EXHIBIT O

DIGITAL RESEARCH
Post Office Box 579, Pacific Grove, California 93950, (408) 373-3403

CP/M Interface Guide

TABLE OF CONTENTS

CP/M INTERFACE GUIDE

1. INTRODUCTION

This manual describes the CP/M system organization including the structure of memory, as well as system entry points. The intention here is to provide the necessary information required to write programs which operate under CP/M, and which use the peripheral and disk I/O facilities of the system.

1.1 CP/M Organization

CP/M is logically divided into four parts:

BIOS - the basic I/O system for serial peripheral control
BDOS - the basic disk operating system primitives
CCP - the console command processer
TPA - the transient program area

The BIOS and BDOS are combined into a single program with a common entry point and referred to as the FDOS. The CCP is a distinct program which uses the FDOS to provide a human-oriented interface to the information which is cataloged on the diskette. The TPA is an area of memory (i.e, the portion which is not used by the FDOS and CCP) where various non-resident operating system commands are executed. User programs also execute in the TPA. The organization of memory in a standard CP/M system is shown in Figure 1.

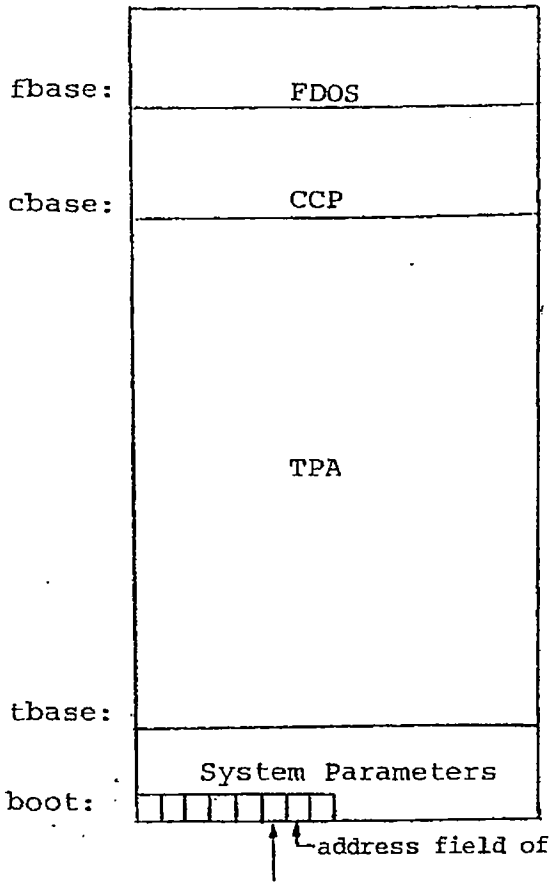
The lower portion of memory is reserved for system information (which is detailed in later sections), including user defined interrupt locations. The portion between tbase and cbase is reserved for the transient operating system commands, while the portion above cbase contains the resident CCP and FDOS. The last three locations of memory contain a jump instruction to the FDOS entry point which provides access to system functions.

1.2 Operation of Transient Programs

Transient programs (system functions and user-defined programs) are loaded into the TPA and executed as follows. The operator communicates with the CCP by typing command lines following each prompt character. Each command line takes one of the forms:

{ <command>
<command> <filename>
<command> <filename>.<filetype> }

Figure 1. CP/M Memory Organization



entry: the principal entry point to FDOS is at location 0005 which contains a JMP to fbase. The address field at location 0006 can be used to determine the size of available memory, assuming the CCP is being overlayed.

Note: The exact addresses for boot, tbase, cbase, fbase, and entry vary with the CP/M version (see Section 6. for version correspondence).

where <command> is either a built-in command (e.g., DIR or TYPE), or the name of a transient command or program. If the <command> is a built-in function of CP/M, it is executed immediately; otherwise the CCP searches the currently addressed disk for a file by the name

<command>.COM

If the file is found, it is assumed to be a memory image of a program which executes in the TPA, and thus implicitly originates at tbase in memory (see the CP/M LOAD command). The CCP loads the COM file from the diskette into memory starting at tbase, and extending up to address cbase.

If the <command> is followed by either a <filename> or <filename>.<filetype>, then the CCP prepares a file control-block (FCB) in the system information area of memory. This FCB is in the form required to access the file through the FDOS, and is given in detail in Section 3.2.

The program then executes, perhaps using the I/O facilities of the FDOS. If the program uses no FDOS facilities, then the entire remaining memory area is available for data used by the program. If the FDOS is to remain in memory, then the transient program can use only up to location fbase as data.* In any case, if the CCP area is used by the transient, the entire CP/M system must be reloaded upon the transient's completion. This system reload is accomplished by a direct branch to location "boot" in memory.

The transient uses the CP/M I/O facilities to communicate with the operator's console and peripheral devices, including the floppy disk subsystem. The I/O system is accessed by passing a "function number" and an "information address" to CP/M through the address marked "entry" in Figure 1. In the case of a disk read, for example, the transient program sends the number corresponding to a disk read, along with the address of an FCB, and CP/M performs the operation, returning with either a disk read complete indication or an error number indicating that the disk operation was unsuccessful. The function numbers and error indicators are given in detail in Section 3.3.

1.3 Operating System Facilities

CP/M facilities which are available to transients are divided into two categories: BIOS operations, and BDOS primitives. The BIOS operations are listed first:**

* Address "entry" contains a jump to the lowest address in the FDOS, and thus "entry+1" contains the first FDOS address which cannot be overlayed.

**The device support (exclusive of the disk subsystem) corresponds exactly to Intel's peripheral definition, including I/O port assignment and status byte format (see the Intel manual which discusses the Intellec MDS hardware environment).

Read Console Character
Write Console Character
Read Reader Character
Write Punch Character
Write List Device Character
Set I/O Status
Interrogate Device Status
Print Console Buffer
Read Console Buffer
Interrogate Console Status

The exact details of BIOS access are given in Section 2. The BDOS primitives include the following operations:

Disk System Reset
Drive Select
File Creation
File Open
File Close
Directory Search
File Delete
File Rename
Read Record
Write Record
Interrogate Available Disks
Interrogate Selected Disk
Set DMA Address

The details of BDOS access are given in Section 3.

2. BASIC I/O FACILITIES

Access to common peripherals is accomplished by passing a function number and information address to the BIOS. In general, the function number is passed in Register C, while the information address is passed in Register pair D,E. Note that this conforms to the PL/M Conventions for parameter passing, and thus the following PL/M procedure is sufficient to link to the BIOS when a value is returned:

```
DECLARE ENTRY LITERALLY    '0005H'; /* MONITOR ENTRY */

MON2:  PROCEDURE (FUNC, INFO) BYTE;
       DECLARE FUNC BYTE, INFO ADDRESS;
       GO TO ENTRY;

       END MON2;
```

```
MON1:   PROCEDURE (FUNC,INFO);
        DECLARE FUNC BYTE, INFO ADDRESS;
        GO TO ENTRY;
        END MON1
```

if no returned value is expected.

2.1 Direct and Buffered I/O.

The BIOS entry points are given in Table I. In the case of simple character I/O to the console, the BIOS reads the console device, and removes the parity bit. The character is echoed back to the console, and tab characters (control-I) are expanded to tab positions starting at column one and separated by eight character positions. The I/O status byte takes the form shown in Table I, and can be programmatically interrogated or changed. The buffered read operation takes advantage of the CP/M line editing facilities. That is, the program sends the address of a read buffer whose first byte is the length of the buffer. The second byte is initially empty, but is filled-in by CP/M to the number of characters read from the console after the operation (not including the terminating carriage-return). The remaining positions are used to hold the characters read from the console. The BIOS line editing functions which are performed during this operation are given below:

break	- line delete and transmit
rubout	- delete last character typed, and echo
control-C	- system rebout
control-U	- delete entire line
control-E	- return carriage, but do not transmit buffer (physical carriage return)
<cr>	- transmit buffer

The read routine also detects control character sequences other than those shown above, and echos them with a preceding "↑" symbol. The print entry point allows an entire string of symbols to be printed before returning from the BIOS. The string is terminated by a "$" symbol.

2.2 A Simple Example

As an example, consider the following PL/M procedures and procedure calls which print a heading, and successively read the console buffer. Each console buffer is then echoed back in reverse order:

```
PRINTCHAR:  PROCEDURE (B);
      /* SEND THE ASCII CHARACTER B TO THE CONSOLE */
      DECLARE B BYTE;
      CALL MON1(2,B);
      END PRINTCHAR;

CRLF:  PROCEDURE;
      /* SEND CARRIAGE-RETURN-LINE-FEED CHARACTERS */
      CALL PRINTCHAR (0DH); CALL PRINTCHAR (0AH);
      END CRLF;

PRINT:  PROCEDURE (A);
      /* PRINT THE BUFFER STARTING AT ADDRESS A */
      DECLARE A ADDRESS;
      CALL MON1(9,A);
      END PRINT;
DECLARE   RDBUFF (130) BYTE;

READ:  PROCEDURE;
      /* READ CONSOLE CHARACTERS INTO 'RDBUFF' */
      RDBUFF=128; /* FIRST BYTE SET TO BUFFER LENGTH */
      CALL MON1(10,.RDBUFF);
      END READ;
DECLARE I BYTE;
CALL CRLF;
      DO WHILE 1; /* INFINITE LOOP-UNTIL CONTROL-C */
                              CALL PRINT (.'TYPE INPUT LINES $');
      CALL CRLF; CALL PRINTCHAR ('*'); /* PROMPT WITH '*' */
      CALL READ; I = RDBUFF(1);
            DO WHILE (I := I -1) <> 255;
            CALL PRINTCHAR (RDBUFF(I+2));
            END;
      END;
```

The execution of this program might proceed as follows:

```
TYPE INPUT LINES
*HELLO,
OLLEH
*WALL WALLA WASH,
HSAW ALLAW ALLAW
*MOM WOW,
WOW MOM
*↑C                    (system reboot)
```

TABLE I

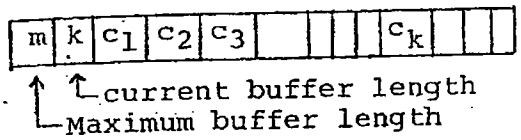
BASIC I/O OPERATIONS

FUNCTION/ NUMBER	ENTRY PARAMETERS	RETURNED VALUE	TYPICAL CALL
Read Console 1	None	ASCII Character	I = MON2(1,0)
Write Console 2	ASCII Character	None	CALL MON1(2,'A')
Read Reader 3	None	ASCII Character	I = MON2(3,0)
Write Punch 4	ASCII Character	None	CALL MON1(4,'B')
Write List 5	ASCII Character	None	CALL MON1(5,'C')
Get I/O Status 7	None	I/O Status Byte	IOSTAT=MON2(7,0)
Set I/O Status 8	I/O Status Byte	None	CALL MON1(8,IOSTAT)
Print Buffer 9	Address of string terminated by '$'	None	CALL MON1(9, .'PRINT THIS $')

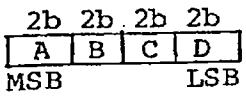
TABLE I (continued)

FUNCTION/ NUMBER	ENTRY PARAMETERS	RETURNED VALUE	TYPICAL CALL
Read Buffer 10	Address of Read Buffer* (See Note$_1$)	Read buffer is filled to maximum length with console characters	CALL MON1(10, .RDBUFF);
Interrogate Console Ready 11	None	Byte value with least significant bit = 1 (true) if console character is ready	I = MON2(11,0)

Note$_1$: Read buffer is a sequence of memory locations of the form:



Note$_2$: The I/O status byte is defined as three fields A,B,C, and D



requiring two bits each, listed from most significant to least significant bit, which define the current device assignment as follows:

D Console = { 0 TTY, 1 CRT, 2 BATCH, 3 - }

C Reader = { 0 TTY, 1 FAST READER, 2 -, 3 - }

B Punch = { 0 TTY, 1 FAST PUNCH, 2 -, 3 - }

A List = { 0 TTY, 1 CRT, 2 -, 3 - }

3. DISK I/O FACILITIES

The BDOS section of CP/M provides access to files stored on diskettes. The discussion which follows gives the overall file organization, along with file access mechanisms.

3.1 File Organization

CP/M implements a named file structure on each diskette, providing a logical organization which allows any particular file to contain any number of records, from completely empty, to the full capacity of a diskette. Each diskette is logically distinct, with a complete operating system, disk directory, and file data area. The disk file names are in two parts: the <filename> which can be from one to eight alphanumeric characters, and the <filetype> which consists of zero through three alphanumeric characters. The <filetype> names the generic category of a particular file, while the <filename> distinguishes a particular file within the category. The <filetype>s listed below give some generic categories which have been established, although they are generally arbitrary:

ASM	assembler source file
PRN	assembler listing file
HEX	assembler or PL/M machine code in "hex" format
BAS	BASIC Source file
INT	BASIC Intermediate file
COM	Memory image file (i.e., "Command" file for transients, produced by LOAD)
BAK	Backup file produced by editor (see ED manual)
$$$	Temporary files created and normally erased by editor and utilities

Thus, the name

X.ASM

is interpreted as an assembly language source file by the CCP with <filename> X.

The files in CP/M are organized as a logically contiguous sequence of 128 byte records (although the records may not be physically contiguous on the diskette), which are normally read or written in sequential order. Random access is allowed under CP/M however, as described in Section 3.4. No particular format within records is assumed by CP/M, although some transients expect particular formats:

(1) Source files are considered a sequence of ASCII characters, where each "line" of the source file is followed by carriage-return-line-feed characters. Thus, one 128 byte CP/M record could contain several logical lines of source text. Machine code "hex" tapes are also assumed to be in this format, although the loader does not require the carriage-return-line-feed characters. End of text is given by the character control-z, or real end-of-file returned by CP/M.

and

(2) COM files are assumed to be absolute machine code in memory image form, starting at tbase in memory. In this case, control-z is not considered an end of file, but instead is determined by the actual space allocated to the file being accessed.

3.2 File Control Block Format

Each file being accessed through CP/M has a corresponding file control block (FCB) which provides name and allocation information for all file operations. The FCB is a 33-byte area in the transient program's memory space which is set up for each file. The FCB format is given in Figure 2. When accessing CP/M files, it is the programmer's responsibility to fill the lower 16 bytes of the FCB, along with the NR field. Normally, the FN and FT fields are set to the ASCII <filename> and <filetype>, while all other fields are set to zero. Each FCB describes up to 16K bytes of a particular file (0 to 128 records of 128 bytes each), and, using automatic mechanisms of CP/M, up to 15 additional extensions of the file can be addressed. Thus, each FCB can potentially describe files up to 256K bytes (which is slightly larger than the diskette capacity).

FCB's are stored in a directory area of the diskette, and are brought into central memory before file operations (see the OPEN and MAKE commands) then updated in memory as file operations proceed, and finally recorded on the diskette at the termination of the file operation (see the CLOSE command). This organization makes CP/M file organization highly reliable, since diskette file integrity can only be disrupted in the unlikely case of hardware failure during update of a single directory entry.

It should be noted that the CCP constructs an FCB for all transients by scanning the remainder of the line following the transient name for a <filename> or <filename>.<filetype> combination. Any field not specified is assumed to be all blanks. A properly formed FCB is set up at location tfcb (see Section 6), with an assumed I/O buffer at tbuff. The transient can use tfcb as an address in subsequent input or output operations on this file.

In addition to the default fcb which is set-up at address tfcb, the CCP also constructs a second default fcb at address tfcb+16 (i.e., the disk map field of the fcb at tbase). Thus, if the user types

PROGNAME X.ZOT Y.ZAP

the file PROGNAME.COM is loaded to the TPA, and the default fcb at tfcb is initialized to the filename X with filetype ZOT. Since the user typed a second file name, the 16 byte area beginning at tfcb + 16_{10} is also initialized with the filename Y and filetype ZAP. It is the responsibility of the program to move this second filename and filetype to another area (usually a separate file control block) before opening the file which begins at tbase, since the open operation will fill the disk map portion, thus overwriting the second name and type.

