),L
        LD      (IX+2),H        ;Install addr into DCB
        LD      HL,0            ;Successful...
        RET
;
PRMERR  LD      HL,PRMERR$
        DB      0DDH
VIASET  LD      HL,VIASET$
        DB      0DDH
NOMEM   LD      HL,NOMEM$
        LD      A,@LOGOT
        RST     40
        LD      HL,-1           ;Indicate extended error
        RET
;
HELLO$  DB      LF,'TRAP filter to trap a character code',CR
PRMERR$ DB      'Bad parameters!',CR
NOMEM$  DB      'High memory is not available!',CR
VIASET$ DB      'Must install via SET!',CR
;
PRMTBL$ DB      80H
        DB      80H.OR.20H.OR.10H.OR.4
        DB      'CHAR'          ;Parameter word
CRESP   DB      0               ;Response byte
        DW      CPARM+1         ;Storage address
        NOP                     ;Table end indicator
;
;*****
;       Actual FILTER routine to shift up to HIGH$
;*****
TRAP    JR      START
OLDHI   DW      $-$             ;HIGH$ before filtering
        DB      MODDCB-TRAP-5
        DB      'TRAP'
MODDCB  DW      $-$             ;Loaded with DCB pointer
        DW      0
;
START   JR      NZ,OUTP1        ;Go if not PUT
        LD      A,C
TRAPBYT SUB     0               ;Space for trap char
        RET     Z               ;Back with Z & A=0 if trapped
OUTP1   PUSH    IX              ;Save current pointer
        LD      IX,(MODDCB)     ;P/u this module's DCB
RX01    EQU     $-2
        LD      A,@CHNIO        ;Chain to the next
        RST     40
        POP     IX
MODEND  RET
LENGTH  EQU     $-TRAP
        END     BEGIN
```

8.15 SLASH0 Filter Illustrated

```
;SLASH0/FLT - Version 6.0 - 05/27/83
;
        COM     '<Copyright 1983 by Roy Soltoff>'
;
;*=*=*
;       This filter will provide slashed zeroes on
;       printers capable of accepting a backspace
;*=*=*
;
LF      EQU     10
CR      EQU     13
@CHNIO  EQU     20
@HIGH$  EQU     100
@DSPLY  EQU     10
@FLAGS$ EQU     101
@LOGOT  EQU     12
        ORG     3000H
BEGIN   PUSH    DE
        POP     IX              ;Get dcb
        LD      (MODDCB),DE     ;Stuff DCB pointer
        LD      HL,HELLO$
        LD      A,@DSPLY        ;Display hello
        RST     40
;*=*=*
;       Check if entry from SET command
;*=*=*
        LD      A,@FLAGS$       ;Get flags pointer
        RST     40
        BIT     3,(IY+'C'-'A')  ;System request?
        JP      Z,VIASET
;*=*=*
;       install new HIGH$ and move filter code
;*=*=*
        LD      HL,0            ;Get current HIGH$
        LD      B,L
        LD      A,@HIGH$
        RST     40
        JR      NZ,NOMEM
        LD      (OLDHI),HL      ;Put in filter header
;*=*=*
;       Relocate internal references in driver
;*=*=*
        LD      IY,RELTAB       ;Point to relocation tbl
        LD      DE,MODEND
        OR      A               ;Clear carry flag
        SBC     HL,DE
        LD      B,H             ;Move to BC
        LD      C,L
RLOOP   LD      L,(IY)          ;Get address to change
        LD      H,(IY+1)
        LD      A,H
        OR      L
        JR      Z,RXEND
        LD      E,(HL)          ;P/U address
        INC     HL
        LD      D,(HL)
        EX      DE,HL           ;Offset it
        ADD     HL,BC
        EX      DE,HL
        LD      (HL),D          ;And put back
        DEC     HL
        LD      (HL),E
        INC     IY
        INC     IY
        JR      RLOOP           ;Loop till done
;*=*=*
;       Move driver into high memory
;*=*=*
RXEND   LD      DE,(OLDHI)      ;Destination address
        LD      HL,MODEND       ;Last byte of module
        LD      BC,LENGTH       ;Length of filter
        LDDR
```