If no file names were specified in the original command, then the fields beginning at tfcb and tfcb + 16 both contain blanks (20H). If one file name was specified, then the field at tfcb + 16 contains blanks. If the filetype is omitted, then the field is assumed to contain blanks. In all cases, the CCP translates lower case alphabetics to upper case to be consistent with the CP/M file naming conventions.

As an added programming convenience, the default buffer at tbuff is initialized to hold the entire command line past the program name. Address tbuff contains the number of characters, and tbuff+1, tbuff+2, ..., contain the remaining characters up to, but not including, the carriage return. Given that the above command has been typed at the console, the area beginning at tbuff is set up as follows:

tbuff:

+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15
12	Ø	X	.	Z	O	T	Ø	Y	.	Z	A	P	?	?	?

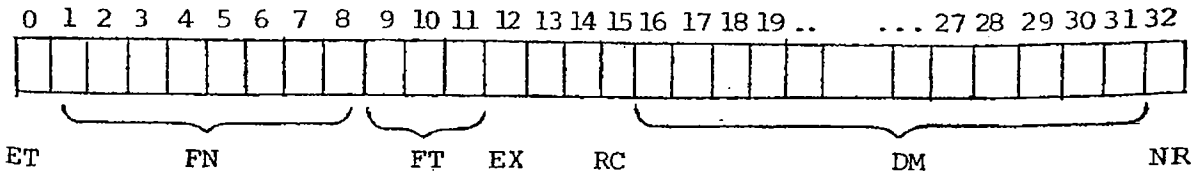
where 12 is the number of valid characters (in binary), and Ø represents an ASCII blank. Characters are given in ASCII upper case, with uninitialized memory following the last valid character.

Again, it is the responsibility of the program to extract the information from this buffer before any file operations are performed since the FDOS uses the tbuff area to perform directory functions.

In a standard CP/M system, the following values are assumed:

boot:	0000H	bootstrap load (warm start)
entry:	0005H	entry point to FDOS
tfcb:	005CH	first default file control block
tfcb+16	006CH	second file name
tbuff	0080H	default buffer address
tbase:	0100H	base of transient area

Figure 2. File Control Block Format



FIELD	FCB POSITIONS	PURPOSE
ET	0	Entry type (currently not used, but assumed zero)
FN	1-8	File name, padded with ASCII blanks
FT	9-11	File type, padded with ASCII blanks
EX	12	File extent, normally set to zero
	13-14	Not used, but assumed zero
RC	15	Record count is current extent Size (0 to 128 records)
DM	16-31	Disk allocation map, filled-in and used by CP/M
NR	32	Next record number to read or write

3.3 Disk Access Primitives

Given that a program has properly initialized the FCB's for each of its files, there are several operations which can be performed, as shown in Table II. In each case, the operation is applied to the currently selected disk (see the disk select operation in Table II), using the file information in a specific FCB. The following PL/M program segment, for example, copies the contents of the file X.Y to the (new) file NEW.FIL:

```
DECLARE RET BYTE;

OPEN:       PROCEDURE (A)
            DECLARE A ADDRESS;
            RET=MON2(15,A);
            END OPEN;

CLOSE:      PROCEDURE (A);
            DECLARE A ADDRESS;
            RET=MON2(16,A);
            END;

MAKE:       PROCEDURE (A);
            DECLARE A ADDRESS;
            RET=MON2(22,A);
            END MAKE;

DELETE:     PROCEDURE (A);
            DECLARE A ADDRESS;
            /* IGNORE RETURNED VALUE */
            CALL MON1(19,A);
            END DELETE;

READBF:     PROCEDURE (A);
            DECLARE A ADDRESS;
            RET=MON2(20,A);
            END READBF;

WRITEBF:    PROCEDURE (A);
            DECLARE A ADDRESS;
            RET=MON2(21,A);
            END WRITEBF;

INIT:       PROCEDURE;
            CALL MON1(13,0);
            END INIT;

/* SET UP FILE CONTROL BLOCKS */
DECLARE FCB1 (33) BYTE
   INITIAL (0,'X       ','Y  ',0,0,0,0),
   FCB2 (33) BYTE
   INITIAL (0,'NEW     ','FIL',0,0,0,0);
```

```
CALL INIT;
/* ERASE 'NEW.FIL' IF IT EXISTS */
CALL DELETE (.FCB2);
/* CREATE''NEW.FIL' AND CHECK SUCCESS */
CALL MAKE (.FCB2);
IF RET = 255 THEN CALL PRINT (.'NO DIRECTORY SPACE $');
   ELSE
   DO; /* FILE SUCCESSFULLY CREATED, NOW OPEN 'X.Y' */
   CALL OPEN (.FCB1);
   IF RET = 255 THEN CALL PRINT (.'FILE NOT PRESENT $');
      ELSE
      DO; /* FILE X.Y FOUND AND OPENED, SET
      NEXT RECORD TO ZERO FOR BOTH FILES */
      FCB1(32), FCB2(32) = 0;
      /* READ FILE X.Y UNTIL EOF OR ERROR */
      CALL READBF (.FCB1); /*READ TO 80H*/
         DO WHILE RET = 0;
         CALL WRITEBF (.FCB2) /*WRITE FROM 80H*/
         IF RET = 0 THEN /*GET ANOTHER RECORD*/
            CALL READBF (.FCB1); ELSE
            CALL PRINT (.'DISK WRITE ERROR $');
         END;
      IF RET < >1 THEN CALL PRINT (.' TRANSFER ERROR $');
      ELSE
         DO; CALL CLOSE (.FCB2);
         IF RET = 255 THEN CALL PRINT (.'CLOSE ERROR$');
         END;
      END;
   END;
EOF
```

This program consists of a number of utility procedures for opening, closing, creating, and deleting files, as well as two procedures for reading and writing data. These utility procedures are followed by two FCB's for the input and output files. In both cases, the first 16 bytes are initialized to the <filename> and <filetype> of the input and output files. The main program first initializes the disk system, then deletes any existing copy of "NEW.FIL" before starting. The next step is to create a new directory entry (and empty file) for "NEW.FIL". If file creation is successful, the input file "X.Y" is opened. If this second operation is also successful, then the disk to disk copy can proceed. The NR fields are set to zero so that the first record of each file is accessed on subsequent disk I/O operations. The first call to READBF fills the (implied) DMA buffer at 80H with the first record from X.Y. The loop which follows copies the record at 80H to "NEW.FIL" and then reports any errors, or reads another 128 bytes from X.Y. This transfer operation continues until either all data has been transferred, or an error condition arises. If an error occurs, it is reported; otherwise the new file is closed and the program halts.

TABLE II

DISK ACCESS PRIMITIVES

FUNCTION/NUMBER	ENTRY PARAMETERS	RETURNED VALUE	TYPICAL CALL
Lift Head 12	None	None Head is lifted from current drive	CALL MON2(12,0)
Initialize BDOS and select disk "A" Set DMA address to 80H 13	None	None Side effect is that disk A is "logged-in" while all others are considered "off-line"	CALL MON1(13,0)
Log-in and select disk X 14	An integer value corresponding to the disk to log-in: A=0, B=1, C=2, etc.	None Disk X is considered "on-line" and selected for subsequent file operations	CALL MON1(14,1) (log-in disk "B")
Open file 15	Address of the FCB for the file to be accessed	Byte address of the FCB in the directory, if found, or 255 if file not present. The DM bytes are set by the BDOS.	I = MON2(15,.FCB)
Close file 16	Address of an FCB which has been previously created or opened	Byte address of the directory entry corresponding to the FCB, or 255 if not present	I = MON2(16,.FCB)

TABLE II (continued)

FUNCTION/NUMBER	ENTRY PARAMETERS	RETURNED VALUE	TYPICAL CALL
Search for file 17	Address of FCB containing <filename> and <filetype> to match. ASCII "?" in FCB matches any character.	Byte address of first FCB in directory that matches input FCB, if any; otherwise 255 indicates no match.	I = MON2(17,.FCB)
Search for next occurrence 18	Same as above, but called after function 17 (no other intermediate BDOS calls allowed)	Byte address of next	I = MON2(18,.FCB)
Delete File 19	Address of FCB containing <filename> and <filetype> of file to delete from diskette	None	I = MON2(19,.FCB)
Read Next Record 20	Address of FCB of a successfully OPENed file, with NR set to the next record to read (see note$_1$)	0 = successful read 1 = read past end of file 2 = reading unwritten data in random access	I = MON2(20,.FCB)

Note$_1$: The I/O operations transfer data to/from address 80H for the next 128 bytes unless the DMA address has been altered (see function 26). Further, the NR field of the FCB is automatically incremented after the operation. If the NR field exceeds 128, the next extent is opened automatically, and the NR field is reset to zero.

TABLE II (continued)

FUNCTION/NUMBER	ENTRY PARAMETERS	RETURNED VALUE	TYPICAL CALL
Write Next Record 21	Same as above, except NR is set to the next record to write	0 = successful write 1 = error in extending file 2 = end of disk data 255 = no more directory space (see note$_2$)	I = MON2(21,.FCB)
Make File 22	Address of FCB with <filename> and <filetype> set. Directory entry is created, the file is initialized to empty.	Byte address of directory entry allocated to the FCB, or 255 if no directory space is available	I = MON2(22,.FCB)
Rename FCB 23	Address of FCB with old FN and FT in first 16 bytes, and new FN and FT in second 16 bytes	Address of the directory entry which matches the first 16 bytes. The <filename>and <filetype> is altered 255 if no match.	I = MON2(23,.FCB)

Note$_2$: There are normally 64 directory entries available on each diskette (can be expanded to 255 entries), where one entry is required for the primary file, and one for each additional extent.

TABLE II (continued)

FUNCTION/NUMBER	ENTRY PARAMETERS	RETURNED VALUE	TYPICAL CALL
Interrogate log-in vector 24	None	Byte value with "1" in bit positions of "on line" disks, with least significant bit corresponding to disk "A"	I = MON2(24,0)
Set DMA address 26	Address of 128 byte DMA buffer	None Subsequent disk I/O takes place at specified address in memory	CALL MON1(26,2000H)
Interrogate Allocation 27	None	Address of the allocation vector for the current disk (used by STATUS command)	MON3: PROCEDURE(...) ADDRESS; A = MON3(27,0);
Interrogate Drive number 25	None	Disk number of currently logged disk (i.e., the drive which will be used for the next disk operation	I = MON2(25,0);

3.4 Random Access

Recall that a single FCB describes up to a 16K segment of a (possibly) larger file. Random access within the first 16K segment is accomplished by setting the NR field to the record number of the record to be accessed before the disk I/O takes place. Note, however, that if the 128th record is written, then the BDOS automatically increments the extent field (EX), and opens the next extent, if possible. In this case, the program must explicitly decrement the EX field and re-open the previous extent. If random access outside the first 16K segment is necessary, then the extent number e be explicitly computed, given an absolute record number r as

$$e = \left\lfloor \frac{r}{128} \right\rfloor$$

or equivalently,

$$e = SHR(r,7)$$

this extent number is then placed in the EX field before the segment is opened. The NR value n is then computed as

$$n = r \bmod 128$$

or

$$n = r \text{ AND } 7FH.$$

When the programmer expects considerable cross-segment accesses, it may save time to create an FCB for each of the 16K segments, open all segments for access, and compute the relevant FCB from the absolute record number r.

4. SYSTEM GENERATION

As mentioned previously, every diskette used under CP/M is assumed to contain the entire system (excluding transient commands) on the first two tracks. The operating system need not be present, however, if the diskette is only used as secondary disk storage on drives B, C, ..., since the CP/M system is loaded only from drive A.

The CP/M file system is organized so that an IBM-compatible diskette from the factory (or from a vendor which claims IBM compatibility) looks like a diskette with an empty directory. Thus, the user must first copy a version of the CP/M system from an existing diskette to the first two tracks of the new diskette, followed by a sequence of copy operations, using PIP, which transfer the transient command files from the original diskette to the new diskette.

NOTE: before you begin the CP/M copy operation, read your Licensing Agreement. It gives your exact legal obligations when making reproductions of CP/M in whole or in part, and specifically requires that you place the copyright notice

Copyright (c), 1976
Digital Research

on each diskette which results from the copy operation.

4.1. Initializing CP/M from an Existing Diskette

The first two tracks are placed on a new diskette by running the transient command SYSGEN, as described in the document "An Introduction to CP/M Features and Facilities." The SYSGEN operation brings the CP/M system from an initialized diskette into memory, and then takes the memory image and places it on the new diskette.

Upon completion of the SYSGEN operation, place the original diskette on drive A, and the initialized diskette on drive B. Reboot the system; the response should be

A>

indicating that drive A is active. Log into drive B by typing

B:

and CP/M should respond with

B>

indicating that drive B is active. If the diskette in drive B is factory fresh, it will contain an empty directory. Non-standard diskettes may, however, appear as full directories to CP/M, which can be emptied by typing

ERA *.*↵

when the diskette to be initialized is active. Do not give the ERA command if you wish to preserve files on the new diskette since all files will be erased with this command.

After examining disk B, reboot the CP/M system and return to drive A for further operations.

The transient commands are then copied from drive A to drive B using the PIP program. The sequence of commands shown below, for example, copy the principal programs from a standard CP/M diskette to the new diskette:

```
A>PIP↵
*B:STAT.COM=STAT.COM↵
*B:PIP.COM=PIP.COM↵
*B:LOAD.COM=LOAD.COM↵
*B:ED.COM=ED.COM↵
```

```
*B:ASM.COM=ASM.COM
*B:SYSGEN.COM=SYSGEN.COM
*B:DDT.COM=DDT.COM
*
A>
```

The user should then log in disk B, and type the command

```
DIR *.*
```

to ensure that the files were transferred to drive B from drive A. The various programs can then be tested on drive B to check that they were transferred properly.

Note that the copy operation can be simplified somewhat by creating a "submit" file which contains the copy commands. The file could be named GEN.SUB, for example, and might contain

```
SYSGEN
PIP B:STAT.COM=STAT.COM
PIP B:PIP.COM=PIP.COM
PIP B:LOAD.COM=LOAD.COM
PIP B:ED.COM=ED.COM
PIP B:ASM.COM=ASM.COM
PIP B:SYSGEN.COM=SYSGEN.COM
PIP B:DDT.COM=DDT.COM
```

The generation of a new diskette from the standard diskette is then done by typing simply

```
SUBMIT GEN
```

5. CP/M ENTRY POINT SUMMARY

The functions shown below summarize the functions of the FDOS. The function number is passed in Register C (first parameter in PL/M), and the information is passed in Registers D,E (second PL/M parameter). Single byte results are returned in Register A. If a double byte result is returned, then the high-order byte comes back in Register B (normal PL/M return). The transient program enters the FDOS through location "entry" (see Section 7.) as shown in Section 2. for PL/M, or

```
CALL entry
```

in assembly language. All registers are altered in the FDOS.

Function	Number	Information	Result
0	System Reset		
1	Read Console		ASCII character
2	Write Console	ASCII character	
3	Read Reader		ASCII character
4	Write Punch	ASCII character	
5	Write List	ASCII character	
6	(not used)		
7	Interrogate I/O Status		I/O Status Byte
8	Alter I/O Status	I/O Status Byte	
9	Print Console Buffer	Buffer Address	
10	Read Console Buffer	Buffer Address	
11	Check Console Status		True if character Ready
12	Lift Disk Head		
13	Reset Disk System		
14	Select Disk	Disk number	
15	Open File	FCB Address	Completion Code
16	Close File	" "	" "
17	Search First	" "	" "
18	Search Next	" "	" "
19	Delete File	" "	" "
20	Read Record	" "	" "
21	Write Record	" "	" "
22	Create File	" "	" "
23	Rename File	" "	" "
24	Interrogate Login		Login Vector
25	Interrogate Disk		Selected Disk Number
26	Set DMA Address	DMA Address	
27	Interrogate Allocation		Address of Allocation Vector

6. ADDRESS ASSIGNMENTS

The standard distribution version of CP/M is organized for an Intel MDS microcomputer developmental system with 16K of main memory, and two diskette drives. Larger systems are available in 16K increments, providing management of 32K, 48K, and 64K systems (the largest MDS system is 62K since the ROM monitor provided with the MDS resides in the top 2K of the memory space). For each additional 16K increment, add 4000H to the values of cbase and fbase.

The address assignments are

boot = 0000H	warm start operation
tfcb = 005CH	default file control block location
tbuff= 0080H	default buffer location
tbase= 0100H	base of transient program area
cbase= 2900H	base of console command processor
fbase= 3200H	base of disk operating system
entry= 0005H	entry point to disk system from user programs

7. SAMPLE PROGRAMS

This section contains two sample programs which interface with the CP/M operating system. The first program is written in assembly language, and is the source program for the DUMP utility. The second program is the CP/M LOAD utility, written in PL/M.

The assembly language program begins with a number of "equates" for system entry points and program constants. The equate

```
BDOS    EQU    0005H
```

for example, gives the CP/M entry point for peripheral I/O functions. The defualt file control block address is also defined (FCB), along with the default buffer address (BUFF). Note that the program is set up to run at location 100H, which is the base of the transient program area. The stack is first set-up by saving the entry stack pointer into OLDSP, and resetting SP to the local stack. The stack pointer upon entry belongs to the console command processor, and need not be saved unless control is to return to the CCP upon exit. That is, if the program terminates with a reboot (branch to location 0000H) then the entry stack pointer need not be saved.

The program then jumps to MAIN, past a number of subroutines which are listed below:

BREAK - when called, checks to see if there is a console character ready. BREAK is used to stop the listing at the console

PCHAR - print the character which is in register A at the console.

CRLF - send carriage return and line feed to the console

PNIB - print the hexadecimal value in register A in ASCII at the console

PHEX - print the byte value (two ASCII characters) in register A at the console

ERR - print error flag #n at the console, where n is

1 if file cannot be opened
2 if disk read error occurred

GNB - get next byte of data from the input file. If the IBP (input buffer pointer) exceeds the size of the input buffer, then another disk record of 128 bytes is read. Otherwise, the next character in the buffer is returned. IBP is updated to point to the next character.

The MAIN program then appears, which begins by calling SETUP. The SETUP subroutine, discussed below, opens the input file and checks for errors. If the file is opened properly, the GLOOP (get loop) label gets control.

On each successive pass through the GLOOP label, the next data byte is fetched using GNB and save in register B. The line addresses are listed every sixteen bytes, so there must be a check to see if the least significant 4 bits is zero on each output. If so, the line address is taken from registers h and l, and typed at the left of the line. In all cases, the byte which was previously saved in register B is brought back to register A, following label NONUM, and printed in the output line. The cycle through GLOOP continues until an end of file condition is detected in DISKR, as described below. Thus, the output lines appear as

```
0000  bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb
0010  bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb
                ...
```

until the end of file.

The label FINIS gets control upon end of file. CRLF is called first to return the carriage from the last line output. The CCP stack pointer is then reclaimed from OLDSP, followed by a RET to return to the console command processor. Note that a JMP 0000H could be used following the FINIS label, which would cause the CP/M system to be brought in again from the diskette (this operation is necessary only if the CCP has been overlayed by data areas).

The file control block format is then listed (FCBDN ... FCBLN) which overlays the fcb at location 05CH which is setup by the CCP when the DUMP program is initiated. That is, if the user types

```
DUMP  X.Y
```

then the CCP sets up a properly formed fcb at location 05CH for the DUMP (or any other) program when it goes into execution. Thus, the SETUP subroutine simply addresses this default fcb, and calls the disk system to open it. The DISKR (disk read) routine is called whenever GNB needs another buffer full of data. The default buffer at location 80H is used, along with a pointer (IBP) which counts bytes as they are processed. Normally, an end of file condition is taken as either an ASCII 1AH (control-z), or an end of file detection by the DOS. The file dump program, however, stops only on a DOS end of file.

```
                        ;       FILE DUMP PROGRAM, READS AN INPUT FILE AND PRINTS IT
                        ;
                        ;       COPYRIGHT (C), DIGITAL RESEARCH, 1975, 1976
                        ;
0100                            ORG     100H
0005 =                  BDOS    EQU     0005H   ;DOS ENTRY POINT
000F =                  OPENF   EQU     15      ;FILE OPEN
0014 =                  READF   EQU     20      ;READ FUNCTION
0002 =                  TYPEF   EQU     2       ;TYPE FUNCTION
0001 =                  CONS    EQU     1       ;READ CONSOLE
000B =                  BRKF    EQU     11      ;BREAK KEY FUNCTION (TRUE IF CHAR READY)
                        ;
005C =                  FCB     EQU     5CH     ;FILE CONTROL BLOCK ADDRESS
0080 =                  BUFF    EQU     80H     ;INPUT DISK BUFFER ADDRESS
                        ;
                        ;       SET UP STACK
0100 210000                     LXI     H,0
0103 39                         DAD     SP
0104 220F01                     SHLD    OLDSP
0107 315101                     LXI     SP,STKTOP
010A C3C401                     JMP     MAIN
                        ;       VARIABLES
010D                    IBP:    DS      2       ;INPUT BUFFER POINTER
                        ;
                        ;       STACK AREA
010F                    OLDSP:  DS      2
0111                    STACK:  DS      64
0151 =                  STKTOP  EQU     $
                        ;
                        ;       SUBROUTINES
                        ;
                        BREAK: ;CHECK BREAK KEY (ACTUALLY ANY KEY WILL DO)
0151 E5D5C5                     PUSH H! PUSH D! PUSH B; ENVIRONMENT SAVED
0154 0E0B                       MVI     C,BRKF
0156 CD0500                     CALL    BDOS
0159 C1D1E1                     POP B! POP D! POP H; ENVIRONMENT RESTORED
015C C9                         RET
                        ;
                        PCHAR: ;PRINT A CHARACTER
015D E5D5C5                     PUSH H! PUSH D! PUSH B; SAVED
0160 0E02                       MVI     C,TYPEF
0162 5F                         MOV     E,A
0163 CD0500                     CALL    BDOS
0166 C1D1E1                     POP B! POP D! POP H; RESTORED
0169 C9                         RET
                        ;
                        CRLF:
016A 3E0D                       MVI     A,0DH
016C CD5D01                     CALL    PCHAR
016F 3E0A                       MVI     A,0AH
0171 CD5D01                     CALL    PCHAR
0174 C9                         RET
                        ;
                        ;
                        PNIB:   ;PRINT NIBBLE IN REG A
0175 E60F                       ANI     0FH     ;LOW 4 BITS
0177 FE0A                       CPI     10
0179 D28101                     JNC     P10
```

```
                        ;       LESS THAN OR EQUAL TO 9
017C C630               ;       ADI     '0'
017E C38301             ;       JMP     PRN
                        ;
                        ;       GREATER OR EQUAL TO 10                  26
0181 C637               P10:    ADI     'A' - 10
0183 CD5D01             PRN:    CALL    PCHAR
0186 C9                 ;       RET
                        ;
                        PHEX:   ;PRINT HEX CHAR IN REG A
0187 F5                 ;       PUSH    PSW
0188 0F                 ;       RRC
0189 0F                 ;       RRC
018A 0F                 ;       RRC
018B 0F                 ;       RRC
018C CD7501             ;       CALL    PNIB    ;PRINT NIBBLE
018F F1                 ;       POP     PSW
0190 CD7501             ;       CALL    PNIB
0193 C9                 ;       RET
                        ;
                        ERR:    ;PRINT ERROR MESSAGE
0194 CD6A01             ;       CALL    CRLF
0197 3E23               ;       MVI     A,'#'
0199 CD5D01             ;       CALL    PCHAR
019C 78                 ;       MOV     A,B
019D C630               ;       ADI     '0'
019F CD5D01             ;       CALL    PCHAR
01A2 CD6A01             ;       CALL    CRLF
01A5 C3F701             ;       JMP     FINIS
                        ;
                        GNB:    ;GET NEXT BYTE
01A8 3A0D01             ;       LDA     IBP
01AB FE80               ;       CPI     80H
01AD C2B401             ;       JNZ     G0
                        ;       READ ANOTHER BUFFER
                        ;
                        ;
01B0 CD1602             ;       CALL    DISKR
01B3 AF                 ;       XRA     A
                        ;
                        G0:     ;READ THE BYTE AT BUFF+REG A
01B4 5F                 ;       MOV     E,A
01B5 1600               ;       MVI     D,0
01B7 3C                 ;       INR     A
01B8 320D01             ;       STA     IBP
                        ;       POINTER IS INCREMENTED
                        ;       SAVE THE CURRENT FILE ADDRESS
01BB E5                 ;       PUSH    H
01BC 218000             ;       LXI     H,BUFF
01BF 19                 ;       DAD     D
01C0 7E                 ;       MOV     A,M
                        ;       BYTE IS IN THE ACCUMULATOR
                        ;
                        ;       RESTORE FILE ADDRESS AND INCREMENT
01C1 E1                 ;       POP     H
01C2 23                 ;       INX     H
01C3 C9                 ;       RET
                        ;
                        MAIN:   ; READ AND PRINT SUCCESSIVE BUFFERS
01C4 CDFF01             ;       CALL    SETUP   ;SET UP INPUT FILE
```

```
01C7 3E80               MVI     A,80H
01C9 320D01             STA     IBP     ;SET BUFFER POINTER TO 80H
01CC 21FFFF             LXI     H,0FFFFH        ;SET TO -1 TO START
                    ;
                    GLOOP:
01CF CDA801             CALL    GNB
01D2 47                 MOV     B,A
                    ;       PRINT HEX VALUES
                    ;       CHECK FOR LINE FOLD
01D3 7D                 MOV     A,L
01D4 E60F               ANI     0FH     ;CHECK LOW 4 BITS
01D6 C2EB01             JNZ     NONUM
                    ;       PRINT LINE NUMBER
01D9 CD6A01             CALL    CRLF
                    ;
                    ;       CHECK FOR BREAK KEY
01DC CD5101             CALL    BREAK
01DF 0F                 RRC
01E0 DAF701             JC      FINIS   ;DON'T PRINT ANY MORE
                    ;
01E3 7C                 MOV     A,H
01E4 CD8701             CALL    PHEX
01E7 7D                 MOV     A,L
01E8 CD8701             CALL    PHEX
                    NONUM:
01EB 3E20               MVI     A,' '
01ED CD5D01             CALL    PCHAR
01F0 78                 MOV     A,B
01F1 CD8701             CALL    PHEX

01F4 C3CF01             JMP     GLOOP
                    ;
                    EPSA:   ;END PSA
                            ;       END OF INPUT
                    FINIS:
01F7 CD6A01             CALL    CRLF
01FA 2A0F01             LHLD    OLDSP
01FD F9                 SPHL
01FE C9                 RET
                    ;
                    ;
                    ;       FILE CONTROL BLOCK DEFINITIONS
005C =              FCBDN   EQU     FCB+0   ;DISK NAME
005D =              FCBFN   EQU     FCB+1   ;FILE NAME
0065 =              FCBFT   EQU     FCB+9   ;DISK FILE TYPE (3 CHARACTERS)
0068 =              FCBRL   EQU     FCB+12  ;FILE'S CURRENT REEL NUMBER
006B =              FCBRC   EQU     FCB+15  ;FILE'S RECORD COUNT (0 TO 128)
007C =              FCBCR   EQU     FCB+32  ;CURRENT (NEXT) RECORD NUMBER (0 TO 127)
007D =              FCBLN   EQU     FCB+33  ;FCB LENGTH
                    ;
                    ;
                    SETUP:  ;SET UP FILE
                    ;       OPEN THE FILE FOR INPUT
01FF 115C00             LXI     D,FCB
0202 0E0F               MVI     C,OPENF
0204 CD0500             CALL    BDOS
                    ;       CHECK FOR ERRORS
0207 FEFF               CPI     255
0209 C21102             JNZ     OPNOK
```

```
                ;       BAD OPEN
020C 0601               MVI     B,1     ;OPEN ERROR
020E CD9401             CALL    ERR
                ;
                OPNOK:  ;OPEN IS OK.
0211 AF                 XRA     A
0212 327C00             STA     FCBCR
0215 C9                 RET
                ;
                DISKR:  ;READ DISK FILE RECORD
0216 E5D5C5             PUSH H! PUSH D! PUSH B
0219 115C00             LXI     D,FCB
021C 0E14               MVI     C,READF
021E CD0500             CALL    BDOS
0221 C1D1E1             POP B! POP D! POP H
0224 FE00               CPI     0       ;CHECK FOR ERRS
0226 C8                 RZ
                ;       MAY BE EOF
0227 FE01               CPI     1
0229 CAF701             JZ      FINIS
                ;
022C 0602               MVI     B,2     ;DISK READ ERROR
022E CD9401             CALL    ERR
                ;
0231                    END
```

The PL/M program which follows implements the CP/M LOAD utility. The function is as follows. The user types

LOAD filename

If filename.HEX exists on the diskette, then the LOAD utility reads the "hex" formatted machine code file and produces the file

filename.COM

where the COM file contains an absolute memory image of the machine code, ready for load and execution in the TPA. If the file does not appear on the diskette, the LOAD program types

SOURCE IS READER

and reads an Addmaster paper tape reader which contains the hex file.

The LOAD program is set up to load and run in the TPA, and, upon completion, return to the CCP without rebooting the system. Thus, the program is constructed as a single procedure called LOADCOM which takes the form

```
0FAH:
   LOADCOM: PROCEDURE;
      /* LIBRARY PROCEDURES */
      MON1: ...
      /* END LIBRARY PROCEDURES */
      MOVE: ...
      GETCHAR: ...
      PRINTNIB: ...
      PRINTHEX: ...
      PRINTADDR: ...
      RELOC: ...
         SETMEM:
         READHEX:
         READBYTE:
         READCS:
         MAKEDOUBLE:
         DIAGNOSE:
      END RELOC;

      DECLARE STACK(16) ADDRESS, SP ADDRESS;
      SP = STACKPTR; STACKPTR = .STACK(LENGTH(STACK));

      ...
      CALL RELOC;
      ...
      STACKPTR = SP;
      RETURN 0;
   END LOADCOM;
   ;
   EOF
```

The label 0FAH at the beginning sets the origin of the compilation to 0FAH, which causes the first 6 bytes of the compilation to be ignored when loaded (i.e., the TPA starts at location 100H and thus 0FAH,...,0FFH are deleted from the COM file). In a PL/M compilation, these 6 bytes are used to set up the stack pointer and branch around the subroutines in the program. In this case, there is only one subroutine, called LOADCOM, which results in the following machine memory image for LOAD

```
0FAH: LXI SP,plmstack      ;SET SP TO DEFAULT STACK
0FDH: JMP pastsubr         ;JUMP AROUND LOADCOM
100H: beginning of LOADCOM procedure
      ....
      end of LOADCOM procedure
      RET

pastsubr:
      EI
      HLT
```

Since the machine code between 0FAH and 0FFH is deleted in the load, execution actually begins at the top of LOADCOM. Note, however, that the initialization of the SP to the default stack has also been deleted; thus, there is a declaration and initialization of an explicit stack and stack pointer before the call to RELOC at the end of LOADCOM. This is necessary only if we wish to return to the CCP without a reboot operation: otherwise the origin of the program is set to 100H, the declaration of LOADCOM as a procedure is not necessary, and termination is accomplished by simply executing a

```
GO TO 0000H;
```

at the end of the program. Note also that the overhead for a system reboot is not great (approximately 2 seconds), but can be bothersome for system utilities which are used quite often, and do not need the extra space.