```
        EX      DE,HL           ;Move new HIGH$ to HL
        LD      A,@HIGH$        ;Set new HIGH$ into the system
        RST     40
        INC     HL              ;Bump to filter entry
        LD      (IX+0),40H.OR.7 ;Stuff TYPE byte
        LD      (IX+1),L
        LD      (IX+2),H        ;Install addr into dcb
        LD      HL,0            ;Successful...
        RET
;
VIASET  LD      HL,VIASET$
        DB      0DDH
NOMEM   LD      HL,NOMEM$
        @@LOGOT
        LD      HL,-1
        RET
;
HELLO$  DB       LF,'SLASH0 Filter'
NOMEM$  DB      'High memory is not available!',CR
VIASET$ DB      'Must install via SET!',CR
;
;*=*=*
;       The SLASH-0 filter
;*=*=*
SLASH   JR      START
OLDHI   DW      $-$             ;HIGH$ before filtering
        DB      MODDCB-SLASH-5
        DB      'SLASH0'
MODDCB  DW      $-$             ;Loaded with DCB pointer
        DW      0
;
START   JR      NZ,OUTP1        ;Go if not PUT
        LD      A,C
        CP      '0'             ;ASCII zero?
        JR      Z,OUTCF         ;Go if so
OUTP1   PUSH    IX              ;Save current pointer
        PUSH    BC              ;Save in case affected downstream
        LD      IX,(MODDCB)     ;P/u this module's DCB
RX01    EQU     $-2
        LD      A,@CHNIO        ;Chain to the next
        RST     40
        POP     BC
        POP     IX
        RET
;*=*=*
;       Do the slashing
;*=*=*
OUTCF   CALL    OUTP1           ;Put the zero
RX02    EQU     $-2
        LD      C,08H           ;Backspace
        CALL    Z,OUTP1
RX03    EQU     $-2
        LD      C,'/'           ;Now put the slash
        JR      Z,OUTP1         ;  unless an error
MODEND  RET
;
LENGTH  EQU     $-SLASH
RELTAB  DW      RX01,RX02,RX03,0
;
        END     BEGIN
```

8.16 DMP-500 BOLDFACE Filter Illustrated

```
;BOLDFACE/ASM -  FILTER to invoke boldfacing on DMP-500 - 03/20/83
        TITLE   '<DMP-500 BOLDFACE Filter>'
;*****
;       This filter uses two trigger toggle characters to turn
;       on and off the boldface mode of the DMP-500 printer.
;       One character called TOGGLE (defaults to tilde) will
;       toggle on/off boldface and output a space in lieu of
;       the toggle character. This is useful to maintain right
;       justification. The other character called NULL (defaults
;       to DELETE, X'7F') toggles the boldface mode but causes
;       no character to be sent in lieu of the toggle character.
;       The boldface mode is automatically turned off when a
;       carriage return (X'0D') is sensed.
;*=*=*
        COM     '<Copyright (C) 1983 by MISOSYS>'
;*=*=*
LF      EQU     10
CR      EQU     13
ESCAPE  EQU     27
BOLDON  EQU     31
BOLDOFF EQU     32
@CHNIO  EQU     20
@HIGH$  EQU     100
@DSPLY  EQU     10
@FLAGS$ EQU     101
@PARAM  EQU     17
@LOGOT  EQU     12
;
        ORG     3000H
BEGIN   PUSH    DE
        POP     IX              ;Get DCB into IX
        LD      (MODDCB),DE     ;Stuff DCB pointer
        PUSH    HL              ;Save INBUF$ pointer
        LD      HL,HELLO$
        LD      A,@DSPLY
        RST     40
        POP     HL              ;Rcvr INBUF$ pointer
;*=*=*
;       Check if entry from SET command
;*=*=*
        LD      A,@FLAGS$       ;Get flags pointer into IY
        RST     40
        BIT     3,(IY+'C'-'A')  ;System request?
        JP      Z,VIASET
;
        LD      DE,PRMTBL$      ;Grab any user parms
        LD      A,@PARAM
        RST     40
        JP      NZ,PRMERR
;*=*=*
;       Transfer requested TOGGLE e/w space to filter
;*=*=*
        LD      A,(TRESP)       ;Ck if any entry
        LD      B,A
TOGGLE  LD      HL,7EH          ;Set default to TILDE
        LD      A,(HL)          ;P/u assumed string
        BIT     5,B             ;String entry?
        JR      NZ,TSTUF
        LD      A,L             ;P/u hex or dec entry
        BIT     6,B             ;Error if switch entry
        JP      NZ,PRMERR
TSTUF   LD      (TILDE1+1),A    ;Stuff it in there
        LD      (TILDE2+1),A
;*=*=*
;       Transfer requested toggle w/o space to filter
;*=*=*
        LD      A,(NRESP)       ;Ck if any entry
        LD      B,A
NULL    LD      HL,7FH          ;Set default to DELETE
        LD      A,(HL)          ;P/u assumed string
        BIT     5,B             ;String entry?
        JR      NZ,NSTUF
```