The procedures listed in LOADCOM as "library procedures" are a standard set of PL/M subroutines which are useful for CP/M interface. The RELOC procedure contains several nested subroutines for local functions, and actually performs the load operation when called from LOADCOM. Control initially starts on line 327 where the stackpointer is saved and re-initialized to the local stack. The default file control block name is copied to another file control block (SFCB) since two files may be open at the same time. The program then calls SEARCH to see if the HEX file exists; if not, then the high speed reader is used. If the file does exist, it is opened for input (if possible). The filetype COM is moved to the default file control block area, and any existing copies of filename.COM files are removed from the diskette before creating a new file. The MAKE operation creates a new file, and, if successful, RELOC is called to read the HEX file and produce the COM file. At the end of processing by RELOC, the COM file is closed (line 350). Note that the HEX file does not need to be closed since it was opened for input only. The data written to a file is not permanently recorded until the file is successfully closed.

Disk input characters are read through the procedure GETCHAR on line 137. Although the DMA facilities of CP/M could be used here, the GETCHAR procedure instead uses the default buffer at location 80H and moves each buffer into a vector called SBUFF (source buffer) as it is read. On exit, the GETCHAR procedure returns the next input character and updates the source buffer pointer (SBP).

The SETMEM procedure on line 191 performs the opposite function from GETCHAR. The SETMEM procedure maintains a buffer of loaded machine code in pure binary form which acts as a "window" on the loaded code. If there is an attempt by RELOC to write below this window, then the data is ignored. If the data is within the window, then it is placed into MBUFF (memory buffer). If the data is to be placed above this window, then the window is moved up to the point where it would include the data address by writing the memory image successively (by 128 byte buffers), and moving the base address of the window. Using this technique, the programmer can recover from checksum errors on the high-speed reader by stopping the reader, rewinding the tape for some distance, then restarting LOAD (in this case, LOADing is resumed by interrupting with a NOP instruction). Again, the SETMEM procedure uses the default buffer at location 80H to perform the disk output by moving 128 byte segments to 80H through 0FFH before each write.

```
 00001  1
 00002  1   0FAH: DECLARE BDOS LITERALLY '0005H';
 00003  1     /* TRANSIENT COMMAND LOADER PROGRAM
 00004  1
 00005  1                COPYRIGHT (C) DIGITAL RESEARCH
 00006  1                        JUNE, 1975
 00007  1    */
 00008  1
 00009  1   LOADCOM: PROCEDURE BYTE;
 00010  2       DECLARE FCBA ADDRESS INITIAL(5CH);
 00011  2       DECLARE FCB BASED FCBA (33) BYTE;
 00012  2
 00013  2       DECLARE BUFFA ADDRESS INITIAL(80H), /* I/O BUFFER ADDR
ESS */
 00014  2           BUFFER BASED BUFFA (128) BYTE;
 00015  2
 00016  2       DECLARE SFCB(33) BYTE,  /* SOURCE FILE CONTROL BLOCK *
/
 00017  2           BSIZE LITERALLY '1024',
 00018  2           EOFILE LITERALLY '1AH',
 00019  2           SBUFF(BSIZE) BYTE  /* SOURCE FILE BUFFER */
 00020  2               INITIAL(EOFILE),
 00021  2           RFLAG BYTE,          /* READER FLAG */
 00022  2           SBP ADDRESS;         /* SOURCE FILE BUFFER POINTER
*/
 00023  2
 00024  2       /* LOADCOM LOADS TRANSIENT COMMAND FILES TO THE DISK F
ROM THE
 00025  2       CURRENTLY DEFINED READER PERIPHERAL.  THE LOADER PLACE
S THE MACH
 00026  2       CODE INTO A FILE WHICH APPEARS IN THE LOADCOM COMMAND
*/
 00027  2    /* ***************** LIBRARY PROCEDURES FOR DISKIO *******
****** */
 00028  2
 00029  2   MON1: PROCEDURE(F,A);
 00030  3       DECLARE F BYTE,
 00031  3       A ADDRESS;
 00032  3       GO TO BDOS;
 00033  3       END MON1;
 00034  2
 00035  2   MON2: PROCEDURE(F,A) BYTE;
 00036  3       DECLARE F BYTE,
 00037  3       A ADDRESS;
 00038  3       GO TO BDOS;
 00039  3       END MON2;
 00040  2
 00041  2   READRDR: PROCEDURE BYTE;
 00042  3       /* READ CURRENT READER DEVICE */
 00043  3       RETURN MON2(3,0);
 00044  3       END READRDR;
 00045  2
 00046  2   DECLARE
 00047  2       TRUE LITERALLY '1',
 00048  2       FALSE LITERALLY '0',
 00049  2       FOREVER LITERALLY 'WHILE TRUE',
 00050  2       CR LITERALLY '13',
```

```
00051  2       LF LITERALLY '10',
00052  2       WHAT LITERALLY '63';
00053  2
00054  2    PRINTCHAR: PROCEDURE(CHAR);
00055  3        DECLARE CHAR BYTE;
00056  3        CALL MON1(2,CHAR);
00057  3        END PRINTCHAR;
00058  2
00059  2    CRLF: PROCEDURE;
00060  3        CALL PRINTCHAR(CR);
00061  3        CALL PRINTCHAR(LF);
00062  3        END CRLF;
00063  2
00064  2    PRINT: PROCEDURE(A);
00065  3        DECLARE A ADDRESS;
00066  3        /* PRINT THE STRING STARTING AT ADDRESS A UNTIL THE
00067  3        NEXT DOLLAR SIGN IS ENCOUNTERED */
00068  3        CALL CRLF;
00069  3        CALL MON1(9,A);
00070  3        END PRINT;
00071  2
00072  2    DECLARE DCNT BYTE;
00073  2
00074  2    INITIALIZE: PROCEDURE;
00075  3        CALL MON1(13,0);
00076  3        END INITIALIZE;
00077  2
00078  2    SELECT: PROCEDURE(D);
00079  3        DECLARE D BYTE;
00080  3        CALL MON1(14,D);
00081  3        END SELECT;
00082  2
00083  2    OPEN: PROCEDURE(FCB);
00084  3        DECLARE FCB ADDRESS;
00085  3        DCNT = MON2(15,FCB);
00086  3        END OPEN;
00087  2
00088  2    CLOSE: PROCEDURE(FCB);
00089  3        DECLARE FCB ADDRESS;
00090  3        DCNT = MON2(16,FCB);
00091  3        END CLOSE;
00092  2
00093  2    SEARCH: PROCEDURE(FCB);
00094  3        DECLARE FCB ADDRESS;
00095  3        DCNT = MON2(17,FCB);
00096  3        END SEARCH;
00097  2
00098  2    SEARCHN: PROCEDURE;
00099  3        DCNT = MON2(18,0);
00100  3        END SEARCHN;
00101  2
00102  2    DELETE: PROCEDURE(FCB);
00103  3        DECLARE FCB ADDRESS;
00104  3        CALL MON1(19,FCB);
00105  3        END DELETE;
00106  2
00107  2    DISKREAD: PROCEDURE(FCB) BYTE;
00108  3        DECLARE FCB ADDRESS;
00109  3        RETURN MON2(20,FCB);
00110  3        END DISKREAD;
```

```
00111  2
00112  2   DISKWRITE: PROCEDURE(FCB) BYTE;
00113  3       DECLARE FCB ADDRESS;
00114  3       RETURN MON2(21,FCB);
00115  3       END DISKWRITE;
00116  2
00117  2   MAKE: PROCEDURE(FCB);
00118  3       DECLARE FCB ADDRESS;
00119  3       DCNT = MON2(22,FCB);
00120  3       END MAKE;
00121  2
00122  2   RENAME: PROCEDURE(FCB);
00123  3       DECLARE FCB ADDRESS;
00124  3       CALL MON1(23,FCB);
00125  3       END RENAME;
00126  2
00127  2    /* ******************* END OF LIBRARY PROCEDURES ********
***** */
00128  2
00129  2       MOVE: PROCEDURE(S,D,N);
00130  3           DECLARE (S,D) ADDRESS, N BYTE,
00131  3           A BASED S BYTE, B BASED D BYTE;
00132  3               DO WHILE (N:=N-1) <> 255;
00133  3               B = A; S=S+1; D=D+1;
00134  4               END;
00135  3           END MOVE;
00136  2
00137  2       GETCHAR: PROCEDURE BYTE;
00138  3           /* GET NEXT CHARACTER */
00139  3           DECLARE I BYTE;
00140  3           IF RFLAG THEN RETURN READRDR;
00141  3           IF (SBP := SBP+1) <= LAST(SBUFF) THEN
00142  3               RETURN SBUFF(SBP);
00143  3           /* OTHERWISE READ ANOTHER BUFFER FULL */
00144  3               DO SBP = 0 TO LAST(SBUFF) BY 128;
00145  3               IF (I:=DISKREAD(.SFCB)) = 0 THEN
00146  4                   CALL MOVE(80H,.SBUFF(SBP),80H); ELSE
00147  4                   DO; IF I<>1 THEN CALL PRINT(.'DISK READ ER
ROR$');
00148  5                   SBUFF(SBP) = EOFILE;
00149  5                   SBP = LAST(SBUFF);
00150  5                   END;
00151  4               END;
00152  3           SBP = 0; RETURN SBUFF;
00153  3           END GETCHAR;
00154  2   DECLARE
00155  2       STACKPOINTER LITERALLY 'STACKPTR';
00156  2
00157  2
00158  2   PRINTNIB: PROCEDURE(N);
00159  3       DECLARE N BYTE;
00160  3           IF N > 9 THEN CALL PRINTCHAR(N+'A'-10); ELSE
00161  3           CALL PRINTCHAR(N+'0');
00162  3       END PRINTNIB;
00163  2
00164  2   PRINTHEX: PROCEDURE(B);
00165  3       DECLARE B BYTE;
00166  3           CALL PRINTNIB(SHR(B,4)); CALL PRINTNIB(B AND 0FH);
00167  3       END PRINTHEX;
00168  2
```

```
00169  2   PRINTADDR: PROCEDURE(A);
00170  3      DECLARE A ADDRESS;
00171  3      CALL PRINTHEX(HIGH(A)); CALL PRINTHEX(LOW(A));
00172  3      END PRINTADDR;
00173  2
00174  2
00175  2    /* INTEL HEX FORMAT LOADER */
00176  2
00177  2   RELOC: PROCEDURE;
00178  3      DECLARE (RL, CS, RT) BYTE;
00179  3      DECLARE
00180  3          LA ADDRESS,     /* LOAD ADDRESS */
00181  3          TA ADDRESS,     /* TEMP ADDRESS */
00182  3          SA ADDRESS,     /* START ADDRESS */
00183  3          FA ADDRESS,     /* FINAL ADDRESS */
00184  3          NB ADDRESS,     /* NUMBER OF BYTES LOADED */
00185  3          SP ADDRESS,     /* STACK POINTER UPON ENTRY TO REL
OC */
00186  3
00187  3      MBUFF(256) BYTE,
00188  3      P BYTE,
00189  3      L ADDRESS;
00190  3
00191  3      SETMEM: PROCEDURE(B);
00192  4          /* SET MBUFF TO B AT LOCATION LA MOD LENGTH(MBUFF)
*/
00193  4          DECLARE (B,I) BYTE;
00194  4          IF LA < L THEN /* MAY BE A RETRY */ RETURN;
00195  4              DO WHILE LA > L + LAST(MBUFF); /* WRITE A PARA
GRAPH */
00196  4                  DO I = 0 TO 127; /* COPY INTO BUFFER */
00197  5                  BUFFER(I) = MBUFF(LOW(L)); L = L + 1;
00198  6                  END;
00199  5              /* WRITE BUFFER ONTO DISK */
00200  5              P = P + 1;
00201  5              IF DISKWRITE(FCBA) <> 0 THEN
00202  5                  DO; CALL PRINT(.'DISK WRITE ERROR$');
00203  6                  HALT;
00204  6                  /* RETRY AFTER INTERRUPT NOP */
00205  6                  L = L - 128;
00206  6                  END;
00207  5              END;
00208  4          MBUFF(LOW(LA)) = B;
00209  4          END SETMEM;
00210  3
00211  3      READHEX: PROCEDURE BYTE;
00212  4          /* READ ONE HEX CHARACTER FROM THE INPUT */
00213  4          DECLARE H BYTE;
00214  4          IF (H := GETCHAR) - '0' <= 9 THEN RETURN H - '0';
00215  4          IF H - 'A' > 5 THEN GO TO CHARERR;
00216  4          RETURN H - 'A' + 10;
00217  4          END READHEX;
00218  3
00219  3      READBYTE: PROCEDURE BYTE;
00220  4          /* READ TWO HEX DIGITS */
00221  4          RETURN SHL(READHEX,4) OR READHEX;
00222  4          END READBYTE;
00223  3
00224  3      READCS: PROCEDURE BYTE;
00225  4          /* READ BYTE WHILE COMPUTING CHECKSUM */
```

```
                    DECLARE B BYTE;
                    CS = CS + (B := READBYTE);
                    RETURN B;
                    END READCS;

MAKE$DOUBLE: PROCEDURE(H,L) ADDRESS;
    /* CREATE A BOUBLE BYTE VALUE FROM TWO SINGLE BYTE
    */
    DECLARE (H,L) BYTE;
    RETURN SHL(DOUBLE(H),8) OR L;
    END MAKE$DOUBLE;

DIAGNOSE: PROCEDURE;

DECLARE M BASED TA BYTE;

NEWLINE: PROCEDURE;
    CALL CRLF; CALL PRINTADDR(TA); CALL PRINTCHAR(':')
;
    CALL PRINTCHAR(' ');
    END NEWLINE;

/* PRINT DIAGNOSTIC INFORMATION AT THE CONSOLE */
    CALL PRINT(.'LOAD  ADDRESS $'); CALL PRINTADDR(TA);
    CALL PRINT(.'ERROR ADDRESS $'); CALL PRINTADDR(LA);

    CALL PRINT(.'BYTES READ:$'); CALL NEWLINE;
    DO WHILE TA < LA;
    IF (LOW(TA) AND 0FH) = 0  THEN CALL NEWLINE;
    CALL PRINTHEX(MBUFF(TA-L)); TA=TA+1;
    CALL PRINTCHAR(' ');
    END;
CALL CRLF;
HALT;
END DIAGNOSE;


/* INITIALIZE */
SA, FA, NB = 0;
SP = STACKPOINTER;
P = 0; /* PARAGRAPH COUNT */
TA,LA,L = 100H;  /* BASE ADDRESS OF TRANSIENT ROUTINES
*/
IF FALSE THEN
    CHARERR:  /* ARRIVE HERE IF NON-HEX DIGIT IS ENCOU
NTERED */
    DO; /* RESTORE STACKPOINTER */ STACKPOINTER = SP;
    CALL PRINT(.'NON-HEXADECIMAL DIGIT ENCOUNTERED $')
;
    CALL DIAGNOSE;
    END;


/* READ RECORDS UNTIL :00XXXX IS ENCOUNTERED */

    DO FOREVER;
    /* SCAN THE : */
        DO WHILE GETCHAR <> ':';
        END;
```

```
00280  4
00281  4          /* SET CHECK SUM TO ZERO, AND SAVE THE RECORD LENG
TH */
00282  4          CS = 0;
00283  4          /* MAY BE THE END OF TAPE */                    37
00284  4          IF (RL := READCS) = 0 THEN
00285  4                  GO TO FIN;
00286  4          NB = NB + RL;
00287  4
00288  4          TA, LA = MAKE$DOUBLE(READCS,READCS);
00289  4          IF SA = 0 THEN SA = LA;
00290  4
00291  4
00292  4          /* READ THE RECORD TYPE (NOT CURRENTLY USED) */
00293  4          RT = READCS;
00294  4
00295  4          /* PROCESS EACH BYTE */
00296  4              DO WHILE (RL := RL - 1) <> 255;
00297  4              CALL SETMEM(READCS); LA = LA+1;
00298  5              END;
00299  4          IF LA > FA THEN FA = LA - 1;
00300  4
00301  4          /* NOW READ CHECKSUM AND COMPARE */
00302  4          IF CS + READBYTE <> 0 THEN
00303  4              DO; CALL PRINT(.'CHECK SUM ERROR $');
00304  5              CALL DIAGNOSE;
00305  5              END;
00306  4          END;
00307  3
00308  3      FIN:
00309  3      /* EMPTY THE BUFFERS */
00310  3      TA = LA;
00311  3          DO WHILE L < TA;
00312  3          CALL SETMEM(0); LA = LA+1;
00313  4          END;
00314  3      /* PRINT FINAL STATISTICS */
00315  3      CALL PRINT(.'FIRST ADDRESS $'); CALL PRINTADDR(SA);
00316  3      CALL PRINT(.'LAST  ADDRESS $'); CALL PRINTADDR(FA);
00317  3      CALL PRINT(.'BYTES READ    $'); CALL PRINTADDR(NB);
00318  3      CALL PRINT(.'RECORDS WRITTEN $'); CALL PRINTHEX(P);
00319  3      CALL CRLF;
00320  3
00321  3      END RELOC;
00322  2
00323  2   /* ARRIVE HERE FROM THE SYSTEM MONITOR, READY TO READ THE
HEX TAPE
00324  2
00325  2    /* SET UP STACKPOINTER IN THE LOCAL AREA */
00326  2   DECLARE STACK(16) ADDRESS, SP ADDRESS;
00327  2   SP = STACKPOINTER; STACKPOINTER = .STACK(LENGTH(STACK));
00328  2
00329  2   SBP = LENGTH(SBUFF);
00330  2      /* SET UP THE SOURCE FILE */
00331  2       CALL MOVE(FCBA,.SFCB,33);
00332  2       CALL MOVE(.('HEX',0),.SFCB(9),4);
00333  2       CALL SEARCH(.SFCB);
00334  2       IF (RFLAG := DCNT = 255) THEN
00335  2           CALL PRINT(.'SOURCE IS READER$'); ELSE
00336  2           DO; CALL PRINT(.'SOURCE IS DISK$');
```

LENG

```
00337  3          CALL OPEN(.SFCB);
00338  3          IF DCNT = 255 THEN CALL PRINT(.'-CANNOT OPEN SOURC
E$');
00339  3          END;                                              38
00340  2       CALL CRLF;
00341  2
00342  2       CALL MOVE(.'COM',FCBA+9,3);
00343  2
00344  2    /* REMOVE ANY EXISTING FILE BY THIS NAME */
00345  2    CALL DELETE(FCBA);
00346  2    /* THEN OPEN A NEW FILE */
00347  2    CALL MAKE(FCBA); FCB(32) = 0; /* CREATE AND SET NEXT RECORD */
00348  2    IF DCNT = 255 THEN CALL PRINT(.'NO MORE DIRECTORY SPACE$'
); ELSE
00349  2          DO; CALL RELOC;
00350  3          CALL CLOSE(FCBA);
00351  3          IF DCNT = 255 THEN CALL PRINT(.'CANNOT CLOSE FILE$
');
00352  3          END;
00353  2    CALL CRLF;
00354  2
00355  2    /* RESTORE STACKPOINTER FOR RETURN */
00356  2    STACKPOINTER = SP;
00357  2    RETURN 0;
00358  2    END LOADCOM;
00359  1    ;
00360  1    EOF
```

EXHIBIT P

A P-I SPECIAL REPORT

Bill Gates: Of Mind and Money

Business ties to IBM start with a new tie

Third of Five Parts

By James Wallace and Jim Erickson
P-I Reporters

It was one of the most important meetings of his life, and Bill Gates didn't have a tie.

He had spent a sleepless night on the red-eye from Seattle to Miami, feeding his photographic memory with bits and bytes of business and technical information for a meeting with IBM executives at their offices up the Florida coast in Boca Raton.

Gates carried with him a final report on how the jeans-and-tennis-shoe boys at Microsoft could work with the white-shirts-and-wingtip crowd at IBM on Project Chess, code name for a secret IBM effort to develop a personal computer.

■ **Fair shares:** A 3-for-2 split of Microsoft stock. **Page B5**

As his rental car sped north from the Miami airport to Boca Raton, Gates, wired with excitement and lack of sleep, stopped at a department store and waited for it to open so he could buy a tie.

Normally, Gates is not big on appropriate dress. But, after all, this was IBM, an American institution with a work force more than half as large as the population of Seattle.

Although he was 15 minutes late, the meeting that morning went well. The 25-year-old Gates, new tie dangling from his neck, confidently answered question after question from much older executives.

A couple of months later, in November 1980, the corporate odd couple signed the papers: Microsoft would develop the operating system for IBM's personal computer. The operating system would be the base layer of software that controlled the computer's internal functions. Application programs used on top of the operating system would provide for specific uses, such as word processing or spreadsheets.

How Gates managed such a feat is largely unknown by the 40

See **BILL GATES**, Page A4

Bush's thyroid at heart of a

Doctors say it can be easily treated

By Julia Malone
Cox News Service

WASHINGTON — President Bush's irregular heartbeat was caused by a "mild" hyperthyroid condition that can be easily treated, his doctors said last night.

Dr. Burton Lee, the president's chief physician, said he was "very pleased" that tests taken over the weekend had traced Bush's episode of an uneven heart rate to thyroid overactivity, which he said could be cured relatively quickly.

Bush will return to Bethesda Naval Medical Center in Maryland today for tests to determine the cause of the overactive thyroid, doctors said.

The White House said the president would not be kept overnight at the hospital.

The condition, biochemical hyperthyroidism, is related to that suffered by Barbara Bush, who had an enlarged thyroid.

The president's doctors said they would continue tests this week to find out the exact type of Bush's thyroid disorder and determine the type of therapy, which they expected to administer next week.

Meanwhile, Lee said of the president, "He's fine, but we're not going to waste any time."

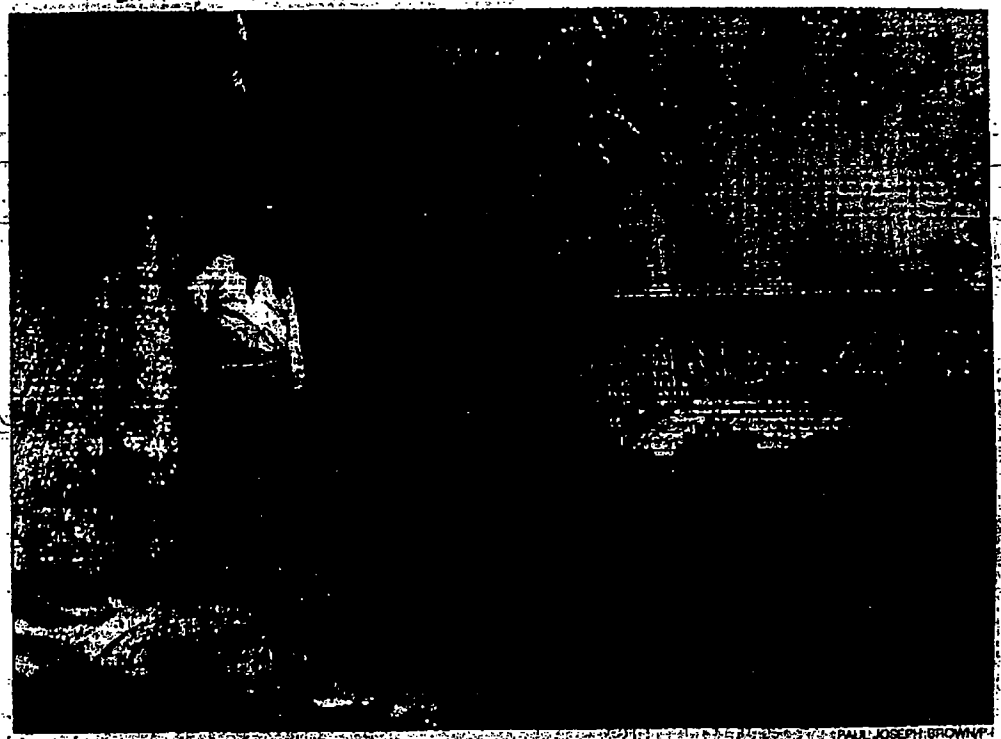
Dr. Kenneth Burman, an Army colonel and Walter Reed endrocrinologist who is also in charge of caring for Mrs. Bush, listed four possible treatments for the president: a radioactive iodine drink, which Mrs. Bush took; pills to counter the excess thyroid actions; surgery; and waiting for the problem to subside.

The president was "elated" when Lee brought the news to him yesterday afternoon, said Marlin Fitzwater, the White

House
Fit
ebrate
Lawn
Blond
"It
shows
heart
findin
ident'
cise -
cups-a
Bu
time
of the

See B

Shantytown's hidden society



PAUL JOSEPH BROWN/P-I

"Ranch" resident Sergio Maranda stands in the doorway of a hut on a hill where 12th Avenue South crosses over South Dearborn Street.

Health hazards doom longtime encampment

By Scott Maier
P-I Reporter

In some respects, Sergio Maranda's home has the trappings of middle-class life.

Nestled beneath a towering apple tree, the home is hidden from the noise and grime of its urban surroundings. A barbecue is at the front entrance. A tinfoil portal provides entry for Maranda's dog and two cats.

Inside are portraits of Marilyn Monroe and the Mona Lisa. By his freshly made bed is the inscription: "May life always bring you many reasons to smile and laugh."

But missing from Maranda's home is: electricity and water. His private commode is a pail and a white plastic bucket. The floor is made of dirt. A putrid stench hangs in the air.

Such are the living conditions for Maranda and a dozen or so other permanent residents of "the ranch," a shantytown concealed on a wooded hill where 12th Avenue South crosses over South Dearborn Street.

Though homeless encampments are not unusual, this community is distinctive because the residents are not transient. Some hold jobs and a few are said to have lived in the makeshift hovels for as long as six years.

Police have informed the residents that they have to go.

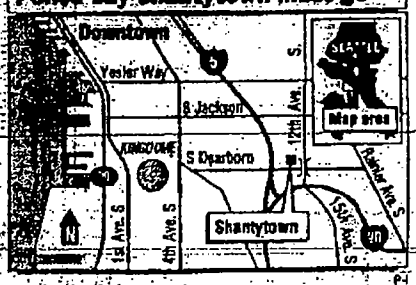
"We simply have to get them out of there because of the health hazards," said Sgt. John Manning, who heads the East Precinct's community police team. Police intend to clear out the area by the end of the month, but are holding off while city officials seek new homes for the residents.

Police have no intention of moving in quickly on the settlement, though the pace of the action depends on the site's private property owners, who had asked for police assistance, officials said.

The property is

See **SHANTYTOWN**, P

Police say shantytown must go



A P-I SPECIAL REPORT

Bill Gates: Of Mind and Money

Ties that bind Microsoft, IBM wearing thin

From Page 1

million or so people who have helped make him a very wealthy man by buying computers that use Microsoft's disk operating system, better known as MS-DOS.

Gates had bought DOS from an unsuspecting Seattle computer company for $50,000, then made the deal of the century with IBM.

Whether by luck or by genius, Chairman Bill had hitched a ride with one of the most respected and feared names in corporate America. It was the break his small Seattle software firm needed. DOS would become as important to the computer industry as the internal combustion engine to modern transportation. Almost overnight, it seemed, Microsoft was the most successful company on the planet, and Gates was the world's software tycoon, the Henry Ford of the computer age.

TYCOON TICKER

- Bill Gates owns 30,460,820 shares of Microsoft stock. Price per share, Tuesday close: $101.25
- Change since Monday: —$1.50
- Gates' loss: —$59.4 million

Don Estridge, the brilliant and maverick IBM manager who directed Project Chess, would later tell his friend Gates that IBM Chairman John Opel was impressed when he learned Microsoft and Big Blue might do some business together.

Even though he had not met Gates, Opel thought well of the young computer whiz from Seattle because he and Mary Gates, Bill's mother, had served together on the national board of United Way. How much this may have helped Microsoft get the IBM deal is not known. Opel, now retired, won't talk.

A lot has changed in the 10 years since.

For one thing, Mary Gates is no longer on United Way's national board. Her son is.

And the ties that bind Microsoft

Here's a betting man who lo

Chairman Bill loves to test his breadth of knowledge against others, especially when there is something to win or lose.

A woman friend remembers being at a restaurant with Gates when he was betting on all sorts of things, like who was older, Jimmy Treybig (founder of Tandem Computer Inc.) or Sam Armacost (then president

A lot has changed in the 10 years since.

For one thing, Mary Gates is no longer on United Way's national board. Her son is.

And the ties that bind Microsoft and IBM are today as threadbare as an old pair of jeans that have gone through too many washes.

The hardware and software titans of the computer world are busily trying to outflank the other, maneuvering like two mighty armies on a battlefield, each seeking an advantage.

Many industry insiders say Gates will win, as he always does. Others, however, are not so sure.

"NO ONE HAS EVER gotten into bed with IBM and failed to get kicked out," says Adam Osborne, who developed the Osborne 1 personal computer in the late 1970s. "I'm waiting to see if this guy (Gates) is going to be so smart that he is going to win and IBM is going to lose."

When IBM first contacted Gates in the summer of 1980, fewer than 30 people were working out of Microsoft's offices on the eighth floor of the Old National Bank building in downtown Bellevue.

Microsoft had moved there the previous year from Albuquerque, N.M.

The IBM representative who visited Microsoft in July 1980 asked a lot of questions, but with typical IBM secrecy gave no clues about what was afoot.

A month later, Gates got another call from IBM. Could he meet with several of the company's engineers the next day? Gates canceled a meeting with the chairman of Atari.

After signing an agreement that he would keep everything discussed at the meeting secret, Gates heard about Project Chess. IBM, he was told, was looking for someone to supply the software.

At the time, the standard operating system for personal computers was something known as Control Program/Monitor, or CP/M. It had been developed by Gary Kildall of Digital Research.

Kildall, whose father owned a navigation shop in Seattle, had received his Ph.D. in computer science from the University of Washington in 1972 and moved to the scenic coastal city of Pacific Grove, Calif. There, four years later, he and his wife formed Intergalactic Digital Research. The name later was shortened to Digital Research.

In 1980, his CP/M operating system had no serious competition.

Steve Wood, one of the original programmers at Microsoft in Albuquerque, says Gates had talked about developing an operating system, but nothing came of it.

"We kept asking ourselves if we should really be sending all this business to Gary," Wood says. "We always came back with the same answer: We have all this other stuff to do."

So when IBM came to Microsoft, Gates once again sent a customer to Kildall.

But the deal for CP/M fell through. Kildall reportedly refused to make changes IBM said it wanted in the operating system. Regardless of what really happened, most of those in the computer industry believe Kildall simply blew it, and thus rolled out the red carpet for the coronation of the new software king, Chairman Bill.

After sending IBM to talk with Kildall, Gates had been able to persuade the Project Chess engineers to design their computer around Intel's new 8086 microcomputer chip, which was far more powerful than other chips in use at the time and would allow more sophisticated software programs.

Gates knew just where to go for an operating system that would work on the 8086 chip: Up the road to Tukwila, to a mom-and-pop computer business called Seattle Computer Products. There, Tim Paterson had designed an operating system called 86-DOS for the company's new computer that used the 8086 chip.

Microsoft cofounder Paul Allen contacted Rod Brock, owner of Seattle Computer Products, and told him Microsoft had a customer for 86-DOS and wanted to further develop and market the operating system.

He couldn't, of course, tell Brock the customer was IBM.

Brock agreed, and Paterson went to work on the changes Microsoft wanted.

And Gates, who must have felt like he just filled an inside straight in poker, flew off to Boca Raton to make his final report to IBM and secure the deal.

"Gates was able to maximize a series of good fortune and lucky breaks," says Seymour Rubinstein, one of the pioneers of the personal computer revolution who founded the software company Wordstar.

"There was no foresight, no imagination, no brilliant maneuvers," he says, "just a lucky break caused by a combination of Digital Research screwing up and Seattle Computer Products having something which wasn't very good that could be modified for IBM."

Kildall, still chief executive of Digital Research, says there were "remarkable similarities" between MS-DOS and his CP/M operating system.