```
        LD      A,L             ;P/u hex or dec entry
        BIT     6,B             ;Error if switch entry
        JP      NZ,PRMERR
NSTUF   LD      (NULL1+1),A     ;Stuff it in there
        LD      (NULL2+1),A
;*=*=*
;       install new HIGH$ and move filter code
;*=*=*
        LD      HL,0            ;get current HIGH$
        LD      B,L
        LD      A,@HIGH$
        RST     40
        JR      NZ,NOMEM
        LD      (OLDHI),HL      ;put in filter header
;*=*=*
;       Relocate internal references in driver
;*=*=*
        LD      IY,RELTAB       ;Point to relocation tbl
        LD      DE,MODEND
        XOR     A               ;Clear carry flag
        SBC     HL,DE
        LD      B,H             ;Move to BC
        LD      C,L
RLOOP   LD      L,(IY)          ;Get address to change
        LD      H,(IY+1)
        LD      A,H
        OR      L
        JR      Z,RXEND
        LD      E,(HL)          ;P/U address
        INC     HL
        LD      D,(HL)
        EX      DE,HL           ;Offset it
        ADD     HL,BC
        EX      DE,HL
        LD      (HL),D          ;And put back
        DEC     HL
        LD      (HL),E
        INC     IY
        INC     IY
        JR      RLOOP           ;Loop till done
;*=*=*
;       Move driver
;*=*=*
RXEND   LD      DE,(OLDHI)      ;Destination address
        LD      HL,MODEND       ;Last byte of module
        LD      BC,LENGTH       ;length of filter
        LDDR
        EX      DE,HL           ;Move new HIGH$ to HL
        LD      A,@HIGH$        ;Set new HIGH$ into the system
        RST     40
        INC     HL              ;Bump to filter entry
        LD      (IX+0),40H.OR.6 ;Stuff TYPE byte
        LD      (IX+1),L
        LD      (IX+2),H        ;install addr into dcb
        LD      HL,0            ;Successful...
        RET
;*=*=*
;       Error message handling
;*=*=*
VIASET  LD      HL,VIASET$      ;'Must install...
        DB      0DDH
NOMEM   LD      HL,NOMEM$       ;'No memory'
        DB      0DDH
PRMERR  LD      HL,PRMERR$      ;'Parameter error'
        LD      A,@LOGOT
        RST     40
        LD      HL,-1
        RET
;*=*=*
;       Data area
;*=*=*
HELLO$  DB      'DMP-500 BOLDFACE Filter Version 6.0a - '
        DB      'Copyright 1983 by Roy Soltoff',LF,CR
PRMERR$ DB      'Parameter error!',CR
```

```
NOMEM$  DB      'High memory is not available!',CR
VIASET$ DB      'Must install via SET',CR
;*=*=*
;       Parameter table
;*=*=*
PRMTBL$ DB      80H!'R'
;
        DB      0F6H,'TOGGLE'   ;Toggle on/off char
TRESP   DB      0
        DW      TOGGLE+1
;
        DB      0F4H,'NULL'     ;Toggle on/off w/o space
NRESP   DB      0
        DW      NULL+1
;
        NOP                     ;End of table
;*=*=*
;       Entry point
;*=*=*
;
BOLD    JR      START           ;Branch around linkage
OLDHI   DW      $-$             ;Last byte used
;
        DB      7,'DMPBOLD'
;
MODDCB  DW      $-$             ;Loaded with DCB pointer
        DW      0
;
START   JR      Z,FILTER        ;Go if @PUT
PUTOUT  PUSH    IX              ;Save current pointer
        PUSH    BC              ;Save in case affected downstream
        LD      IX,(MODDCB)     ;P/u this module's DCB
RX01    EQU     $-2
        LD      A,@CHNIO        ;Chain to the next
        RST     40
        POP     BC
        POP     IX
        RET
FILTER  EQU     $
SWITCH  LD      A,0             ;P/u switch
        OR      A               ;Is flag on?
        JR      NZ,SWISON       ;Go if switch is on
        LD      A,C             ;Is char a tilde?
TILDE1  CP      7EH
        JR      Z,TONSPA        ;Go if got to turn on
NULL1   CP      7FH             ;Turn on w/o space?
        JR      Z,TURNON
        JR      PUTOUT          ;Send the char
;*=*=*
;       Got a flag to turn switch on/off
;*=*=*
TURNON  LD      C,BOLDON
        JR      TURNA
TURNOFF XOR     A
        LD      C,BOLDOFF
TURNA   LD      (SWITCH+1),A    ;Turn off the switch
RX02    EQU     $-2
;
        PUSH    BC              ;Save toggle control code
        LD      C,ESCAPE
        CALL    PUTOUT          ;Put the ESCAPE
RX03    EQU     $-2
        POP     BC              ;Restore and PUT
        JR      PUTOUT          ;  the toggle code
TOFFSPA CALL    TURNOFF
RX04    EQU     $-2
        JR      PUT_SPA
TONSPA  CALL    TURNON
RX05    EQU     $-2
PUT_SPA LD      C,' '           ;Put space for tilde
        JR      PUTOUT          ; and stuff a space
;*=*=*
;       Flag is on - what should we do?
;*=*=*
```

```
SWISON  LD      A,C             ;Do we close the switch?
TILDE2  CP      7EH
        JR      Z,TOFFSPA
NULL2   CP      7FH             ;Turn off w/o space?
        JR      Z,TURNOFF
        CP      CR              ;Turn off on EOL
        JR      NZ,PUTOUT
        CALL    TURNOFF
RX06    EQU     $-2
        LD      C,CR
        JR      PUTOUT
MODEND  EQU     $-1
LENGTH  EQU     $-BOLD
RELTAB  DW      RX01,RX02,RX03,RX04,RX05,RX06,0
;
        END     BEGIN
```

@

C

D