"Ask Bill why function code 6 (in DOS) ends with a dollar sign," Kildall says. "No one in the world knows that but me." He says the unusual symbol was a carryover from his days writing mainframe computer languages, and was used in his CP/M.

The Post-Intelligencer was not able to put that question to Gates, whose interview time was limited.

Only a computer programmer familiar with DOS could really understand what Kildall is talking about, but the implication is clear.

Paterson acknowledges there was some "low-level borrowing." But he points out that MS-DOS contained substantial improvements over Digital Research's operating system.

THE FIRST PROTOTYPE of the new IBM computer arrived at Microsoft shortly after Thanksgiving 1980.

Dave Bradley, an IBM engineer on the Project Chess team, delivered the computer to Microsoft in nine large boxes. He rented a station wagon at Sea-Tac for the trip from the airport to downtown Bellevue.

Over the next few months, Bradley would make the 4,000-mile trip between Seattle and Boca Raton more than anyone else.

"Every time I made that trip it rained," says Bradley, still with IBM in Boca Raton.

Gates had a corner office in the downtown bank building with a view of the Cascades.

"On every visit, he would tell me that if it weren't so cloudy I'd be able to see Mount Rainier," Bradley says. "I never did."

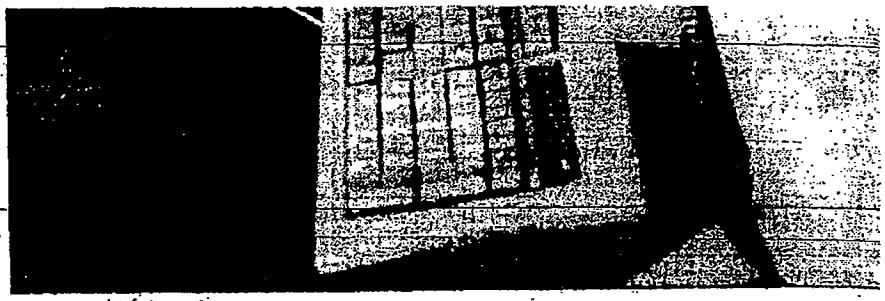
A few years later, Bradley took a vacation to Seattle just to see for himself there really was a Mount Rainier. He stayed at Paradise Inn on the mountain's flanks.

During one trip to Microsoft to bring a replacement part for the prototype, Bradley was told by his office to pick up the new BASIC Microsoft had been working on for the computer.

Bradley had an early morning flight back to Boca Raton, so about 6 a.m. he went to Microsoft's office to get the BASIC. He found Gates sprawled across the floor, going through a huge printout with a red pen. He had been up all night making last-minute fixes in the program.

Paterson left Seattle Computer Products in May 1981 and went to work for Microsoft, where he officially learned for the first time what he already suspected — the customer for his operating system was IBM.

On July 27, 1981, Allen and Brock, owner of Seattle Computer Products,



Gary Kildall of Digital Research says there were "remarkable similarities" between MS-DOS and his CP

Tim Paterson, tem allowed Bil deal of the centu signed-a-contra all rights to 86- than a month Astoria Hotel in unveiled its new

As part of Gates was able to other hardw result was a cl puters, all using system.

It was the n Chairman Bill more competit game of Monop was no longer the operating sy

"There was tion on Bill's pa the market," sa had worked wi que on softwar first personal c his own softwa Soft, in New Yo

Speaking ge utation as a h software game, it's a highly While he is try on top of the are maneuverin

"The comp against are oft nies," Gates say are sitting ther we do to get the moving busines

And Gates than his compet

"They are j ple," Kildall sa

Scout's honor

Chairman Bill was a Boy Scout, a member of Troop No. 186, which met at Sandpoint Elementary School near Magnuson Park. "Bill was a fun guy to have along (on camping trips). He was always very humorous and cheerful," says Karl Oles, a Lakeside friend and member of the Scout troop. Oles is now a Seattle attorney.

Here's a betting man who loves to play the game

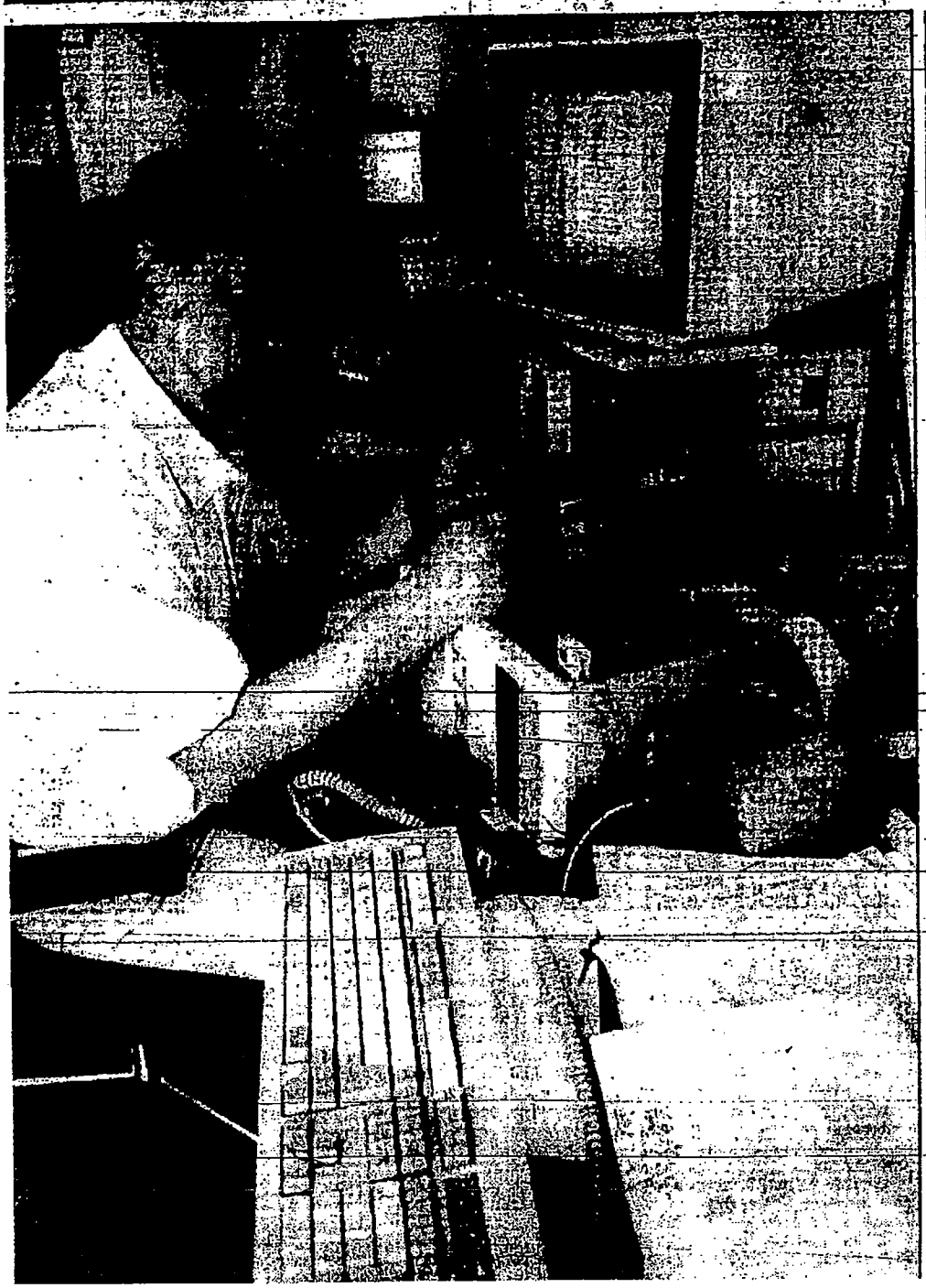
Chairman Bill loves to test his breadth of knowledge against others, especially when there is something to win or lose.

A woman friend remembers being at a restaurant with Gates when he was betting on all sorts of things, like who was older, Jimmy Treybig (founder of Tandem Computer Inc.) or Sam Armacost (then president of Bank of America). The bets were small, $10 to $20.

But the betting usually escalates into double or nothing, with a few side bets.

Once, Gates was betting with friends in a bar about what year the MGM Hotel in Las Vegas burned down. They had to call the MGM and ask.

Chairman Bill lost about $1,200 that night. ■



But he lands a stinging jab at his friend Gates when he adds: "Bill is a very bright person, but he's more of a businessman than innovator."

Bill Gates does not like to hear that he is not an innovator.

"IT DRIVES HIM NUTS," says Stewart Alsop, a well-known industry observer and computer newsletter author. He has known Gates for years.

"It's a dangerous subject to bring up with him, but it's true."

Alsop says Microsoft is not an innovative company because it does applied research, rather than long-term research and development like IBM and Apple.

He has raised this issue with Gates several times.

"Bill will sit there, and say 'Hey, what about this,' and start rattling off everything Microsoft has ever done," Alsop says.

The IBM deal was clearly a case where Gates was opportunistic rather than innovative. He took existing operating system software and reworked it to Microsoft's advantage.

> **"Gates was able to maximize a series of good fortune and lucky breaks. There was no foresight, no imagination, no brilliant maneuvers; just a lucky break..."**
>
> — Seymour Rubinstein
> founder of Wordstar.

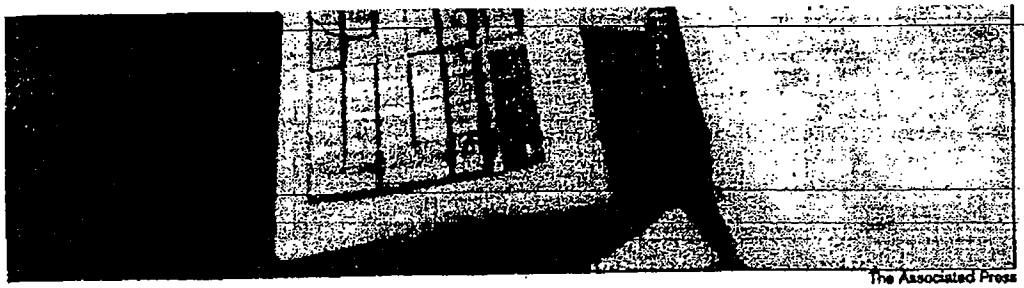
Not long after the IBM project was done, Gates received a form letter from Big Blue: "Dear Vendor, You've done a fine job."

It wasn't the warmest way to begin a marriage, but in a way that letter came to reflect the relationship over the decade. It was just business. They were never close.

Gates' relationship with IBM has been characterized by whoever was responsible for IBM's desktop computer division.

He got along famously with Estridge, the leader of the Project Chess team. They trusted each other. Estridge understood the technology. But he was killed in the Delta Airlines crash at Dallas/Forth Worth Airport in 1985.

Estridge was replaced by Bill Lowe, who didn't understand the technology as well. He and Gates



The Associated Press

Kildall of Digital Research says there were "remarkable similarities" between MS-DOS and his CP/M operating system.

oping an operating system, but
ng came of it.
Ve kept asking ourselves if we
d really be sending all this
ess to Gary," Wood says. "We
's came back with the same
er: We have all this other stuff to

› when IBM came to Microsoft,
once again sent a customer to
ll.
it the deal for CP/M fell through.
ll reportedly refused to make
ges IBM said it wanted in the
ting system. Regardless of what
happened, most of those in the
uter industry believe Kildall
y blew it, and thus rolled out the
arpet for the coronation of the
oftware king, Chairman Bill.
ter sending IBM to talk with
ll, Gates had been able to
ade the Project Chess engineers
esign their computer around
s new 8086 microcomputer chip,
was far more powerful than
chips in use at the time and
l allow more sophisticated soft-
programs.
ates knew just where to go for an
ting system that would work on
086 chip: Up the road to Tuk-
to a mom-and-pop computer
ess called Seattle Computer
ucts. There, Tim Paterson had
ned an operating system called
OS for the company's new com-
that used the 8086 chip.
licrosoft cofounder Paul Allen
cted Rod Brock, owner of Seat-
omputer Products, and told him
soft had a customer for 86-DOS
wanted to further develop and
et the operating system.
e couldn't, of course, tell Brock
stomer was IBM.
ock agreed, and Paterson went
ork on the changes Microsoft
d.
d Gates, who must have felt like
st filled an inside straight in
, flew off to Boca Raton to make
al report to IBM and secure the

Gates was able to maximize a
of good fortune and lucky
s," says Seymour Rubinstein,
f the pioneers of the personal
uter revolution who founded the
are company Wordstar.
"There was no foresight, no imagi-
n; no brilliant maneuvers," he
"just a lucky break caused by a
ination of Digital Research
ing up and Seattle Computer
ucts having something which
t very good that could be modi-
or IBM."
Kildall, still chief executive of
l Research, says there were
arkable similarities" between
MS-DOS and his CP/M operating system.

"Ask Bill why function code 6 (in DOS) ends with a dollar sign," Kildall says. "No one in the world knows that but me." He says the unusual symbol was a carryover from his days writing mainframe computer languages, and was used in his CP/M.

The Post-Intelligencer was not able to put that question to Gates, whose interview time was limited.

Only a computer programmer familiar with DOS could really understand what Kildall is talking about, but the implication is clear.

Paterson acknowledges there was some "low-level borrowing." But he points out that MS-DOS contained substantial improvements over Digital Research's operating system.

THE FIRST PROTOTYPE of the new IBM computer arrived at Microsoft shortly after Thanksgiving 1980.

Dave Bradley, an IBM engineer on the Project Chess team, delivered the computer to Microsoft in nine large boxes. He rented a station wagon at Sea-Tac for the trip from the airport to downtown Bellevue.

Over the next few months, Bradley would make the 4,000-mile trip between Seattle and Boca Raton more than anyone else.

"Every time I made that trip it rained," says Bradley, still with IBM in Boca Raton.

Gates had a corner office in the downtown bank building with a view of the Cascades.

"On every visit, he would tell me that if it weren't so cloudy I'd be able to see Mount Rainier," Bradley says. "I never did."

A few years later, Bradley took a vacation to Seattle just to see for himself there really was a Mount Rainier. He stayed at Paradise Inn on the mountain's flanks.

During one trip to Microsoft to bring a replacement part for the prototype, Bradley was told by his office to pick up the new BASIC Microsoft had been working on for the computer.

Bradley had an early morning flight back to Boca Raton, so about 5 a.m. he went to Microsoft's office to get the BASIC. He found Gates sprawled across the floor, going through a huge printout with a red pen. He had been up all night making last-minute fixes in the program.

Paterson left Seattle Computer Products in May 1981 and went to work for Microsoft, where he officially learned for the first time what he already suspected — the customer for his operating system was IBM.

On July 27, 1981, Allen and Brock, owner of Seattle Computer Products,

P-I/1986

Tim Paterson, whose operating system allowed Bill Gates to secure the deal of the century with IBM.

signed a contract that gave Microsoft all rights to 86-DOS for $50,000. Less than a month later, in the Waldorf-Astoria Hotel in New York City, IBM unveiled its new computer.

As part of the deal with IBM, Gates was able to relicense MS-DOS to other hardware companies. The result was a clone industry of computers, all using Microsoft's operating system.

It was the new industry standard. Chairman Bill had eliminated one more competitor in this real-life game of Monopoly. Digital Research was no longer a significant player in the operating system market.

"There was absolute determination on Bill's part to take them out of the market," says Eddie Curry, who had worked with Gates in Albuquerque on software for the Altair, the first personal computer. He now has his own software company, ImageSoft, in New York.

Speaking generally about his reputation as a hardball player in the software game, Gates points out that it's a highly competitive business. While he is trying to keep Microsoft on top of the hill, other companies are maneuvering to knock him off.

"The companies we compete against are often very fine companies," Gates says, "and you know they are sitting there thinking, 'What can we do to get these guys' . . . It's a fast-moving business."

And Gates usually moves faster than his competitors.

"They are just good business people," Kildall says of Microsoft.

team. They trusted each other. Estridge understood the technology. But he was killed in the Delta Airlines crash at Dallas/Forth Worth Airport in 1985.

Estridge was replaced by Bill Lowe, who didn't understand the technology as well. He and Gates clashed over Microsoft's development of Windows.

Lowe was replaced by Jim Cannavino in 1989. By then, there was great suspicion of Microsoft within IBM, and the relationship became even more strained.

Last September, InfoWorld, a highly regarded trade publication, reported that Gates took a group of executives from Lotus Development Corp., the world's second largest software company, to dinner, and after a few too many drinks began trashing IBM and Cannavino, saying IBM wouldn't be around in 10 years, and he, Gates, would rule all.

Sort of a Microsoft Uber Alles.

One of the Lotus executives went home and wrote a memo about the incident. It eventually found its way to Cannavino's desk, according to InfoWorld.

The story was picked up by the national press, and Microsoft issued a strong denial that Chairman Bill had ever said such things about their good friends at IBM.

But Alsop, editor of P.C. Letter, the national computer newsletter, tells a story that suggests otherwise.

LAST FALL, Gates was supposed to address a software entrepreneurs forum in Palo Alto, Calif., but he was running late after an all-day meeting with Cannavino in Milwaukee, Wis. Gates arrived about five minutes before he was to speak to about 600 people.

Afterwards, Gates and Alsop went to a bar at the Il Fornaio in Palo Alto, next to the hotel Gates was booked in.

The speech had gone well, and Gates was really up and having a good time. Alsop, knowing IBM and Microsoft were going through a terrible strain, decided it was the right time to ask about Cannavino.

"Whenever you talk to him," Alsop says, "if you raise any subject about personal computers he will immediately focus on it, and start rocking back and forth, looking you straight in in the eye, asking the fundamental questions. You've got to remember, he really knows everything.

"So I asked him, 'What do you think of Jim Cannavino?' By luck, I hit a hot button. He started ranting and raving, saying 'you would not believe what this guy was telling me, how Cannavino was telling him, Bill Gates, how he should be running Microsoft. . . .

"It was clear," Alsop says, "that Gates had to restrain himself from saying more about Jim."

Alsop believes Gates may be making his first serious blunder at Microsoft by believing he no longer needs those ties to IBM.

"If I'm right, and Gates believes he can operate away from IBM, the very thing that made Microsoft successful will have changed," Alsop says. "Perhaps Gates can pull it off, perhaps not."

■ **Tomorrow:** The silicon bully.

EXHIBIT Q

Design goals and implementation of the new High Performance File System. (includes related article on B-Trees and B+ Trees). Roy Duncan.
Microsoft Systems Journal v4.n5 (Sept 1989): pp1(13).

Abstract:

The High Performance File System (HPFS) enhancement to OS/2 Version 1.2 solves all the problems of the File Allocation Table (FAT) file system and is designed to meet the demands expected into the next few decades. HPFS not only serves as a way to organize data on random access block storage devices, but is also a software module that translates file-oriented requests from applications programs to device drivers. HPFS is also an example of an installable file system, which makes it possible to access several incompatible volume structures on the same OS/2 system simultaneously. Excellent throughput is achieved by the use of advanced data structures such as intelligent caching, read-ahead and write-behind. Disk space is managed more economically by the use of sectoring. HPFS also includes greatly improved fault tolerance. Applications programs need only simple modifications to make use of extended attributes and long filenames.

THE HPFS IS A WAY OF ORGANIZING DATA ON A RANDOM ACCESS BLOCK STORAGE DEVICE. IT IS ALSO A SOFTWARE MODULE THAT TRANSLATES FILE-ORIENTED REQUESTS FROM AN APPLICATION PROGRAM INTO MORE PRIMITIVE REQUESTS THAT A DEVICE DRIVER CAN UNDERSTAND.

The High Performance File System (hereafter HPFS), which is making its first appearance in the OS/2 operating systemVersion 1.2, had its genesis in the network division of Microsoft and was designed by Gordon Letwin, the chief architect of the OS/2 operating system. The HPFS has been designed to meet the demands of increasingly powerful PCs, fixed disks, and networks for many years to come and to serve as a suitable platform for object-oriented languages, applications, and user interfaces.

The HPFS is a complex topic because it incorporates three distinct yet interrelated file system issues. First, the HPFS is a way of organizing data on a random access block storage device. Second, it is a software module that translates fi le-oriented requests from an application program into more primitive requests that a device driver can understand, using a variety of creative techniques to maximize performance. Third, the HPFS is a practical illustration of an important new OS/2 feature known as installable file systems.

This article introduces the three aspects of the HPFS. But first, it puts the HPFS in perspective by reviewing some of the problems that led to the system's existence.

FAT File System

The so-called FAT file system, which is the file system used in all versions of the MS-DOS™ operating system to date and in the first two releases of OS/2' (Versions 1.0 and 1.1), has a dual heritage in Microsoft's earliest programming language products and the Digital Research(R) CP/M(R) operating system-software originally written for 8080-based and Z-80-based microcomputers. It inherited characteristics from both ancestors that

have progressively tumed into handicaps in this new era of multitasking, protected mode, virtual memory, and huge fixed disks.

The FAT file system revolves around the File Allocation Table for which it is named. Each logical volume has its own FAT, which serves two important functions: it contains the allocation information for each file on the volume in the form of linked lists of allocation units (clusters, which are power-of-2 multiples of sectors) and it indicates which allocation units are free for assignment to a file that is being created or extended.

The FAT was invented by Bill Gates and Marc McDonald in1977 as a method of managing disk space in the NCR version of standalone Microsoft* Disk BASIC. Tim Paterson, at that time an employee of Seattle (R) Computer Products (SCP), was introduced to the FAT concept when his company shared a booth with Microsoft at the National Computer Conference in 1979. Paterson subsequently incorporated FATs into the file system of 86-DOS, an operating system for SCP's S-100 bus 8086 CPU boards. 86-DOS was eventually purchased by Microsoft and became the starting point for MS-DOS2 Version 1.0, which was released for the original IBM* PC in August 1981.

When the FAT was conceived, it was an excellent solution to disk management, mainly because the floppy disks on which it was used were rarely larger than I Mb. On such di s, the FAT was small enough to be held in memory at all times, allowing very fast random access to any part of any file. This proved far superior to the CP/M method of tracking disk space, in which the information about the sectors assigned to a file might be spread across many directory entries, which were in turn scattered randomly throughout the disk directory.

When applied to fixed disks, however, the FAT began to look more like a bug than a feature. It became too large to be held entirely resident and had to be paged into memory in pieces; this paging resulted in many superfluous disk head movements as a program was reading through a file and degraded system throughput. In addition, because the information about free disk space was dispersed across many sectors of FAT, it was impractical to allocate file space contiguously, and file fragmentation became another obstacle to good performance. Moreover, the use of relatively large clusters on fixed disks resulted in a lot of dead space, since an average of one-half cluster was wasted for each file. (Some network servers use clusters as large as 64Kb.)

The FAT file system's restrictions on naming files and directories are inherited from CP/M. When Paterson was writing 86DOS, one of his primary objectives was to make programs easy to port from CP/M to his new operating system. He therefore adopted CP/M's limits on filenames and extensions so the critical fields of 86-DOS File Control Blocks (FCBs) would look almost exactly like those of CP/M. The sizes of the FCB filename and extension fields were also propagated into the structure of disk directory entries. In due time, 86-DOS became MSDOS and application programs for MS-DOS proliferated beyond anyone's wildest dreams. Since most of the early programs depended on the structure of FCBs, the 8.3 format for filenames became irrevocably locked into the system.

During the last couple of years, Microsoft and IBM have made valiant attempts to prolong the useful life of the FAT file system by lifting the restrictions on volume sizes, improving allocation strategies, caching pathnames, and moving tables and buffers into expanded memory. But these can only be regarded as temporizing measures, because the fundamental data structures used by the FAT file system are simply not well suited to large random access devices.

The HPFS solves the FAT file system problems mentioned here and many others, but it is not derived in any way from the FAT file system. The architect of the HPFS started with a clean sheet of paper and designed a file system that can take full advantage of a multitasking environment, and that will be able to cope with any sort of disk device likely to arrive on microcomputers during the next decade.

HPFS Volume Structure

HPFS volumes are a new partition type-type 7-and can exist on a fixed disk alongside of the several previously defined FAT partition types. IBM-compatible HPFS volumes use a sector size of 512 bytes and have a maximum size of 2199Gb (2" sectors). Although there is no particular reason why floppy disks can't be formatted as HPFS volumes, Microsoft plans to stick with FAT file systems on floppy disks for the foreseeable future. (This ensures that users will be able to transport files easily between MS-DOS and OS/2 systems.)

An HPFS volume has very few fixed structures (Figure 1). Sectors 0-15 of a volume (8Kb) are the BootBlock and contain a volume name, 32-bit volume ID, and a disk bootstrap program. The bootstrap is relatively sophisticated (by MS-DOS standards) and can use the HPFS in a restricted mode to locate and read the operating system files wherever they might be found.

Sectors 16 and 17 are known as the SuperBlock and the SpareBlock respectively. The SuperBlock is only modified by disk maintenance utilities. It contains pointers to the free space bitmaps, the bad block list, the directory block band, and the root directory. It also contains the date that the volume was last checked out and repaired with CHKDSK/F. The SpareBlock contains various flags and pointers that will be discussed later; it is modified, although infrequently, as the system executes.

The remainder of the disk is divided into 8Mb bands. Each band has its own free space bitmap in which a bit represents each sector. A bit is 0 if the sector is in use and I if the sector is available. The bitmaps are located at the head or tail of a band so that two bitmaps are adjacent between alternate bands. This allows the maximum contiguous free space that can be allocated to a file to be 16Mb. One band, located at or toward the seek center of the disk, is called the directory block band and receives special treatment (more about this later). Note that the band size is a characteristic of the current implementation and may be changed in later versions of the file system.

Files and Fnodes

Every file or directory on an HPFS volume is anchored on a fundamental file system object called an Fnode (pronounced "eff node"). Each Fnode occupies a single sector and contains control and access history information used internally by the file system, extended attributes and access control lists (more about this later), the length and the first 15 characters of the name of the associated file or directory, and an allocation structure (Figure 2). An Fnode is always stored near the file or directory that it represents.

The allocation structure in the Fnode can take several forms, depending on the size and degree of contiguity of the file or directory. The HPFS views a file as a collection of one or more runs or extents of one or more contiguous sectors. Each run is symbolized by a pair of doublewords-a 32-bit starting sector number and a 32-bit length in sectors (this is referred to as runlength encoding). From an application program's point of view, the extents are invisible; the file appears as a seamless stream of bytes.

The space reserved for allocation information in an Fnode can hold pointers to as many as eight runs of sectors of up to 16Mb each. (This maximum run size is a result of the band size and free space bitmap placement only; it is not an inherent limitation of the file system.) Reasonably small files or highly contiguous files can therefore be described completely within the Fnode (Figure 3).

HPFS uses a new method to represent the location of files that are too large or too fragmented for the Fnode and consist of more than eight runs. The Fnode's allocation structure becomes the root for a B+ Tree of allocation sectors, which in turn contain the actual pointers to the file's sector runs (see Figure 4 and the sidebar, "BTrees and B+

Trees"). The Fnode's root has room for 12 elements. Each allocation sector can contain, in addition to various control information, as many as 40 pointers to sector runs. Therefore, a two-level allocation B+ Tree can describe a file of 480 (12*40) sector runs with a theoretical maximum size of 7.68Gb (12*40*16Mb) in the current implementation (although the 32-bit signed offset parameter for DosChgFilePtr effectively limits file sizes to 2Gb).

In the unlikely event that a two-level B+ Tree is not sufficient to describe a highly fragmented file, the file system will introduce additional levels in the tree as needed. Allocation sectors in the intermediate levels can hold as many as 60 internal (nonterminal) B+ Tree nodes, which means that the descriptive ability of this structure rapidly grows to numbers that are nearly beyond comprehension. For example, a threelevel allocation B+ Tree can describe a file with as many as 28,800 (12*60*40) sector runs.

Run-length encoding and B+ Trees of allocation sectors are a memory-efficient way to specify a file's size and location, but they have other significant advantages. Translating a logical file offset into a sector number is extremely fast: the file system just needs to traverse the list (or B+ Tree of lists) of run pointers until it finds the correct range. It can then identify the sector within the run with a simple calculation. Run-length encoding also makes it trivial to extend the file logically if the newly assigned sector is contiguous with the file's previous last sector; the file system merely needs to increment the size doubleword of the file's last run pointer and clear the sector's bit in the appropriate freespace bitmap.

Directories

Directories, like files, are anchored on Fnodes. A pointe to the Fnode for the root directory is found in the SuperBlock. The Fnodes for directories other than the root are reache through subdirectory entries in their parent directories.

Directories can grow to any size and are built up from 2Kb directory blocks, which are allocated as four consecutive sectors on the disk. The file system attempts to allocate directory blocks in the directory band, which is located at or near the seek center of the disk. Once the directory band is full, the directory blocks are allocated wherever space is available.

Each 2Kb directory block contains from one to many directory entries. A directory entry contains several fields, including time and date stamps, an Fnode pointer, a usage count for use by disk maintenance programs, the length of the file or directory name, the name itself, and a B-Tree pointer. Each entry begins with a word that contains the length of the entry. This provides for a variable amount of flex space at the end of each entry, which can be used by special versions of the file system and allows the directory block to be traversed very quickly (Figure 5).

The nuniber of entries in a directory block varies with the length of names. If the average filename length is 1 3 characters, an average directory block will hold about 40 entries. The entries in a directory block are sorted by the binary lexical order of their name fields (this happens to put them in alphabetical order for the U.S. alphabet). The last entry in a directory block is a dummy record that marks the end of the block.

When a directory gets too large to be stored in one block, it increases in size by the addition of 2Kb blocks that are organized as a B-Tree"B-Trees and B+ Trees"). When searching for a specific name, the file system traverses a directory block until it either finds a match or finds a name that is lexically greater than the target. In the latter case, the file system extracts the BTree pointer from the entry. If there is no pointer, the search failed; otherwise the file system follows the pointer to the next directory block in the tree and continues the search.

A little back-of-the-envelope arithmetic yields some impressive statistics. Assuming 40 entries per block, a two-level tree of directory blocks can hold 1640 directory entries and a three-level tree can hold an astonishing 65,640 entries. In other words, a particular file can be found (or shown not to exist) in a typical directory of 65,640 files with a maximum of three disk hits-the actual number of disk accesses depending on cache contents and the location of the file's name in the directory block B-Tree. That's quite a contrast to the FAT file system, where in the worst case more than 4000 sectors would have to be read to establish that a file was or was not present in a directory containing the same number of files.

The B-Tree directory structure has interesting implications beyond its effect on open and find operations. A file creation, renaming, or deletion may result in a cascade of complex operations, as directory blocks are added or freed or names are moved from one block to the other to keep the tree balanced. In fact, a rename operation could theoretically fail for lack of disk space even though the file itself is not growing. In order to avoid this sort of disaster, the HPFS maintains a small pool of free blocks that can be drawn from in a directory emergency; a pointer to this pool of free blocks is stored in the SpareBlock.

Extended Attributes

File attributes are information about a file that is maintained by the operating system outside the file's overt storage area. The FAT file system supports only a few simple attributes (read only, system, hidden, and archive) that are actually stored as bit flags in the file's directory entry; these attributes are inspected or modified by special function calls and are not accessible through the normal file open, read, and write calls.

The HPFS supports the same attributes as the FAT file system for historical reasons, but it also supports a new form of fileassociated, highly generalized information called Extended Attributes (EAs). Each EA is conceptually similar to an environment variable, taking the form

name- -value

except that the value portion can be either a null-terminated (ASCIIZ) string or binary data. In OS/2 1.2, each file or directory can have a maximum of 64Kb of EAs attached to it. This limit may be lifted in a later release of OS/2.

The storage method for EAs can vary. If the EAs associated with a given file or directory are small enough, they will be stored right in the Fnode. If the total size of the EAs is too large, they are stored outside the Fnode in sector runs, and a B+ Tree of allocation sectors can be created to describe the runs. If a single EA gets too large, it can be pushed outside the Fnode into a B+ Tree of its own.

The kernel API functions DosQFileInfo and DosSetFileInfo have been expanded with new information levels that allow application programs to manipulate extended attributes for files. The new functions DosQPathInfo and DosSetPathInfo are used to read or write the EAs associated with arbitrary pathnames. An application program can either ask for the value of a specific EA (supplying a name to be matched) or can obtain all of the EAs for the file or directory at once.

Although application programs can begin to take advantage of EAs as soon as the HPFS is released, support for EAs is an essential component in Microsoft's long-range plans for object-oriented file systems, Information of almost any type can be stored in EAs, ranging from the name of the application that owns the file to names of dependent files to icons to executable code. As the HPFS evolves, its facilities for manipulating EAs are likely to become much more sophisticated. It's easy to imagine, for example, that in future versions the API might be extended with EA functions that are analogous to DosFindFirst and DosFindNext and EA data might get organized into B -Trees.

I should note here that in addition to EAs, the LAN Manager version of HPFS will support another class of file-associated information called Access Control Lists (ACLs). ACLs have the same general appearance as EAs and are manipulated in a similar manner, but they are used to store access rights, passwords, and other information of interest in a networking multiuser environment.

Installable File Systems

Support for installable file system has been one of the most eagerly anticipated features of OS/2 Version 1.2. It will make it possible to access multiple incompatible volume structures-FAT, HPFS, CD ROM, and perhaps even UNIX(R)--on the same OS/2 system at the same time, will simplify the life of network implementors, and will open the door to rapid file system evolution and innovation. Installable file systems are, however, only relevant to the HPFS insofar as they make use of the HPFS optional. The FAT file system is still embedded in the OS/2 kernel, as it was in OS/2 1.0 and 1.1, and will remain there as the compatibility file system for some time to come.

An installable file system driver (FSD) is analogous in many ways to a device driver. An FSD resides on the disk in a file that is structured like a dynamic-link library (DLL), typically with a SYS or IFS extension, and is loaded during system initialization by IFS= statements in the CONFIG.SYS file. IFS= directives are processed in the order they are encountered and are also sensitive to the order of DEVICE= statements for device drivers. This lets you load a device driver for a nonstandard device, load a file system driver from a volume on that device, and so on.

Once an FSD is installed and initialized, the kernel communicates with it in terms of logical requests for file opens, reads, writes, seeks, closes, and so on. The FSD translates these requests-using control structures and tables found on the volume itself-into requests for sector reads and writes for which it can call special kernel entry points called File System Helpers (FsHlps). The kernel passes the demands for sector 1/0 to the appropriate device driver and returns the results to the FSD (Figure 6).

The procedure used by the operating system to associate volumes with FSDs is called dynamic mounting and works as follows. Whenever a volume is first accessed, or after it has been locked for direct access and then unlocked (for example, by a FORMAT operation), OS/2 presents identifying information from the volume to each of the FSDs in turn until one of them recognizes the information. When an FSD claims the volume, the volume is mounted and all subsequent file 1/0 requests for the volume are routed to that FSD.

Performance Issues

The HPFS attacks potential bottlenecks in disk throughput at multiple levels. It uses advanced data structures, contiguous sector allocation, intelligent caching, read-ahead, and deferred writes in order to boost performance.

First, the HPFS matches its data structures to the task at hand: sophisticated data structures (B-Trees and B+ Trees) for fast random access to filenames, directory names, and lists of sectors allocated to files or directories, and simple compact data structures (bitmaps) for locating chunks of free space of the appropriate size. The routines that manipulate these data structures are written in assembly language and have been painstakingly tuned, with special focus on the routines that search the freespace bitmaps for patterns of set bits (unused sectors).

Next, the HPFS's main goal -its prime directive, if you will- is to assign consecutive sectors to files whenever possible. The time required to move the disk's read/write head from one track to another far outweighs the other possible delays, so the HPFS works hard to avoid or minimize such head movements by allocating file space contiguously and by keeping control structures such as Fnodes and freespace bitmaps near the things

they control. Highly contiguous files also help the file system make fewer requests of the disk driver for more sectors at a time, allow the disk driver to exploit the multisector transfer capabilities of the disk controller, and reduce the number of disk completion interrupts that must be serviced.

Of course, trying to keep files from becoming fragmented in a multitasking system in which many files are being updated concurrently is no easy chore. One strategy the HPFS uses is to scatter newly created files across the disk-in separate bands, if possible-so that the sectors allocated to the files as they are extended will not be interleaved. Another strategy is to preallocate approximately 4Kb of contiguous space to the file each time it must be extended and give back any excess when the file is closed.

If an application knows the ultimate size of a new file in advance, it can assist the file system by specifying an initial file allocation when it creates the file. The system will then search all the free space bitmaps to find a run of consecutive sec tors large enough to hold the file. That failing, it will search for two runs that are half the size of the file, and so on.

The HPFS relies on several different kinds of caching to minimize the number of physical disk transfers it must request. Naturally, it caches sectors, as did the FAT file system. But unlike the FAT file system, the HPFS can manage very large caches efficiently and adjusts sector caching on a perhandle basis to the manner in which a file is used. The HPFS also caches pathnames and directories, transforming disk directory entries into an even more compact and efficient inmemory representation.

Another technique that the HPFS uses to improve performance is to preread data it believes the program is likely to need. For example, when a file is opened, the file system will preread and cache the Fnode and the first few sectors of the file's contents. If the file is an executable program or the history information in the file's Fnode shows that an open operation has typically been followed by an immediate sequential read of the entire file, the file system will preread and cache much more of the file's contents. When a program issues relatively small read requests, the file system always fetches data from the file in 2Kb chunks and caches the excess, allowing most read operations to be satisfied from the cache.

Finally, the OS/2 operating system's support for multitasking makes it possible for the HPFS to rely heavity on lazy writes (sometimes called deferred writes or write behind) to improve performance. When a program requests a disk write, the data is placed in the cache and the cache buffer is flagged as dirty (that is, inconsistent with the state of the data on disk). When the disk becomes idle or the cache becomes saturated with dirty buffers, the file system uses a captive thread from a daemon process to write the buffers to disk, starting with the oldest data.

In general, lazy writes mean that programs run faster because their read requests will almost never be stalled waiting for a write request to complete. For programs that repeatedly read, modify, and write a small working set of records, it also means that many unnecessary or redundant physical disk writes may be avoided. Lazy writes have their dangers, of course, so a program can defeat them on a per-handle basis by setting the writethrough flag in the OpenMode parameter for DosOpen, or it can commit data to disk on a perhandle basis with the DosBufReset function.

Fault Tolerance

The HPFS's extensive use of lazy writes makes it imperative for the HPFS to be able to recover gracefully from write errors under any but the most dire circumstances. After all, by the time a write is known to have failed, the application has long since gone on its way under the illusion that it has safely shipped the data into disk storage. The errors may be detected by the hardware (such as a "sector not found" error returned by the disk adapter), or they may be detected by the disk driver in spite of the hardware during a

read-after-write verification of the data.

The primary mechanism for handling write errors is called a hotfix. When an error is detected, the file system takes a free block out of a reserved hotfix pool, writes the data to that block, and updates the hotfix map. (The hotfix map is simply a series of pairs of doublewords, with each pair containing the number of a bad sector associated with the number of its hotfix replacement. A pointer to the hotfix map is maintained in the SpareBlock.) A copy of the hotfix map is t written to disk, and a warning message is displayed to let the user know that all is not well with the disk device.

Each time the file system requests a sector read or write from the disk driver, it scans the hotfix map and replaces any bad sector numbers with the corresponding good sector holding the actual data. This lookaside translation of sector numbers is not as expensive as it sounds, since the hotfix list need only be scanned when a sector is physically read or written, not each time it is accessed in the cache.

One of CHKDSK's duties is to empty the hotfix map. For each replacement block on the hotfix map, it allocates a new sector that is in a favorable location for the file that owns the data, moves the data from the hotfix block to the newly allocated sector, and updates the file's allocation information (which may involve rebalancing allocation trees and other elaborate operations). It then adds the bad sector to the bad block list, releases the replacement sector back to the hotfix pool, deletes the hotfix entry from the hotfix map, and writes the updated hotfix map to disk.

Of course, write errors that can be detected and fixed on the fly are not the only calamity that can befall a file system. The HPFS designers also had to consider the inevitable damage to be wreaked by power failures, program crashes, malicious viruses and Trojan horses, and those users who tum off the machine without selecting Shutdown in the Presentation Manager Shell. (Shutdown notifies the file system to flush the disk cache, update directories, and do whatever else is necessary to bring the disk to a consistent state.)

The HPFS defends itself against the user who is too abrupt with the Big Red Switch by maintaining a Dirty FS flag in the SpareBlock of each HPFS volume. The flag is only cleared when all files on the volume have been closed and all dirty buffers in the cache have been written out or, in the case of the boot volume (since OS2.INI and the swap file are never closed), when Shutdown has been selected and has completed its work.

During the OS/2 boot sequence, the file system inspects the DirtyFS flag on each HPFS volume and, if the flag is set, will not allow further access to that volume until CHKDSK has been run. If the DirtyFS flag is set on the boot volume, the system will refuse to boot; the user must boot OS/2 in maintenance mode from a diskette and run CHKDSK to check and possibly repair the boot volume.

In the event of a truly major catastrophe, such as loss of the SuperBlock or the root directory, the HPFS is designed to give data recovery the best possible chance of success. Every type of crucial file object-including Fnodes, allocation sectors, and directory blocks-is doubly linked to both its parent and its children and contains a unique 32-bit signature. Fnodes also contain the initial portion of the name of their file or directory. Consequently, CHKDSK can rebuild an entire volume by methodically scanning the disk for Fnodes, allocation sectors, and directory blocks, using them to reconstruct the files and directories and finally regenerating the freespace bitmaps.

Application Programs and the HPFS

Each of the OS/2 releases thus far have carried with them a major discontinuity for application programmers who learned their trade in the MS-DOS environment. In OS/2 1.0, such programmers were faced for the first time with virtual memory, multitasking, interprocess communications, and the protected mode restrictions on addressing and

direct control of the hardware and were challenged to master powerful new concepts such as threading and dynamic linking. In OS/2 Version 1.1, the stakes were raised even further. Programmers were offered a powerful hardware-independent graphical user interface but had to restructure their applications drastically for an event-driven environment based on objects and message passing.

In OS/2 Version 1.2, it is time for many of the file-oriented programming habits and assumptions carried forward from the MS-DOS environment to fall by the wayside. An application that wishes to take full advantage of the HPFS must allow for long, free-form, mixed-case filenames and paths with few restrictions on punctuation and must be sensitive to the presence of EAs and ACLs. After all, if EAs are to be of any use, it won't suffice for applications to update a file by renaming the old file and creating a new one without also copying the EAs.

But the necessary changes for OS/2 Version 1.2 are not tricky to make. A new API function, DosCopy, helps applications create backups-it essentially duplicates an existing file together with its EAs. EAs can also be manipulated explicitly with DosQFileInfo, DosSetFileInfo, DosQPathInfo, and DosSetPathInfo. A program should call DosQSysInfo at run time to find the maximum possible path length for the system, and ensure that all buffers used by DosChDir, DosQCurDir, and related functions are sufficiently large. Similarly, the buffers used by DosOpen, DosMove, D o s G e t M o d N a m e , DosFindFirst, DosFindNext, and like functions must allow for longer filenames. Any logic that folds cases in filenames or tests for the occurrence of only one dot delimiter in a filename must be rethought or eliminated

The other changes in the API will not affect the average application. The functions DosQFileInfo, DosFindFirst, and DosFindNext now return all three sets of times and dates (created, last accessed, last modified) for a file on an HPFS volume, but few programs are concerned with time and date stamps anyway. DosQFsInfo is used to obtain volume labels or disk characteristics just as before, and the use of DosSetFsInfo for volume labels is unchanged. There are a few totally new API functions such as DosFsCtl (analogous to DosDev IOCtl but used for communication between an application and an FSD), DosFsAttach (a sort of explicit mount call), and DosQFsAttach (determines which FSD owns a volume); these are intended mainly for use by disk utility programs.

In order to prevent old OS/2 applications and MS-DOS applications running in the DOS box from inadvertently damaging HPDS files, a new flag bit has been defined in the EXE file header that indicates whether an application is HPFS-aware. If this bit is not set, the application will only be able to search for, open, or create files on HPFS volumes that are compatible with the FAT file system's 8.3 naming conventions. If the bit is set, OS/2 allows access to all files on an HPFS volume, because it assumes that the program knows how to handle long, free-form filenames and will take the responsibility of conserving a file's EAs and ACLs.

Summary

The HPFS solves all of the historical problems of the FAT file system .It achieves excellent throughput even in extreme cases-many very small files or a few very large files-by means of advanced data structures and techniques such as intelligent caching, read-ahead, and write-behind. Disk space is used economically because it is managed on a sector basis. Existing application programs will need modification to take advantage of the HPFS's support for extended attributes and long filenames, but these changes will not be difficult. All application programs will benefit from the HPFS's improved performance and decreased CPU use whether they are modified or not. This article is based on a prerelease version of the HPFS that was still undergoing modification and tuning. therefore, the final release of the HPFS may differ in some details from the description given here--Ed.

B- Trees and B+ Trees

Most programmers are at least passingly familiar with the data structure known as a binary tree. Binary trees are a technique for imposing a logical ordering on a collection of data items by means of pointers, without regard to the physical order of the data.

In a simple binary tree, each node contains some data, including a key value that determines the node's logical position in the tree, as well as pointers to the node's left and right subtrees. The node that begins the tree is known as the root; the nodes that sit at the ends of the tree's branches are sometimes called the leaves.

To find a particular piece of data, the binary tree is traversed from the root, At each node, the desired key is compared with the node's key; if they don't match, one branch of the node's subtree or another is selected based on whether the desired key is less than or greater than the node's key. This process continues until a match is found or an empty subtree is encountered (see Figure A).

Such simple binary trees, although easy to understand and implement, have disadvantages in practice. If keys are not well distributed or are added to the tree in a non-random fashion, the tree can become quite asymmetric, leading to wide variations in tree traversal times.

In order to make access times uniform, many programmers prefer a particular type of balanced tree known as a B-Tree. For the purposes of this discussion, the important points about a B-Tree are that data is stored in all nodes, more than one data item might be stored in a node, and all of the branches of the tree are of identical length (see Figure B).

The worst-case behavior of a B-Tree is predictable and much better than that of a simple binary tree, but the maintenance of a B-Tree is correspondingly more complex. Adding a new data item, changing a key value, or deleting a data item may result in the splitting or merging of a node, which in tum forces a cascade of other operations on the tree to rebalance it.

A B+ Tree is a specialized form of B-Tree that has two types of nodes: internal which only point to other nodes, and external, which contain the actual data (see Figure C).

The advantage of a B+ Tree over a B- Tree is that the internal nodes of the B+ Tree can hold many more decision values than the intermediate-level nodes of a B-Tree, so the fan out of the tree is faster and the average length of a branch is shorter. This makes up for the fact that you must always follow a B+ Tree branch to its end to get the data for which you are looking, whereas in a B-Tree you may discover the data at an intermediate node or even at the root.

Source Citation: Duncan, Roy. "Design goals and implementation of the new High Performance File System. (Includes related article on B-Trees and B+ Trees)." *Microsoft Systems Journal* 4.n5 (Sept 1989): 1(13). *Expanded Academic ASAP.* Thomson Gale. University of Washington. 9 Jan. 2007 <http://find.galegroup.com/itx/infomark.do?&contentSet=IAC-Documents&type=retrieve&tabID=T003&prodId=EAIM&docId=A7585454&source=gale&srcprod=EAIM&userGroupName=wash_eai&version=1.0>.

Thomson Gale Document Number:A7585454